Cent. Eur. J. Comp. Sci. • 2(3) • 2012 • 283-299
DOI: 10.2478/s13537-012-0015-7

VERSITA

Central European Journal of **Computer Science**

Supporting multiple configuration sources using abstraction

Research Article

Milan Nosáľ*, Jaroslav Porubän†

*Department of Computers and Informatics,
Faculty of Electrical Engineering and Informatics,
Technical University of Košice,
Letná 9, 042 00 Košice, Slovakia*

Received 31 January 2012; accepted 17 August 2012

Abstract: Software engineers have long recognized the need to shift focus from developing systems to developing system families. One way to develop software family is to develop configurable systems. A configuration (initial settings of a program), written in application-specific language, can be expressed using many different formats, such as XML, YAML, attribute-oriented programming, etc., each one having pros and cons. Often the target group of users is too wide to meet their expectations by using only one format. This paper analyzes options that system providers have in supporting multiple configuration languages or sources. An enhanced abstraction tool is chosen as the best solution, and its architecture is briefly presented. The main contribution to the tool's design is advocation of the declarative representation of mapping of input languages to output format.

Keywords: configuration • multiple sources • configuration language • declarative approach to language mapping

1. Introduction

In the 1980's software engineers started to move from developing software systems to developing system *families* [3, 5]. Configuration became the first tool in this movement. In this paper we use the term *configuration* to mean the initial settings of a computer program, written in an application-specific language. This application-specific language (*configuration language*) is interpreted in deployment time. Using configuration in software systems allows configurable system to be modeled, customized or personalized to conform to specific requirements of customer or to be adapted to special circumstances or environment [2].
On the side of the system's provider, utilizing configuration brings advantages in a form of satisfying more clients, strengthening competitive advantages, providing more flexibility, robustness, quality and transparency in system. A system's ability to evolve is crucial to the user of a system, and therefore to its provider, too (e.g., his income may depends on his user's satisfaction) [1].

* *E-mail: milan.nosal@tuke.sk (Corresponding author)*
† *E-mail: jaroslav.poruban@tuke.sk*

On the other side stands the user, who wants to customize the system to meet her or his goals, preferences, abilities and skills the best [4]. This way a configurable system can be generic enough (the system is customizable) but still satisfying users' individuality and raising effectivity of their work (the system can be customized in a way the user wants) [6]. Also, using a configurable system instead of a custom one means spending less money, because customizing a ready configurable system is usually cheaper than developing a new custom one (and also the evolution of such a system is easier to manage) [5].
As this speculation suggests, configuration is an important part of software development. This paper concerns possibilities in supporting multiple heterogenous configuration languages to increase a user's satisfaction and this way possibly widen the target group of users.
The paper has following structure. Section 2 motivates our work. Section 3 analyses possible solutions to the problem with regard to existing work and advocates a common abstraction of configuration sources as the best solution. Section 4 introduces and advocates utilization of declarative definition of the language mapping. Section 5 presents designed architecture of the abstraction tool dealing with the issue presented in the paper. Section 6 introduces experimental implementation of the tool and states observed conclusions. Section 7 describes current state and future perspectives in presented work. The paper concludes with the summarization of the work.

2. Motivation

A *configuration structure* is a set of configurations. This set defines what can be configured in a system family. A configuration is a concrete member of a software system family. When implementing a configurable system, its author has to design suitable configuration language for expressing concrete configurations of the system (we consider configuration language an application-specific language interpreted in deployment time). A configuration as a concrete instance of a configurable system is expressed as a sentence in a custom domain-specific language [7] (DSL), called configuration language. In the same way the language matches the configuration structure, defining a set of all sentences that express possible configurations.
In the design of the language a provider needs to consider many barriers and problems with adapting configurable systems that are connected to the used format of the configuration language (for instance, ones presented in [3, 6]). There are not merely technical aspects to consider. Each user prefers a configuration language that best meets her or his needs and taste.
During the design process of the suitable configuration language, there are considered aspects as a simplicity of the language, its verbosity and sententiousness, complexity of a configuration process, domain abstraction, etc. [3, 4]. But user's preferences are based on subjective motives as well as on objective. It is easier to learn a new XML configuration language than an annotation-based one, when you are familiar with XML but not with attribute-oriented programming (abbreviated @OP, it became very popular format for configuration languages [8, 11]). The wider the range of users, the more conflicts in requirements on a configuration language may occur.
To encourage the user to utilize configuration, it is best to give him or her the option of expressing a concrete configuration in a language with which the user is most familiar. The most straightforward way to deal with the problem is to choose one of the available formats (XML, YAML, INI, @OP, etc.) and to design a configuration language that best suits the needs of potential users. But usually there is no format that would meet all important requirements and its drawback would be negligible.

2.1. Sample comparison of qualities of two configuration formats

As an example we can consider Java annotations and XML documents. In Fig. 1 there is listed a snippet of an annotation-based configuration of Java Persistence API. The equivalent partial configuration in XML is listed in Fig. 2. We can see that the annotations are less verbose. XML needs both opening and closing tags and it duplicates names of source code elements (classes, methods, etc.). Annotations are easier to write as they are located directly in the source code. This allows to configure simultaneously with programming. Configuration in annotations is tangled with sources, so it can not be "lost" so easily (comparing to accidental deletion of a configuration file). On the other hand, XML centralizes configuration — the whole configuration can be read much easier than by inspecting source code. Another strong advantage of XML documents is that changing a configuration file does not require a recompilation of the program (unlike changing annotations).

```
package pckg;

@Entity(name = "Person")
@Table(name = "PERSON")
public class Person {

   @Id
   @Column(name = "ID", length="255")
   private String id;
   ..
```

Figure 1. Snippet of annotation-based configuration of JPA.

```
<entity class="pckg.Person" name="Person">
   <table name="PERSON"/>
   <attributes>
      <id name="id">
         <column name="ID" length="255"/>
      </id>
      ...
</entity>
```

Figure 2. Equivalent snippet of XML-based configuration of JPA.

In Figures 1 and 2 we can find a one-to-one matching between an XML-based and an annotation-based configuration language. Annotation `@Entity` is mapped to an XML element `entity` with attribute `class` referring to the anotated class. Property `name` of the `@Entity` annotation is mapped to the `name` attribute of the `entity` element. In the same way the `@Table` annotation is mapped to the `table` element, that is a child to the `entity` element for corresponding annotated class. In the same manner as in case of the `@Entity` annotation we can see mapping of the `name` property to the `name` attribute of the `table` element.
The `entity` element has the `attributes` element that contains all attributes of the entity. In our example we can see that class `Person` has a field `id` annotated with annotations `@Id` and `@Column`. The `@Id` annotation is mapped to the `id` element in XML, its `name` attribute is referring to the annotated field – "id" is the name of the field. The `@Column` annotation is mapped to the `column` element with corresponding attributes expressing the values of its properties.
The same mapping could be found in languages' definitions — XML Schema for XML language and annotation types for annotations. One can find even matches between default values for many XML attributes and annotation properties, like the `length` with the default value 255 in the case of the JPA configuration.
For the sake of our simple example both XML and annotations code snippets described the same configuration. But it is important to state that we want to address pieces of information in multiple configurations that are not the same (or to be more precise, are not necessarily the same). We usually want data from one configuration to override the data from the others, but the pieces of information that are not stated in the first one to be supplemented from the others. To model the situation in our example, if the XML language was prioritized but did not state the length of "id" column, we would want to get that information from the secondary configuration in annotations.
In general, we might want to take two totally different configuration languages (meaning their semantics do not intersect) and want them to be abstracted. In this case the output from abstraction would be a model that would carry information from both configurations.

2.2. Suitability of configuration language for given configuration structure

There are also other aspects that complicate the decision for one configuration language. A configuration needed for localization of an application can be as trivial as listed in Fig. 3 (using .properties files format). This configuration is not dependent on source code elements, and therefore it is not natural to define it through @OP. And imagine how cumbersome would be designing XML language for it (does not Fig. 3 feel more natural than Fig. 4?). Configuration structure is sometimes so simple, that using XML is unnecessary complicated (of course, when hierarchy is needed, .properties are insufficient).

```
locale = en-gb
numberPrecision = 2
```

Figure 3. .properties localization configuration.

```
<local>
   <locale>
      en-gb
   </locale>
   <numberPrecision>
      2
   </numberPrecision>
</local>
```

Figure 4. XML-based localization configuration.

2.3. Summary

We came to the conclusion, that differences between available formats demand usage of multiple configuration languages. But implementing processing of more configuration languages requires more resources. The code that process configuration becomes larger, and its maintenance harder and error-prone (due to mixed processing of multiple configuration languages). There is also a negative impact on evolution of used configuration languages, because a change of languages requires changes in processing code.
The situation can be summarized in the following two statements:

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

This reasoning brings up a question: **How to support multiple configuration languages without a significant raise of costs and decrease of processing code simplicity?**

3. Analysis

Let's suppose the system supports multiple configuration languages. Then it should be able to process configuration in any of supported languages. In other words, if the system supports for example INI files and XML documents, user must be able to freely choose between these two formats. A second, stronger requirement is the option of randomly mixing configuration languages. The complete configuration is expressed as composition of partial configurations expressed in different languages.
These composition means that there might be two configurations, both in different languages, that would not even overlap, but joined together they must form complete configuration of a system. This means that there has to be some way to

define a model of these languages that can be used to model any sentence in these languages and even to model the union/intersection of these sentences. We will come back to this idea later later in section 4.1.
So far, we can identify three solutions: Ad-hoc solutions, source transformations and common abstraction of configuration sources.

3.1. Ad-hoc solutions

As the title suggests, a provider needs to implement processing of all desired configuration languages. No dedicated tool or framework is used. The concept of this solution is outlined in Fig. 5. Adding a new supported configuration language is basically implementing a whole operation of processing configurations in a given language (the 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 the existing configuration interface (processing code for other languages) makes the implementation even more difficult. The code processing one language may interleave with the code for other languages, what results in worse maintenance implementation.

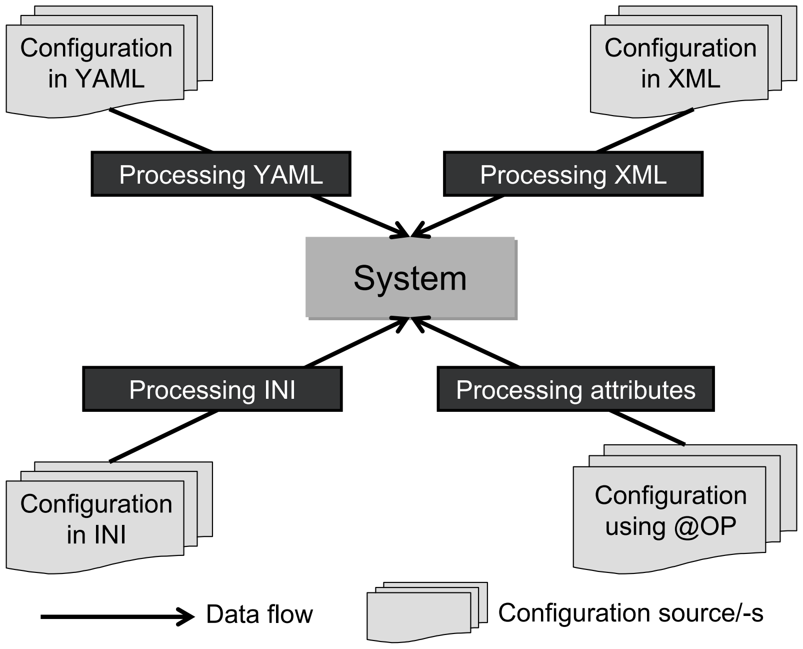


Figure 5. Example of Ad-hoc solution with four configuration languages.

A clear advantage of this approach is having the ability to optimize code for performance.

3.1.1. Related work

This approach is the most common, but it is also the most inefficient. As it is implemented in a general-purpose language, it does not require working with new tool and developers are free to defining their own custom policy of mixing partial configuration into complete one. This approach can be recognized in many frameworks (e.g. Java Enterprise Edition, Microsoft Enterprise Library, GCore [9]), applications (Apache Tomcat), games (Fallout 2), and so on.

3.2. Source transformations

More elegant approach are source transformations. Instead of implementing processing for each configuration language, one processes merely one language. For the other languages compilers are provided (that can be considered dedicated tools for translation) that translate configuration sources in unsupported languages to the supported one. The concept is outlined in Fig. 6.

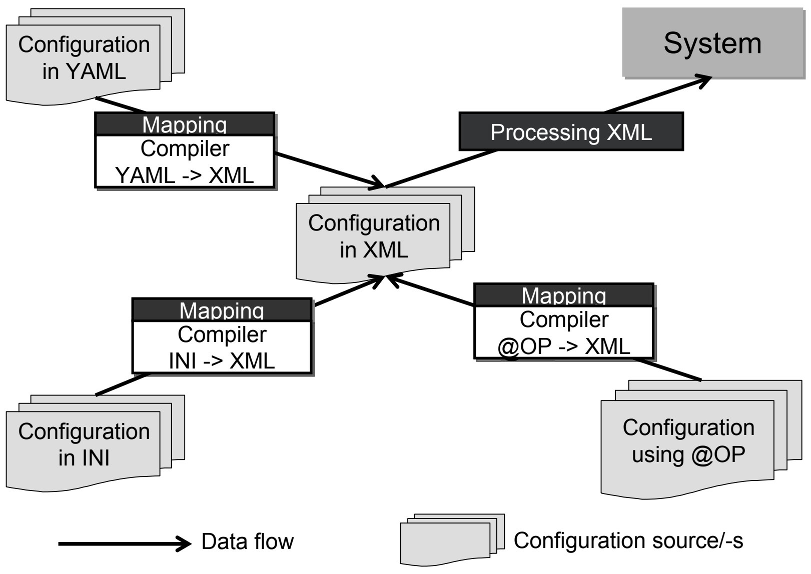


Figure 6. Example of source transformations with four configuration languages.

According to this approach there is a manual implementation for one language; processing of other languages is transformed into translations between them and the supported language. A description of translation is represented as a mapping of one configuration language to another. Generally, translation description of a language is shorter and therefore cheaper than implementing its processing. What's more, usage of the 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).

There is also a greater flexibility in this solution. It is possible for the user to write in an arbitrary configuration language, so long as he or she has a compiler for the language to translate it to the supported language (or other language that can be translated to the supported one, creating a chain of compilers).

The drawback of this approach is absence of support for combining (mixing) of configurations in multiple languages. Usually compilers just translate sentence from an input language to an output language. And even with compilers capable of this functionality, the provider needs to deal with synchronization of the translations to ensure proper priority of configuration languages. Fulfilling the stronger requirement for supporting multiple configuration languages is quite difficult. Moreover, as in the next approach to be discussed, the source transformation approach can result in bad performance, because solutions like this can hardly be optimized to every possible situation while remaining general. But this performance drawback is negligable because the settings are read and processed during deployment when better performance is not that critical.

3.2.1. Related work

This solution was used in our earlier project, A2X (Annotations to XML generator), but was found insufficient and later matured to the common abstraction of configuration sources. A2X was a project for the first author's Bachelor's thesis.

3.3. Common abstraction of configuration sources

This paper suggests common abstraction of configuration sources as the best solution to the problem. A dedicated abstraction tool is a program that processes the sources in multiple languages and by combining them generates a complete model of a configuration in an output language as a virtual configuration source in memory. This virtual source is used by a configurable system for customization. Fig. 7 presents this solution. The complexity of a mapping description between configuration languages and an output language can be decreased if one of the input languages is chosen as the output language (mapping between this input language and the output language is trivial as they are the same). This way resources (meaning the code needed to specify language mappings) needed to use this solution are the same as with the source transformations. For example, in case of four input languages there would be need to define three mappings (not four, because one of the input languages is also the output language and the approach does not need the mapping between them), just as in case of the source trasformations. The approach carries the same performance drawback as source transformations, but its significance lies in clear, comprehensible and maintainable code.

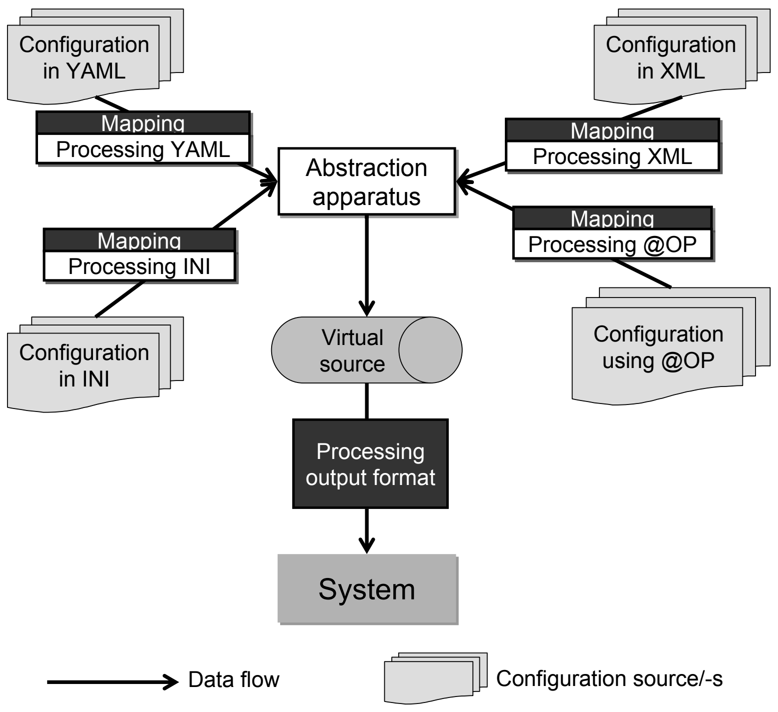


Figure 7. Example of common abstraction of configuration sources.

3.3.1. Related work

An experimental tool, Mixer (developed at our department as a Bachelor's thesis project), abstracts configurations in annotations and XML. But this tool does not allow any changes to the default mapping between XML and annotation-based configuration languages.
As commercial examples of this approach we mention Zend Config framework and Apache Commons Configuration. These examples abstract configuration in different formats (e.g. INI, XML, .properties). However, both of these lack effective means to freely define how the concrete syntaxes of the supported languages should look like. While Zend Config strictly requires 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 demanding for practical purposes.

4. Declarative definition of language mappings

The drawback of supporting multiple configuration languages lies in a larger and more tangled processing code. Using an abstraction tool (or the source transformation approach) allows one to decrease code length and to clearly separate processing of a complete configuration (processing of a virtual source) and processing of multiple languages (definition of

language mappings). The problem with existing tools is that they lack sufficient support for different mappings between languages in multiple formats. Zend Config supports only one default mapping between formats or in case of Apache Commons Configuration changing default mapping is too demanding. We believe that this demand is at least partially caused by procedural definition of mapping.
Our point can be elucidated by following figures. Fig. 8 shows some C#-like pseudocode that implements simple mapping between two languages. In the first prioritized configuration language there are tables represented by nodes with the "table" name. In the second language, the hierarchy is the same, merely name of tables' representation is changed to "Table". This figure shows how their combination may be implemented.

```
public ConfigurationNode combine(ConfigurationNode firstConf,
                                 ConfigurationNode secondConf){
        // other code for combination
        ..
        // For each child with name "Table"
        foreach(ConfigurationNode child in secondConf.getChildren("Table")) {
                // if there is not coressponding child in first configuration, add it
                // (determining whether the node is
                // corresponding may differ from case to case)
                if(!firstConf.containsSame(child)) {
                        child.setName("table");
                        firstConf.addChild(child);
                }
        }
}
```

Figure 8. Procedural definition of partial mapping between two languages in C#-like code.

Counterpart of the procedural mapping is its declarative representation. In Fig. 9 we can see a definition of the same mapping in some DSL for mapping definition. Both figures represent only incomplete mapping (we can think of it as a difference from default mapping), but they already show how can declarative representation be shorter and easier to comprehend.

```
map "Table" to "table"
```

Figure 9. Declarative definition of mapping between two languages in DSL.

4.1. Metamodel

In light of the comparison of declarative and procedural definition of mapping we propose using a metamodel of configurations. A metamodel of configurations is a model of configuration models. It defines a common configuration structure and mappings of the configurations with different concrete syntaxes to it. Model of a configuration can be presented as a sentence in some configuration language, and therefore we can think of metamodel as a model of the configuration languages. In other words, ametamodel defines abstract syntax of a configuration language, with mappings to its concrete syntaxes and policies of their combination. Sentence in the language is parsed into an abstract syntax graph, which is a model of the configuration.
"Common configuration structure" in this sense does not mean the intersection of the configuration languages. Instead we want a union of the languages so that we would be able to express everything that any of the configuration languages is able to express. The languages do not necessarily need to be able to express the same configuration – we want to use them concurrently so they could complement each other.
A metamodel of a configuration structure needs to be expressed somehow. We have to develop a declarative language that would be able to express a sentence describing a concrete metamodel.

Currently we use Java classes as a language model definition. At runtime these classes are supplemented with Java objects carrying information about mappings and combining policies. Currently we find this decision unfortunate, because dealing with this objects may be quite cumbersome, and it lowers manipulation of the metamodel to using procedural definition of mapping (we need to define how the objects pf Java classes are created). We implemented an annotation and an XML-based interface for definition and manipulation of these objects, but today we believe that using a native declarative approach would be a better idea (such an approach is presented in Subsection 7.1). Our current implementation is more elaborated in Subsection 6.1 under the paragraph Metamodel details.

5. Architecture of the tool

Fig. 10 shows a basic architecture of the tool we designed basing on an analysis of the existing tools and considering utilization of the metamodel. The tool should be composed of these modules:

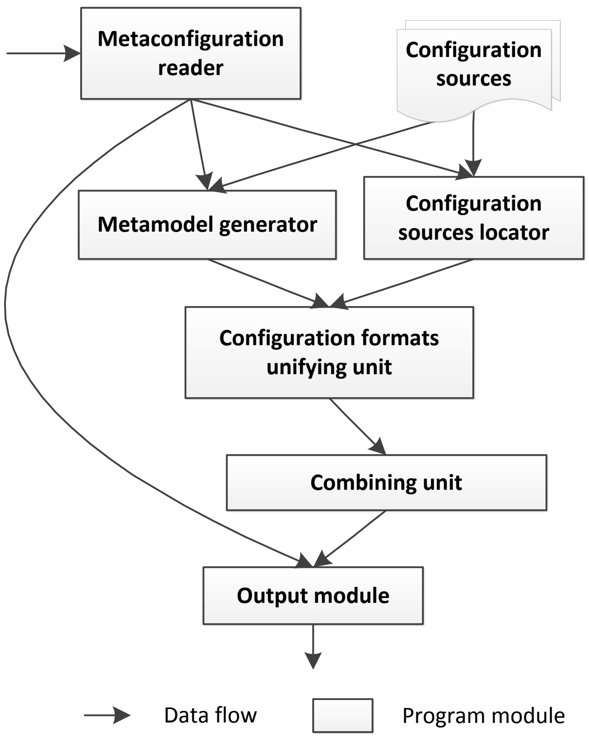


Figure 10. Architecture of the tool.

- The **metaconfiguration reader**'s task is to process metaconfiguration - a configuration of the tool. Metaconfiguration contains all necessary information for the tool's operating. This module represents an interface for the system's provider to define required metamodel and other system-specific settings such as configuration sources locations.
- The **metamodel generator** uses information mediated by the **metaconfiguration reader** to build up an in-memory representation of the metamodel.
- The **configuration sources locator** takes care of locating and preparing sources for their further processing. Purpose of this module is to separate locating and preparing of the sources from the translation process.

- When everything is prepared, the configurations from multiple sources are translated into an internal format using the metamodel as a guide. This format unification is performed by the **configuration formats unifying unit**. The result of this process is a set of configuration models in the internal format. Each of the models represents a whole or a partial configuration of the system. The **configuration formats unifying unit** is basically a set of compilers, making the abstraction tool an expansion of the source transformations solution.
- According to a combining policy defined in a metamodel (created by the *metamodel generator* using metaconfiguration) the **combining unit** combines all models into one that represents a complete configuration. Result of this combining can be one of the models created by *configuration formats unifying unit*, if source format of the model has the highest priority and the model represents the complete configuration (so no information has to be added from other source).
- In the end of the process comes to play the **output module**. Its purpose is to return a unified model in a requested format (defined in metaconfiguration) to the user of the tool. If the requested format is not identical with internal, the module has to perform an additional translation.

A modular approach brings better scalability of the tool. Upgrading the tool can be easily performed module by module. Adding support for new configuration format demands providing a compiler for this format and upgrading a metamodel to take the new format into consideration. This is an another scenario when a careful design of the metamodel can contribute, making this process smoother.

6. Experimental implementation

To provide a practical basis for the arguments stated in this paper, we implemented an experimental tool and carried out few experiments. This section presents this tool and conclusions from the experiments.

6.1. *Bridge To Equalia*

Bridge To Equalia (BTE) is our proof-of-concept implementation of the abstraction tool.[1] Its purpose is to provide abstraction of configuration sources for configuration languages based on Java annotations (implementation of @OP) and XML documents, the two most commonly used configuration formats on Java platform. A part of the BTE's design is the concept of metamodel to make it easier to customize the tool for specific requirements of tool's users. This metamodel idea came from experiences with the A2X implementation mentioned in Subsection 3.2, especially by cumbersome definition of mappings between XML and annotations.
As far as we know, BTE and Mixer 3.3 are the only implemented tools that abstract annotation and XML-based configuration sources. Mixer does not allow user to change the default mapping to concrete syntaxes of annotations and XML. Therefore we believe BTE is, today, the most flexible and general tool abstracting XML documents and Java annotations.

6.1.1. Related work

BTE is implemented in the Java programming language. The *metaconfiguration reader* module uses the implementation of BTE (a kind of a recursion) to read metaconfiguration written through Java annotations and/or XML documents. For easier and more effective access to annotations, tool's *configuration sources locator* uses the Scannotation tool[2]. XML documents are read using standard Java DOM parsers.
The structure of an internal format is defined through our Java classes. An object tree of their instances represents a model of a configuration. Structure of the metamodel is defined by Java classes too. The metamodel describes details of creation of a configuration model in internal format from configuration in Java annotations or XML documents (in

[1] *BTE was designed and developed during 2010 as a part of the first author's Dimploma thesis under supervision of assoc. prof. Porubä.*
[2] *http://scannotation.sourceforge.net*

other words, it describes concrete syntaxes). The default metamodel is created according to annotation types that define configuration language in annotations. In this sense usage of BTE is annotation-centric. The tool's user defines input configuration languages by designing annotation types of @OP-based configuration language and providing its mapping to an XML-based language. BTE provides a default mapping to XML, but it can be changed by altering the metamodel (through the metaconfiguration).

6.1.2. Metamodel details

To give some intuition about the current metamodel implemenation, we now present its basics. Fig. 11 shows a small excerpt from the code of Java class `Information` which is an implementation of a node of a configuration model. In some sense this is a part of the metamodel, defining abstract syntax of the configuration languages. We can see that a node carries information about configuration value (that might be null value, if the node has only sense in creating hierarchy), referres to source code elements that are target for the configuration information, carries reference to and additional metamodel information, and provides many other important data, such as references to parent and children of the node in configuration hierarchy.

```
// Name of the source code element that this configuration information is bound to
private String targetQualifiedName;

// Configuration value of this node
private final String value;

// Reference to metamodel object for this information
private final ConfigurationType MMConfiguration;
```

Figure 11. Excerpt from the class defining configuration model.

The `Information` nodes are enough to represent the abstract syntax of the most (if not all) of common XML or annotation-based configuration languages. Important role in the metamodel plays the `ConfigurationType` class representing additional details of the metamodel. Its main purpose is to define mapping of the configuration model to the configuration language — in the words of language engineers, to define concrete syntaxes from abstract syntax. Fig. 12 shows a part of the `ConfigurationType` implementation that defines mapping to XML. There is a name of the corresponding XML element/attribute, possible default value (that can be used to generate XML Schema definition for the XML language), and a tag that defines whether the information is extracted from an XML element or an XML attribute.

```
// The name of the XML element/attribute this information is mapped to
private String name;

// Default value
private String defaultValue;

// Enum value defining whether the information is mapped to the element or attribute
private XMLProcessing XMLOutputType;
```

Figure 12. Excerpt from the metamodel class concerning concrete syntax in XML.

In a similar manner as in the Fig. 13 there is an excerpt from the `ConfigurationType` implementation that concerns model mapping to source code (or we can say that defines concrete syntax for an annotation-based configuration language). There are pieces of information that say which configuration annotation types corresponding `Information` node is mapped to, whether the information is mapped to an annotation or its declared property, and so on. This information is necessary to generate a configuration model from the annotations.
`ConfigurationType` objects create a tree that defines abstract syntax and its concrete syntaxes in XML and annotations. This tree is created from configuration annotation types by the **Metamodel generator** module. Modifications to the default

```
// Class of configuration annotation that is source of the information
private final Class confAnnotation;

// Qualified name of the source of the information (annotation or its property)
private final String qualifiedNameOfSource;

// Enum type of the information source - annotation, its property or custom
// (there is option to define custom information extractor)
private final SourceType sourceType;
```

Figure 13. Excerpt from the metamodel class concerning concrete syntax in annotations.

metamodel can be carried out in memory on Java objects of `ConfigurationType` using some kind of a tree visitor, but as it was said in Subsection 4.1, this would lower the metamodel utilization to the procedural declaring of the language mappings. Therefore we implemented a visitor that considers metaconfiguration and modifies the metamodel according to it. This way a metamodel modifications on the default state can be stated declaratively as annotations or XML document (an example of such @OP based configuration will be introduced and discussed on the Fig. 14 in the next section).
A model is generated through the compilers we implemented. An XML compiler is basically a wrapper around a standard DOM model generator in Java. This wrapper translates the DOM model to our model using metamodel (tree of `ConfigurationType`) as a guide. The same way a model for annotations is generated, but the source are annotations that are obtained using annotation scanner. We implemented our own annotation scanner, based on the Scannotation tool, that provides an interface similar to the interface of Annotation Processing Tool[3] by Oracle (Sun) that works in compile time.
A default output format is the XML-based input configuration language. Thanks to the fact that XML is used as the output format, there is need to define only one mapping, the mapping of annotations types to XML. XML to XML mapping is trivial (if we provided only one XML document to the tool, output would be the same document).
Using JAXB technology, our output can be translated into a tree of Java objects. In this transformation, the default mapping by JAXB is used.

6.2. Experiments

To prove assumptions about three properties of the tool (usability, flexibility and effectivity), we performed various experiments.

6.2.1. Usability

Our implementation of the *metaconfiguration reader* module using the tool itself allows user to define metaconfiguration of BTE using both XML documents and Java annotations. This implementation provides few most common mapping patterns between annotations and XML. BTE's metamodel is an in-memory object tree (Subsection 6.1), and therefore it is somewhat inconvenient to alter it. In the current state there are pluggable processors that can work with metamodel. But it is easier with metaconfiguration through anotations or XML that is processed by one generic processor (again, processors as procedural definition versus annotations/XML as declarative definition). Fig. 14 shows how the annotations are used to express the metaconfiguration. Annotation type `Table` is mapped to an XML element `table` (by default its mapped to element with the same case-sensitive name, in this case Table). Its declared property `name` is mapped to an XML attribute with the same name (it is not changed via the `MapsTo` annotation).
This metaconfiguration in a form of annotations and/or XML documents is processed using BTE. The example shows just a little part of the metaconfiguration structure. As we were able to cover the whole metaconfiguration structure, we consider this implementation a proof of the usability of the tool.

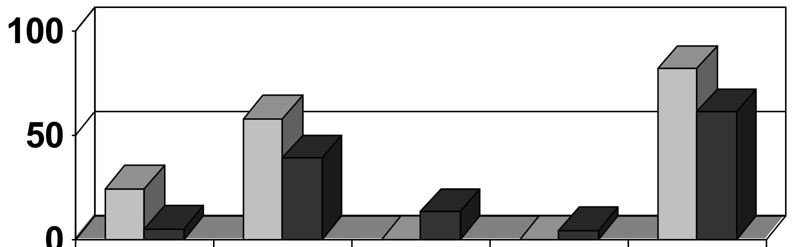
[3] *http://docs.oracle.com/javase/6/docs/technotes/guides/apt/index.html*

```
@MapsTo(name="table")
public @interface Table {
    @Attribute
    String name();
}
```

Figure 14. Sample metaconfiguration for mapping of annotation type Table to XML in BTE.

6.2.2. Flexibility and effectivity

To test flexibility, our tool was used to process a configuration of several Java EE technologies, that use both XML and Java annotations as configuration sources formats. Tests were performed for parts of the Servlet, the Java Server Faces and the Java Persistence API configuration and all were successful. Despite great differences from the default mapping between annotations and XML, the tool successed in fully abstracting both sources. We needed merely to alter default metamodel and did not have to change the tool. This flexibility was mainly caused by adapting a metamodel concept in the tool's design (there were changes that would go beyond the abilities of the XML and annotation-based metaconfiguration, but using processor on the in-memory metamodel was enough).
At last, we carried our an experiment to compare direct processing of configuration to BTE-mediated ones. The purpose of the experiment was to show a decrease of code length (measured in LOC - lines of code) and to confirm assumptions about usage of the abstraction tool. Not only was the code simpler (code that process configuration had to deal merely with XML instead of XML plus annotations) but it was shorter too (Fig. 15). As the analysis showed, the benefit would be even more notable in less trivial case than in our example.



	Accessing sources	Processing of conf.	Metaconfigu ration	Mapping	Total
▫ Direct processing / lines of code	24	58	0	0	82
■ BTE / lines of code	5	39	13	4	61

Figure 15. Comparison of direct and BTE-mediated configuration processing.

In the same experiment we implemented translation of the default models of both formats to common internal model in a procedural manner (as one would need using approach of Apache Commons Configuration). This imlementation was used to override the translation guided by the metamodel to analyze our assumption about declarative and procedural expression of languages' mappings. The translation implementation for this particular case has around 120 LOC (lines of code) and is much more complicated than 13 + 4 LOC (Fig. 15) of the annotation-based declarative definition of mappings. In fact, it is even larger and more complicated than ad-hoc solution (more complicated because it requires additional knowledge of our model implementation). We consider this an important fact proving our assumption about declarative definition of languages' mappings from Section 4.

6.3. Conclusions of experiments

The experiments led to the following conclusions:

1. The significance of the benefits of using the abstraction tool rises with the increasing number of supported configuration languages. This is a natural conclusion because abstraction of each language saves some code length and therefore more abstracted languages means more saved code and less tangled configuration processing.

2. Let's suppose that we have a given number of input configuration languages. A larger abstract syntax of the configuration requires more complicated processing (one has to process more semantically diverse information) and therefore more code for each supported language. But if the processing of languages is abstracted, this extra code is saved (the first conclusion). So larger abstract syntax of configuration (in other words, larger configuration structure) results in more code saving.

3. Supposing we have a given number of input 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 the less effective usage of the tool (there are resources saved on code, but they have to be used to define the metaconfiguration).

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

Third conclusion brings urge to tool's author for finding the most common mapping between formats supported by the tool and using it as default. By using the most common mapping as default there is a lower risk for users to need to change it.
To sum it up, for practical purposes using the abstraction tool instead of the ad-hoc solution 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 a default mapping or one close to the 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. But one has to take into consideration that it can make further upgrades and extensions easier. In short, using the tool should be carefully considered in design phase of software development, especially in development of small systems.

7. Current status and future perspectives

As we have mentioned in Section 6, BTE is a proof-of-concept implementation of the tool. Its primary purpose was to test the concepts of the source abstraction and the declarative approach to the 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. The experiments we performed were designed merely to confirm theses about importance of the common abstraction of configuration sources and the declarative approach to the language mapping.
The BTE tool need to be put through tests to confirm its real-world applicability. We performed a few tests to find and remove major bugs, but we cannot guarantee that BTE is bug-free. In practical applications a bug can cause a crash or an undefined behavior of the system.
Although the experiments were based mainly upon the LOC metrics, proving that using the tool decreases LOC needed to abstract multiple configuration sources, the main potential and strength of the tool lies in better scalability of the system. Extending configuration structure or adding more sources is much easier than with the ad-hoc solutions, because the implementation processes merely output format and so it is shorter and not tangling. Therefore we believe using this apporach in practice would be beneficial for system developers.
We also think that adapting redesigned metamodel implementation presented in next section would give BTE more more chances in real world. This new metamodel implementation would be easier to learn and to work with.

7.1. Future perspective

Currently, BTE uses a metamodel based on our Java classes. One concrete metamodel is a tree of objects in memory. This approach is sufficient, but makes life harder for the tool's user. When the user wants to use BTE, first he or she has to define an annotation-based language by defining annotation types. Then he/she has to define mapping, which can be easy if the defaults are sufficient, but can become quite hard when he/she has to alter directly metamodel in memory (he/she would have to study our implementation and understand our abstractions). And at last, output format is XML, so he/she would have to implement processing of the XML-based language (there is also an option to use JAXB object model). In this use case, user would have to deal with both annotations and XML, and moreover our own metamodel.
We want to experiment with a new metamodel representation. Instead of the Java objects we propose using custom annotated classes. A tree of Java classes (using an object-oriented composition relationship) would represent common abstract syntax of configuration languages (or probably better term would be the language model as in [10]). Details about its mapping to concrete languages would be expressed by annotations. An internal model and output format of configuration would be an object tree of these classes.

7.1.1. Vision in detail

Fig. 16 shows an example of this vision. There is a class representing a configuration entity Table. This entity has some properties, represented by fields of the `Table` class. One of the fields is list of `Column` objects. In this scenarion, Column would be another configuration entity (represented by `Column` class). This way the user can describe the whole configuration structure using merely Java classes.

```
// Metamodel class Table
@AnnotationType(type = "config.annotations.Table")
public class Table {

    @XmlAttribute
    @DeclaredProperty
    protected String name;

    @XmlElement
    @DeclaredProperty
    protected String table;

    @XmlElement
    @Annotations(relationship = Annotations.ON_FIELDS)
    protected List<Column> columns;

    // getters and setters
}
```

Figure 16. Vision of new BTE metamodel.

7.1.2. Mapping annotations

Annotations annotating the `Table` class and its fields are mapping annotations. `@XmlAttribute` and `@XmlElement` are annotations for concrete syntax of an XML configuration language. Annotations `@AnnotationType`, `@DeclaredProperty` and `@Annotations` define concrete syntax for an annotation-based language. If we suppose that the default mapping of a field would be to an XML element in XML and a declared property in annotations, this source code would be even shorter. We could omit annotations `@DeclaredProperty` and `@XmlElement`. If we had the default mapping set to the annotation type with the same name as is the class', we could omit annotation @AnnotationType.
We can also define a default relationship between annotations representing a class and annotations representing class' fields. In our example the configuration annotation `@Table` would be used on classes. Annotations of the `@Column` type would be annotating fields of a class annotated by `@Table`. This is just one of multiple possible relationships between annotations' usages (for more annotations' usage constraints see [12]).

In the end, we can get to the state where for many common configuration structures there will be no need to change the default mapping. Only advanced users would need to annotate the metamodel. This reasoning can be considered an example of how the default mapping should be worked out.

7.1.3. Benefit

Annotated metamodel classes would be used to generate annotation types for an annotation-based configuration language and an XML schema for an XML-based configuration language. This generative approach would help tool's user not to deal with the details of concrete syntaxes. If the default mapping would be fitting, the user would not need to get in touch with neither annotations nor XML. The user would be working only on abstract level of metamodel classes (output format would be Java objects of metamodel classes) – and these classes are his/her implementation. We believe this approach lowers the requirements on skills and resources of the users.

8. Conclusion

This paper concerns the configuration from multiple sources. Its main purpose is to show and explain importance of the common abstraction of configuration from multiple sources in comparison with other approaches. Paper presents new approach in designing the abstraction tool which lies in declarative way of defining mapping of abstracted configuration languages to an output language. This design targets issues with tools usage caused by extensive configuration of the tool (in cases, when mapping between abstracted configuration languages is too far from default mapping supported by the tool). The effect of this approach is proven by experiments performed with a proof-of-concept implementation of the tool for abstracting Java annotations and XML documents. Also, future perspectives are stated.

Acknowledgment

This work was supported by VEGA Grant No. 1/0305/11 - Co-evolution of the artifacts written in domain-specific languages driven by language evolution.

References

[1] Bennett K., Layzell P., Budgen D., Brereton P., Macaulay L., Munro M., Service-based software: the future for flexible software, In: Proceedings of the Seventh Asia-Pacific Software Engineering Conference, ser. APSEC '00. Washington, DC, USA: IEEE Computer Society, 214, 2000

[2] Dreiling A., Rosemann M., van der Aalst W., Heuser L., Schulz K., Model-based software configuration: patterns and languages, EJIS, 15, 583-600, 2006

[3] Gross P.H., Ginzberg M.J., Barriers to the adoption of application software packages, Information Systems Working Papers Series, 4, 211-226, 1984

[4] Hui B., Liaskos S., Mylopoulos J., Requirements analysis for customizable software goals-skills-preferences framework, In: Proceedings of the 11th IEEE International Conference on Requirements Engineering. Washington, DC, USA: IEEE Computer Society, 117, 2003

[5] Lucas H.C., Walton E.J., Ginzberg M.J., Implementing packaged software, Manage. Inf. Syst. Q., 12, 537-549, 1988

[6] Mackay W.E., Triggers and barriers to customizing software, In: Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology, ser. CHI '91. New York, NY, USA: ACM, 153-160, 1991

[7] Mernik M., Heering J., Sloane A.M., When and how to develop domain-specific languages, ACM Comput. Surv., 37, 316-344, 2005

[8] Newkirk J., Vorontsov A.A., How .NET's Custom Attributes Affect Design, IEEE Software, 19, 18-20, 2002

[9] Passos E.B., Sousa J.W.S., Clua E.W.G., Montenegro A., Murta L., Smart composition of game objects using dependency injection, Comput. Entertain., 7, 53:1-53:15, 2010

[10] Porubän J., Forgáč M., Běhálek M., Annotation based parser generator, Computer Science and Information Systems: Special Issue on Advances in Languages, Related Technologies and Applications, 7, 291-307, 2010
[11] Rouvoy R., Merle P., Leveraging Component-Oriented Programming with Attribute-Oriented Programming, In: 11th International ECOOP Workshop on Component-Oriented Programming (WCOP'06), 11, 10-18, 2006
[12] Ruska Š., Porubän J., Defining Annotation Constraints in Attribute Oriented Programming, AEI, 10, 89-93, 2010