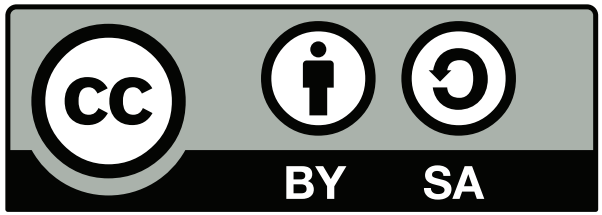
Configuration Management

Markus Raab

Institute of Information Systems Engineering, TU Wien

23.3.2018



Organization

Today:

23.3.2018: **teams found together, dates of talks**

Next dates:

13.4.2018: **homework submitted, topics of team exercise**

27.4.2018: lecture

4.5.2018: lecture

18.5.2018: guest lecture

25.5.2018: team exercise submitted

1.6.2018: lecture

8.6.2018: lecture

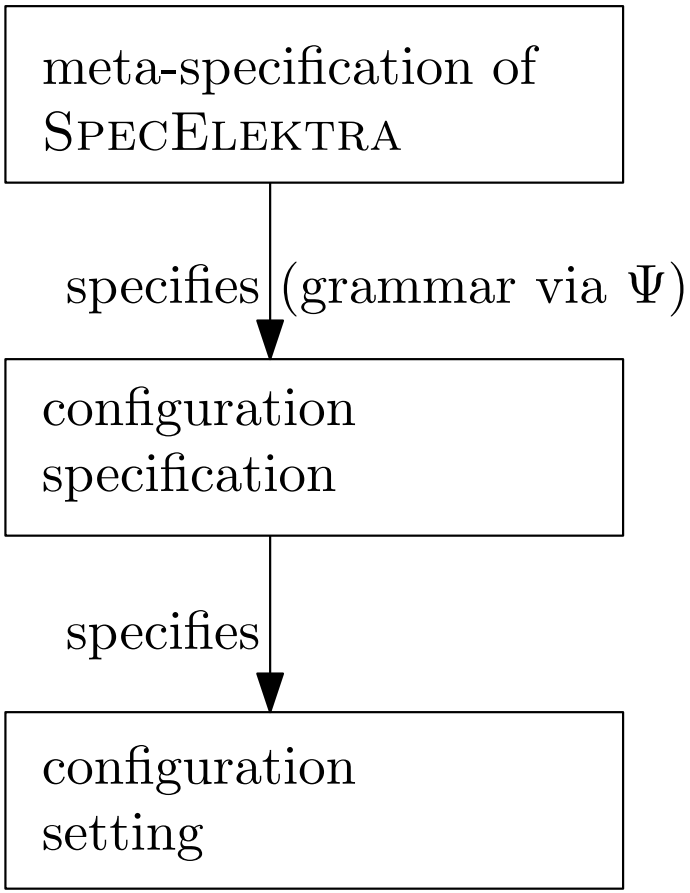
15.6.2018: **last corrections of team exercise**

22.6.2018: **test**

Recapitulation

- alarming trend in number and complexity of configuration settings
- sharing, visibility and default value calculation often helps
- needs abstraction: configuration specification
- but also more courageous decisions and periodical reevaluation
- different ways to reduce configuration space

Recapitulation (Metalevels)



Recapitulation (Requirements of SpecElektra)

- formal and informal
- should strive for completeness
- should be extensible
- should be external to application
- open for introspection (for tooling)
- should talk to users
- should allow generation of artefacts

SpecElektra

SpecElektra is a modular ***configuration specification language*** for configuration settings. In SpecElektra we use properties to specify configuration settings and configuration access. SpecElektra enables us to specify different parts of Elektra.

Popular Topics

- 4 validation
- 4 user interface
- 3 tools (benefits?)
- 3 testability
- 3 complexity reduction (when conf. needed?)
- 3 architectural decisions
- 2 Puppet
- 2 modularity
- 2 environment variables
- 2 documentation
- 2 configuration specification
- 2 command-line args
- 2 code generation
- 1 variability
- 1 self-description
- 1 round-tripping
- 1 introspection
- 1 dependences
- 1 auto-detection
- 1 early
- 1 context-awareness
- 1 administrators

Goals for today

- modularity on system level
  - horizontal
  - vertical
- system-wide introspection
- avoiding dependences
- auto-detection

Modularity

Configuration Access APIs

An ***application programming interface (API)*** defines boundaries on source code level. Better APIs make the execution environment easier and more uniformly accessible.

Configuration access is the part of every software system concerned with fetching and storing configuration settings from and to the execution environment. There are many ways to access configuration [1, 2, 4]. ***Configuration access APIs*** are APIs that enable configuration access.

Configuration Access APIs

Task

Which configuration access APIs do you know?
What are the differences between these APIs?

For example:

- `char * getenv (const char * key)`
- `ConfigStatus xf86HandleConfigFile(Bool autoconfig)`
- `long pathconf (const char *path, int name)`
- `long sysconf (int name)`
- `size_t confstr (int name, char *buf, size_t len)`

Configuration Access Points

Within the source code the ***configuration access points*** are configuration access API invocations that return configuration values.

```
1 int  main()
2 {
3       getenv  ("PATH");
4 }
```

Configuration Libraries

Configuration libraries provide implementations for a configuration access API.
Trends:

- flexibility to configure configuration access (e.g., `https://commons.apache.org/proper/commons-configuration/`)
- more type safety (e.g., `http://owner.aeonbits.org/`, code generation in next lecture)
- try to unify something (UCI, Augeas, Elektra)

Types of Modularity

Vertical modularity describes how strongly separated the configuration accesses of different applications is. *Horizontal modularity* describes how strongly separated modules implementing configuration access for a single application is.

Vertical Modularity [3]

Vertical modularity is the degree of separation between different applications. If all applications use the same key database with a single backend or a single configuration file, applications would be coupled tightly. [...]
If coupling between applications is low, for example every application uses a different configuration library or a different backend, we have a high degree of vertical modularity.

Retain Vertical Modularity [3]

Elektra provides two mechanisms to retain vertical modularity:

- **Mounting** configuration files facilitates different applications to use their own backend and their own configuration file. Furthermore, mounting enables integrating existing configuration files into the key database. Configuration specifications written in SpecElektra allow different applications to share their configuration files with each other in a controlled way.
- Having frontends that implement existing **APIs** decouple applications from each other. These applications continue to use their specific configuration accesses, but Elektra redirects their configuration accesses to the shared key database.

Vertical Modularity [3]

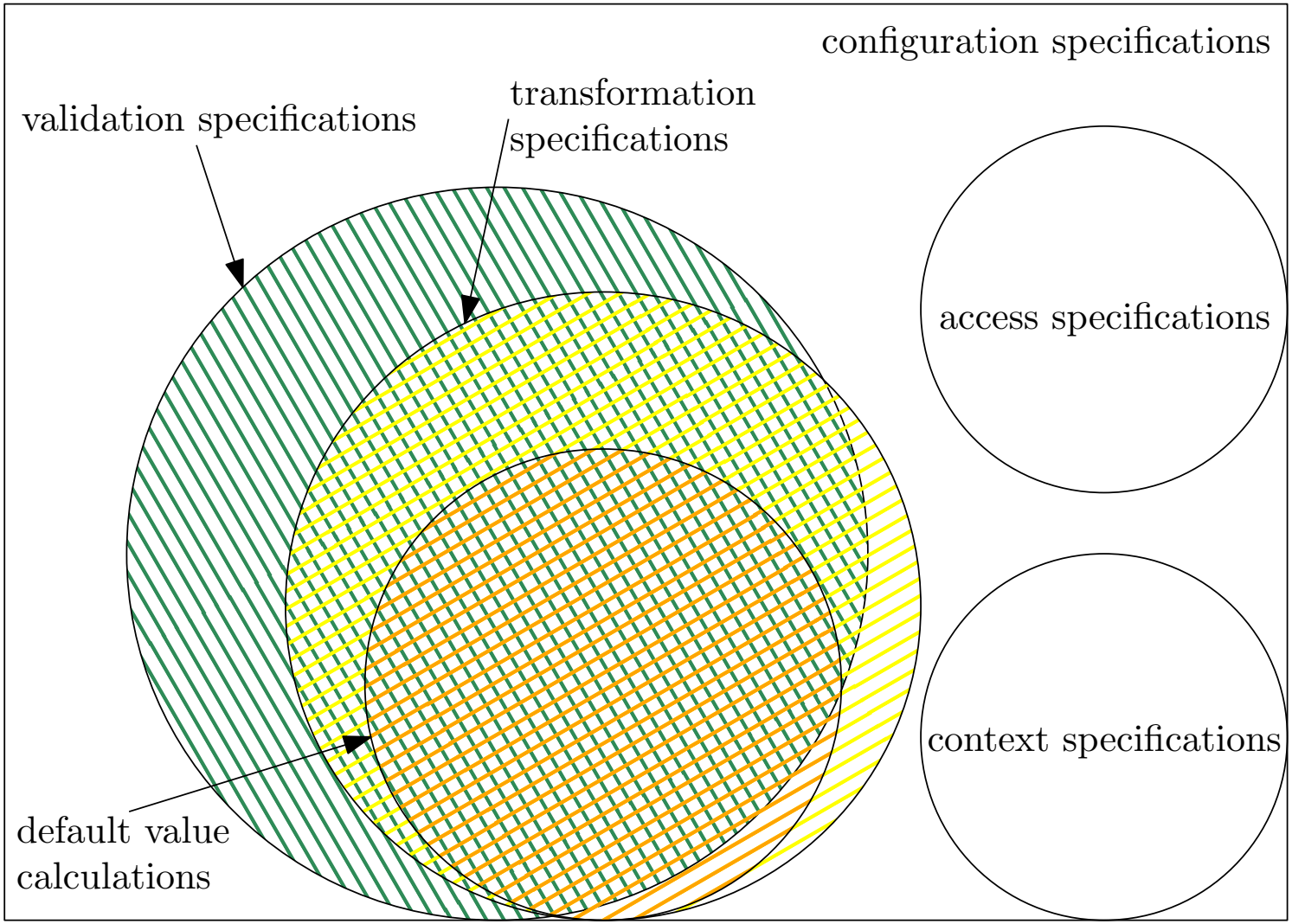
Mountpoints can also be a part of the specification:

```
1 [ntp]
2   mountpoint:=ntp.conf
3 [sw/libreoffice]
4   mountpoint:=libreoffice.conf
```

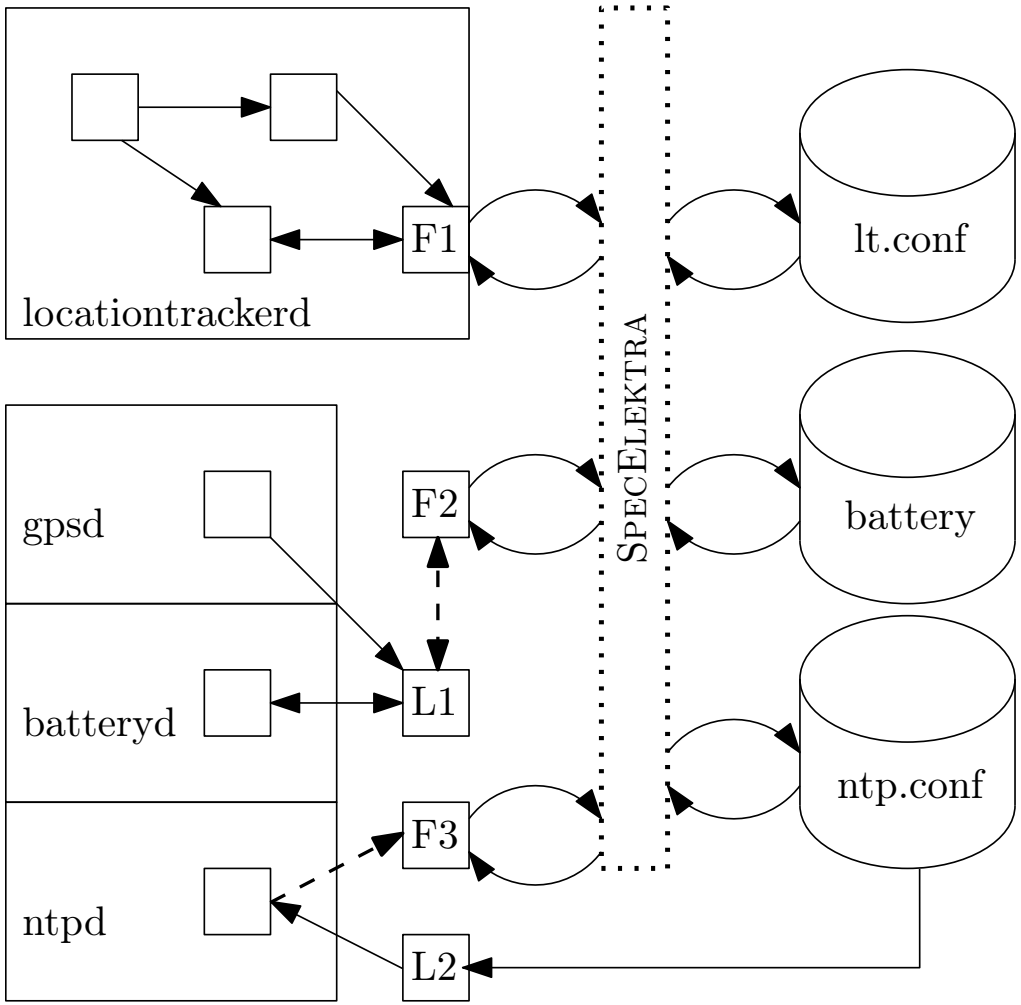
Task

Which type of specification is this?

Types of Specifications



Vertical Modularity



Needed to keep applications independently.
Boxes are applications, cylinders are configuration files, F? are frontends or frontend adapters, L? are configuration libraries [3].

Horizontal Modularity [3]

Horizontal modularity is "the degree of separation in configuration access code" [3]. A higher degree of horizontal modularity allows us to better separate configuration access code and plug the code together as needed.

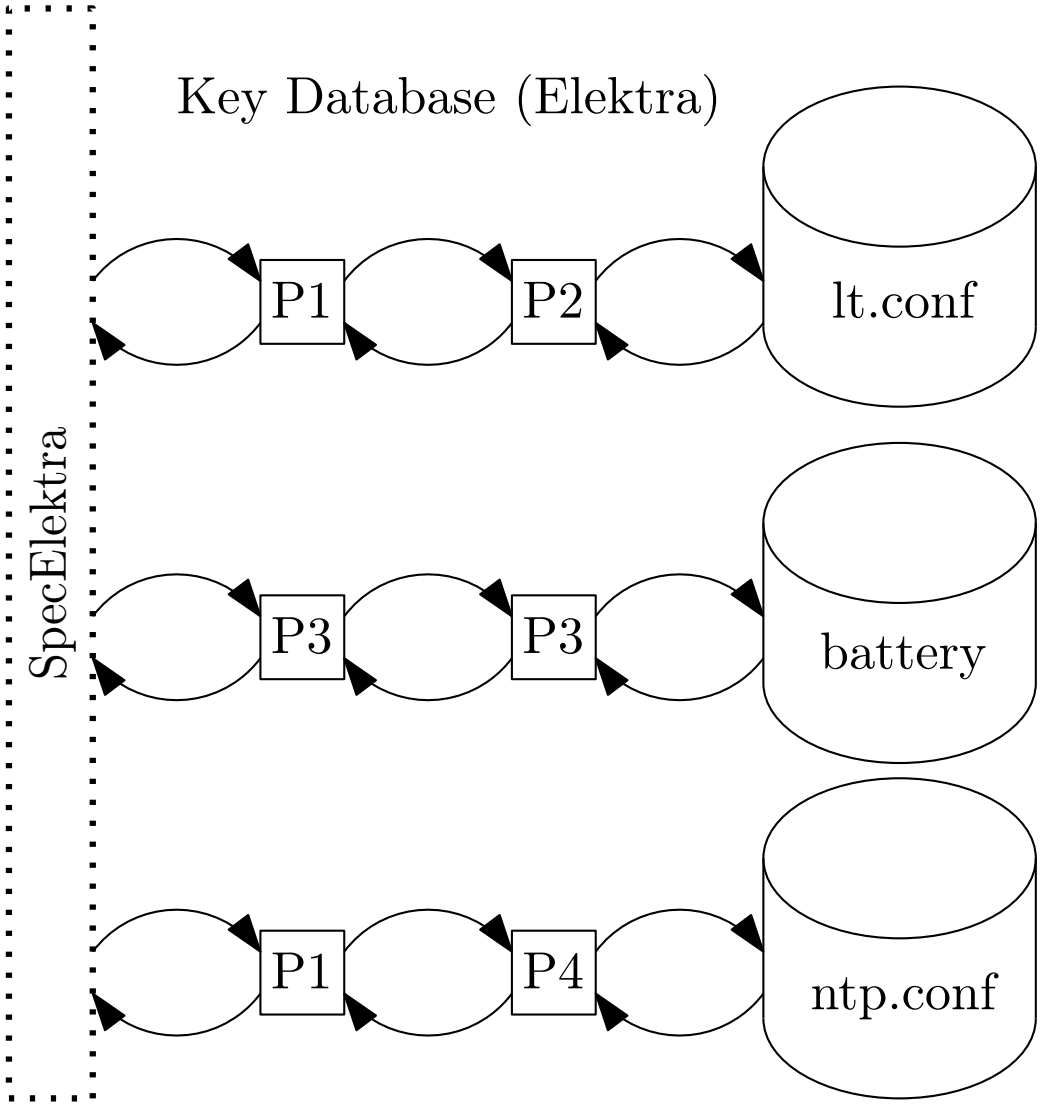
Three factors of SpecElektra improve horizontal modularity:

1. Using SpecElektra, applications are completely decoupled from configuration specifications.
2. Specifications and their implementation are decoupled.
3. Abstract dependences within the implementation of specifications.

Task

This is very vague.
Can you describe a system that would (not) fulfil this?

Horizontal Modularity



Needed mainly for validation.

Cylinders are configuration files, P? are plugins [3]

Plugins

Acceptable Effort

Q: "Which effort do you think is worthwhile for providing better configuration experience?"

- 44 % would use other configuration access APIs next to `getenv`.
- 30 % would use OS-specific sources.
- 21 % would use dedicated libraries.
- 19 % would read other application's configuration settings,
- 16 % would use external configuration access APIs that add new dependences.

Why?

Finding

Q: Most developers have concerns adding dependences for more validation (84 %) but consider good defaults important (80 %).

Requirement

Dependences exclusively needed to validate configuration settings must be avoided.

Rationale

Why is it difficult to have good defaults?

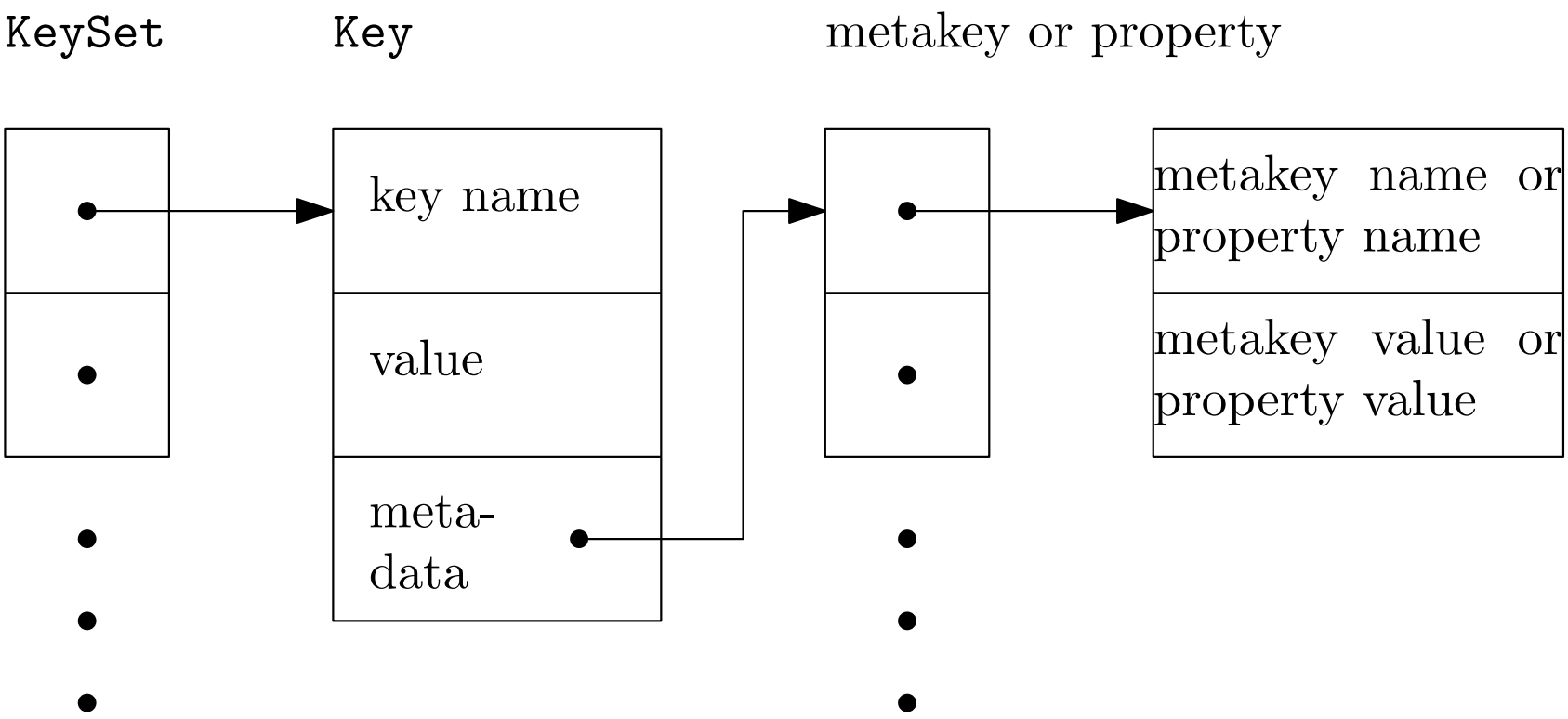
- **Modularity:** diverse and conflicting requirements between applications. Especially in validation, for example, constraint solvers vs. type systems vs. model checkers.
- **System-level:** specification must always be enforced. Examples:
  - which desktop is the application started in?
  - how many CPUs does the system have?
  - get the correct proxy of the system.
  - get available network bandwidth.
  - is the filesystem local?

Plugins are filters, sinks, and sources processing a key set. We aim at SpecElektra to be as modular as possible and make extensive use of plugins:

1. SpecElektra does not have any built-in feature, all features are (or can be) implemented as plugins.
2. Elektra works completely without SpecElektra's specifications.
3. Configuration specifications are present within the execution environment. Thus any tool and plugin can introspect and use the specifications.

KeySet

The common data structure between plugins:



Plugin Assembly

automatic assembling of plugins:

- iterate over the specification and collect all key words
- iterate over all plugins and check if they offer key words
- check contract between plugins and specification
- of the remaining plugins: use best suited or rated

(implemented in `kdb mount` / `kdb spec-mount` in Elektra)

SpecElektra is a dependency injection mechanism:

- By extending the specification, new plugins are being injected into the system.
- The *provider* abstractions in the dependences between the plugins abstract over concrete implementations of configuration access code.
- We have a modular implementation of SpecElektra.

Task

Which kind of modularity does *provider* improve?

Answer

3rd point of horizontal modularity on Slide 21

Examples

calculation with context:

```
1 [gps/status]
2 assign:=(battery > 'low') ? ('on') : ('off')
3 [battery]
4 plugins:=battery
```

other example: how to resolve names of configuration files on an operating system

Elektra

Elektra as Virtual Filesystem

- configuration files are seen like “block devices”
- are mounted with respective filesystem drivers into the filesystem
- many tools and APIs evolved to work with files

Why is Elektra not a Filesystem then?

- API semantics: key/value get/set
- namespaces: based on established semantics
- many features essential for misconfiguration hardening:
  - validation
  - visibility
  - defaults
  - ... (extensible specification)

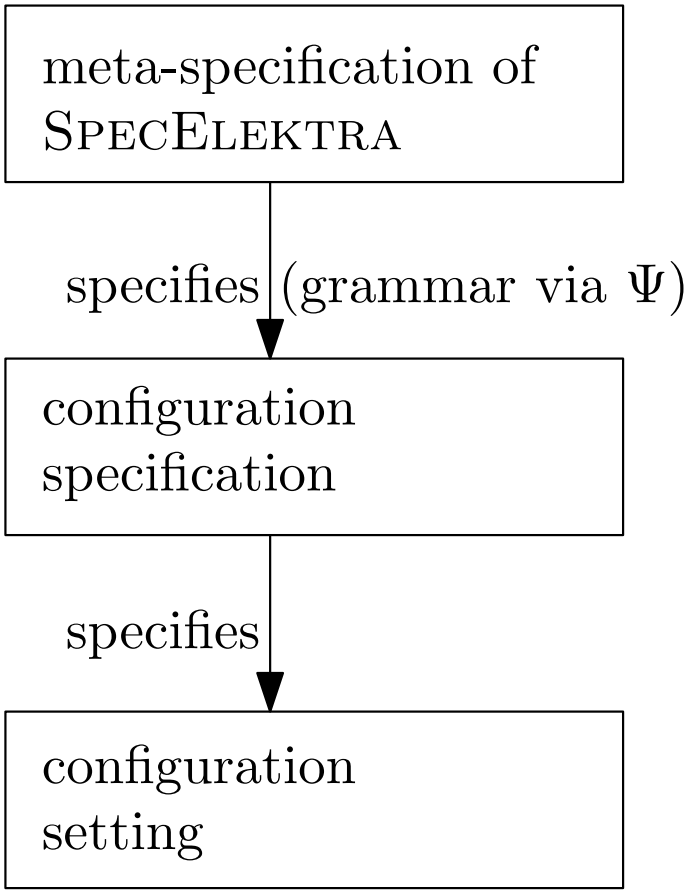
kdb.open(): The first step is to bootstrap into a situation where the necessary plugins can be loaded.

kdb.get(KeySet): The application (initially) fetches and (later) updates its configuration settings as a key set of type `KeySet` from the execution environment by one or many calls to `kdb.get`. If all relevant configuration files are unmodified since the last invocation, `kdb.get` will do nothing.

kdb.set(KeySet): When a user finishes editing configuration settings, `kdb.set` is in charge of writing all changes back to the key database. This function atomically persists a whole key set in involved parts of the execution environment. In the case of an error no action takes place.

kdb.close(): The last step is to close the connection to the key database.

Recapitulation (Metalevels)



We will now walk through metalevels bottom-up.

Configuration Settings

A configuration file may look like:

```
1       a=5
2       b=10
3       c=15
```

We apply these configuration settings imperatively using:

```
1       kdb set /a 5
2       kdb set /b 10
3       kdb set /c 15
```

And we list them with `kdb ls /`.

Specifications

For specifications such as:

```
1     [slapd/threads/listener]
2       check/range :=1,2,4,8,16
3       default :=1
```

We apply the specifications imperatively using:

```
1     kdb setmeta /slapd/threads/listener\
2         check/range 1,2,4,8,16
3     kdb setmeta /slapd/threads/listener\
4             default 1
```

(automatically uses spec namespace)

Meta-Specifications

For meta-specifications such as:

```
1     [visibility]
2     type:=enum critical important user\
3           advanced developer debug disabled
4     description:=Who should see this\
5          configuration setting?
```

We apply the meta-specifications imperatively using:

```
1     kdb setmeta /elektra/meta/\
2         visibility type enum ...
3     kdb setmeta /elektra/meta/\
4         visibility description "Who ...
```

(see doc/METADATA.ini, disclaimer: 1.0 not yet released)

Introspection

- unified get/set access to (meta*)-key/values
- access via applications, CLI, GUI, web-UI, ...
- access via any programming language (similar to file systems)
- GUI, web-UI can semantically interpret metadata

Conclusion

- definitions: APIs, modularity
- goal:
  - good defaults
  - system-wide introspection
- how?
  - We use a system-level dependency injection.
  - We get reusable plugins operating on key/value pairs.
  - We get operating system abstractions of context.
  - We avoid to have dependences in the applications.
- Elektra has no dependence to other libraries but only concrete plugins introduce dependences.

Preview

next lecture after eastern:

- code generation vs. introspection
- testability

[1] Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 215–224, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2768-8. doi: 10.1145/2591062.2591191. URL `http://dx.doi.org/10.1145/2591062.2591191`.

[2] Emre Kiciman and Yi-Min Wang. Discovering correctness constraints for self-management of system configuration. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 28–35. IEEE, May 2004. doi: 10.1109/ICAC.2004.1301344.

[3] Markus Raab. Improving system integration using a modular configuration specification language. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 152–157, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4033-5. doi: 10.1145/2892664.2892691. URL http://dx.doi.org/10.1145/2892664.2892691.

[4] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 244–259. ACM, 2013.