Safe Management of Software Configuration

Markus Raab

Vienna University of Technology
Institute of Computer Languages, Austria
markus.raab@complang.tuwien.ac.at
Supervisor: Franz Puntigam

Abstract. We tend to write software in a parameterized way with parameter values specified in configuration files. Such configurability allows us to deploy software in a context not initially thought of, but it also has the downside that it introduces a new class of hard-to-track faults. This issue arises because the use of configurations in programs is not integrated into configuration management needed by administrators. We propose a light-weight specification language to be used by both parties. From this specification we generate configuration access code that includes compile-time checks. Furthermore, we use the same specification to add run-time checks for safe configuration management. We expect that our approach averts many failures configurable software faces today. Additionally, we think it improves the quality altogether, because the documentation resulting from the specification leads to a better understanding of the overall system.

1 Introduction

Today many behavioral aspects of applications are not fixed at compile-time, but are determined at run-time by examining configuration files, environment variables, and command-line arguments. Even for an average software system the configurability is complicated due to a huge number of possibilities, constraints and dependencies.

As studies revealed [8], [13] much time and money is wasted because of configuration errors. Misconfigurations are one of today's major causes of system failures. Faulty configuration files sometimes trigger crashes and make services unavailable. These problems lead to downtimes, severe outages and a frustrating process of debugging configuration problems.

The state of the art in software configuration is to use schemata to describe the data in the *key databases* (they facilitate access to software configuration) and type systems to describe the corresponding variables in the programming languages. These two worlds are disconnected. We think that this gap causes most of these failures.

A recent paper supports our view and argues that users are not the ones to blame for misconfiguration [12]. The authors found evidence that the data-flow path from variable initialization to variable use contains many potential

errors. We think that these errors are only the symptom of the state-of-the-art development. We are positive that consequent use of a software configuration specification mitigates these issues.

In the thesis discussed in this paper we design and study a novel specification language and its integration in a key database. The specification should include constraints for both the key database and the variables in order to achieve following benefits and goals:

- Provide safe use of variables within programming languages containing values of configuration files by exploiting compile-time checks using type systems.
- Adding run-time checkers when compile-time checking is not possible, e.g. for managing the key databases.
- From the specification other artifacts can be derived, yielding improvement compared to state-of-the-art systems. E.g., in PostgreSQL[1] 5 artifacts needs to be maintained in the software engineering process.

The specification facilitates code generation in the programming languages used by the applications. When the application and the generated code is compiled, compile-time checks detect many problems at an early stage. The specification allows software architects, developers and end-users to have a better understanding of software configuration, e.g., it deals with documentation and improves traceability. So the specification can even lead to an entirely better software system.

For example, the OpenLDAP 2.4.39 daemon crashes when "listener-threads" is configured to be larger than 15 [12]. The documentation for this configuration item does not even mention this limit nor that these values are internally changed to be a power of 2. In our approach we solve this issue by writing a specification:

```
[/openldap/listener-threads]
  type=enum 1 2 4 8
```

We have the identifier `/openldap/listener-threads` and one *property* `type`. Because of this property, we know which values are permitted. When the user changes the value of "listener-threads" to 16, the key database tells him/her that 16 is not one of the allowed values 1, 2, 4 or 8. Using this specification OpenLDAP would not crash and the difficult process of debugging is avoided. For a developer the approach is intuitive: configuration items can be used like variables, e.g., the following C++ code prints the value of the configuration item:

```
std::cout << openldap.listener-threads << std::endl;
```

The library *libelektra* provides access to the key database. The code generator *genelektra* makes sure that configuration items used by the developer always match with the specifications. We also generate documentation that includes the type information from the specification. Any other property in addition to `type` can be added, which means the specification is extensible.

[1] Version 9.1.12, see `http://doxygen.postgresql.org/guc_8c_source.html`

We expect, given powerful properties in the specification, it is simple to write a specification that avoids crashes, because 90% of all options are covered with a dozen types [8]. Using this approach, mismatches are ruled out and faulty use of the variable is detected by the compiler. The substantial gain is that it enhances type safety without leaving the familiar programming environment. Other potential benefits of our approach are improved software maintenance and evolution as well as reduced duplication of code.

The rest of the paper is structured as follows: Section 2 describes the details of our approach. In Section 3 we show how we plan to validate or falsify our research questions. In Section 4 we talk about expected results based on our current knowledge. Finally, in Section 5 we compare our approach to related work before drawing our conclusions in Section 6.

2 Elektra

Our approach, called *Elektra*, introduces a specification for configuration. Instead of many places containing constraints and types, Elektra defines a clear way how to specify configuration. The key database *libelektra* enforces the constraints at run-time and a code generator *genelektra* ensures program code conforms to it.

The approach is still in its infancy and thus many vital questions are not yet answered. The aim of our thesis is to answer following question: **What kind of influence has the use of our configuration specification framework, i.e. Elektra, on software?** The two subquestions, to solve or at least alleviate the problems stated in Section 1, are:

1. Which properties in the specification have the strongest influence on avoiding software failures caused by invalid configuration files?
2. How does the specification interact during software engineering processes with software architectures, software evolution, and software quality?

Elektra's Architecture

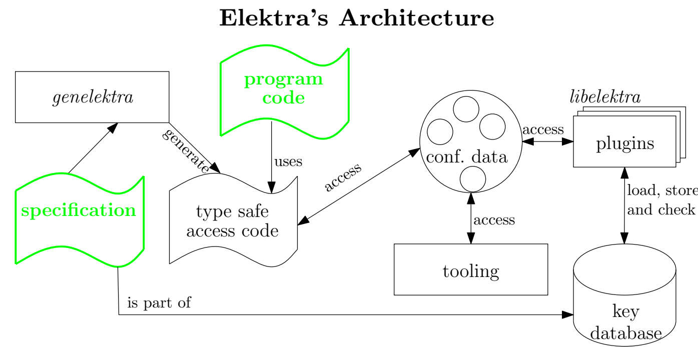

Fig. 1. Boxes represent software artifacts. The bold boxes show artifacts developers need to implement. Our *Elektra* implementation provides the other libraries and tools.

In Fig. 1 we see how to apply our approach and which artifacts it consists of. We immediately spot the configuration data structure in the center. It contains

all key/value pairs representing configuration items. As used in every configuration parser they form a generic, but unsafe, container. In our approach we pass it around between tools, plugins and the type safe access code to avoid tight coupling between these components.

2.1 Key Database

The *key database* is responsible for retrieval and storage of configuration items. We inherit Elektra's key database and plugin system from our earlier work [5]. Configuration is retrieved and stored using different plugins. While some of the plugins are responsible for the obvious tasks, e.g. parsing configuration files, others take care of cross-cutting concerns or implement run-time checkers. Latter plugins use the specification as input and perform run-time checks and validate input before it is stored in the key database.

2.2 Specification

As we see in Fig. 1 the *specification* is deployed as part of the key database. By including the specification in the key database, we build up an information system which supports administrators in the process of creating correct software configuration. Constraints, types and links support administrators in this process. The specification states how a valid configuration is structured and which values are permitted.

For our thesis we are particularly interested in specification validation. The specification validation needs to fulfill the following tasks:

1. Check if the specification is consistently typed and has no conflicting constraints.
2. Compile a minimal list of plugins that can perform the run-time checks.
3. Check if the plugins will work together. For run-time checkers it is known that such a check is a non-trivial task.
4. Check if the specification has a safe upgrade path from its previous version.
5. Ensure that the particular configuration file (syntax and structure) works with this specification by checking if they have a common supertype.

2.3 Code Generation

The *code generator* is used to generate all other configuration-related artifacts from the specification. We are especially interested in the generation of *type safe access code*. In earlier work, context awareness turned out to be useful to provide a type safe access code using C++ [6], [7]. The types used in the specification must be mapped to types or generated classes within programming languages for code generation.

The compile-time safety of the approach stems from the fact that no identifier string nor self written type conversions exist in the application's source code. Instead, the developer prefers to use generated variables. Without our approach, file names and other identifiers usually exist as strings in the code.

3 Validation & Methods

The scientific foundation and starting point of our research is in the area of modularity [5]. To validate our first question, i.e. which properties have the strongest influence, we first must find out which properties are good choices for our problem domain and need an implementation of them. The following *run-time checkers* are candidates as properties in the specification:

- structure validation with CORBA data types (as shown in our thesis [5]),
- more powerful data types, e.g. units of measurement,
- novel ways to define subtyping,
- types inference using unification,
- global constraints, e.g. using Gecode, Coinor and Z3,
- schemas, e.g. Relax NG Schema and XSD,
- Data Format Description Languages,
- configuration value deduction and
- any combination of the approaches above.

With the described tooling and an implementation of run-time checkers the validation of the first question in Section 2 is straight forward:

1. By analyzing real-world problems we find out which kinds of typical and sophisticated configuration errors occur in practice, e.g.:
   (a) Typos (e.g. insertion, substitution, transposition),
   (b) Structural errors (e.g. missing sections, parameters in wrong sections),
   (c) Semantic errors (e.g. wrong version, documentation, confusing similar applications) and
   (d) Domain-specific errors (e.g. no such resource)
2. We build a model [3] that allows us to construct such configuration errors.
3. We build a run-time checker that permits us to reject erroneous configuration based on a promising technique, i.e., one of those listed above.
4. We evaluate the run-time checker, e.g., by comparing the expressiveness and usability of the specification.

The question of the influence during software development asked in Section 2 is much more challenging, because it involves user studies. Case studies of individual attempts can provide valuable insight. The following validation plan is even more precious:

1. We create an assignment (a list of requirements) that is specifically designed to have a non-trivial, but not too complex configuration. To reduce the effort for the participants, we implement most parts of the application, except of the configuration relevant parts.
2. We train *all* participants how to use our approach. The explanation includes how to write the specification.
3. We randomly choose two groups A and B out of the participants:
   (a) Group A solves the task by using a specification (with the best checkers from the previous validation step present).

   (b) Group B solves the task without a specification.
4. During the development we make snapshots of the work. Each snapshot will be tested by injection of erroneous configuration and running unit tests.
5. Finally, the participant fills out a questionnaire to answer the usefulness of the specification and checkers on a Likert scale.

Using this method, we can answer the questions if there is a difference between group A and B regarding:

1. The needed efforts.
2. Which applications are more safe.
3. If the participants think the specification was useful.

We identified following risks and threads to validity:

1. The selection of participants might be biased.
2. The participants may not have many years of experience and their learning curve might not be representative.
3. When we teach the specification we might give a group an unfair advantage.
4. The number of participants might be too small to give results beyond the group.

To mitigate these issue we add graduates and employees to our pool. Additionally, we will use case studies and benchmarks to show other properties.

4 Expected Results

4.1 Performance

In previous work [7], we showed that the access of the variables representing the configuration values does not impact performance compared to the use of native variables. Because many applications use strings at run-time, we expect that applications will even benefit from our approach in respect of run-time. For initial startup we expect that only a reasonable overhead will be added. Some additional startup time compared to hard-coded solutions, however, is unavoidable because of the abstraction Elektra provides: no configuration file names are fixed at compile-time and a generic container is used.

4.2 Specification

We expect that the specification will present a powerful way to precisely define all influencing parts of the software configuration. We also think that the quality of documentation will rise as a result of less duplication. The properties of the specification, that includes type information, will give valuable hints often not available in today's systems. More assumptions will be stated explicitly.

We expect the availability of the specification in the key database to play a crucial role for interoperability: It will allow us to facilitate validation on every access, even by applications not aware of a specific specification.

Moreover the specification will allow us to add traceability links to architectural decisions [2]. As a result, we expect our approach to improve the traceability and decision making process.

4.3 Safety

Type safety means that a system prevents certain kinds of errors. Because of the additional compile-time and run-time checks, we expect applications using our approach to be safer in respect to the problems mentioned in Section 1. We think that most problems can be solved by adding a minimal amount of properties in the specification. For some issues, more effort will be required from the developers.

4.4 Less Effort

We expect that a key database with integrated specification will make it easier for administrators to make the right decisions in shorter time. We also think that validation sometimes even will avoid the necessity of debugging configuration problems.

The integration with the key database will allow us to change many configuration items in a safe way across applications without manual intervention. We expect this property to have a similar effect as has the use of DNS names instead of IP addresses.

5 Related Work

Currently, to the best of our knowledge, no other approach permits us to specify configuration independent of the used technology (e.g. XSD works with XML). Configuration parsers (e.g. Apache commons configuration) need the specification of configuration data additional to the specification of configuration variables. They do not detect mismatches between code accessing configuration and the schemata of the data. We conclude the use of these libraries leads to all issues described in Section 1. Moreover, they do not provide means to abstract over file location and syntax, but need this information hard-coded.

Pluggable types [4] tackle some issues mandatory type systems have and are still an active research topic. These type systems are both used for popular dynamic and static programming languages, but are currently not available for specification of software configuration systems.

ConfErr [3] is able to detect configuration errors by injecting wrong configurations before starting the application. The main difference to our approach is, that ConfErr does not use a specification. We cannot directly extend ConfErr for our benchmarks because it uses an internal representation which does not support all configuration standards Elektra supports.

Range Fixes [11] make use of constraints in order to support the administrator in the decision making process, but the authors did not tackle the problem of wrong use of configuration items in the code of applications.

AutoBash [9] and ConfAid [1] have similar goals as Elektra. In these approaches predicates, that test the application, must be available on the productive system. We think that testing should not happen on the productive system,

but instead earlier in the software engineering process. In our approach, possible problems will be ruled out by the specification so that they cannot occur in the productive system.

Spex [12] can infer parts of the specification by analyzing the code. This approach is complementing our approach in the sense that it can be used for initial construction of the specification for legacy code. It is, however, not suitable for a software engineering process. Even though Spex is the best tool available at the moment, it can only detect less than 40% of bad reactions. Because in our approach constraints are explicitly defined in the specification, the number is expected to be much higher, only limited by mistakes in the specification.

Software product lines often assume that different products have different deliveries. In our approach, the same binary can be used in different deployments. In approaches that delay variability up to the execution of the application [10] our work complements product lines by increasing safety on configuration changes.

6 Conclusion

In this paper we discussed further directions of a thesis with the objective to improve integration and safety of key databases. We propose a simple configuration specification language that is only data integrated in a key database. The specification provides support for administrators configuring the system. Additionally, the specification allows us to synthesize code in order to eliminate potential incorrect use of configuration items in the application.

So far, we have achieved:

1. A fully working key database [5] with several dozens of plugins to support many configuration file standards and to provide some run-time checkers.
2. A fully working code synthesis tool [7] with support for thread-local and global context awareness for embedded systems [6].
3. No overhead when reading configuration items [7].
4. An implementation of Elektra (see `http://www.libelektra.org`) is freely available and can be used to see current progress of our work. Elektra already includes all components as shown in Fig. 1.

These contributions are significant, because they lead to a specification language for code synthesis and run-time checkers that mitigate the issues as mentioned in Section 1. They are also practically relevant, because they provide stakeholders a good understanding of their system's configurability and might even reduce crashes and downtime.

In the next steps we will:

1. further define a specification language and its properties,
2. implement tooling to verify specifications and configurations (run-time and compile-time checkers), and
3. conduct the implementation and study as outlined in Section 3.

References

1. Attariyan, M., Flinn, J.: Automating configuration troubleshooting with dynamic information flow analysis. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. pp. 1–11. OSDI'10, USENIX Association, Berkeley, CA, USA (2010)
2. Harrison, N.B., Avgeriou, P., Zdun, U.: Using patterns to capture architectural decisions. Software, IEEE 24(4), 38–45 (2007)
3. Keller, L., Upadhyaya, P., Candea, G.: Conferr: A tool for assessing resilience to human configuration errors. In: Dependable Systems and Networks With FTCS and DCC, 2008. pp. 157–166. IEEE (2008)
4. Papi, M.M., Ali, M., Correa Jr, T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for java. In: Proceedings of the 2008 international symposium on Software testing and analysis. pp. 201–212. ACM (2008)
5. Raab, M.: A modular approach to configuration storage. Master's thesis, Vienna University of Technology (2010)
6. Raab, M.: Global and thread-local activation of contextual program execution environments. In: Proceedings of 11th International IEEE/IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems. pp. 1–8. IEEE (2015)
7. Raab, M., Puntigam, F.: Program execution environments as contextual values. In: Proceedings of 6th International Workshop on Context-Oriented Programming. pp. 8:1–8:6. ACM, NY, USA (2014), `http://doi.acm.org/10.1145/2637066.2637074`
8. Rabkin, A., Katz, R.: Static extraction of program configuration options. In: Software Engineering (ICSE), 2011 33rd International Conference on. pp. 131–140. IEEE (2011)
9. Su, Y.Y., Attariyan, M., Flinn, J.: Autobash: improving configuration management with operating system causality analysis. ACM SIGOPS Operating Systems Review 41(6), 237–250 (2007)
10. Van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on. pp. 45–54. IEEE (2001)
11. Xiong, Y., Hubaux, A., She, S., Czarnecki, K.: Generating range fixes for software configuration. In: Proceedings of the 34th International Conference on Software Engineering. pp. 58–68. ICSE '12, IEEE Press, Piscataway, NJ, USA (2012), `http://dl.acm.org/citation.cfm?id=2337223.2337231`
12. Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., Pasupathy, S.: Do not blame users for misconfigurations. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 244–259. ACM (2013)
13. Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L.N., Pasupathy, S.: An empirical study on configuration errors in commercial and open source systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 159–172. SOSP '11, ACM, New York, NY, USA (2011)