Configuration Management

Markus Raab

Institute of Information Systems Engineering, TU Wien

13.4.2018

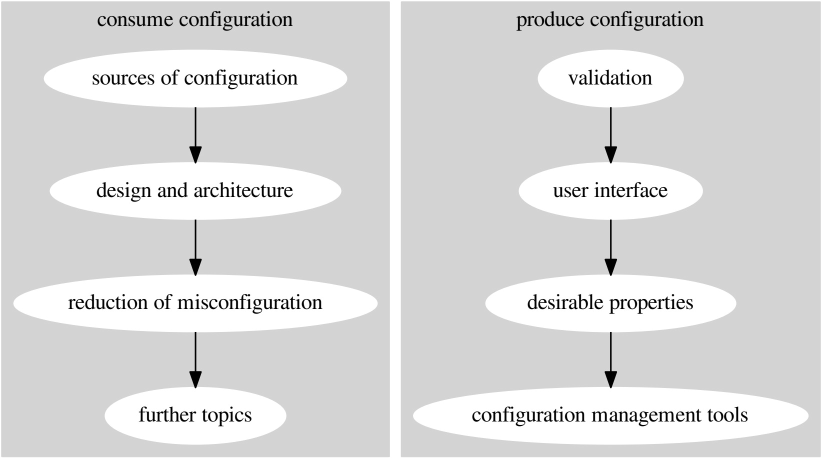
Organization

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

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 early
- 1 introspection
- 1 dependences
- 1 auto-detection
- 1 context-awareness
- 1 administrators


consume configuration
sources of configuration
design and architecture
reduction of misconfiguration
further topics
produce configuration
validation
user interface
desirable properties
configuration management tools

Configuration Access (Recapitulation)

Configuration Access (Recapitulation)

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 [2, 3, 7]. ***Configuration access APIs*** are APIs that enable configuration access.

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

Trend (Recapitulation)

- alarming trend in number and complexity of configuration settings

Trend (Recapitulation)

- alarming trend in number and complexity of configuration settings
- sharing, visibility and default value calculation often helps

Trend (Recapitulation)

- alarming trend in number and complexity of configuration settings
- sharing, visibility and default value calculation often helps
- needs abstraction: configuration specification

Trend (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

Trend (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

SpecElektra (Recapitulation)

SpecElektra (Recapitulation)

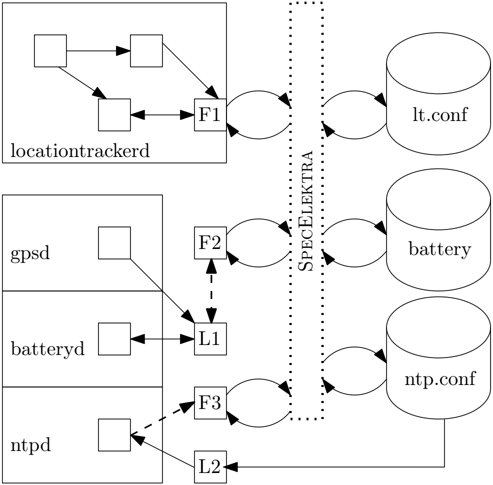
SpecElektra is a modular ***configuration specification language*** for configuration settings. In SpecElektra we use properties to specify configuration settings and configuration access. SpecElektra enables us to specify different parts of Elektra.

Modularity (Recapitulation)

Modularity (Recapitulation)

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



Needed to keep applications independently. Boxes are applications, cylinders are configuration files, F? are frontends or frontend adapters, L? are configuration libraries [5].

Plugins (Recapitulation)

Plugins (Recapitulation)

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.

Introspection (Recapitulation)

Introspection (Recapitulation)

- 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

Code Generation

Task

How to ensure that configuration access points match with present configuration settings?

Rationale (Partly Recapitulation)

Configuration Specification:

- without specification you and others do not even know which settings are available
- needed for any further techniques we will discuss:
  - code generation guarantees that configuration access points match with specification
  - validation guarantees that configuration settings match with specification
- essential for ***no-futz computing*** Holland et al. [1]
- the foundation for any advanced tooling like configuration management tools
- needed as communication of producers and consumers of configuration

Task

Brainstorming: Which artefacts can we produce with code generation?

Artefacts:

- generate examples/documentation
- auto-completion/syntax highlighting/IDE support
- tooling (GUI, Web UI)
- validation code
- configuration management tool code
- configuration access APIs

Current Challenges

Configuration Access Code usually has:

- code duplications
- hard-coded default values
- unexpected transformations
- no introspection facilities

Example

```
if (!strcasecmp(token, "on")) {
     *var = 1;
} else {
     *var = 0;
} /* src/cache_cf.cc from Squid */
```

Goal

Goal

Configuration settings should adhere the specification from source to destination.

Requirement

The specification must enable code generation and inconsistencies must be ruled out during compilation.

Code Generation

The code generator GenElektra reads SpecElektra specifications and emits high-level APIs to be used in applications. GenElektra facilitates the key names to generate unique API names.

Possible Properties

For example, SpecElektra has following properties:

- **type** represents the type to be used in the emitted source code.
- **opt** is used for short command-line options to be copied to the namespace proc.
- **opt/long** is used for long command-line options, which differ from short command-line options by supporting strings and not only characters.
- **readonly** yields compilation errors when developers assign a value to a contextual value within the program.
- **default** enables us to start the application even if the backend does not work.

With the specification:

```
[foo/bar]
  default:=Hello
  type:=string
  opt:=b
  readonly:=1
```

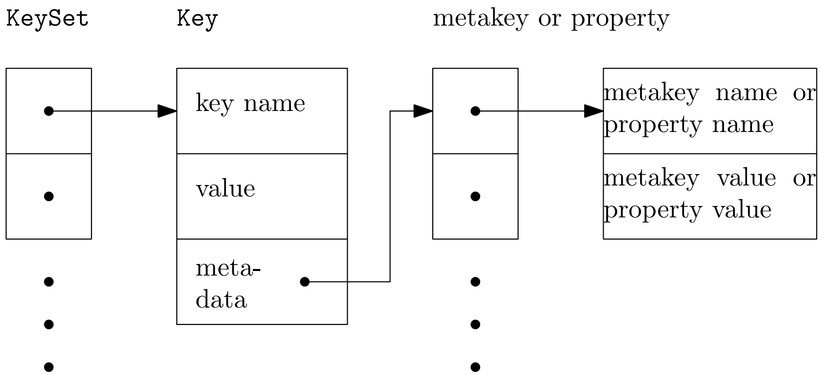
GenElektra gives the user read-only access to the object `env.foo.bar`:

```
1      std::cout << env.foo.bar;
2      env.foo.bar = "Other world"; // comp. error
```

Line 1 prints the configuration value of `/foo/bar` or `"Hello"` (without quotes) by default. When invoking the application with `application -b "This world"`, the application would print `"This world"` (without quotes). Line 2 leads to a compilation error because of the property readonly.

KeySet (Recapitulation)

The common data structure between plugins:



KeySet Generation

Question

Idea: What if the configuration file format grammar describes source code?

KeySet Generation

Question

Idea: What if the configuration file format grammar describes source code?

⟨*KeySet*⟩ ::= 'ksNew'␣(' { ⟨*Key*⟩ ', ↩' } { '␣' } 'KS_END);'

⟨*Key*⟩ ::= 'keyNew ␣(" ' ⟨*key name*⟩ '"' , ↩' [⟨*Value*⟩]
⟨*properties*⟩ 'KEY_END)'

⟨*Value*⟩ ::= { '␣' } 'KEY_VALUE, ␣" ' ⟨*configuration value*⟩ '"' ,
↩'

⟨*properties*⟩ ::= { { '␣' } ⟨*property*⟩ ', ↩' }

⟨*property*⟩ ::= 'KEY_META, ␣ ' <property name> ' ,
␣ ' <property value> ' '

Example

Given the key `spec:/slapd/threads/listener`, with the configuration value 4 and the property default $\mapsto$ 1, GenElektra emits:

```
ksNew (keyNew ("spec:/slapd/threads/listener",
               KEY_VALUE, "4",
               KEY_META, "default", "1",
               KEY_END),
          KS_END);
```

Example

Given the key `spec:/slapd/threads/listener`, with the configuration value `4` and the property default $\mapsto$ 1, GenElektra emits:

```
ksNew (keyNew ("spec:/slapd/threads/listener",
               KEY_VALUE, "4",
               KEY_META, "default", "1",
               KEY_END),
        KS_END);
```

Result

We have source code representing the settings! And if we instantiate it, we have a data structure representing the settings!

Which Configuration Access API?

First approach, one class (or function) per configuration setting:

```
class SlapdThreadsListener : public Value<long,
    WritePolicyIs<ReadOnlyPolicy>> {
    ... keyNew ("/slapd/threads/listener",
            KEY_META, "type", "long",
            KEY_META, "readonly", "1",
            KEY_END) ...
};
```

Which Configuration Access API?

Bad idea, manual instantiation and long names necessary:

```
KeySet config;
Context c;
long foo ()
{
    SlapdThreadsListener slapdThreadsListener (co
    slapdThreadsListener++;
    return slapdThreadsListener;
}
```

Which Configuration Access API?

Use hierarchy with namespaces or nasted classes:

```
namespace slapd
{
namespace threads
{
class Listener : public Value<long> {};
} // <continues on the next page>
class Threads : public Value<none_t>
{threads::Listener listener;};
} // end namespace slapd
class Slapd : public Value<none_t>
{slapd::Threads threads;};
class Environment {Slapd slapd;};
```

Which Configuration Access API?

Much easier to use:

```
long foo(slapd::Threads const & threads)
{
    threads.listener++;
    Context & c = threads.context (); // access co
    return threads.listener;
}

int main()
{
    KeySet config;
    Context c;
    Environment env (config, c);
    long x = foo (env.slapd.threads);
}
```

Which Configuration Access API?

In C, we use identifiers to be passed to the API:

```
elektraGetString (elektra, ELEKTRA_TAG_X);
```

Where `ELEKTRA_TAG_X` is a struct for that type.

Guarantees by code generation:

- Every configuration setting is specified.
- Configuration access with defaults is always successful.
  Reason: We compile in a KeySet and use it if everything else fails.

Missing Guarantee: Is every specified setting actually used?

Introspection vs. Generation

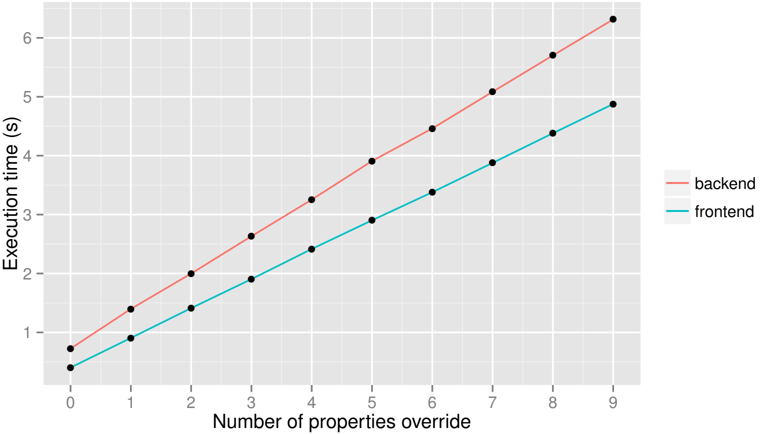
Question

Introspection vs. Code Generation?

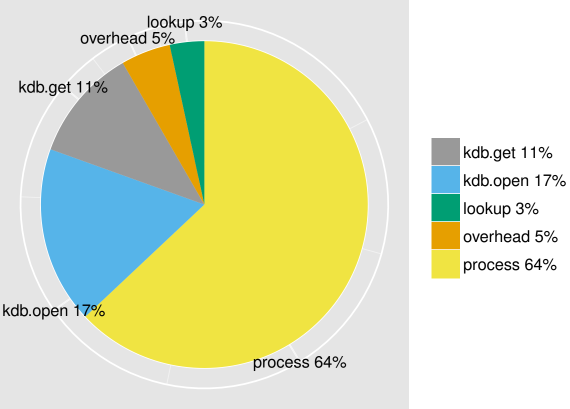
Limitations of introspection:

- no static checks
- no whole-program optimizations (API barriers)

Overhead without code generation (=backend) is 1.8x higher [4]:

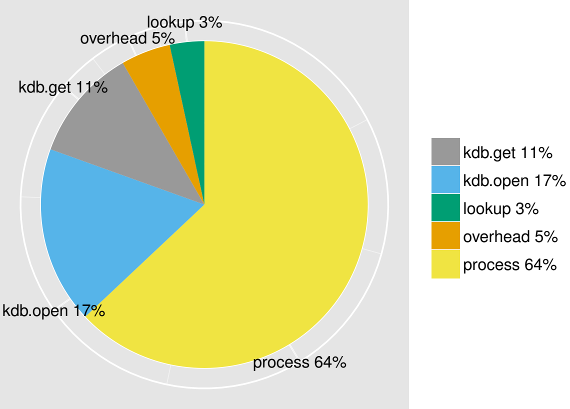


But it might not matter because configuration access might not be a bottleneck [4], for example, a word counting application:



But:

But it might not matter because configuration access might not be a bottleneck [4], for example, a word counting application:



But: Configuration access points within loops might be a bottleneck.

Advantages of introspection:

- specification can be updated live on the system without recompilation
- tooling has generic access to all specifications
- new features the key database (e.g., better validation) are immediately available consistently

Implication

We generally prefer introspection

Requirement

Configuration settings and specifications must be introspectable.

Testability

1. Code Generation
   - Why?
   - How?
2. Introspection vs. Generation
3. Testability

[2]

- Wants to improve configuration-ware testing and debugging
- Manual investigations for three applications
- Finds 1957 settings in Firefox ($2^{846} * 3^{1111}$) and 36322 in LibreOffice ($2^{4433} * 3^{31889}$)
- Finds unused settings: settings only in the source code
- Finds unsynchronized configuration settings (see in "early")

Requirement

Configuration setting traceability is a necessity.

Idea

Code generation helps to trace settings and to find unused settings.

Find Unused Settings

The first (optional) step of the algorithm is:

- Run all tests with code coverage.
- Check if generated code is executed.
- If it is, we know that the configuration setting is used in a test case. Otherwise, we know it is not tested by the test suite. All these untested configuration settings are remembered as candidates for the second step.

```
KeySet findUnusedSettings (KeySet untestedSettings,
                           KDB kdb,
                           Builder build)
{
    KeySet unusedSettings = {};
    KeySet configurationSpecification;
    kdb.get (configurationSpecification);

    for (candidate: untestedSettings)
    {
        configurationSpecification.remove (candidate);
        kdb.set (configurationSpecification);
        build.recompile ();
        if (build.wasSuccessful ())
        {
            unusedSettings.append (candidate);
        }
        configurationSpecification.append (candidate);
    }

    kdb.set (configurationSpecification);
    return unusedSettings;
}
```

Conclusion

- Challenges: duplications, transformations, ...
- Configuration access APIs with code generation
- Guarantees of configuration access points
- We reuse properties of SpecElektra (`type`, `default`)
- We prefer hierarchies and tags to long function names
- Usually introspection preferred, except for static type safety

Preview

- Puppet-Libelektra talk
- Early Detection of Misconfiguration

[1] David A. Holland, William Josephson, Kostas Magoutis, Margo I. Seltzer, Christopher A. Stein, and Ada Lim. Research issues in no-futz computing. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 106–110. IEEE, May 2001. doi: 10.1109/HOTOS.2001.990069.

[2] 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`.

[3] 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.

[4] Markus 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.

[5] 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`.

[6] Markus Raab and Gergö Barany. Introducing context awareness in unmodified, context-unaware software. In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, pages 218–225. INSTICC, ScitePress, 2017. ISBN 978-989-758-250-9. doi: 10.5220/0006326602180225.

[7] 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.