BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2016
http://excel.fit.vutbr.cz

DSL for Configuration Files Verification

Matěj Mareček*

```
'''print("Demo template!");'''
[general]
{exactlyOnce} pvss_path = "C:/Siemens/" || "/opt/Siemens"
proj_path = ''(value.toLowerCase().startsWith("c:/projects/"))''
[ui]
showActiveShapes = 0
checkADAuthIntervall = >= 60
```

Abstract

CERN (*European Organization for Nuclear Research*) is one of the biggest research organizations in the world. It heavily uses SCADA (*Supervisory Control And Data Acquisition*) software for their scientific and industrial machines. This paper tackles a problem of verifying that configuration files used by CERN's SCADA (*WinCC Open Architecture*) software are correct and comply with CERN standards.

The aim was to develop a tool that gives SCADA developers and administrators ability to easily create templates that describe how certain configuration files should look like and determine whether templates match configuration files.

This paper introduces a tool that solves the problem by using a specially designed domain-specific programming language and an interpreter of the language.

The language itself is based on declarative paradigm and its fundamental capabilities can be extended by JavaScript injection. As for the interpreter, it uses Xtext-based parser to convert configuration files and templates into form of abstract syntax trees (ASTs). The execution itself is a combination of AST interpretation, translation of certain parts of AST into a JavaScript code and running JavaScript code on top of the Java Virtual Machine.

Keywords: Domain-Specific Languages — Templates — Configuration Verification — Xtext — Parsers — Interpreters — WinCC OA — SCADA — JavaScript — JVM — CERN

Supplementary Material: IDE Documentation — IDE Videos

*matej.marecek@outlook.com, *Faculty of Information Technology, Brno University of Technology & CERN*

1. Introduction

CERN (*European Organization for Nuclear Research*) is one of the biggest research organizations in the world. It uses many scientific, industrial and custom built machines that are supervised/operated using SCADA (*Supervisory Control And Data Acquisition*) software, which is mainly developed for WinCC Open Architecture platform. The platform uses specific C-like programming language for development and special configuration files responsible for setting up the SCADA applications.

In order to make development of SCADA applications at CERN easier and safer, Eindhoven University of Technology conducted a research and created a prototype of an Eclipse-based IDE, primarily focused on writing, analyzing and debugging the C-like code. While I was working at CERN, my task was to continue in this project to improve capabilities of the IDE and make it ready for real-world usage.

One of the requirements for the IDE was that it has to be able to verify correctness of configuration files for SCADA applications. This functionality is very important because SCADA applications at CERN are used among others for critical systems (accelera-

Figure 1. Usage of SCADA applications at CERN.

tors, cryogenics, electricity etc.), and using incorrect configuration files could have serious consequences. In addition, the configuration files are often extensive and complex. It means that it is time consuming and difficult to manually check whether a configuration is correct.

The purpose of this paper is to introduce a part of the IDE that is responsible for the verification of configuration files. In order to do so, we need to understand the requirements:

- Correct configurations should be described using some kind of **templates in text form**.
- The **templates should be familiar** to people who know configuration files and have basic knowledge of programming.
- The implemented tool should be **integrated into the IDE**.

After analyzing this demands, it soon became clear that there is no existing solution that could be easily reused, and the most direct way of tackling the problem is a specially designed programming language and its interpreter (in other words, the template is interpreted in order to verify correctness of a configuration file).

The basic idea here is that the core syntax of the new domain-specific language (hereinafter referred to as DSL) is similar to the syntax of configuration files. Therefore, the language has steep learning curve and users can copy-paste parts of configuration files directly into templates.

Technologies used for the language implementation are Eclipse, Xtext, Java and JavaScript. The usage of the Eclipse Platform is obvious choice since the tool should be part of the Eclipse-based IDE. Xtext is a very convenient framework that simplifies parser implementation and integration with Eclipse. Java and JavaScript are general-purpose programming languages used for the implementation itself.

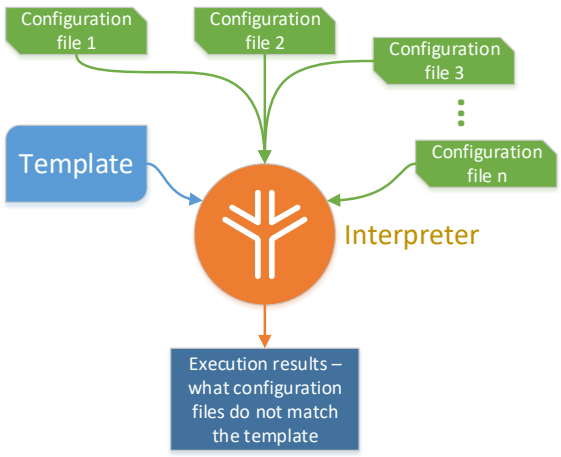

Figure 2. Basic principle of how configuration files are verified to be correct. The figure shows that the interpreter can compare one or many configuration files with a template at the same time and produces a report stating what configuration files do not match the template and where is the problem.

The proposed DSL and implemented software has proven to be working in real-world environment at CERN. It also demonstrates modern and progressive trends of using JavaScript as means of execution instead of bytecode and machine code.

2. Configuration files

In order to design a template language for configuration files, we need to understand the structure of configuration files. The basic syntax is shown in Figure 3 (*Extended Backus–Naur Form* of metasyntax notation is used) and a simple example of a configuration file is shown in Figure 4. For purposes of this paper, it is enough to know that the configuration files are very similar to INI files (although they are not exactly the same), and their content consists of 0..*n sections* and every *section* contains 0..*m properties*, each with 1..*k values*.

3. Template language design

Designing the language itself is the first step that needs to be done after requirements are gathered. It consists of syntax definition and semantics definition. Both (syntax and semantics) have to be unambiguous.

It is important to realize that language design can have a big impact on learning curve, interpreter performance, parser complexity (complex languages require complicated parsers and such parsers are usually slower), language extensibility etc. Therefore, the core syntax of the DSL was designed to be as simple as

```
ConfigModel:
  sections+=Section*
;

Section:
  '[' name=KEY ']'
  content+=Property*
;

Property:
  ('('param=(SUBSTITUTION|KEY)')')?
  name=KEY '=' values+=Value+
;

Value:
   NumberValue
  |StringValue
;
```

Figure 3. This code represents the basic syntax of the configuration file written in Xtext grammar language. Such language description is then used to generate a parser and a set of Java classes that represent the AST model.

```
[general]
proj_path = "C:\MyProject\scripts"
pmonPort = 5687
[dist]
distPeer = "PC1324" -3
```

Figure 4. An example of a simple configuration file. In this case, the content consists of two sections (*general* and *dist*). The first section has two properties. One with a *string* value and one with an *integer* value. The second section has only one property with two values (*string* and *integer*).

possible; to benefit from following advantages:

- Designing a language is usually an iterative process that requires feedback from users. Simple languages are easier to explain to users and can be changed rapidly.
- Simple languages are usually consistent and intuitive.
- Extending the syntax of a simple language is much easier than trying to extend complicated syntax.
- Parsing and interpretation of simple languages is relatively easy to implement and optimize.

```
[general]
proj_path = startsWith "C:\"
{never} pmonPort = < 1024
```

Figure 5. This example demonstrates the basic concept of the rules. The first rule starts on line 1. Its meaning is: search the configuration file, find all sections with name *general* and on each section apply the sub-rules (proj_path and pmonPort). This rule also has to succeed at least once and it can never fail. More formally, *(S>0 ∧ F=0, findSections("general", AST), {proj_path:Rule, pmonPort:Rule})* The last (sub)rule is probably most self-explanatory. It means that in currently tested *general* section, there can never be *pmonPort* property with an integer value less than 1024. Formally, *(S=0 ∧ F ≥ 0, findProperties("pmonPort", AST-general-subtree), {value < 1024})*.

3.1 Basic concepts of the DSL

The core of the language is built around several concepts, and each concept serves the purpose of the language.

[Everything is a rule] In the language everything is a rule. There are several types of rules. Each type serves different purpose so they differ by their syntax and context of usage. Despite the differences, all rules can be defined as follows:

A rule is a triple (λ, φ, R) where:

- λ is a predicate that returns *true/false*, depending on how many times the elements of R succeeded (S) or failed (F) in φ. This predicate decides whether the entire rule succeeded or failed.
- φ is a function that takes as its input the AST of a configuration file (or its subtree) and returns a set of subtrees of the given AST.
- R is a set of sub-rules/code that is evaluated in the context given by φ.

An example of how rules can look like when implemented in a template, is shown in Figure 5. There is also a possibility to use more advanced syntax for definition of the rules. This syntax then allows programmers to customize the rules and use it in special cases (by default it is recommended to stick with the basic language constructs).

[Declarative paradigm] The language is based on declarative paradigm. In common cases, programmers declare only rules and do not care about sequences of rules execution etc. This makes possible (in an ideal situation) to reuse the same rules among multiple templates without having to worry about affecting other

```
#following basic constructs
[general]
{never} pmonPort = < 1024

#alternative equivalent version
   using JS injection
[''(sectionName == "general")'']
{''(passed == 0 && failed >= 0)''}
   pmonPort = ''(value < 1024)''
```

Figure 6. This example demonstrates the ability of the language to replace most of its high-level constructs with JavaScript code. Notice that variables used in JavaScript code are context-dependent. For instance, if we want to replace the part responsible for success/fail counting ({*never*}), we can access the information of how many times the rule passed/failed via values stored in variables *passed* and *failed*, provided by the language interpreter at runtime.

rules.

[JavaScript integration] To support the variety of different features requested by the users, the language supports JavaScript code injection. This means that all basic language constructs can be replaced by JavaScript code. An example demonstrating this usage of JavaScript can be seen in Figure 6.

[Separated scopes] Every type of the rule is responsible for providing a scope for its children rules. The root of the rules-tree provides a shared scope for all of its child-rules, and child-rules should inherit this scope, optionally add some variables to it and again pass it to their child-rules (if they have any).

Theoretically, rules can use their parent-scope to run JS code (so they can change their parent-scope). Nevertheless, in reality, all types of rules (except global rules) run their JS code in a local scope inherited from the global scope that is shared among global rules (we can think about it as accessing the parent-scope in "read-only mode" so it cannot be changed).

A simple example: Let us have a global rule that defines variable *a* (*var a = 5;*). Then all rules executed after this global rule can access variable *a*. Another global rules can change its value (*a = "different value";*), the rest of the rules can only read it (*print(a);*) or shadow it, without affecting the original value (*var a = true; // this will not change the value of variable 'a' in the global scope*).

The reason why the rules have separated scopes is their portability. In most cases, we can simply take a rule from template *A* and put it into template *B*, and we can be sure that the original rules in template *B* are not affected by the newly added rule.

```
terminal SCRIPT_CODE:
   "'''" -> "'''"
;

terminal ALGORITHM_CODE:
   "''{" -> "}''"
;

terminal EXPRESSION_CODE:
   "''(" -> ")''"
;
```

Figure 7. Terminals responsible for JavaScript handling. Their semantic is that everything between the three introducing and closing symbols is a JavaScript code: `'''JS'''`, `''{JS}''`, `''(JS)''`.

4. Template language parser

When we have a properly designed language, or at least a language concept, we can implement a parser.

In case of the template language, there are two different parsers. The first parses the high-level language constructs (see Section 4.1) and the second takes care of pieces of injected JavaScript code (see Section 4.2).

4.1 Xtext-based parser

The high-level parser is Xtext-based. It means that a grammar of the template language is written in Xtext grammar language and then automatically translated into a Java parser and AST model (EMF model).

The language grammar is around 260 lines of code long and not responsible for JavaScript code parsing. Instead, on places where JavaScript syntax is expected, it uses so called *until token* to consume everything until a certain token occurs. The grammar terminals responsible for JavaScript code handling are shown in Figure 7.

4.2 Nashorn engine parser

The Xtext-based parser is able to create an AST out of a template code and thanks to the terminals presented in Figure 7, it can store JavaScript code as plain-text (*string* value) in the AST.

JavaScript code stored as part of the template AST is not parsed immediately, but rather at the time of template execution. The reason is that the template language parser does not perform any semantic analysis which would require JavaScript AST, so the JavaScript parsing can be postponed until execution time.

Another argument for parsing JavaScript at runtime is that some pieces of injected JavaScript code do not have to be executed at all. Imagine a template rule

that is executed for every *proj_path* property in *general* section. If the tested configuration file does not have a section named *general* or it has no *proj_path* property, then the content of the rule (i.e. JavaScript code) is never executed. Therefore, it makes no sense to eagerly parse it. What makes sense is to lazily parse/-compile the code at the point when it is needed for the first time, store the compiled form of JavaScript and then reuse it when the rule is executed again.

[Coexistence of two parsers] There exist several reasons why there are two different types of parsers used to parse the whole template language:

- **Time-saving** - Implementing JavaScript parser by using Xtext would be difficult and time-consuming. By reusing already created JavaScript parser which is part of JDK 8, we shorten the implementation phase.
- **Performance** - As mentioned before, many programmatically generated parsers are not very efficient (this applies also to Xtext). The lack of performance is especially noticeable when parsing complex languages and large pieces of source code. For this reason, it makes sense to create an Xtext-based parser only for high-level constructs (relatively easy and fast to parse) and use more optimized parser for JavaScript parts.
- **No need for JavaScript AST** - One relative advantage of Xtext is that it also automatically generates AST models for parsers. Such models can be used for linking, code analysis etc. However, in this case we do not perform any analysis related to JavaScript code so it would be pointless to even use Xtext to create a JavaScript AST[1].

5. Template language interpreter

The language interpreter (see Figure 8) is a hybrid composed of several technologies and interpretation approaches.

Generally speaking, interpreters often use three ways of code interpretation. They either interpret an AST or bytecode or they compile code (at runtime) into some kind of machine code that can be executed by underlying hardware.

The template language interpreter uses all three variants of interpretation.

5.1 AST interpretation

At the beginning of the interpretation process, the interpreter parses the template and the configuration file. The results of parsing are ASTs. One AST represents the template itself and the other represents the tested configuration file.

Both types of ASTs can be theoretically interpreted directly, but their form (EMF models) is not very suitable for interpretation. For this reason, the template AST is converted into an immutable tree-like model of *rules* and *sub-rules*, and the configuration AST is converted into an immutable tree-like model that can convert itself and its subtrees into JSON format (see *exec. model* and config. *file JSON model* in the upper part of Figure 8).

With the new models ready, the real interpretation begins. The essence of the interpretation algorithm is shown in Figure 10. The basic idea of the algorithm is that it takes a *rules model* (i.e. *exec. model* in Figure 8) and *configuration model* in the new form and applies the rules from rules model to the configuration model.

[Configuration file AST in JSON] The reason, why configuration model is done in such way that it can be easily converted into JSON, is the interoperability with JavaScript. Let us have a look at Figure 6. The JavaScript code in the figure uses variables like *sectionName* and *value*. These variables are context-dependent and they point to certain parts of the *configuration AST* (an example of how a configuration file AST in JSON format may look like is in Figure 9).

5.2 JavaScript engine and JVM execution

The algorithm responsible for AST/rules model interpretation converts big part of the rules into JavaScript and combines it with the configuration AST in JSON format.

The algorithm then takes the generated JavaScript and JSON, creates appropriate scope for it and executes it via Nashorn engine. The engine parses the generated JavaScript code and compiles it into Java bytecode. The bytecode is then passed directly to the JVM and now it is up to the JVM to decide whether it will interpret the bytecode or use just-in-time (JIT) compilation.

From the performance point of view, the overall template interpreter speed is sufficient. Although the whole interpretation process consists of 4-5 steps (template/configuration source code → ASTs → rules (JSON) → JavaScript code → bytecode → machine code), most of the steps are not computationally intensive. The Xtext-based parsers for the template language and configuration files are lightweight, and the algorithm for AST/rules model interpretation is effective because majority of its job is to generate JavaScript code (concatenation of strings). It leaves the heavy lifting to Nashorn engine and the JVM and both of them

[1] If, for some reason, an access to a JavaScript AST is needed, it is possible to use Nashorn engine to obtain the AST. See: *JDK Enhancement Proposal 236, Parser API for Nashorn*

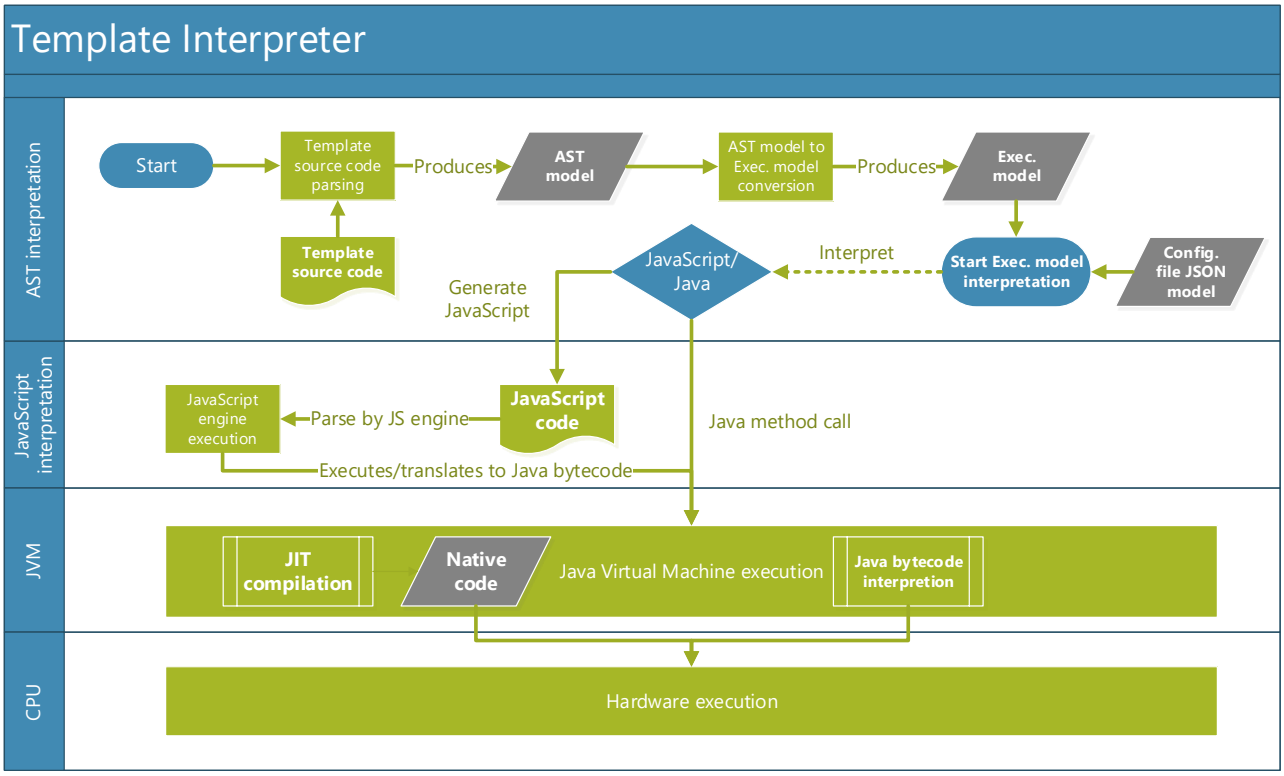

Figure 8. A schema of configuration template interpretation.

```
{
   "general": [{
      "name": "proj_path",
      "values": [
         "C:\\MyProject\\scripts"
      ]
   }, {
      "name": "pmonPort",
      "values": [
         45687
      ]
   }],
   "dist": [{
      "name": "dist",
      "values": [
         "PC1324", -3
      ]
   }]
}
```

Figure 9. The content of this figure represents a configuration file AST in JSON format (the original configuration file is in Figure 4).

have really good performance.

Also the template language execution is not a time-critical task. In case we have many templates and configuration files, we can run it on servers during night. Additionally, the interpreter itself provides means of parallel execution (that is why the models created from ASTs are immutable) and it effectively uses modern multi-core CPUs.

[Compilation to JS] It seems to be a trend of past few years that high-level languages are translated into JavaScript. For example, modern languages like Go, Kotlin, TypeScript and Dart are either primarily compiled into JavaScript or they have experimental compilers that can do it. The advantage of this way of execution is that JavaScript engines are available for various platforms and performance of some of them (V8, Chakra) is generally considered to be excellent.

6. Conclusions

This paper describes the design, parser and interpreter of a domain-specific language, whose purpose is to verify that configuration files for WinCC OA projects are correct and comply with the CERN-defined rules. The language design is based on declarative paradigm (rules), and it allows programmers to inject JavaScript code. The parser is created by using the Xtext framework, and the interpreter is combination of various approaches and technologies (AST interpretation, JVM, Nashorn).

The practical result of this work is a software tool that is used at CERN for verification of configuration files correctness. It is a part of the effort to improve the overall code quality and safety of SCADA systems.

This work is unique, because the DSL presented here is probably the only existing programming language that is specially designed for complex description and manipulation of configuration files (INI files). Additionally, the language itself provides a specific

```
type SuccessAndFail = {
    Success : int;
    Fail : int
}

type Rule = {
    CounterPredicate : SuccessAndFail->bool;
    ASTFilter : ConfigurationASTNode->ConfigurationASTNode;
    Execute : list<Executable>
}
and Executable =
    | Rule of Rule
    | Code of Code

type ExeResult = Option<FailStackTrace>

let inline (+) (acc: SuccessAndFail) (exeResult: ExeResult) =
    if exeResult.IsSome then {Success=acc.Success; Fail=acc.Fail+1}
    else {Success=acc.Success+1; Fail=acc.Fail}


let interpret (rule:Rule) (configASTPart:ConfigurationASTNode) =
    let filteredAST = filterAST rule.ASTFilter configASTPart
    let execResults = filteredAST |> List.map (internalExecution rule.Execute)
    let counter = execResults |> List.fold (fun acc elem -> acc+elem) {Success=0; Fail=0}

    if rule.CounterPredicate counter then ExeResult.None
    else getCounterFail counter execResults

let interpretTemplateAndCOnfiguration template configuration =
    let rule = convertTemplateToRule template
    let configAST = convertConfigurationToAST configuration
    interpret rule configAST
```

Figure 10. Simplified version of algorithm responsible for template AST interpretation. [F# language]

syntax for several levels of abstraction, allowing programmers to write easy-to-read high-level code and on the other hand access low-level features (even configuration file AST) if needed. The design of the DSL also offers a possibility to write code using declarative and imperative paradigm, while making sure that both paradigms seamlessly work together. Another unusual thing presented here is the effective way of how various technologies (Xtext, Nashorn and JVM) can be combined and serve as a programming language interpreter, whose architecture has proved to be very flexible and used technologies reliable.

As for the future development, it would be worth it to explore the performance limits of the interpreter design and used technologies. The implemented interpreter already uses several optimizations (reusing/sharing of immutable models, JSON-AST caching, parallel execution and partially lazy evaluation), and the paper shows that the performance is sufficient in case of offline verification of configuration files. However, there are still aspects that can be improved (for example caching and reusing of compiled JavaScript), and it could be useful to see, where are the actual limits, and whether this concept can be applied also in cases where performance truly matters.

Acknowledgements

I would like to thank Zbyněk Křivka for his advices and feedback.

References

[1] MAREČEK, Matěj. *IDE for SCADA Development at CERN*. Brno, 2016. Master's thesis. Brno University of Technology. Supervisor Zbyněk Křivka.

[2] *Nashorn - OpenJDK Wiki* [online]. Redwood Shores, Redwood City, California, United States: Oracle Corporation and/or its affiliates, 2016 [cit. 2016-03-12]. Available from: https://wiki.openjdk.java.net/display/Nashorn/Main

[3] *Xtext - Documentation* [online]. Ottawa, Canada: Eclipse Foundation, 2015, 2016-03-12 [cit. 2016-03-12]. Available from: https://eclipse.org/Xtext/documentation/