REL

Rules Engine Language

White Paper

Sudhakar Perumal
(skp2111@columbia.edu)

Introduction

Rules Engine Language (REL) is a simple language to represent condition and action statements in an effective manner and at the same time it can be interpreted and executed dynamically during runtime.

1.1 Background

One of the most common challenge that is encountered during software application design and development is efficiently managing the rules. Rules can be found in so many forms in software programs, sometimes they are the if-then-else statements that control the flow of logic and sometimes they are the statements that validate a data model. Often these rules make up large percentage of the code. Programmers spend most of their time to organize the rules so that the code becomes manageable and easy to debug. And in some applications its often required to change the rules frequently, for example, a sales application may change their rules based on the customer feedback and market conditions. In those applications its required to externalize the rules from the application source, so that the programmers need not recompile the entire code base when those rules change. This introduces another challenge that is how can these rules be represented so that it can be dynamically interpreted during runtime and executed. It forces the programmers to formulate syntax to represent these rules and write programs that can interpret these statements and execute during runtime. This type of software architectures don't follow any standard way of representing the rules statements, which makes it very difficult to learn the syntax and write the rules. Also, it becomes very difficult to manage the program, which interprets and executes these rules. Common Programming languages like c++ and java doesn't provide an concise way to represent and load these rules dynamically.

REL provides a clear solution to this problem. This language provides an easier way to represent the rules in java like syntax. So that it can be understood by anybody who is familiar with java like language. REL interpreter provides a way to load these rules dynamically during runtime, which allows the programmer to externalize the rules into a separate text file and not be part of the main source code. That gives the ability to modify the rules easily without having to recompile the source code for every little change in rules.

1.2 GOAL

Generally, rules consists of a conditional part, which tells what condition should be satisfied, and the action part, which tells what to do when that condition is satisfied. Main objective of this language is to provide a way to represent these rules easily and provide an interpreter to parse the rules and convert into bytecode and then execute them in JVM.

1.2.1 Portability

Since REL converts the rules into bytecode and then executes it, REL can be used in all the platforms supported by SUN JVM.

1.2.2 Efficiency

Since the rules are converted into bytecode before execution, REL provides an efficient way to execute the rules dynamically.

1.3 Main language features

1.3.1 Data types

Basic data types supported by this language are int, float, string and boolean. It also supports a collection – map. In order to allow grouping of rules, REL provides an additional datatype - ruleset.

1.3.2 Statements

The language supports Boolean expressions, if-then-else statements, for loops.

1.3.3 Internal Functions

The language provides basic functions like print, loadRules and eval (to execute rules).

1.4 Tutorial

Here is a simple program that will validate a data model. This program demonstrates how a dataset with certain elements related to person can be validated against ruleset that dictates what should be the value of each element. In this example, its required to enter name, add1, city, state and zip. Email is optional. And zip must be either 5 or 9 characters in length.

Let us assume the following script is stored as “formvalidation.script”

```
function boolean isInvalidField(string value) {
        if (value == null)
                return true;
        if (value == "")
                return true;
}
```

```
ruleset Validate_PersonalInfo {
        {isInvalidField(name)}  -> {error “Please enter name”}
        {isInvalidField(add1)}  -> { error “Please enter add1”}
        {isInvalidField(city)}  -> { error “Please enter city”}
        {isInvalidField(state)} -> { error “Please enter state”}
        { (zip.length() != 5) || (zip.length() != 9)} -> { error “Invalid zip” }
}
```

Following program will make use of above script, performs validation and.displays whether the provided datamodel is valid or not.

```
loadRules(“formvalidation.script”);

map input, output;

input.name = "test";
input.add1 = "test";
input.city = "test";
input.state = null;
input.zip = “123123”;

boolean ret = eval(“Validate_PersonalInfo”, input, output);

if (!ret)
        print(“Invalid model”);

// output will contain all the fields that resulted in error and corresponding message if any

while (map.hasMoreElements()) {
        map.next();
        print(map.key + map.value);
}
```