Improving System Integration using a Modular Configuration Specification Language

Markus Raab
Vienna University of Technology
Institute of Computer Languages
markus.raab@complang.tuwien.ac.at

Abstract

In today's systems we often plug together configurable standard components in a modular way. Most software, however, does not specify its configuration in a way suitable for other software. The aim of our configuration specification language SpecElektra is to fill this gap. It allows us to externally specify the configuration items of non-standardized configuration files. In SpecElektra we assign properties that enable additional validations and transformations. As a result, we can safely and easily configure software at run-time. The approach integrates standard software while retaining its modularity. We demonstrate how high-level configuration items help us to cope with changes in system-oriented goals.

Categories and Subject Descriptors D.2.12 [*Software Engineering*]: Interoperability

Keywords System Integration, Modularity, System-oriented goals

1. Introduction

In a world where systems get increasingly complex we take special care to not miss ever-shifting system-oriented goals. Modularity presents a well-established mechanism to cope with complexity. Instead of building every system from scratch, we aim towards configuring existing components in order to create new systems.

Most standard software does not consider to be part of an integral whole. Instead it often provide run-time configurability only via their specific configuration files. We have to configure each application individually. Such an endeavour can be complex and error-prone. Configuration files use different syntax and software to access them are written in different languages. Our aim for system integration is to have options to tune the whole system.

As running example we will employ an embedded location-tracking device. On this device we install the time synchronization daemon `ntpd`. At startup `ntpd` considers a configuration file named `ntp.conf`. Certain context changes require us to reconfigure `ntpd`. Suppose the power-source of the device switches to battery. When such a system-wide context changes, we want to modify many settings in order to save energy. One of the possible settings to be changed is the reduction of ntp-polling. We might directly communicate new settings via inter-process communication. To make changes persistent, however, we need to change the poll settings which are located in `ntp.conf`. Thus the more generic approach is to change `ntp.conf` and notify `ntpd` to reread its configuration.

To elegantly deal with changing context the use of context-oriented programming (COP) was proposed [6]. It enables modularization of dynamically changing behaviours. The usage of COP, however, is challenging if we want to work with unmodified standard software. Additionally, in COP we need to decide during implementation which contexts we want to consider. We cannot directly apply COP-techniques, the focuses are too different. We want to investigate how to postpone such decisions until system integration. Our goal is a language that externally specifies configuration. By changes in the specification we adapt to system-oriented goals.

Currently, there are two major directions for run-time configuration: (1) Some embedded and/or proprietary systems have global registries. In this scenario every application interacts with the whole configuration of the system. It poses the advantage of very good integration capabilities. The downside is that all applications need to be rewritten to work with the global registry. (2) In most systems every application has its own configuration files. Every application has full control over every aspect of run-time configuration access. So we have the extreme opposite: a fully modular system without compromise. The specification of the configuration happens within every application, e.g., using XSD.

With all advantages the fully modular approach has, it completely fails with one system-oriented goal: It gets practically impossible that an application has access to the configuration of every other application. As a consequence applications do not access others configurations for better integration at a large scale. Thus if you want to configure a system you have to tinker with the configuration of each application. Our vision is that a specification language defines the configuration access. If done in a modular way it should be able to fulfill today's diverse requirements. Application specific adapters are necessary to avoid a lowest common denominator.

According to our vision we propose the configuration specification language SpecElektra. Specifications in it describe a global key-database that mounts application-specific configuration files. The aim is to integrate previously unrelated components via their run-time configuration. For example, using a specification in which the key-database integrates all the values of `ntp.conf`, we transform the battery status present on the system to other values, including one suitable for `ntpd`:

```
1 [battery/level]
2 check/enum = critical, low, high, full
3 [ntp]
4 mountpoint = ntp.conf
5 transform/batterytontp = battery/level maxpoll
6 [locationtracker]
7 transform/batterytotracker = battery/level
```

The specification language is semi-structured. The syntax used in our examples resembles INI files. It only has two constructs: Within `[...]` in lines 1, 3 and 6 we write keys that refer to values, found in configuration files but not in the specification. The other lines specify properties of the keys. Using the property `mountpoint` (line 4), all key/values below `[ntp]` will be persisted in the configuration file `ntp.conf`. The property `check/enum` (line 2) validates values of `[battery/level]`. The lines `transform/*` in lines 5 and 7 describe two transformation properties. Because of line 7 the configuration value of `[locationtracker]` is calculated from `[battery/level]`'s value. Functionality of properties are implemented as plugins that process configuration during access. The plugins sometimes facilitate COP techniques.

COP describes the current context by the combination of layers. Each layer is associated with specific behavior at run-time. COP demands a distinction of objects and context. Explicit layers make developers more aware of the contextual situation. The gestalt of the context obviously depends on the requirements. In our specification language every key can be interpreted as context. We can easily connect the data in unanticipated ways to tackle new goals.

For example, we use the components of the example above in a vehicle instead of a wearable device. Then the context of the polling time would be different. The speed of the vehicle replaces the battery level as dominant context. When such a system-oriented goal changes, we only need to rewrite parts of the short specification. But we can continue to use the same standard software components without modifications. Our only assumption is that applications are configureable to reach the system-oriented goals. We do not assume that options are specified or standardized in any specific way.

One novelty of the configuration specification language is that it has nearly no built-in language constructs. Instead all language features are negotiated with plugins at deployment time. This paper contributes to the question of the interaction of system-oriented goals and modularity. The problem is significant because applications often have unused options that would improve the system.

In Section 2 we describe how the configuration specification language SpecElektra works. In Section 3 we discuss the perspectives, and in Section 4 we evaluate the overhead. In Section 5 we discuss related work, and finally in Section 6 we conclude.

2. SpecElektra

SpecElektra is a highly-modular configuration specification language. Its main purpose is to specify configuration data and its constraints for configuration problems [4]. It is different from other configuration specification languages because: (1) there is hardly any feature built in the language, (2) the specification is always present on the target system, and (3) it also specifies the configuration access. As other languages SpecElektra aims at ease of use.

We observe the presence of two modularity dimensions in most systems: vertical and horizontal modularity. We begin by describing our observations. Later we explain how SpecElektra improves modularity in both dimensions.

2.1 Vertical Modularity

Vertical modularity is the degree of separation between different applications. If they all use the same configuration registry, we would couple them tightly. For example, (see Figure 1) the applications `gpsd` and `batteryd` use the configuration library `L1`. A configuration library fetches the configuration at startup. Thus we cannot deploy `gpsd` or `batteryd` without `L1`. We want, however, high vertical modularity: any component should be interchangeable and the configuration framework should not interfere. If the applications and configurations coupling is low, we would have a high degree of modularity.

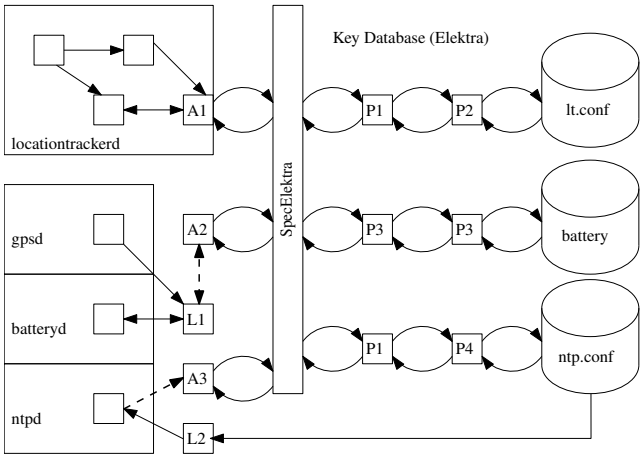

Figure 1. Horizontal and vertical modularity of locationtracker device. Boxes are applications, cylinders are configuration files, A? are adapter libraries, L? are configuration libraries, P? are plugins.

SpecElektra provides three main mechanisms to improve vertical modularity. The first mechanism is shown in Figure 1 with `ntpd`. The application `ntpd` directly accesses its configuration file `ntp.conf` using the library `L2`. Additionally, the configuration file `ntp.conf` is accessible as specified via `P1` and `P4`. The plugin `P4` directly parses and generates `ntp.conf` and `P1` acts as an filter. *Mountpoints* permit us to integrate the configuration file `ntp.conf`:

```
1 [ntp]
2 mountpoint = ntp.conf
3 infos/plugins = P1 P4
```

In the above example, the values of `ntp.conf` are mapped into the key `[ntp]` and its subkeys. A mountpoint establishes a connection between configuration data in a configuration file and the global key database. *Plugins* are pieces of code that share the same interface. The specified plugins `P1` and `P4` are responsible for mapping `ntp.conf` into the global key database. Using mountpoints the modularity is improved indirectly: we prevent that the applications directly access `ntp.conf`. Changes in keys below the key `[ntp]`, e.g. the key `[ntp/maxpoll]`, are automatically reflected in the file `ntp.conf`. Using links and transformations [9] we can even avoid dependencies on the configuration structure.

We prefer the second mechanism used in `locationtrackerd`. Here we use the *adapter library* `A1`. Adapter libraries provide the interface the application needs. They either transform or directly access the data structure from SpecElektra. `A1` is directly included and thus `locationtrackerd` is integrated in our approach without extra steps. We favor adapters that facilitate code generation because it guarantees that the configuration access code matches the specification. The advantage of this (second) mechanism is that SpecElektra always enforces all properties of the specification.

The third mechanism is shown on the example with `L1`. Suppose this library would usually read the configuration file `L1.conf`. Because `A2` intercepts the system call "`open`", `L1` will actually use SpecElektra when configuration is read. Then every call to work with `L1.conf` will actually be redirected via adapter `A2`. All applications using `L1` (i.e. `gpsd` and `batteryd`) will participate, too. Most operating systems provide ways to intercept library calls without recompilation, e.g., with `ld.preload` on Linux. Thus `gpsd`, `batteryd` and `L1` do not need any modifications.

As last example, the application `ntpd` uses the library call `getenv("NTPD_UPDATEINTERVAL")`. `A3` again is an adapter which intercepts `getenv()`-calls. With `A3` in place, `ntpd` actually requests `[getenv/NTPD_UPDATEINTERVAL]` instead. Even though

ntpd is unmodified, it participates in a unified configuration system. We conclude that SpecElektra has mechanism to achieve a high degree of vertical modularity.

2.2 Horizontal Modularity

The *horizontal* modularity is the degree of separation in configuration access code. A high degree of horizontal modularity allows us to plug together any configuration access code. This modularity is needed, because configuration access works different for every application. E.g. some applications validate configuration extensively, others use exotic configuration files. Our goal is to reuse configuration access code nevertheless. For horizontal modularity SpecElektra uses the pipes-and-filters pattern (see P1–P4 in Figure 1). The specification abstracts from the assembling of plugins:

```
1 [getenv/ntpd_updateinterval1]
2 check/type = long
3 rename/toupper = 1
```

The check/type property in lines 2 will load appropriate type checker plugins. The property in line 3, will make sure that a rename plugin will be loaded. Rename plugins make sure that getenv within ntpd works with NTPD_UPDATEINTERVAL, although the key is written lower-case in the configuration file. The argument of the property rename/ toupper ensures that the string getenv/ will not be converted. Nearly any set of plugins can be combined for configuration access. We see SpecElektra provides a good degree of horizontal modularity.

2.3 Properties and Beyond

Properties are an elegant way to specify keys. We only need to list properties, and plugins are assembled automatically to ensure these properties. For example, we can assign a new value conditionally:

```
1 [gps/status]
2 assign/condition = (battery/level > 'low') ?
3                    ('active') : ('inactive')
```

Currently, the plugin "conditionals" is the only one that provides the property assign/condition. Thus the plugin "conditionals" will be loaded during configuration access and it will assign the value according to the condition in line 2 and 3.

One might wonder, how dynamically the system reacts, e.g. turns off GPS, when the battery status changes. In SpecElektra, different notification strategies are implemented with different plugins. Thus we can make use of every mechanism to reread configuration files that applications provide. If applications do not listen to any notification, we restart them.

Some functionality, however, is not related to a single key, but to a mountpoint, or even globally. In these cases we need to manually specify plugins (as done with P1 and P4 before):

```
1 [ntp]
2 mountpoint = ntp.conf
3 infos/plugins = augeas lens=ntpd.lns syslog
```

In line 2 we declare that the key is a mountpoint. In line 3 we directly specify plugins needed for the mountpoint. We pass a configuration to the so called augeas plugin. The plugin configuration lens=ntpd.lns is the passed when the plugin is loaded at runtime. A lens is a bidirectional program [1] that describes how the syntax of the configuration file is mapped into our key database. The plugin syslog will log every configuration change.

It is sometimes useful to include the same plugin several times (P3 in Figure 1). For example, we want to encrypt the values of the configuration file with different ciphers. If the default cipher is AES and we first want DES, then AES:

```
1 [locationtracker/secret]
2 infos/plugins = crypto cipher=DES crypto
```

We see that SpecElektra also provides means for a more manual specification to load plugins. This is necessary for situations where properties are not related to individual keys. Using such mountpoint properties (or global ones not described here) are useful to fulfill cross-cutting concerns.

2.4 Contracts

A contract is a description of a plugin's functionality. It complements to the specification. It describes not only what a plugin offers but also the needs of the plugin. The plugin's contract has some overlapping but also different properties compared to the specification. The main difference is that properties of the contract only refer to the plugin itself, not to individual keys. Thus the contract does not contain keys written in [...].

The most important part of the contract lists all properties the plugin handles. E.g., a rename plugin lists rename/toupper, rename/tolower as supported properties.

Another important property of the contract is the development status. By weighting the labels we map the status to a number. For example, the contract can contain infos/status = productive tested memleak. Here the given status productive and tested increases the assigned number, but not memleak.

Plugins sometimes require other plugins to work properly. To avoid undesired direct dependencies between plugins, we use the dependency inversion principle. For example, a plugin offers in its contract infos/provides = rename and another plugin requests infos/needs = rename. Then the dependency is fulfilled, even when the developer did not consider that particular rename plugin.

The last concept needed for assembling plugins is their ordering. Its core principle is to list which plugins are not admissible to be present when the plugin is loaded. They have to be loaded earlier. For example, we would write in the contract infos/ordering = rename to reorder the plugin so that the respective plugin is inserted before rename.

2.5 Algorithm

We already mentioned that a particular property causes a specific plugin to be loaded. But often many plugins are suitable to ensure properties of keys. We now go into the details of how the algorithm assembles all plugins needed for a configuration specification.

A specialty of SpecElektra is the negotiation for every property. With the negotiation we decide which plugin is best suitable for a specific property in SpecElektra. To select the best plugin out of many, we use the status information from the contract:

```
1 Plugin findBestPlugin (prop) {
2     p = findPluginsThatOffer (prop);
3     if (p.empty()) throw NoPluginFound ();
4     sort (p, cmpBy ("infos/status"));
5     return p[0];
6 }
```

The algorithm in the line 2 finds all suitable plugins, e.g., those which offer the property rename/toupper. In the line 4 we sort the plugins by their development status as given by the contract infos/status. We use the plugin with the highest ranking in line 5. We refrained from only having a "default" concept because we want to avoid assumptions about availability of plugins. Our algorithm always determine a best plugin. We also use the same algorithm findBestPlugin (provide) for finding the best provider.

Every specification has potential to contain conflicting properties. For example, we can specify contradicting properties:

```
[getenv/NTPD_updateintervall]
rename/toupper = 1
rename/tolower = 1
```

It is unclear if `NTPD_updateintervall` should be transformed to upper or lower case. The function `hasConflict(props)` implements rules based on properties to detect such conflicts. Thus the contract of plugins does not bother about conflicts.

For assembling the plugins we solve an instance of the configuration problem [4]. We have given the contracts of plugins (domain description) and the properties in the specification (specific requirements). We want the consistent, valid and irreducible configuration of plugins. Given how we find the best plugins and detect conflicts the algorithm is straight forward:

```
1  Mountpoint assemblePlugins(keys) {
2      plugins = {}
3      for (key : keys) {
4          if (hasConflict (allProperties (key)))
5              throw PropertyConflict ();
6          for (prop : allProperties (key)) {
7              p = findBestPlugin (prop);
8              plugins.addIfMissing (prop, p);
9          }
10     }
11     sort (plugins, cmpBy ("infos/ordering"));
12     return Mountpoint (plugins);
13 }
```

The algorithm iterates over all properties it finds in the specification (line 6). For every specification entry, we find the best suitable plugin. In line 7 we add it if the property is not handled already. Because the framework makes plugins idempotent we accept several plugins that handle the same property. Line 11 makes sure that the plugins have no ordering violation for the pipes-and-filters.

We see that SpecElektra heavily relies on the concept of orchestrating plugins by properties. Our strategy, especially the absence of conflicts in contracts, leads to a simplification of the configuration problem. With a constant number of plugins, we even find a solution for all properties in a specification in linear time $O(prop)$.

2.6 Generic Plugins

We say a plugin is *generic*, if its feature set is beyond what can be described in a single contract. We identified two different types of generic plugins. The first type of generic plugins accepts programs, behavioral descriptions, or scripts as plugin configuration. They allow solutions for different issues without a dedicated plugin for each property. If the plugin's configuration changes, the contract can differ, too. For example, the generic plugin `lua` has the plugin configuration `script` that points to a file containing a lua-script:

```
[locationtracker]
infos/plugins = lua script=batterytotracker.lua
```

Depending on the script, the behavior of the plugin can be completely different. We have to be very careful when we think about equality of plugins. Thus, SpecElektra requires an instance name to distinguish generic plugin where the contracts differ.

The other type of generic plugins, uses compile-time conditionals. Then the variability is already resolved in the target code. They are useful for three purposes: (1) If you prefer performance to flexibility, (2) during bootstrapping when we lack plugin configuration [9], and (3) if we want to have different plugins with a very high degree of code reuse. Every combination of differently defined macros is a new compilation variant, with possibly distinct contracts. For example, with a compilation variant we decide which cryptography library should be used in the plugin `crypto`.

In conclusion, we have different types of generic plugins. One type resolves variability early during compilation, the other at runtime. This feature is contrasting to COP, because in COP we cannot decide when variability is resolved.

3. Perspectives

In this section we will discuss the perspectives of SpecElektra considering its modularity and system-wide goals.

Unified configuration access is the system-oriented goal to have a single API to modify the configuration of the whole system. SpecElektra directly provides a specification for a unified configuration access. This feature presents advantages, e.g., if a user interface (UI) for system-wide configuration access is required. Suppose we have the specification:

```
[battery/level]
check/type = long
check/type/min = 1
check/type/max = 100
description = The battery level in %
```

We immediately have the necessary information to generate UIs that present the configuration nicely to users. In a project we implemented a UI that was able to display the configuration of all system's components. Except for some issues with layouts, one can even generate such UIs. Because of SpecElektra's abstraction the UI does not need any knowledge of concrete underlying configuration storages.

Reusing standard components is a common goal for nearly every modular system. Because SpecElektra provides vertical modularity, applications do not need to be coupled to participate in the system. A lightweight and useful solution is the integration of configuration files using existing plugins:

```
[ntp]
mountpoint = ntp.conf
infos/plugins = augeas lens=ntpd.lns
```

With more than 50 plugins SpecElektra supports more than 150 different types of configuration files, e.g., JSON and XML. Every application that directly modifies configuration files risks the system integrity. Thus only using mountpoints does not provide full control over configuration access.

A better form of integration of standard components is the reimplementation of the application's configuration API. We already implemented one such adapter for the `getenv()` call. We achieved booting a Debian system where every `getenv()` call gets intercepted and redirected to SpecElektra. For other applications, SpecElektra provides adapters utilizing code generation.

We think that adapters are possible for any data structure and API. We conclude that SpecElektra provides many practicable ways to integrate unmodified applications.

For intra-application integration the unified configuration access and integration of standard components is precondition. To go further, we want the configuration of one standard component to influence another standard component. We have developed several plugins to achieve that goal. The core idea is that some keys are not read from configuration files, but calculated dynamically. For example, we want to derive the polling time:

```
[locationtracker/polling]
description = GPS polling time in seconds
transform/batterytotracker = battery/level
```

The property `transform/batterytotracker` makes sure that the value `[locationtracker/polling]` is transformed to a value suitable for `[battery/level]`. With lower battery, the poll time slows down. Sometimes we prefer to use conditionals:

```
[gps/status]
assign/condition = (battery/level > 'low') ?
                   ('active') : ('inactive')
```

In this example, we will completely turn off GPS after we passed a threshold. We see that these techniques help us to achieve the goal to save battery, without the need to modify the applications. In [9] we describe improvements of intra-application integration.

Cross-cutting concerns are often related to configuration access. We want to demonstrate how SpecElektra handles a typical cross-cutting concern: logging. For example, we want that our tracker application logs every configuration change. Everything should be logged twice, once remote with syslog and once locally. Because we already call `openlog()` in our tracker-application, we do not want the plugin to reopen the log. In SpecElektra we write:

```
[locationtracker]
mountpoint = lt.conf
infos/needs = syslog dontopensyslog= journald
```

With the specification above the tracker-application logs every change to both syslog and journald. Note, that the property `dontopensyslog` needs an = so that we know it is not a plugin, but plugin configuration. We can easily add any code to handle cross-cutting concerns. The only prerequisite to use some code as plugin is to provide an entry point.

Plugin reuse: Sometimes cross-cutting concerns are complex. Then we want to implement them using several plugins in appropriate languages. Plugins can be written in any programming language with bindings for SpecElektra. Currently, most plugins are written in C and C++; others are implemented in Java, Lua and Python.

The logging example above with two logging plugins already gives you an intuition of plugin reuse. Now we want to explore one more situation where plugins even require the functionality of another one. For example, suppose all shared files should be encrypted. We need to compress the file to satisfy performance criteria. A configuration file happens to be in the shared folder:

```
[locationtracker/shared]
mountpoint = /shared_folder/lt.conf
infos/plugins = compress crypto
```

Compression and encryption algorithms are both non-trivial and not always used together. Instead of writing one plugin that implements both features we can make use of the pipes-and-filter pattern. Then the `crypto` plugin uses functionality of the `compress` plugin. We also can easily connect standard plugins along with plugins developed specifically for an application.

Fulfill changing system-goals: We already showed in previous examples how we can facilitate configuration to improve battery life. Now suppose the system-goal has shifted because we deploy a location tracker in a vehicle. The battery is much larger, but now the tracker resolution at high speed is a problem. Additionally, we want the same embedded system to continue its work on smart phones and wearable computing devices. We introduce a key that contains the device information to cope with the different goals:

```
[device]
check/enum = 'wearable','smartphone','vehicle'
```

The basic idea is to introduce high-level configuration items representing goals. From them, we calculate what it actually means to fulfill the system-oriented goal:

```
[powersaving/gps]
assign/condition = (device != vehicle) ?
                   (battery/level) : ('full')
[gps/resolution]
assign/condition = (device == 'vehicle') ?
                   ('high') : ('low')
```

To change a goal we only need a single modification of one such high-level configuration value. Embedded devices often even have this knowledge detectable, e.g., written in EEPROM. Then we can deploy identical images fulfilling diverse system-oriented goals. Due to the possibility to change the specification during system-integration we do not need to know all possible devices beforehand.

4. Evaluation

We benchmarked SpecElektra on a hp® EliteBook 8570w using the central processor unit Intel® Core™ i7-3740QM @ 2.70GHz. The operating system was Debian GNU/Linux Wheezy 8.2 with the architecture amd64. We used the compiler gcc 4.9.2 with the option `-O2`.

Vertical modularity: In this benchmark we want to evaluate the overhead of vertical modularity. In particular, we benchmark the cost of an increasing number of mountpoints, e.g., 2 mountpoints:

```
[benchmark/0]
mountpoint = /tmp/file0
[benchmark/1]
mountpoint = /tmp/file1
```

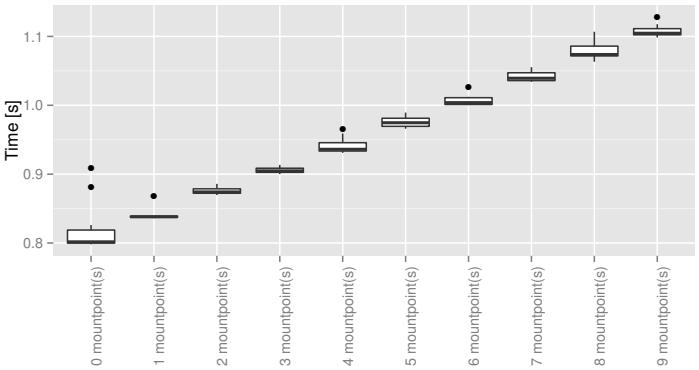

Figure 2. Access time with 1000 keys with 100000 iterations and an increasing number of mountpoints. Note that the high number of iterations means that the relative time is very low.

In Figure 2 we see a linear correlation between time and number of mountpoints. This is not surprising because reading and writing to a higher number of files is expected to need more time. Zero mountpoints means that everything is written into the root mountpoint. We conclude that the overhead for vertical modularity is acceptable, even with frequent configuration changes.

Horizontal modularity: In this benchmark we want to evaluate the overhead of horizontal modularity. We consider the read and write performance. We use the `iterate` plugin that iterates over all keys and search for the property `iterate` in every key. We increase the number of mounted plugins in one mountpoint:

```
[benchmark]
mountpoint = /tmp/file
infos/needs = iterate#0 iterate#1 ...
```

In a nutshell no overhead could be measured. The parsing of the configuration file is by far the dominant factor. This statement does not change with an increasing number of keys or properties. We conclude that SpecElektra imposes no reason to avoid modularity. If useful, every feature should be implemented as a separate plugin.

5. Related Work

Software product lines are a holistic alternative, even able to quickly adapt to new goals [7]. They are, however, not able to integrate standard components as easily. Instead they require to rewrite software if it was not based on produce lines before.

Chiba et al. [2] argues that we sometimes do not need to extend syntax for improved modularity. We agree and similar to their concerns-browser, SpecElektra can be seen as hierarchical data.

Lenses [1] ease the support of many different configuration files. With augeas SpecElektra has a plugin that facilitates lenses. Unfortunately, augeas does not provide the level of configuration abstraction as often needed. It also has limitations supporting hierarchical nested data. E.g., XML or JSON are syntactical alternatives to the INI format we used in this paper. If preferred, SpecElektra allows us to use XML or JSON for its specification.

Configuration management (CM) aims at specifying configuration centrally for all nodes [3]. Bypassing CM to directly change configuration files is considered as anti-pattern. For smartphones and other personal computers it is, however, often desirable to directly reconfigure the devices. SpecElektra supports such local configuration changes while still enforcing the specification.

In context-oriented programming [6, 8] we typically need to decide in the implementation what the context is. Thus we think it is sometimes less flexible when the system-oriented goal changes.

Other configuration specification languages [4, 5] have sophisticated support for validation. To the best of our knowledge, however, no specification languages consists entirely of plugins. Thus in other configuration specification languages the user is bound to the features the configuration specification language has and cannot extend it easily. Additionally, other configuration specification languages are typically not deployed on the systems. Thus they cannot be modified effortlessly to change system-oriented goals.

6. Conclusion

Configuration specification languages are a sweet spot: In SpecElektra modularity helps system developers to reach system-oriented goals in many ways. We simply specify properties and the tool automatically assembles plugins that enforce the properties. Modularization in the configuration access helps applications to include application-specific validation. Then we achieve an improvement of the validation of the whole system. SpecElektra also allows us to externally handle many different cross-cutting concerns related to configuration for the whole system.

By benchmarks we showed that the overhead of configuration integration is acceptable. Even better, the impact of increasing the number of plugins is not measurable.

The benefits of the approach are:
(1) allows validation and documentation of configurations,
(2) avoids lowest common denominator by specific plugins,
(3) enables a unified way to access configuration,
(4) has plugins with compile-time and run-time variability, and.
(5) allows us to introduce high-level configuration options that influence system-oriented goals.

More importantly, a modular configuration specification language allows us to integrate different unmodified applications. Then we can achieve system-wide goals by changing high-level configuration items. Using plugins we easily can propagate the high-level items to concrete configuration. Because all applications are specified at one location, we can easily refer to any option.

SpecElektra is used in practice, and the latest release 0.8.15 can be downloaded freely from `http://www.libelektra.org`. The specifications are semi-structured in any syntax supported by a plugin. They are directly available on the target system and enable unmodified applications to be integrated.

Our contributions are:

- observations and improvements for vertical and horizontal modularity for configurable components,
- an implementation and benchmarks for a fully-modular configuration specification language, and
- examples for interactions between modularity and system-oriented goals.

These contributions are significant. Up to now such integration endeavors were less modular. Thus they could not cope with requirement changes as easily.

As further work we plan to better integrate context-awareness and to conduct larger case studies.

Acknowledgments

I would like to thank Franz Puntigam, Andreas Falkner, Harald Geyer, and the anonymous reviewers for a detailed review of this paper. Additionally, many thanks to all people contributing to Elektra.

References

[1] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. 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. ISBN 978-1-59593-689-9. doi: http://doi.acm.org/10.1145/1328438.1328487.

[2] S. Chiba, M. Horie, K. Kanazawa, F. Takeyama, and Y. Teramoto. Do we really need to extend syntax for advanced modularity? In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 95–106. ACM, 2012.

[3] T. Delaet, W. Joosen, and B. Vanbrabant. A survey of system configuration tools. LISA'10, pages 1–8. USENIX, 2010. URL `http://dl.acm.org/citation.cfm?id=1924976.1924977`.

[4] G. Friedrich and M. Stumptner. *Consistency-based configuration*. 1999.

[5] S. Günther, T. Cleenewerck, and V. Jonckers. Software variability: the design space of configuration languages. In *Proceedings of the 6th Workshop on Variability Modeling of Software-Intensive Systems*, pages 157–164. ACM, 2012.

[6] T. Kamina, T. Aotani, H. Masuhara, and T. Tamai. Context-oriented software engineering: A modularity vision. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 85–98, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2772-5. doi: 10.1145/2577080.2579816. URL `http://doi.acm.org/10.1145/2577080.2579816`.

[7] A. Nascimento, C. Rubira, and F. Castor. Arcmape: A software product line infrastructure to support fault-tolerant composite services. In *High-Assurance Systems Engineering, 2014 IEEE 15th International Symposium on*, pages 41–48, Jan 2014. doi: 10.1109/HASE.2014.15.

[8] M. 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. doi: 10.1109/ISORCW.2015.52.

[9] M. Raab. Sharing software configuration via specified links and transformation rules. In *Technical Report from KPS 2015*, volume 18. Vienna University of Technology, Complang Group, 2015.