# COMMON ABSTRACTION OF CONFIGURATION FROM MULTIPLE SOURCES

Jaroslav PORUBÄN, Milan NOSÁĽ
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice,
Letná 9, 042 00 Košice, Slovak Republic, tel.: +421 55 602 4179, e-mail: {milan.nosal, jaroslav.poruban}@tuke.sk

**ABSTRACT**

*Configuration is an important part of design of software systems. There are many different configuration formats, such as XML, YAML, attribute-oriented programming, etc., that allow system provider to design configuration language according to his requirements. Target group of system users is often very wide and one configuration language can not meet all user requirements. The paper introduces and analyzes idea of supporting multiple configuration sources using common abstraction of configuration sources in order to meet more user requirements without severe increase of provider's costs. The paper presents design of tool providing common abstraction of configuration sources. Design is based on analysis of existing tools, and it is extended with idea of declarative representation of mapping of configuration languages to output format and of process of their combining. At last, the paper presents proof-of-concept implementation of the tool called Bridge To Equalia and states several conclusions based on experiments realized with Bridge To Equalia.*

***Keywords:*** *configuration, abstraction, multiple sources, configuration language, Java annotations, XML documents*

## 1. INTRODUCTION

Idea of configuration in software systems allows configurable system to be modeled, customized or personalized to meet specific requirements of customer or to be adapted to special circumstances or environment.

From the provider's point of view, implementing configurable system brings advantages in form of satisfying more clients, strengthening competitive advantages, providing better flexibility, robustness, quality and transparency in system. The system's ability to evolve is crucial to the user of system, and so it is to provider (as his income depends on user's satisfaction). [3]

On the other side, user profits from customizing system to her goals, preferences, abilities ands skills [5]. This way a configurable system satisfies user's individuality and raises efficiency of her work [8]. Also, using a configurable system instead of custom one means spending less money, because customizing configurable system is usually cheaper than developing a new custom system (also evolution of system is more manageable) [7].

### 1.1. Motivation

Although there are many benefits of configurable systems in comparison with custom, one can run upon many problems with expressing configuration while adapting a configurable system (for instance ones presented in [4, 8]). Configuration as a concrete instance of configurable system is expressed as a sentence in a custom domain-specific language [9] (DSL), called configuration language. Each user prefers configuration language that meets her taste and needs the best.

While choosing a suitable configuration language, there are considered aspects as simplicity of language, its verbosity and sententiousness, complexity of configuration process, domain abstraction, etc. [4, 5]. But user's preferences are based on subjective motives as well as on objective. It is easier to learn new XML configuration language than an annotations-based one, when you are familiar with XML but not with attribute-oriented programming (@OP is new and very popular format of configuration languages [13]).

To encourage user to utilize configuration, it is best to give her option of expressing configuration in the language that she is most familiar with. The most straightforward way to deal with the problem is to choose one of available formats (such as XML, YAML, INI, @OP, etc.) and to design configuration language that suits the needs of potential users the best. But usually there is not a format, that would meet all important requirements and its drawback would be negligible. For instance, source code annotations (@OP) are shorter than XML documents, but changing configuration in annotations requires recompilation. On the other hand, configuration for some programs is so simple and small, that it is easiest to use command line arguments, .INI files or .properties. These and more similar arguments lead to realization of need for support of multiple configuration languages.

Naturally, implementing processing of more configuration languages costs more resources. Code, that process configuration, becomes larger, and its maintenance harder and error-prone (due to mixed processing of more configuration languages). There is also negative impact on evolution of used configuration languages (changing language requires changes in processing code too).

Situation can be summarized in following two statements:

- **1** supported configuration language – risk of **dissatisfied users**.
- **Multiple** supported configuration languages – **increased costs** of implementation and maintenance of system.

This reasoning brings up a question: How to support multiple configuration languages without significant raise of costs and decrease of processing code simplicity (readability, maintainability)? This paper is trying to provide answer to the problem.

## 2. ANALYSIS

If some system is supposed to support multiple configuration languages, then it should to be able to process configuration in any of supported languages. For example, if the system supports INI files and XML documents, user must be able to freely choose between these formats. A stronger requirement is an option of random mix of configuration languages. The complete configuration is then composed of partial configurations expressed in different languages. So far, three solutions have been recognized: Ad-hoc solutions, source transformations and common abstraction of configuration sources.

### 2.1. Ad-hoc solutions

As the title suggests, provider needs to implement processing of all desired configuration languages ad-hoc. The concept of this solution is outlined in Fig. 1. Adding a new supported configuration language is basically implementing a whole operation of processing configuration in given language (difficulty is comparable to situation, when the system supports only one configuration language). But the fact, that the processing code needs to be integrated into existing configuration interface (code for other languages), makes the implementation even more difficult. The code processing one language may interleave with the code for other languages, and this results in worse maintainable implementation.

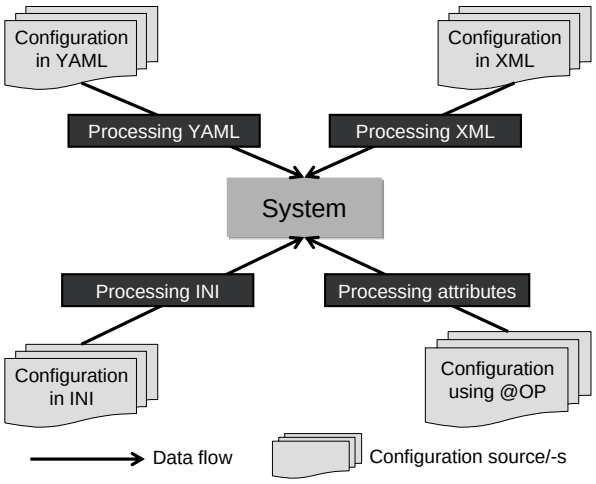


**Fig. 1** Example of Ad-hoc solution with four configuration languages

This approach appears to be the most inefficient, but the most common too. It is implemented in general-purpose language and therefore it does not require working with new tool. Also developers feel freedom in defining a policy of mixing partial configuration into complete one. This approach is used in many frameworks (e.g. Java EE [11], Microsoft Enterprise Library [10], GCore [12]), applications (Apache Tomcat [2]), games (Fallout 2 [6]), and other software systems.

### 2.2. Source transformations

More elegant way to allow user usage of multiple configuration languages are source transformations. Instead of implementing processing for each configuration language, there is implemented processing of only one language. For the other languages there are provided compilers that translate configuration sources in unsupported languages to supported one. The concept is introduced in Fig. 2.

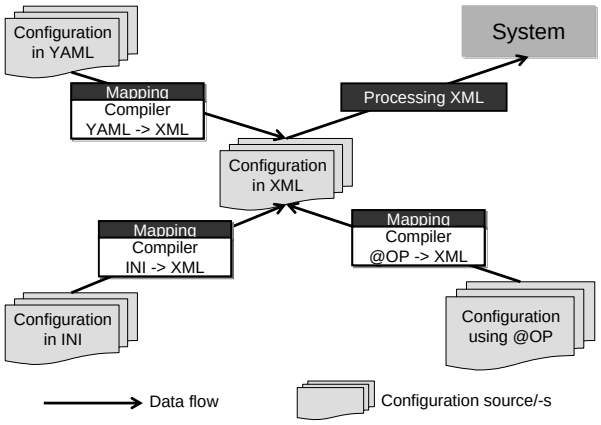


**Fig. 2** Example of source transformations with four configuration languages

This way, provider needs to take care only of processing one language and of describing translation of other languages to supported language. Description of translation is represented as mapping of one configuration language to another. Generally, translation description of language is shorter and therefore cheaper than implementing its processing. What's more, usage of compilers makes configuration processing code simpler (it processes only one language) and easier to maintain (as processing of each language is clearly separated from others).

The drawback of this approach is absence of support for combining (mixing) of configurations in multiple languages. Usually compilers just translate sentence from input language to output. And even with compilers capable of this functionality, the provider needs to deal with the synchronization of the translations to ensure proper priority of configuration languages.

### 2.3. Common abstraction of configuration sources

The solution suggested by this paper is common abstraction of configuration sources. Abstraction apparatus is a tool that processes the sources in multiple languages and by combining them generates complete model of configuration in output language as a virtual source in memory. This virtual source is handed over to system for processing. Fig. 3 presents this solution. And if one of the input languages is same as the output language, thanks to its trivial mapping is the complexity of mapping description the same as in case of source transformations.

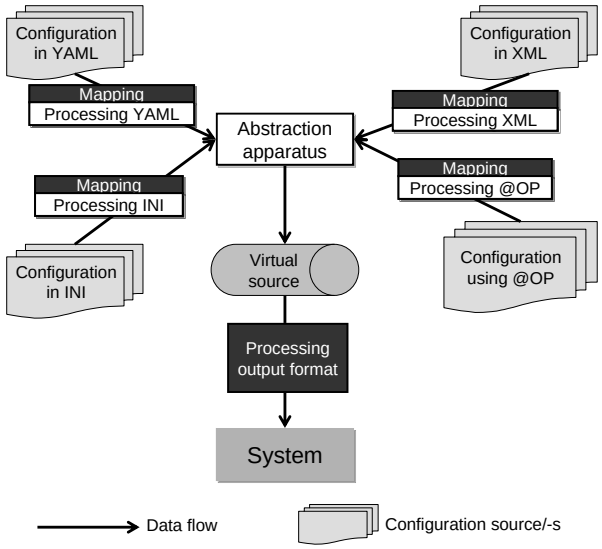


**Fig. 3** Example of common abstraction of configuration sources

As examples of this kind of tools can be mentioned Zend Config [14] and Apache Commons Configuration [1]. These provide abstraction of configuration from different formats (e.g. INI, XML, .properties). However, both of these lack effective means to choose freely abstracted languages of supported formats. While Zend Config requires strictly default mapping between selected input formats (for instance, for one XML language there is only one supported language in INI files), Commons Configuration allows modification of default mapping in procedural way that appears to be too exhausting for practical purposes.

## 3. DESIGN OF THE TOOL

This paper introduces a new design of abstraction tool that reduces drawback of its usage. The need to learn to use new tool, limitations in mapping of configuration languages (only one default mapping between different formats in Zend Config) or challenging means of changing default mapping between languages (Commons Configuration) could be identified as the most significant drawbacks of usage of given tool. The proposed design targets difficulties in definition of mapping of languages.

### 3.1. Metamodel

We suggest enabling change of default mapping between configuration languages that would lead to greater freedom in used languages and therefore raise the potential of satisfying individualities of tool's users. On the other hand, we find procedural representation of mapping (or changes in default mapping) and translation too demanding and therefore we suggest utilizing declarative representation. As this concept represents a model of a model of a configuration, we call it metamodel. Metamodel describes how to create models of configuration in internal format from all input configuration languages and how to combine them into complete configuration. This way metamodel also defines abstract syntax of configuration. Metamodel should be carefully designed to be as general as possible while remaining simple brief (in order to keep benefits of short representation of languages' mappings).

### 3.2. Conceptual design

Fig. 4 shows a conceptual design of tool based on analysis of existing tools and considering utilization of metamodel. The tool is composed of these modules:

- **Metaconfiguration reader**'s task is to process metaconfiguration - configuration of the tool. Metaconfiguration contains all necessary information for tool's operation. This module represents interface for system provider to define required metamodel and other system-specific settings.

- **Metamodel generator** uses information mediated by metaconfiguration reader to build up in-memory representation of metamodel.

- **Configuration sources locator** takes care of locating and preparing sources for further processing. Purpose of designing this module is to separate preparation of sources from translation.

- When everything is prepared, configurations from multiple sources are translated into internal format using metamodel as a guide. This format unification is performed by **configuration formats unifying unit**. Result of this process is set of configuration models in internal format. Each of the models represents whole or partial configuration of the system.

- Following combining policy defined in metamodel (created by *metamodel generator* using metaconfiguration) the **combining unit** combines all models into one representing complete configuration. Result of this combination process can be one of the models created by *configuration formats unifying unit*, if this model represents complete configuration (it is not missing any required information) and source format of the model has highest priority (information provided by this model is the most important).

- At the end of the process comes to play **output module**. Its purpose is to return unified model in requested format (defined in metaconfiguration) to user of the tool. If requested format is not the same as internal, it has to perform additional translation.

This modular approach allows easier debugging and maintenance.

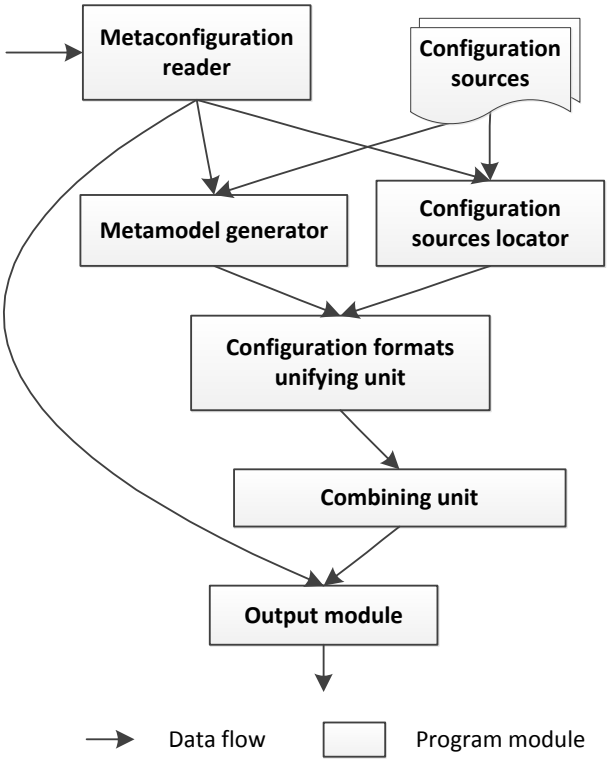


**Fig. 4** Design of the tool

## 4. EXPERIMENTS

To provide practical support for arguments stated in this paper, we implemented experimental tool and carried out few experiments.

### 4.1. *Bridge To Equalia*

*Bridge To Equalia* (BTE) is a proof-of-concept implementation of the abstraction tool. BTE abstracts configuration through Java annotations (implementation of @OP) and XML documents, two most commonly used configuration formats on Java platform. BTE utilizes concept of metamodel to make it easier to customize the tool for individual purposes. Internal format is represented by custom Java classes, and so is metamodel. Default metamodel is created for annotation types defining language in annotations. This metamodel can be altered to customize tool's behavior.

We performed a few experiments to test tool's flexibility and usability. As first, *metaconfiguration reader* module was implemented using the tool itself. This allows user to define metaconfiguration of BTE using both XML documents and Java annotations and it also shows usability of the tool. To test flexibility, tool was used to abstract configuration of several Java EE technologies that use both XML and Java annotations. Tests were performed for parts of Servlet, Java Server Faces and Java Persistence API configuration and all were successful.

At last, we realized an experiment to compare direct processing of configuration to BTE-mediated. Its purpose was to show benefits of using the tool and to confirm assumptions about using abstraction tool. Not only the code was simpler (code processing configuration had to deal only with XML instead of XML and annotations) but it was shorter too (Fig. 5). As the analysis showed, the benefit would be even more notable in less trivial case than used example.

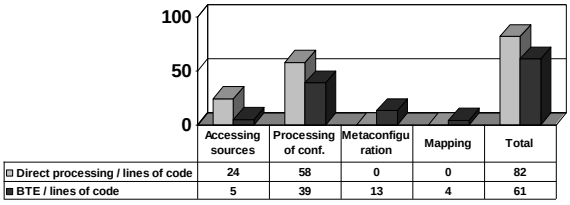


**Fig. 5** Comparison of direct and BTE-mediated configuration processing

So far, BTE can be considered the most flexible and general tool abstracting XML documents and Java annotations (to be honest, there are no such commercial tools, and BTE is successor of two of our older projects).

### 4.2. Conclusions of experiments

Experiments led to following conclusions:

1. Significance of the benefits of using abstraction tool rise with increasing number of supported configuration languages. This is pretty natural conclusion, abstraction of each language saves some code length and therefore more languages means more saved code (in comparison with ad-hoc solution). And there are other benefits than merely a shorter code - the code is also simplier and easier to maintain.

2. Let's suppose that we have given number of input configuration languages. Larger abstract syntax of the configuration induces more complicated processing (one has to process more semantically different information) and therefore more code for each supported language. This results in saving even more code, because with the tool system provider needs to implement only processing of the output format.

3. Supposing we have given number of configuration languages, greater distance of language mappings from default induce metaconfiguration growth. This is due to bigger effort needed in describing changes in default mapping. Of course, larger metaconfiguration means less efficient usage of the tool. Therefore it is very important for tool's author to find most common mapping between formats supported by tool and to use it as default (to ensure that in most cases the tool would not need an excessive language mapping definition).

4. Metamodel allows easier and faster changes to default tool's usage (in comparison with procedural expression used for instance in Apache Common Configuration).

To sum it up for practical purposes, the tool is not the best choice in every case. Instead, the usage of abstraction tool is suggested in case of more supported configuration languages and configuration with larger abstract syntax, while languages are mapped to output format (or internal, depending on metaconfiguration policy) using default mapping or one close to default. On the other hand, with simple abstract syntax of configuration but complicated mapping of languages to output format, the tool's usage is not recommended and ad-hoc solution might be better choice (but one should consider possible benefits of the tool in case of future extension of system and its configuration).

## 5. CURRENT STATUS AND FUTURE PERSPECTIVES

Primary purpose of BTE implementation was to test the concepts of source abstraction and declarative approach to language mapping. Although the tool was tested on aforementioned experiments, it would be necessary to put it through complex testing and debugging process before using it in "real-life" projects.

As next step in this area we see possible movement of metamodel definition from instance variables of metamodel classes to custom annotations. This way, metamodel would not consist of our Java classes, but definition of metamodel would be totally upon the tool's user. Instead of defining configuration language (as it is currently in BTE, user defines annotation-based language and its mapping on XML), user would write directly abstract syntax of configuration languages in form of custom classes (tree of their objects would represent an abstract syntax tree). In other words, user would define, what is configurable in his system without concerning about concrete syntax of configuration languages. Concrete syntax with combining policies would be added later, for example in form of Java annotations.

## 6. CONCLUSION

This paper concerns about enabling configuration from multiple sources. Its main purpose is to show and explain importance of common abstraction of configuration from multiple sources in comparison with other approaches. Paper presents conceptual design of abstraction tool with emphasis put on declarative way of defining mapping of abstracted configuration languages to output language. The effect of this approach is tested by experiments performed with proof-of-concept implementation of the tool for abstracting Java annotations and XML documents.

## ACKNOWLEDGEMENT

This work is the result of the project implementation: Development of the Center of Information and Communication Technologies for Knowledge Systems (ITMS project code: 26220120030) supported by the Research & Development Operational Program funded by the ERDF.

## REFERENCES

[1] Apache Software Foundation: *Apache Commons Configuration*. `http://commons.apache.org/configuration/`, available on 7.7.2011.

[2] Apache Software Foundation: *Apache Tomcat*. `http://tomcat.apache.org/`, available on 7.7.2011.

[3] BENNETT, K. – LAYZELL, P. – BUDGEN, D. – BRERETON, P. – MACAULAY, L. – MUNRO, M.: *Service-Based Software: The Future for Flexible Software*. In: Proceedings of Software Engineering Conference, 2000, APSEC, pp. 214–221.

[4] GROSS, P. HB – GINZBERG, M. J.: *Barriers To The Adoption Of Application Software Packages*. In: Systems, Objectives, Solutions, Vol. 4, No. 4, 1984, pp. 211–226.

[5] HUI, B. – LIASKOS, S. – MYLOPOULOS, J.: *Requirements Analysis for Customizable Software Goals-Skills-Preferences Framework*. In: Proceedings of 11th IEEE International Requirements Engineering Conference, 2003, p. 117.

[6] Interplay: *Fallout, Fallout 2, Fallout Tactics*. `http://www.interplay.com/games/fallout.php`, available on 7.7.2011.

[7] LUCAS, H. C. Jr. – WALTON, E. J. – GINZBERG, M. J.: *Implementing packaged software*. In: Management Information Systems Quarterly, Vol. 12, No. 4, December 1988, pp. 537–549.

[8] MACKAY, W. E.: *Triggers and barriers to customizing software*. In: Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology, April 27–May 02, 1991, New Orleans, Louisiana, United States, pp. 153–160.

[9] MERNIK, M., HEERING, J., SLOANE, A. M.: *When and how to develop domain-specific languages*. In ACM Computing Surveys (CSUR), Vol. 37, No. 4, December 2005, pp. 316–344.

[10] Microsoft: *Microsoft Enterprise Library*. `http://msdn.microsoft.com/en-us/library/ff648951.aspx`, available on 7.7.2011.

[11] Oracle Corporation: *JSR 316: Java Platform, Enterprise Edition 6 (Java EE 6) Specification*. `http://jcp.org/en/jsr/detail?id=316`, available on 7.7.2011.

[12] PASSOS, E. B. – SOUSA, J. W. S., CLUA, E. W. G. – MONTENEGRO, A.– MURTA, L.: *Smart composition of game objects using dependency injection*. In: Computers in Entertainment (CIE) - SPECIAL ISSUE: Games, Vol. 7, No. 4, December 2009.

[13] ROUVOY, R. – MERLE, P.: *Leveraging Component-Oriented Programming with Attribute-Oriented Programming*. In: Proccedings of WCOP 2006, Nantes, France, July 2006.

[14] Zend Technologies Ltd.: *Progammers Reference Guide: Zend_Config*. `http://framework.zend.com/manual/en/zend.config.html`, available on 7.7.2011.

Received October 13, 2011, accepted November 2, 2011

## BIOGRAPHIES

**Jaroslav Porubän** is Associate professor at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in Computer Science in 2000 and his PhD. in Computer Science in 2004. Since 2003 he is the member of the Department of Computers and Informatics at Technical University of Košice. He was involved in the research of profiling tools for process functional programming language. Currently the main subject of his research is the computer language engineering concentrating on design and implementation of domain-specific languages and computer language composition and evolution.

**Milan Nosáľ** is PhD. student at the Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his MSc. in Informatics in 2011. Currently his research focuses on attribute-oriented programming and its roles and usage options in programming and metaprogramming.