Configuration Management

Markus Raab

Institute of Information Systems Engineering, TU Wien

27.4.2018

Organization

Next dates:

- 27.4.2018: **concepts for team exercise**
- 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 detection
- 1 introspection
- 1 dependences
- 1 auto-detection
- 1 context-awareness
- 1 administrators

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 [3, 5, 9]. ***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
- 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

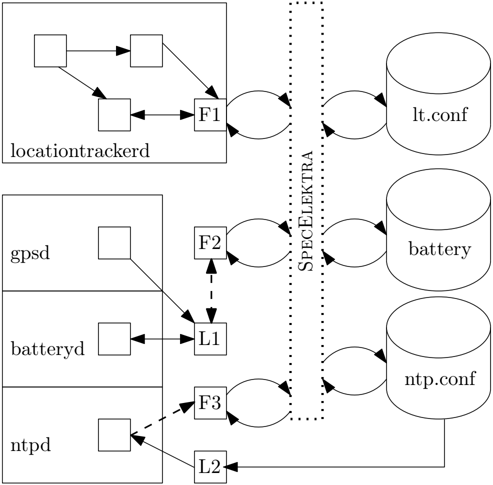
SpecElektra (Recapitulation)

SpecElektra 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)

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)



Needed to keep applications independently. Boxes are applications, cylinders are configuration files, F? are frontends or frontend adapters, L? are configuration libraries [7].

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.

Code Generation vs. Introspection

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

Rationale (Recapitulation)

Task

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

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

Internal Specification

For example, OWNER:

```java
import org.aeonbits.owner.Config;

public interface ServerConfig extends Config {
    int port();
    String hostname();
    @DefaultValue("42")
    int maxThreads();
}
```

Task

Why do we need an external specification?

Introspection:

- needed as communication of producers and consumers of configuration
- the foundation for any advanced tooling like configuration management tools
- essential for ***no-futz computing*** Holland et al. [2]

External Specification

```
[port]
type:=long
[hostname]
default:=42
[threads/max]
type:=long
```

External Specification:

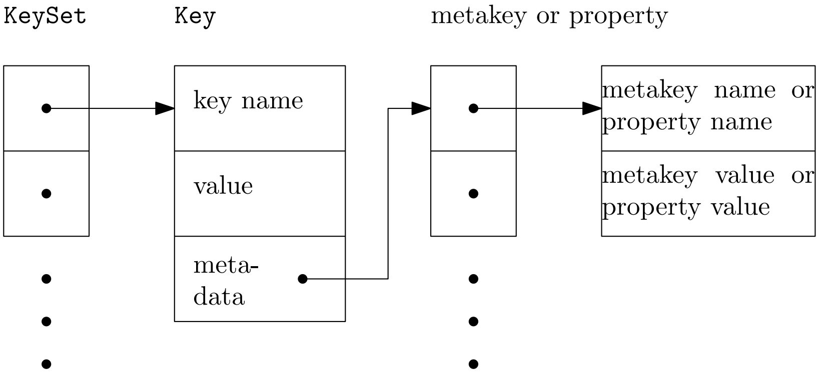
- are read and writable by other applications
- we still can generate the internal specification
- furthermore, we fulfill needs for configuration management tools

Other Artefacts (Recapitulation):

- examples (e.g., defaults)
- documentation
- auto-completion/syntax highlighting/IDE support
- tooling (GUI, Web UI)
- validation code
- configuration management tool code

KeySet (Recapitulation)

The common data structure between plugins:



KeySet Generation (Recapitulation)

Question

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

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

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

Finding

We get source code representing the settings.

Possible Properties (Recapitulation)

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.

(Recapitulation)

Question

Introspection vs. Code Generation?

- − more techniques for performance improvements with code generation
- + 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.

Conclusion (Code Generation)

- Challenges: duplications, transformations, ...
- KeySet equivalence: settings are instantiated configuration files
- 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

Testability

1. Code Generation vs. Introspection
2. Testability
3. Early Detection

Question

What do we want to test?

- That settings do what they should (devs and admins)
- That settings are properly validated (devs [9])
- Regression tests [6]

- Are all settings implemented?
- Are all settings used in tests?
- Are there unused settings in the code?

Matt Welsh from Google wrote in 2013:[1]

"Of course we have extensive testing infrastructure, but the 'hard' problems always come up when running in a real production environment, with real traffic and real resource constraints. Even integration tests and canarying are a joke compared to how complex production-scale systems are."

Most of these problems are still not well understood.

[1] What I wish systems researchers would work on. Retrieved from http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html.

Jin et al. [3]

- Wants to improve configuration-aware 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

Requirement

Configuration setting traceability is a necessity.

Idea

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

Testing by developers:

- ConfErr [4] uses models of key board layout, psychology and linguistics. Tool injects possible misconfiguration.
- Spex [9] analyzes the source code to find misconfigurations. As by-product it extracts internal specifications (including transformation bugs).
- External specification can be directly used to generate test cases.
- Find unused configuration 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;
}
```

Early Detection

1. Code Generation vs. Introspection
2. Testability
3. Early Detection

When are settings used?

Implementation-time configuration accesses are hard-coded settings in the source code repository. For example, architectural decisions [1] lead to implementation-time settings.

Compile-time configuration accesses are configuration accesses resolved by the build system while compiling the code.

Deployment-time configuration accesses are configuration accesses while the software is installed.

Load-time configuration accesses are configuration accesses during the start of applications.

Run-time configuration accesses are configuration accesses during execution not limited to the startup procedure.

Latent Misconfiguration

Phases when we can detect misconfigurations:

- Compilation stage in configuration management tool
- Writing configuration settings on nodes
- Starting applications (load-time)
- When configuration setting is actually used (run-time)

Problem

More context vs. easier to detect and fix.

As shown in [10]:

- 12 % – 39 % configuration settings are not used at all during initialization.
- Applications often have latent misconfigurations (14 % – 93 %)
- Latent misconfigurations are particular severe (75 % of high-severity misconfigurations)
- Latent misconfiguration needs longer to diagnose

Using code generation

Code generation makes sure that only specified configuration settings are used.

Checkers as plugins

Using checkers as plugins exclude whole classes of errors such as:

- Invalid file paths using the plugin *"path"*.
- Invalid IP addresses or host names using the plugins *"network"* or *"ipaddr"*.

Because the checks occur before the resources are actually used, the checks are subject to race conditions.
For example, a path that was present during the check, can have been removed when the application tries to access it.
In some situations facilities of the operating system help,[1] in others we have fundamental problems.[2]

[1] For example, we open the file during the check and pass `/proc/<pid>/fd/<fd>` to the application. This file cannot be unlinked, but unfortunately the file descriptor requires resources.

[2] For example, if the host we want to reach has gone offline after validation.

Example [10]

Squid uses `diskd_program` but not before requests are served. Latent misconfiguration caused 7h downtime and 48h diagnosis effort.

Finding

Configuration from all externals programs need to be checked, too.

Conclusion

- provide external specifications for other tooling and configuration management
- use code generation to keep internal specifications consistent with external specifications
- implement checkers as plugins
- execute checkers as early as possible

[1] Neil B Harrison, Paris Avgeriou, and Uwe Zdun. Using patterns to capture architectural decisions. *Software, IEEE*, 24(4): 38–45, 2007. ISSN 0740-7459. doi: 10.1109/MS.2007.124.

[2] 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.

[3] 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.

[4] Lorenzo Keller, Prasang Upadhyaya, and George Candea. Conferr: A tool for assessing resilience to human configuration errors. In *Dependable Systems and Networks With FTCS and DCC, 2008.*, pages 157–166. IEEE, 2008.

[5] 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.

[6] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 75–86, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. doi: 10.1145/1390630.1390641. URL `http://doi.acm.org/10.1145/1390630.1390641`.

[7] 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`.

[8] 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.

[9] 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.

[10] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, USA, November 2016.