Configuration Management

Markus Raab

Institute of Information Systems Engineering, TU Wien

08.05.2018

Lecture is every week Wednesday 09:00 - 11:00.

06.03.2019: topic, teams
13.03.2019: TISS registration, initial PR
20.03.2019: other registrations, guest lecture
27.03.2019: PR for first issue done, second started
03.04.2019: first issue done, PR for second
10.04.2019: mid-term submission of exercises
08.05.2019: Different Location: Complang Libary
15.05.2019:
22.05.2019: all 5 issues done
29.05.2019:
05.06.2019: final submission of exercises
12.06.2019:
19.06.2019: last corrections of exercises
26.06.2019: exam

Tasks for today

(until 08.05.2019 23:59)

Task

Incorporate feedback for teamwork and homework.

Task

Third PR done, PR for fourth issue created and write some text in your last issue (if 5 issues are not yet assigned to you).

Tasks for next week

(until 15.05.2019 23:59)

Task

Fourth PR done, PR for fifth issue created.

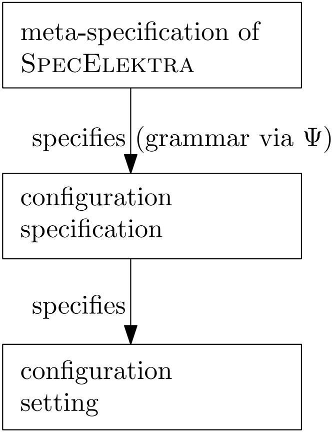
Popular Topics

14	tools	4	design
9	testability	4	cascading
9	code-generation	4	architecture of access
7	context-awareness	3	configuration sources
6	specification	3	config-less systems
6	misconfiguration	2	secure conf
6	complexity reduction	2	architectural decisions
5	validation	1	push vs. pull
5	points in time	1	infrastructure as code
5	error messages	1	full vs. partial
5	auto-detection	1	convention over conf
4	user interface	1	CI/CD
4	introspection	0	documentation

Metalevels (Recapitulation)

Question

Describe the three Metalevels in Elektra.



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.

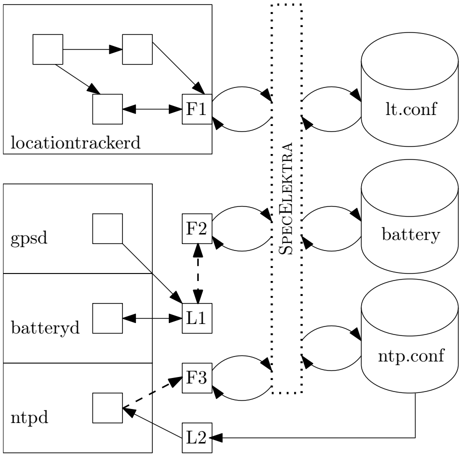
Recapitulation (Requirements of SpecElektra)

- 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

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 (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)

- 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

Goals for today

learning outcome:

- evaluate a configuration system and decide about use of
    - code generation
    - system-wide introspection

Code Generation

1. Code Generation
   - Why?
   - How?
2. Introspection vs. 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:

- examples (e.g., defaults)
- 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

```c
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.

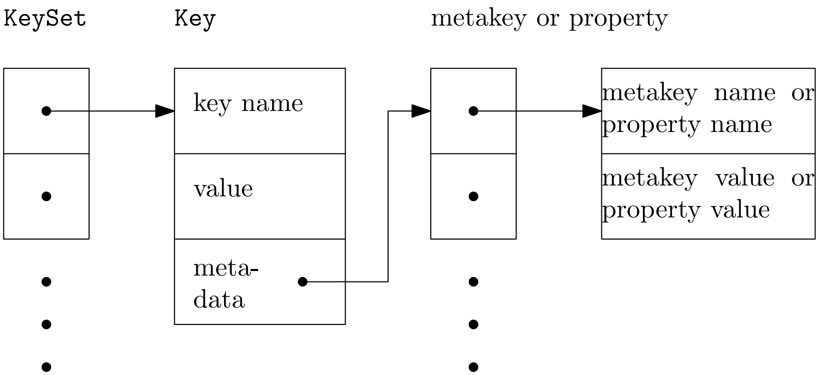
Code Generation

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.

But how?

KeySet (Recapitulation)

The common data structure between plugins:



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>` ‘ ’

Task

Break.

Example

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);
```

Finding

We have source code representing the settings. And if we instantiate it, we have a data structure representing the settings. Plugins emitting such "configuration files" are code generators.

Implementation Strategies

- Using `print` (only for very small generators)
- Using generative grammars

```
query = '{' >> *(pair) > '}';
pair = '{' >> key_name > '=' >> key_value >>
           *('{' >> metakey_name > '=' >> metakey_
           > '}';
```

- Using template languages (RubyERB, Cheetah, Mustache)

```
@for n in hierarchy.name.split('/')[1:-1]
namespace $support.nsnpretty($n)
{
class ${hierarchy.prettyclassname(support)}
{
typedef $support.typeof($hierarchy.info) type;
@if $support.typeof($hierarchy.info) != "kdb::none_t"
static type get(kdb::KeySet &ks, kdb::Key const& spec)
{
    type value $support.valof($hierarchy.info)
    Key found(ckdb::ksLookup(ks.getKeySet(), *spec,
                ckdb::elektraLookupOptions::KDB_O_SPEC));
    return found.get<$support.typeof($hierarchy.info)>();
}
```

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.

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

1. Code Generation
   - Why?
   - How?

2. Introspection vs. Generation

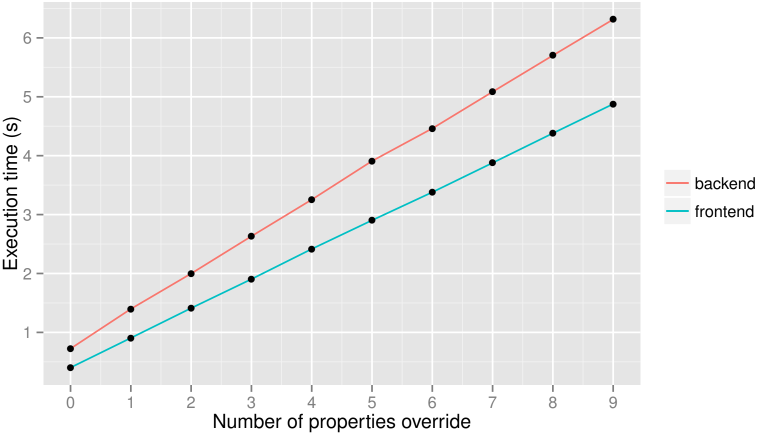
Question

Introspection vs. Code Generation?

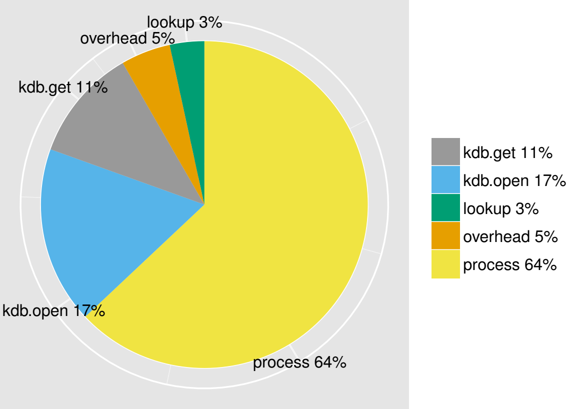
Limitations of introspection:

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

Overhead without code generation (=backend) is 1.8x higher [2]:



But it might not matter because configuration access might not be a bottleneck [2], 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, except for a very thin configuration access API.

Requirement

Configuration settings and specifications must be introspectable.

Preview

- Testing
- 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] 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.

[3] 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, SciteePress, 2017. ISBN 978-989-758-250-9. doi: 10.5220/0006326602180225.