L06 Strategies for Validation and Modularization

Markus Raab

Institute of Information Systems Engineering, TU Wien

Validation

Learning Outcomes

Students will be able to

- write simple checker plugins.

Checking Configurations

Following properties of configuration settings can be checked:

- values (data types)

Checking Configurations

Following properties of configuration settings can be checked:

- values (data types)
- structure

Checking Configurations

Following properties of configuration settings can be checked:

- values (data types)
- structure
- constraints

Checking Configurations

Following properties of configuration settings can be checked:

- values (data types)
- structure
- constraints
- semantic checks (e.g., IP, folder)

Checking Configurations

Following properties of configuration settings can be checked:

- values (data types)
- structure
- constraints
- semantic checks (e.g., IP, folder)
- domain-specific checks (e.g., databases)

Checking Configurations

Following properties of configuration settings can be checked:

- values (data types)
- structure
- constraints
- semantic checks (e.g., IP, folder)
- domain-specific checks (e.g., databases)
- requirements (suitable configurations)

Checking Configurations

Following properties of configuration settings can be checked:

- values (data types)
- structure
- constraints
- semantic checks (e.g., IP, folder)
- domain-specific checks (e.g., databases)
- requirements (suitable configurations)
- context (context-aware configurations)

ELEKTRA supports many other data types, each implemented in its own plugin(s):

- **check/type** allows us to specify CORBA data types. Checking “any” (default) is always successful. The record and enum types defined by CORBA are not part of this plugin but of others as explained below.
- **check/enum** supports a list of supported values denoted by array indexes.
- **check/ipaddr** checks if a string is a valid IP address.
- **check/path** checks presence, permissions, and type of paths in the file system.
- **check/date** supports to check date formats such as POSIX, ISO8601, and RFC2822.
- **check/range** allows us to check if numerical values are within a range.
- **check/validation** checks the configuration value with regular expressions.
- **check/condition** checks using conditionals and comparisons.
- **check/math** checks using mathematical expressions.
- **trigger/error** allows us to express unconditional failures.
- **check/rgbcolor** allows us to check for RGB colors.
- **check/macaddr** allow us to check for MAC addresses.

Logfile Example

```
[slapd/logfile]
  check/path:=file
```

Logfile Extensions

```
[slapd/logfile]
  check/path:=file
  check/validation:=^/var/log/
  check/validation/message:=Policy violation:
    log files must be below /var/log
```

Checking Specifications

The goals of checking SpecElektra are:

- Defaults must be present for safe lookups. This goal also implies that there must be at least one valid configuration setting.
- Types of default values must be compatible with the types of the keys.
- Every contextual interpretation of a key must yield a compatible type.
- Links must not refer to each other in cycles.
- Every link and the pointee must have compatible types.

Example

```
[sw/org/abc/has_true_arg]
  type:=boolean
  default:=0
  override/#0:=/sw/org/abc/arg0
  override/#1:=/sw/org/abc/arg1
```

Error Messages

Error messages are extremely important as they are the main communication channel to system administrators.

Example specification:

```
[a]
  check/type:=long
[b]
  check/type:=long
[c]
  check/range:=0-10
  assign/math:=../a+../b
```

Error Messages

Problems:

- Generic vs. specific plugins

Error Messages

Problems:

- Generic vs. specific plugins
- General principles of good error messages [1]

Error Messages

Problems:

- Generic vs. specific plugins
- General principles of good error messages [1]
- Give context

Error Messages

Problems:

- Generic vs. specific plugins
- General principles of good error messages [1]
- Give context
- Precisely locate the cause:

Error Messages

Problems:

- Generic vs. specific plugins
- General principles of good error messages [1]
- Give context
- Precisely locate the cause:

```
a=5   ; unmodified
b=10  ; modification bit in metadata
      ; is only set here
c=15  ; unmodified by user but changed
      ; later by assign/math
```

Example Error Message

```
Sorry, I was unable to change the configuration settings!
I tried to modify b to be 10 but this caused c to be
outside of the allowed range (0-10).
```

With additional verbose/debug output:

```
Module: range
At: sourcefile.c:1234
Mountpoint: /test
Configfile: /etc/testfile.conf
```

L06 Strategies for Validation and Modularization

Markus Raab

Institute of Information Systems Engineering, TU Wien

Modularity

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.

Status Quo in Free Systems

- nearly all applications use their own configuration system

Status Quo in Free Systems

- nearly all applications use their own configuration system
- immense differences in configuration file formats and configuration access

Status Quo in Free Systems

- nearly all applications use their own configuration system
- immense differences in configuration file formats and configuration access
- very high modularity

Status Quo in Frameworks and Proprietary Systems

- obvious ways how to deal with configuration

Status Quo in Frameworks and Proprietary Systems

- obvious ways how to deal with configuration
- no differences in configuration access

Status Quo in Frameworks and Proprietary Systems

- obvious ways how to deal with configuration
- no differences in configuration access
- very low modularity

Vertical Modularity [2]

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 [2]

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 [2]

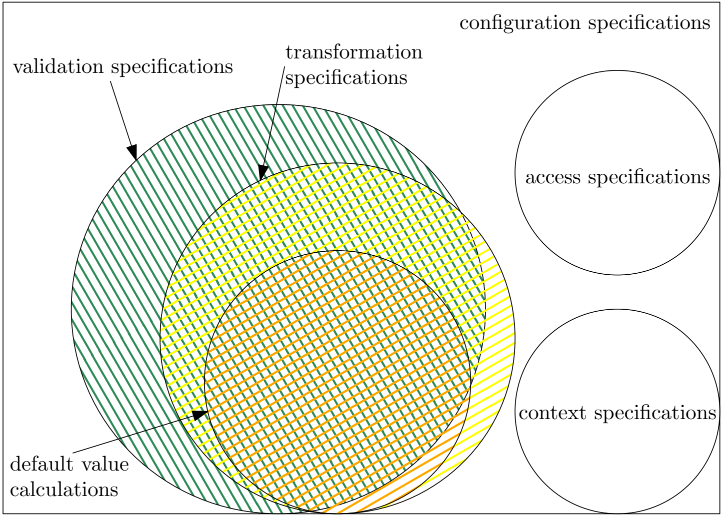
Mountpoints can also be a part of the specification:

```
[ntp]
  mountpoint:=ntp.conf
[sw/libreoffice]
  mountpoint:=libreoffice.conf
```

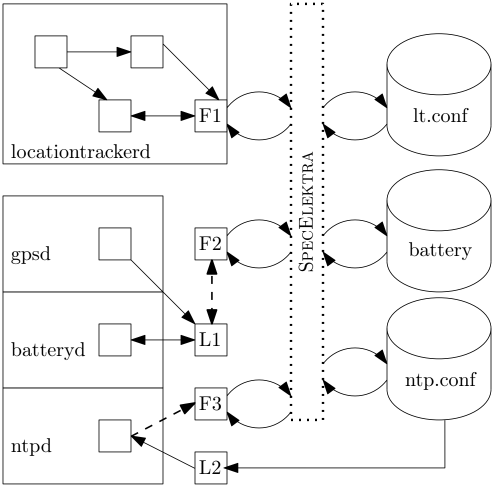
Task

Which type of specification is this?

Types of Specifications



Vertical Modularity

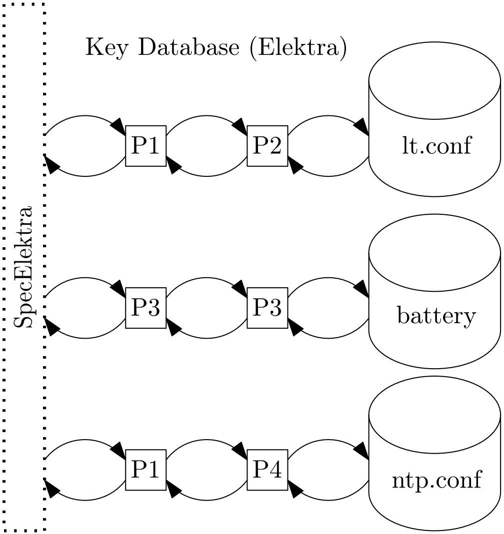


Needed to keep applications independently. Boxes are applications, cylinders are configuration files, F? are frontends or frontend adapters, L? are configuration libraries [2].

Horizontal Modularity [2]

Horizontal modularity is “the degree of separation in configuration access code” [2]. A higher degree of horizontal modularity allows us to better separate configuration access code and plug the code together as needed.

Horizontal Modularity



Needed for validation, auto-detection, . . .

Cylinders are configuration files, P? are plugins [2]

L06 Strategies for Validation and Modularization

Markus Raab

Institute of Information Systems Engineering, TU Wien

Plugins

Why?

Finding

Q: Most developers have concerns adding dependences for more validation (84 %).

Requirement

Dependences exclusively needed to validate configuration settings must be avoided.

Further Requirements

- **Modularity:** diverse and conflicting requirements between applications. Especially in validation, for example,
  constraint solvers vs. type systems.

Further Requirements

- **Modularity:** diverse and conflicting requirements between applications. Especially in validation, for example,
  constraint solvers vs. type systems.
- **System-level:** specification must be enforced. Examples:

Further Requirements

- **Modularity:** diverse and conflicting requirements between applications. Especially in validation, for example,
  constraint solvers vs. type systems.
- **System-level:** specification must be enforced. Examples:
  - which desktop is the application started in?

Further Requirements

- **Modularity:** diverse and conflicting requirements between applications. Especially in validation, for example,
  constraint solvers vs. type systems.
- **System-level:** specification must be enforced. Examples:
  - which desktop is the application started in?
  - how many CPUs does the system have?

Further Requirements

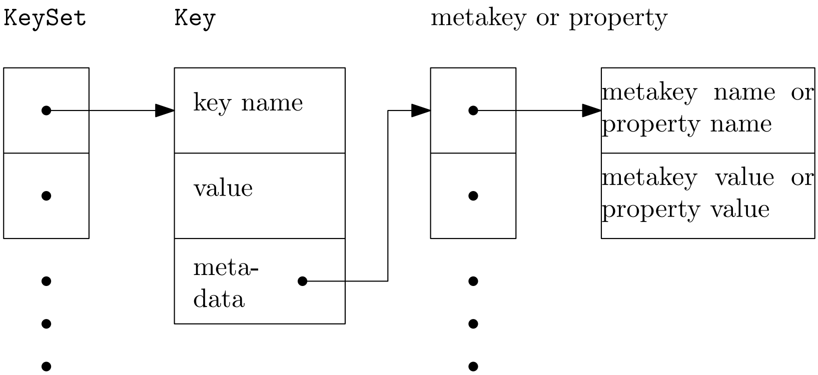
- **Modularity:** diverse and conflicting requirements between applications. Especially in validation, for example,
  constraint solvers vs. type systems.
- **System-level:** specification must be enforced. Examples:
  - which desktop is the application started in?
  - how many CPUs does the system have?
  - 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:

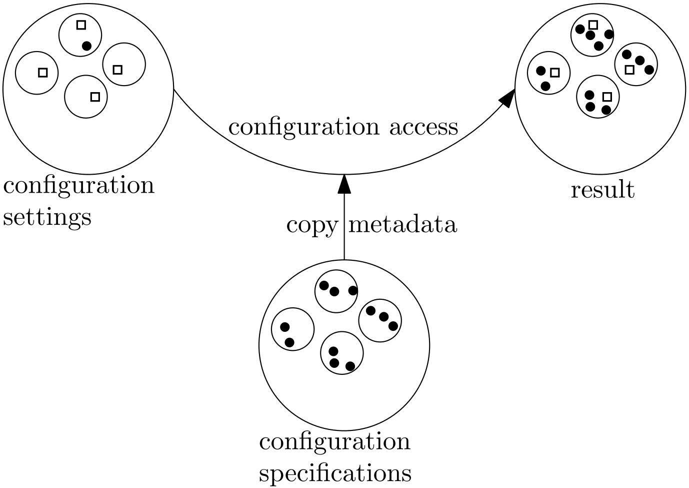
1. SpecElektra does not have any built-in feature, all features are (or can be) implemented as plugins.
2. Elektra works completely without SpecElektra's specifications.
3. Configuration specifications are present within the execution environment. Thus any tool and plugin can introspect and use the specifications.

KeySet

The common data structure between plugins:



Metadata


configuration access
configuration
settings
result
copy metadata
configuration
specifications

Plugin Assembly

automatic assembling of plugins:

- iterate over the specification and collect all key words

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

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

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

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.

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.

Examples

resolve names of configuration files

```
[example]
  mountpoint:=/example.ini
```

depending on operating system, e.g. UNIX:

namespace	resolved path
spec	/example.ini
dir	${PWD}/example.ini
user	${HOME}/example.ini
system	/example.ini

L06 Strategies for Validation and Modularization

Markus Raab

Institute of Information Systems Engineering, TU Wien

Meeting

1 Validation

2 Modularity
- Vertical
- Horizontal

3 Plugins
- Why?
- How?

4 Meeting
- Recapitulation
- Assignments
- Preview

Learning Outcomes

Students will be able to

- write simple checker plugins.

Checking Configurations vs. Specifications

Checking Configurations vs. Specifications

E.g. following properties of configuration settings can be checked:

- values (data types)
- structure
- constraints
- semantic checks (e.g., IP, folder)
- domain-specific checks (e.g., databases)

E.g. following properties of configuration specifications can be checked:

- Types of default values must be compatible with the types of the keys.
- Defaults must be present for safe lookups.

Brainstorming: What makes good/bad error messages

Error Messages

Problems:

- Generic vs. specific code?

Error Messages

Problems:

- Generic vs. specific code?
- Context?

Error Messages

Problems:

- Generic vs. specific code?
- Context?
- Cause?

Error Messages

Problems:

- Generic vs. specific code?
- Context?
- Cause?
- Help?

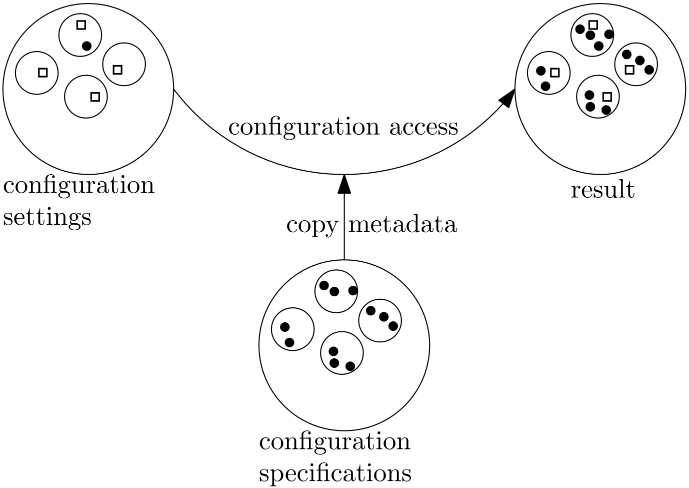
Task

Break.

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.

Metadata


configuration access
configuration
settings
result
copy metadata
configuration
specifications

Think, Pair, Share: Strategies for Validation

Task

Break.

H2 corrections: Write Readme for others!

P0 corrections

Which CM language do you want to use for H3?

Feedback

- Videos?
- Response time and helpfulness?
- More or less materials?
- Any other suggestions for improvements?

Outlook

- Terms and Properties
- Pitfalls
- Unused Settings

[1] Michael J. Lee and Andrew J. Ko. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the Seventh International Workshop on Computing Education Research*, ICER '11, pages 109–116, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0829-8. doi: 10.1145/2016911.2016934. URL `http://dx.doi.org/10.1145/2016911.2016934`.

[2] 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`.