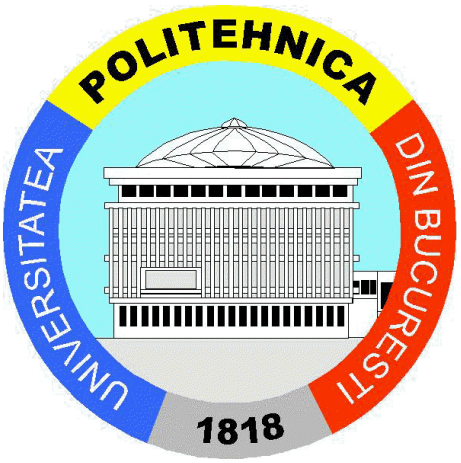
AUTONOMOUS AND DYNAMIC DECISION ENGINE

CĂTĂLIN IONUŢ RAMAŞCANU

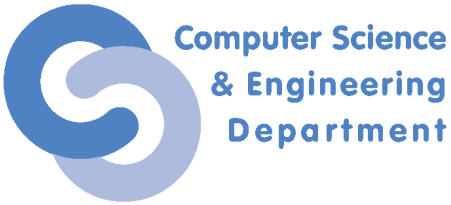


A Rule Engine With More Support For Decision-Making

Bachelor Thesis

Computer Science and Engineering Department
Faculty of Automatic Control and Computers
University Politehnica of Bucharest

September 2015



[September 8, 2015 at 21:51]

SUPERVISORS:
Ing. Tudor Scurtu
Şl.dr.ing Răzvan Deaconescu

LOCATION:
Bucharest

TIME FRAME:
September 2015

ABSTRACT

The demands of executing business rules on a given set of data are various and frequent in computer applications. Typically, a software component called rule engine is used to validate business rules at runtime. Even though most rule engines offer good performance for rule validations and data analysis, there is no proper support for decision-making and they are not easy to configure or to deploy for particular types of applications.

We propose in this thesis a new type of engine, which is not only capable of rule validation but also of decision-making. We describe the engine's functionality, how input data, rules, decisions are represented and how easy it can be configured for the user's needs. The project aims to cover a wide variety of use cases, offering an innovative and dynamic solution for business rule validation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Rule engines are very practical and effective for their representational simplicity and optimized performance, but their limited expressiveness and web unfriendliness restrict their usability.

— Minsu Jang, Joo-Chan Sohn [6]

In the field of software development, it is often that a developer is required to solve a problem that is just too fiddly for traditional code, the logic of algorithm changes too often or is beyond any obvious algorithm based solution. For this type of problems, a rule-based solution might solve all of the above issues. This solution is called a Rule Engine and it is a component that is used to validate rules at runtime. It is often used in order to be able to easily change rules without having to recompile software.

However, even though most rule engine solutions offer good performance for rule validations and data analysis, there is no proper support for decision-making and they are not easy to configure or to deploy for particular types of applications.

We propose in this thesis a new type of engine, which is not only capable of rule validation but also of decision-making.

In the following section we present the motivation for choosing to build a decision engine, what are the objectives that we aim to achieve and give a summary of the next chapters that follow.

1.1 MOTIVATION

The idea of the thesis was designed and implemented in collaboration with Adobe Romania. One of the developing teams in the company used a rule engine prototype software to monitor the state of their product. The monitoring software was developed internally in Adobe and it was designed to retrieve data from an external website, process the data and display different alerts based on some predefined rules. However, the prototype contained some limitations. It was complicated to configure, the configuration file required for setting up the engine was difficult to understand and the team wanted the software to also be able to perform other types of actions besides displaying alerts.

Choosing a rule engine for building a monitoring software is a good idea. Most programmers tend to use a rule-based solution when implementing an algorithm or an application mostly because it is very

easy to define rules and an engine is quite capable of handling an enormous amount of data and rule conditions.But what happens once a rule is triggered? The developer is responsible for writing blocks of code, which represents the action or the decision that the rule executes.

A problem appears in the way the rules are defined. Most rule engines expects the user to define rules in a text file using a given syntax. When defining a rule, code is written to represent the decision and it implies that there is no clear separation between the action and the rule condition. Rules are expected to be as much human-readable as possible and this is hard to achieve when combining the syntax for rule definition and the one of the programming language that dictates the action. A challenge also appears when trying to define the same action to multiple rules. Most engines do not offer a way to create aliases for decisions and by not doing so the developer has to copy the same block of code into each rule definition.

Another point of interest is the way of creating rules. Most rule engines permit the user to define a rule in a specialized file using a given syntax but other implementations don't offer a way to create the definition in a programmatic manner. Having this capability can prove very useful because the rules and actions are kept at the same syntax level and managing the engine from the code becomes an easier task.

A great opportunity for rule engines is to add proper support for decision-making. It would be a great feature for a rule engine to be capable of offering predefined actions like logging, sending data to a web server or sending email notifications to users. An engine that offers these capabilities can no longer be considered just a rule engine but rather a decision engine. At the moment, there are no implementations of such engines out there.

1.2 OBJECTIVE

The purpose of the thesis is to offer an easy-to-configure solution that is capable of executing different decisions or actions based on an input and on certain conditions specified by the user. It is necessary to be agnostic of the processed data interpretation, relying only on the configuration defined by the user. The solution itself can be described as a dynamic and autonomous decision engine.

Dynamic suggests that the engine offers multiple ways for the user to interact with it. The configuration of the engine can be defined programmatically or by using a configuration file that requires a syntax which is human-readable and easy to understand. A configuration is composed of definitions of rules, decisions and the types of input that need to be accepted by the engine. When it comes to inserting data, users have the possibility to do it programmatically by calling a

method or they can configure the engine to automatically fetch data from an external endpoint using a special agent called fetcher. Another useful way of inserting data is through an URL query string.

Dynamic also represents multiple ways of usage. The engine is capable of monitoring data flows that can be inserted by the user or automatically fetched. With rule definitions, it can detect anomalies or any other conditions defined by the user. Once detection has been made, the user can choose to trigger predefined decisions. Another alternative for the user is to define its own custom action which is fired by the engine on a given condition.

Autonomous suggests the engine can be deployed as a standalone component capable of retrieving data from an external endpoint, validating predefined rules and executing different types of decision. The engine keeps on running in case it encounters unexpected data types other than the ones configured by the user. It has proper logging support and it does not stop its execution once improper data is inserted.

1.3 THESIS OVERVIEW

In chapter two, we give a brief overview of other existing rule engine solutions. The decision engine must have a component to evaluate the rules defined by the user. Therefore, it was implemented as a wrapper around an existing rule engine. We describe multiple solutions for multiple languages and at the end we select one of them to be integrated in the decision engine.

Chapter three gives an overview of the decision engine. It covers use cases, how the engine can be configured and how the architecture was designed.

The fourth chapter goes deeper in the implementation of the engine and shows how it works internally.

The testing and results are presented in the fifth chapter. Here we show the capabilities of the current implementation and we compare it with other existing rule engine solutions.

The last chapter covers future development where we describe what features can be further developed in order to achieve a better implementation of the engine.

2 EXISTING RULE-BASED SOLUTIONS

If the rules are likely to change over time due to the nature of the application, then consider using a rules engine-the flexibility is worth the overhead.

— George Rudolph [1]

Rule engines have been around since the early 1990s and at this moment there are numerous implementations available in mostly any programming language. In this section, we analyze a few examples of rule engines in the following programming languages: Java, C/C++ and JavaScript. We have chosen these particular languages because they are the most popular and offer the greatest flexibility.

Each example contains a short description with a series of advantages and disadvantages.

At the end of the section, we select one of the rule engines to be used as the component responsible for rule evaluation and explain the selection.

2.1 RULE ENGINES IN JAVA

Java is one of the programming languages which offer the largest number of rule engines. Based on popularity, we have chosen the following implementations: Jess, Drools and Esper.

2.1.1 *Jess*

"The Rete algorithm is implemented by building a network of nodes, each of which represents one or more tests found on a rule left-hand-side"[1]

Jess is a rule engine written by Ernest Friedman-Hill at Sandia National Laboratories [2]. It uses an enhanced version of the Rete algorithm [1] to implement the processing and validation of rules. Features like backwards chaining and working memory queries are the the reason for why Jess is one of the most popular and powerful rule engines in Java.

One of the main characteristics we have noticed is that the syntax used to define rules in Jess is very similar to the one used by CLIPS (C/C++ rule engine).

Advantages

- The syntax for defining rules is easy to understand and for a CLIPS user it represents a major advantage.

1 http://www.jessrules.com/guidelines.shtml
2 http://herzberg.ca.sandia.gov/

- Well-structured documentation and numerous tutorials on how to get accommodated with Jess in a quick period of time.
- It is considered to offer much better performance than other popular rule engines written in C, especially on large problems.
- It offers a scripting language which enables the user to create Java objects and methods without compiling any Java code.

Disadvantages

- By using the Rete algorithm, Jess can be considered to be a memory-intensive rule engine. In order to achieve a high speed, the algorithm uses a significant amount of space.
- The rule engine offers an API in order for the user to create rules and queries directly from Java. However, this process is complex and it is an undocumented process. It is recommended to define rules using the Jess language.
- By offering a scripting language for rule definitions, Jess creates a separation between Java and the rule engine itself. For creating a stand-alone rule solving component, Jess can be a great choice. On the other hand, if the user decides to include the solving component in an existing Java environment he or she has to learn a new language and determine what is the proper way to adapt the component to the project.

2.1.2 *Esper*

EsperTech developed a complex event processing component also for .NET which is called NEsper.

Esper is not considered to be a rule engine but rather a component for complex event processing(CEP) and event series analysis. It's being developed by EsperTech[3] and it is a great choice for developing systems which processes large volumes of incoming messages or events.

For analyzing the event stream, it uses a language called Event Processing Language(EPL) which is very similar to Structured Query Language(SQL). By using EPL, the user can create a Big Data processing engine for any type of real-time arriving data.

Esper can be used as a rule engine because rules are considered to be a subset of CEP techniques. The EPL can easily be used to define if-then statements which are the basis for any rule engine.

Advantages

- The language provided by Esper is similar to SQL, facilitating the definition and configuration of rules which are applied on the data stream.

3 http://www.espertech.com/esper/

- The processing engine which the user can build is considered to be highly scalable, memory-efficient and it offers minimal latency data processing.
- The EPL statements can either be defined as string values or constructed in a programmatic manner using an API provided by Esper.
- It can run in any architecture and it has no dependencies on external services.

Disadvantages

- Esper does the computing in-memory which makes it memory bound. Processing too many events can potentially cause Esper to run out of memory.
- Although the documentation is quite through and complete, it lacks the practical examples and it makes it look like working with a Esper is complicated. It does not include any tutorials to get accustomed with the engine.
- EPL can be familiar to many developers because it is very much like SQL but this type of structure can become unsuitable for describing complex CEP patterns and it can require more writing effort.

2.1.3 *Drools*

Drools is part of the JBoss community and it categorizes itself as a Business Rules Management System (BRMS) with a forward and backward chaining inference based rules engine [4]. The system is composed of several components:

- Drools Guvnor - Business rules manager
- Drools Expert - Rule engine
- Drools Workbench - Authoring and rules management application
- Drools Fusion - Complex event processing
- Drools Planner - Optimizes automated planning
- Drools Flow - Workflow and business processes
- Eclipse IDE plugin for developers

4 http://www.drools.org/

The rules are defined in a Drools file, a plain text file with .dlr extension short for Drools Rule Language. It can contain multiple rules, queries and functions, as well as some resource declarations like imports, globals and attributes that are assigned and used by the rules and queries. However, the user is also able to spread the rules across multiple rule files.[5]

Advantages

- Drools provides template based rule definitions which contains a simple and human readable structure.
- Wealth of online information (documentation, tutorials, books) about Drools. Very simple for a user to get accustomed to the system.
- Drools is integrated in 3rd party systems. Ex: Spring, Camel.

Disadvantages

- Drools provides an editor which is a plugin to Eclipse IDE. Unfortunately, using Eclipse is the only way to build the rules. Even if the user builds from the command line, it actually runs Eclipse headless mode. The user is not tied to the editor, he or she can write all the rules in any other editor and then use the libraries to build them.
- Technically is difficult to debug Drools Rule Engine files.
- It provides performance at the cost of memory.
- The system does not offer the possibility to create,edit and delete rules at runtime. There was an API which the user could use to build rules from Java code but it has been categorized as unstable and deprecated.
- Drools is not a lightweight system and it is not a good choice if the user is planning on building a system which requires validation of simple rules.

5 https://docs.jboss.org/drools/release/5.2.0.Final/drools-expertdocs/html/ch05.html

2.2 RULE ENGINES IN C/C++

In C/C++, there are not as many rule engine implementations as there are in Java. We have chosen to analyze one of the oldest rule engines ever implemented: CLIPS.

2.2.1 *CLIPS*

CLIPS has been actively developed for over 30 years.

CLIPS stands for "C Language Integrated Production System" and it was first developed in 1984 at NASA's Johnson Space Center. At this moment, CLIPS is maintained independently from NASA as public domain software.The project represents a tool which provides a complete environment of rule and/or object based expert systems.It is still actively supported by Gary Riley who designed and developed the rule-based portion of CLIPS.[2]

Although it is a rule engine written in C, there are special versions of CLIPS which can be used in other programming languages. One special version is Clipsmm which is a C++ CLIPS interface. It offers the possibility to pass C++ objects into the rule engine. [6]

The rules are defined in Lisp-like language. A program written in CLIPS may consist of rules, facts and objects. The inference engine decides which rules should be executed and when. A rule-based expert system written in CLIPS is a data-driven program where the facts and objects if desired, are the data that stimulate execution via the inference engine. [3]

Advantages - extracted from the official website [2]

- CLIPS is lightweight and has received widespread acceptance because of its portability and low cost capabilities.
- It is written in C for portability and speed. The engine has been installed on many different operating systems (tested on Windows XP, Mac OS X and Unix). CLIPS can be ported to any system which has an ANSI compliant C or C++ compiler.
- CLIPS is widely used throughout the academia and there many resources which can be acquired in order to get accustomed to it really fast. It also comes with extensive documentation including a Reference Manual and a User's Guide.
- It supports three different programming paradigms: rule-based, object-oriented and procedural. Rule-based programming allows knowledge to be represented as heuristics, which specify a set of actions to be performed for a given situation. Object-oriented programming allows complex systems to be modeled as modular components The procedural programming capabilities pro-

6 http://sourceforge.net/projects/clipsmm/

vided by CLIPS are similar to capabilities found in languages such as C, Java, Ada, and LISP.

Disadvantages

- CLIPS can be considered to have a steep learning curve and because of this reason it can be seen as a tool for specialists.
- It is single threaded and it does not support backward chaining. The basic control flow is forward chaining. If the user would like to implement other kinds of reasoning, he or she has to manipulate tokens in working memory.
- Clipsmm offers a poor documentation and there are no examples on how to use the library. Getting accustomed to it can be time consuming.
- CLIPS's performance can rapidly decrease if the engine needs to deal with large amount facts and changeable facts. The following paper [4] offers multiple optimization techniques in order to solve this problem and boost the performance of CLIPS.

2.3 RULE ENGINES IN JAVASCRIPT

The main reason for choosing JavaScript as programming language is because most of the rule engine implementations use JSON format as a way to create rules. This can be a real advantage if we want to achieve our objective of defining an easy and human readable syntax for rule and decision definitions.

We noticed that the rule-based solutions in JavaScript are very lightweight and have been developed very recently.

We analyze the following rule engines: Nools and Business-rules-engine.

2.3.1 *Nools*

Nools is a business rule engine based on the Rete Algorithm and it was developed by Doug Martin [7]. The main deployment for this engine is to a Node.js server but it is also prospectively deployable in most browsers too.

The engine's workflow can be described easily by the following internal components:

- Rule - Constraints that must be satisfied in order for an action to execute.
- Action - The code that executes when all of the rule's constraints have been satisfied.

7 https://github.com/C2FO/nools

- Flow - It represents a collection of rules.
- Session - An instance of an object of type Flow.
- Fact - An object inserted into a session in which the rule's constraints match against.

A rule in Nools has three parts:

- The types of data it applies to.
- The filter to apply when the data types match.
- The logic to apply when the filters match.

Advantages

- Nools is a very simple and lightweight rule engine. Rules can either be defined programmatically or using its own rule definition language called DSL.
- Documentation is quite small and simple but it has all the examples a user needs to get accustomed with it. Not being a complex engine, it is very easy to understand how Nools works.
- Support for Node.js and deployable on most browsers.
- Nools creates a semantic separation between the logic and the filters. It can possible lead to a performance optimization by allowing to retrieve rules based on the data type.

Disadvantages

- The engine is two years old and it may contain possible flaws. It is still an early version of the engine.
- When building a big application, the API can become a bit complicated to use and the code base can become complex.
- In a distributed system, Nools does not allow to serialize a state, deserialize it and resume rules validations.
- Using its own domain specific language (DSL) instead of functional JavaScript can become an obstacle in keeping the engine as powerful as possible.
- There are no records about the engine's performance.

2.3.2 *Business-Rules-Engine*

Business-rules-engine is a lightweight JavaScript library developed by Roman Samec and it offers easy rule definition. The project started in 2014 and it has been actively maintained since then.

The main benefit of this solution is the fact that it is not tight to HTML DOM or any other UI framework. This validation engine is UI agnostic and that is why it can be used as an independent representation of business rules of a product, contract, etc. It can be easily reused by different types of applications, libraries.[8]

It supports declarative and imperative validation rules definition:

- declarative JSON schema with validation keywords.
- declarative raw JSON data annotated with meta data - using keywords from JQuery validation plugin.
- imperative - validation API.

Advantages

- Rules can be defined using JSON format.
- Simple engine with straight forward documentation.
- It uses promises to support asynchronous validation rules.
- It supports assigning validation rules to collection-based structures - arrays and lists.

Disadvantages

- The engine can only be used with Node.js server and it is not deployable on browsers.
- There are no records about the engine's performance.
- It's being developed for a year and it may contain possible flaws.
- There are not many articles on the internet which demonstrates that this engine was integrated in any other projects.

8 https://github.com/rsamec/businessrules- engine

2.4 CHOOSING THE RIGHT RULE ENGINE

After analyzing all rule-based solutions, we have decided to use Esper as the rule validation component for our decision engine.

The main reason for choosing this project is because it offers the possibility to define and manage rules straight from the Java code at runtime. This is a huge advantage for defining our own language for configuring the decision engine. It can easily wrap around the EPL syntax, defining its own rule and decision definitions language.

Being a project developed on Java, it also can be seen as an advantage because it would be much easier for us to cover all the use cases mentioned in Chapter 3.

Therefore, Esper acts as the main component for input data processing and rule evaluation in order for decisions to be executed. The decision engine practically acts as a wrapper around Esper offering a more broad support for decision making and a simple,easy to read syntax used for the configuration of the engine.

3 DECISION ENGINE OVERVIEW

There are two ways of constructing a software design:
One way is to make it so simple that there are
obviously no deficiencies, and the other way is to make it
so complicated that there are no obvious deficiencies.

— Professor Sir Charles Anthony Richard Hoare

As mentioned in Chapter 1, this project aims to solve the problem of hard to configure rule engines and the lack of options for decision making.

We have already decided to use Esper as a component for rule evaluation and have chosen Java for it's flexibility and portability. The next step is to define how will the user interact with the decision engine, what are the types of use cases it can have and how it will be built so that it will cover all of the objectives that we have defined at the beginning of this thesis.

The decision engine was developed using the Git version control and the repository is published on GitHub.

In the following sections we cover the use cases for the decision engine, the configuration methods that the user has at its disposal and the high level architecture which gives us an oversight on how the engine works internally.

3.1 USE CASES

Being dynamic is one of the key characteristics of the decision engine. We want the user to have many possibilities when it comes to usage. He or she might want to use the component as a library and integrate it into another project. Or just use it immediately as an independent component. Overall the decision engine aims to cover as many types of use cases as possible having also the ability to easily deploy and switch between states.

Therefore, we have defined the following three uses cases.

3.1.1 *Framework/Library*

The decision engine puts at the user's disposal an API which can be used in order to easily integrate the component into any other Java projects. The API includes methods whose parameters can be interpreted as either input data for the engine or definitions of rules and actions. One particular method can be used in order to retrieve values when a rule has been triggered.

3.1.2 *Stand-Alone Component*

The decision engine can be deployed as a stand-alone process, being capable of retrieving data from an external endpoint and taking different actions based on certain rules.

The user can achieve this by writing a configuration file where he or she would define the input data, rules, actions and data-fetchers. Once the file is completed, the decision engine can be easily deployed by simply specifying the path to the configuration file. The engine would then retrieve the data using the special entities called fetchers, process the data, evaluate the rules and execute the actions if any rule condition has been satisfied.

3.1.3 *Web Application*

The decision engine can be exposed as a web application server using the SpringBoot framework.[5]

One of the functionalities that was implemented in the engine is the option to insert input data using the parameters of an URL's query. Once the data is inserted, an "Accepted" HTTP status is returned as a result.

Example: `http://127.0.0.1/insert_input?inputId="primitive-elements"&int_element=2`

A monitor page is also included when deploying it as a web application. This page can be used to view the current configuration of the engine, analyze the logs and insert input data through a form.

3.2 ENGINE CONFIGURATION

When it comes to configuration, the decision engine needs to have a generic and dynamic way of interaction. Therefore, it offers the possibility to be configured in a programmatic manner, by creating it's own API and by using a configuration file which uses JSON format. JSON has the advantage of being easy to read and to understand. A user who has no knowledge of how the decision engine works internally will be able to understand easily what is the current configuration and how it can modify the state of the engine.

The configuration model is mainly consisted of the following components:

- Input - The type of input data the engine will accept and will process in order to see if a rule condition has been satisfied.
- Rules - Collection of constraints which apply to a specified type of input. It will also be associated to a list of decisions that will be executed when the constraint is fulfilled.

- Actions - Collection of decisions which will be available to be included in the definition of a rule. Certain types of decisions will be available to use.
- Fetchers - Special components which are capable of retrieving data from an external source and injecting it into the decision engine.

In the following subsections, we go through the process of defining all of the configuration components both through JSON format and in a programmatic manner using Java code.

3.2.1 *Input Definition*

At this step the user specifies a list of input domains which are accepted by the decision engine. Each input domain contains an "input-id" field to uniquely identify each domain and a "data" field which is a list of data definitions.

Each data definition has two properties:

- Name - It describes the input data.
- Type - It specifies what type is the particular input data. It can be one of the following: int, float, long, double, string.

Listing 1 illustrates how input data can be defined using JSON format. The decision engine accepts two input domains: one which contains only primitive types and another one which only has an object type of data. When inserting data, the engine needs to know to which input definition is the data meant for in order to check its type and validate it.

Listing 1: Input Configuration Using JSON

```
"input":[
  {
    "input-id":"primitive-elements",
    "data":[
      {"name":"int_element",      "type":"int"},
      {"name":"float_element",    "type":"float"},
      {"name":"long_element",     "type":"long"},
      {"name":"double_element",   "type":"double"},
    ]
  },
  {
    "input-id":"object-element",
    "data":[
      {"name":"string_element",   "type":"string"}
    ]
  }
]
```

Listing 2 shows the exact configuration as Listing 1 but now using Java code. In order for the user to define configuration components programmatically, he or she needs to use special Java classes which act as a model.

Listing 2: Input Configuration Using Java Code

```
public void defineInputThroughAPI(){
    // Get decision engine instance
    DecisionEngine decisionEngine =
        FactoryDecisionEngine.getSingletonInstance();

    // Define primitive elements
    InputDefinitionModel inputModel1 =
        new InputDefinitionModel();
    inputModel1.setInputID("primitive-elements");

    DataDefinitionModel dataModel1 = new DataDefinitionModel();
    dataModel1.addDefinition("int_element", "int");
    dataModel1.addDefinition("float_element", "float");
    dataModel1.addDefinition("long_element", "long");
    dataModel1.addDefinition("double_element", "double");

    inputModel1.setDataDefinition(dataModel1);
    decisionEngine.addInputDefinition(inputModel1);

    // Define object element
    InputDefinitionModel inputModel2 =
        new InputDefinitionModel();
    inputModel2.setInputID("object-element");

    DataDefinitionModel dataModel2 = new DataDefinitionModel();
    dataModel2.addDefinition("string_element", "string");

    inputModel2.setDataDefinition(dataModel2);
    decisionEngine.addInputDefinition(inputModel2);
}
```

3.2.2 *Rule Definition*

A rule configuration component is characterized by the following fields:

- Rule-id : Uniquely identifies a rule definition.
- Input-domains: A list of input domains to which the rule applies. Once the rule has been defined, the engine always checks its condition each time data is inserted for the input domains specified in the list.
- Actors: A list of names which identifies the data definitions of an input domain. A data definition's name needs to be specified

here in order to be used in a condition statement. The decision engine reads this list and knows which data from an input domain needs to be analyzed so that the condition can be checked.

- Condition: Represents the rule constraint. It can be either evaluated true or false.
- Actions: A list of actions that are executed if the conditions is evaluated to true. The list contains IDs which are used to indetify the action itself.

In the following listings, we show how a rule can be defined both in a programmatic manner and through a configuration file.

Listing 3: Rule Configuration Using JSON

```
"rules":[
  {
    "rule-id":"rule-elements",
    "input-domains":["primitive-elements", "object-element"],
    "actors":[
      "int_element",
      "string_element"
    ],
    "condition": "int_element > 0 && string_element == 'Hello'",
    "actions" : ["printHelloAction"]
  }
]
```

Listing 4: Rule Configuration Using Java Code

```
public void defineRuleThroughAPI(){
    // Get decision engine instance
    DecisionEngine decisionEngine =
        FactoryDecisionEngine.getSingletonInstance();

    // Define rule
    RuleModel ruleModel = new RuleModel();
    ruleModel.setRuleID("rule-elements");
    ruleModel.addInputDomain("primitive-elements");
    ruleModel.addInputDomain("object-element");
    ruleModel.addActor("int_element");
    ruleModel.addActor("string_element");
    ruleModel.setCondition("int_element > 0 && string_element ==
        'Hello'");
    ruleModel.addAction("printHelloAction");
    // Add rule to engine
    decisionEngine.addNewRule(ruleModel);
}
```

3.2.3 *Action Definition*

The decision engine supports two types of actions: built-in and custom. A built-in action is directly implemented in the engine and it needs only to be defined like any other configuration component. Custom actions are represented by Java classes which are created by the user. In order to build its own custom action, the user needs to extend an abstract class called "Action" and implement a method called "executeAction()". Once this is done, it can successfully use the newly created class in an action definition.

Each definition of an action must contain the following fields

- Action-id: Uniquely identifies an action definition.
- Action-Type: Specifies if this definition is for a built-in action or a custom action.
- Class: This field is used to specify the Java class which represents the action. It is loaded and instantiated by the decision engine.
- Arguments: Each class that is loaded by the engine can take certain arguments which can be either optional or mandatory.

In the following subsections, we describe the built-in actions and the custom one.

3.2.3.1 *Print-Message Action*

This is an easy type of action which can be used in order to print a message to different destinations. It takes the following constructor parameters:

- Message: The actual message specified by the user.
- Target: Defines the destination of the message. It can be one of the following: "STDOUT" (standard output), "STDERR" (standard error), "FILE" (a text file).
- File-name: This argument is mandatory if "FILE" is specified as target. It represents the name of the text file in which the message will be written.

Listing 5 contains an example of how this type of action can be configured. We imagine that we have attached our decision engine into a project which is responsible for authenticating users to a specific service. We want the engine to print a log message into a file if the rate of authentication is above a certain value.

Listing 5: Print-Message Action Configuration

```
"actions":[
    {
      "action-id":"authRateAction",
      "action-type":"built-in",
      "class":"com.adobe.primetime.adde.output.PrintMessageAction
          ",
      "arguments": {
        "message": "We have a problem with authentication rate.",
        "target": "STDOUT"
      }
    }
]
```

3.2.3.2 *Send-Data-Via-Socket Action*

The decision engine is capable of sending data to an external server using the Send-Data-Via-Socket action. It uses a socket to a connect to a given IP address and port. Once the conection is established, the engine is able to send data. It takes the following arguments:

- Destination-ip-address: The IP address of the destination server.
- Destination-port: The port number of the destination server.
- Data-source-type: The source of the data can either be a file or an argument. It takes one of the two values: "FILE", "ARGUMENT";
- Data-source: If data-source-type is set to "ARGUMENT" then this field represents the actual data that is sent to the destination server. Else it represents the name of the source file which contains the data.

Listing 6 illustrates the configuration of a Send-Data-Via-Socket action. The purpose is to be able send a JSON file to a server situated at the following address 72.100.32.55:2222.

Listing 6: Send-Data-Via-Socket Action Configuration

```
"actions":[
  {
    "action-id":"sendJSONToServer",
    "action-type":"built-in",
    "class":"com.adobe.primetime.adde.output.
        SendDataViaSocketAction",
    "arguments":{
      "destination-ip-address":"72.100.32.55",
      "destination-port":"2222",
      "data-source-type":"file",
      "data-source":"src/test/JsonFile"
```

```
    }
  }
]
```

3.2.3.3 *Send-Email-Via-Smtp Action*

The decision engine is capable of sending email nottifcations to a given set of email addresses using the SMTP protocol. Setting up this type of action can be a bit complicated but using the arguments specified below makes the configuration more intuitive and easier.

- Smtp-properties: It represents a list of properties specific to the SMTP protocol which needs to be set in order to properly connect to a SMTP mail server. A property is composed of a name and a value. A list of SMTP properties can be found at the following source: [1]
- Receiver-list: It contains the destination email addresses.
- Subject: The subject of the email.
- Message: The actual email message.
- Username and Password: If "mail.smtp.auth" property is set to true then authentication is required in order to connect to the SMTP mail server.

Listing 7 illustrates an example of Send-Email-Via-Smtp action which connects to Gmail and sends an email to the specified address in the receiver-list.

Listing 7: Send-Email-Via-Smtp Action Configuration

```
"actions":[
  {
    "action-id": "sendEmailToEmployers",
    "action-type": "built-in",
    "class": "com.adobe.primetime.adde.output.
        SendEmailViaSmtpAction",
    "arguments": {
      "smtp-properties" :[
        { "prop-name" : "mail.smtp.connectiontimeout",
          "prop-value":"200"},
        { "prop-name" : "mail.smtp.auth",
          "prop-value":"true"},
        { "prop-name" : "mail.smtp.starttls.enable",
          "prop-value":"true"},
        { "prop-name" : "mail.smtp.host",
          "prop-value":"smtp.gmail.com"},
```

1 https://javamail.java.net/nonav/docs/api/com/sun/mail/smtp/package-summary.html

```
        { "prop-name" : "mail.smtp.port",
          "prop-value":"587"}
      ],
      "username":"some_username",
      "password":"some_password",
      "receiver-list": ["reader@thesis.com"],
      "subject": "This is the subject line!",
      "message": "This is the actual message!"
    }
  }
]
```

3.2.3.4 *Return-Value Action*

This special type of action can be used in order to access the "actors" in a rule programmatically.For this action to be executed, input data must be inserted using the special method called "addInputDataWithReturnValue()". The method inserts data in the engine, waits to see if any rules have been evaluated as true and it returns the values of the "actors" specified in the configuration of the action.

Return-Value action takes only one argument: "actors-to-return". The arguments contains a list of all the "actors" which are returned in a programmatic manner once the special method is called.

Listing 8 and 9 offer a simple example of how this type can be configured and used.

Listing 8: Return-Value Action Configuration

```
"actions":[
  {
    "action-id":"return-elements",
    "action-type":"built-in",
    "class":"com.adobe.primetime.adde.output.ReturnAction",
    "arguments":{
      "actors-to-return":["int_element","float_element"]
    }
  }
]
```

Listing 9: Return-Value Action Usage

```
public void useAddInputWithReturnValue(){
    // Get decision engine instance
    DecisionEngine decisionEngine =
        FactoryDecisionEngine.getSingletonInstance();

    // We use a HashMap for associating the values with the
        proper names of the input data
    Map<String, Object> elementMap = new HashMap<>();
    elementMap.put("int_element", 10);
```

```
    elementMap.put("long_element", 20);
    elementMap.put("float_element", 10.20);
    elementMap.put("double_element", 20.10);

    // Insert data and retrieve the result
    Map<String,Map<String,Object>> result =
        decisionEngine.addInputDataWithReturnValue(
            "primitive-elements",
            "return-elements",
            elementMap);

    System.out.println(result);
}
```

3.2.3.5 *Custom Action*

The "class" field of an action definition is used to load a Java class which acts as the decision that is executed by the engine. A custom written Java class can be passed as a value for the field but it needs to extend the abstract class "Action" and implement the "executeAction()" method in order to be recognized as a decision by the engine.

When configuring the action definition, the user can pass the "constructor-args" argument to the custom Java class. The argument represents a list of values which are passed as constructor parameters when the class is loaded and instantiated. The type of the values must match with the definition of the constructor for the Java class.

Listing 10 illustrates an example configuration of a custom action. The "CustomAction" class contains a constructor which takes as parameters two strings and an integer.

Listing 10: Custom Action Configuration

```
"actions":[
  {
    "action-id":"my-action",
    "action-type":"custom",
    "class":"com.adobe.primetime.adde.CustomAction",
    "arguments": {
      "constructor-args":[
        "My custom action ",
        "has been trigered. ",
        2015
      ]
    }
  }
]
```

3.2.4 *Fetcher definition*

As mentioned above, a fetcher is an agent capable of retrieving data from an external source and insert it into the decision engine as input data.

The fetcher can be easily configured by using the following fields:

- Fetcher-id: Uniquely identifies a fetcher definition.
- Receiver-input-id: This field represents the input domain which acts as a destination for the data retrieved by the fetcher. The format of the retrieved data must match with the format established in the definition of the input domain.
- Url: The url which is accessed by the fetcher in order to retrieve the available data.
- Interval: The fetcher can be configured to retrieve data at a given interval time. It can be a combination of seconds, minutes or hours.
- Num-of-fetches: The number of retrievals can be limited using this field. If it is set to zero then the fetcher makes unlimited data retrievals.
- Fetcher-parser: In case the retrieved data does not match the format established in the input domain, a parser class can be written by the user and specified in the definition of the fetcher. The parser class must implement the "FetcherParser" interface in order to be used.

Listing 11 contains an example configuration of a fetcher that retrieves data three times at an interval of ten seconds. The retrieved data is processed and parsed before inserted into the decision engine.

Listing 11: Fetcher Configuration

```
"fetchers":[
    {
        "fetcher-id":"dataFetcher",
        "receiver-input-id":"primitive-elements",
        "url":"https://sp.data.webcenter.com/data.json",
        "interval":"10s",
        "num-of-fetches":"3",
        "fetcher-parser":"com.adobe.primetime.adde.CustomParser"
    }
]
```

The complete example of a configuration file for the engine can be seen in listing 12.

Listing 12: Full Engine Configuration File

```
{
  "input":[
    {
      "input-id":"primitive-elements",
      "data":[
        {"name":"int_element",      "type":"int"},
        {"name":"float_element",    "type":"float"},
        {"name":"long_element",     "type":"long"},
        {"name":"double_element",   "type":"double"}
      ]
    },
    {
      "input-id":"object-element",
      "data":[
        {"name":"string_element",   "type":"string"}
      ]
    }
  ],
  "rules":[
    {
      "rule-id":"rule-elements",
      "input-domains":["primitive-elements", "object-element"],
      "actors":[ "int_element","string_element" ],
      "condition": "int_element > 0 && string_element=='Hello'",
      "actions" : ["printHelloAction"]
    }
  ],
  "actions":[
    {
      "action-id":"printHelloAction",
      "action-type":"built-in",
      "class":"com.adobe.primetime.adde.output.PrintMessageAction
          ",
      "arguments": {
        "message": "We have a problem with authentication rate.",
        "target": "STDOUT"
      }
    }
  ],
  "fetchers":[
    {
      "fetcher-id":"dataFetcher",
      "receiver-input-id":"primitive-elements",
      "url":"https://sp.data.webcenter.com/data.json",
      "interval":"10s",
      "num-of-fetches":"3",
      "fetcher-parser":"com.adobe.primetime.adde.CustomParser"
    }]
}
```

3.3 ARCHITECTURE

The architecture of the engine is composed of four main modules, each one having a specific role in the workflow.

3.3.1 *Configuration Parser*

This module is responsible for parsing and validating the configuration file. Once the file is validated, the module converts the JSON definitions in actual Java classes which are passed as configuration components to the Decision Engine module. A configuration component can be a definition of an input data, rule , fetcher or action.

3.3.2 *Fetcher Manager*

Fetcher Manager is responsible for coordinating the fetchers and it receives the necessary definition from the Decision Engine module.

It contains two sub-modules:

- Fetcher Agent: The role of the agent is to establish a connection to the URL specified in the fetcher definition and retrieve the data.
- Fetcher Parser: It is the interface which the user can implement in order to configure the engine to parse the retrieved data so that the format is identical with the receiving input domain.

3.3.3 *Web Server*

This module is used when the user wants to deploy the decision engine as a web application. It contains two submodules:

- Web Controller: The purpose of this submodule is to respond to HTTP requests sent by the user through a web browser. Inserting data through an URL query is implemented here.
- Web Monitor: The submodule contains the implementation of the web page monitor in order for the user to view the state of the engine.

3.3.4 *Output*

All Java classes for the built-in actions are stored in this module. It also contains the Action abstract class which can be used to implement a custom action.

3.3.5 *Decision Engine*

The Decision Engine module represents the core component of the project. It is a wrapper built around the Esper event-processing component and it interacts with the rest of the modules.

The main responsibilities of this module are the following:

- Providing an API to the user in order to easily interact with the engine in a programmatic manner. Through the API, the user can configure it and insert input data programmatically.
- When receiving input data, rule and action definitions from the Configuration Parser module, it successfully converts them to a format which is accepted by Esper. Fetcher definitions are passed to the Fetcher Manager module . Once Esper evaluates a rule as being true, the module triggers the necessary actions from the Output module.
- Capable of setting up the Web Server module. It provides information about the state of the engine in order for the user to monitor it online through a web page.

Listing 12 describes the architecture of the engine.

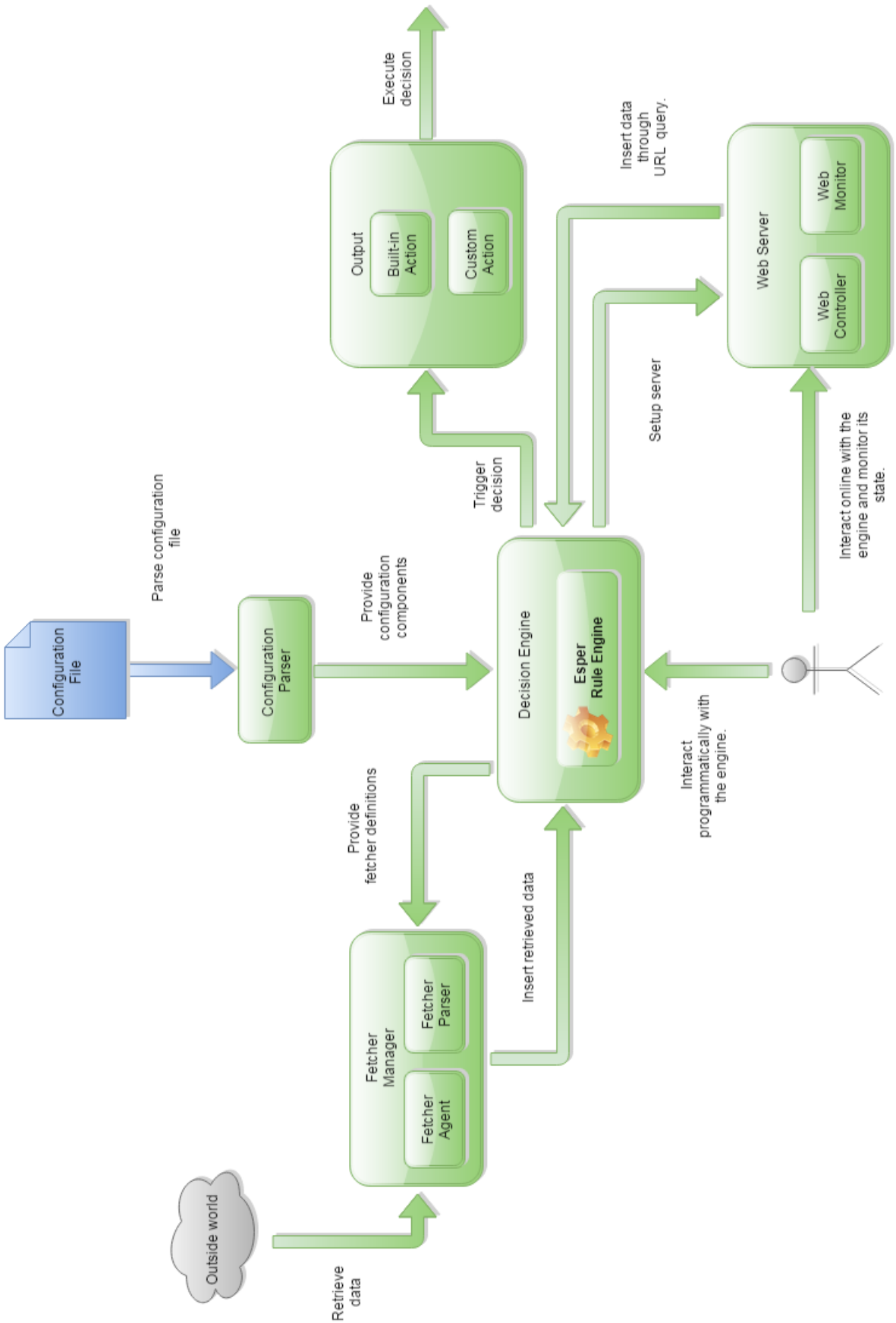


Figure 1: High-Level Diagram of the Architecture

4 DECISION ENGINE IMPLEMENTATION

A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.

— Antoine de Saint-Exupéry

The following sections describes the internal workflow of each main module of the decision engine's architecture. The inner of logic of a module is highlighted using code snippets.

4.1 CONFIGURATION PARSER MODULE

One of the main characteristics of the decision engine is to have an easy way of configuration. The reason for choosing JSON format (instead of XML for example) is the fact that it is easier to read and understand. It is also very simple to write a configuration in JSON due to the fact that it requires lesser tags. An XML item for example needs to be wrapped in an open and close tag whereas JSON uses only a name tag once.

The module itself uses a library called "Google HTTP Client Library for Java" which is developed by Google [1]. It is a flexible, efficient, and powerful Java library for accessing any resource on the web via HTTP. But one of the main features that it has is the efficient data model for parsing files written in JSON format. The model makes it very easy to map a JSON object to a Java class. Therefore, every configuration component defined in the JSON file is converted into its associated Java object which acts as an entity of the component.

For example, each rule definition is mapped to a RuleJson Java class which contains the values for the rule specified in the definition. Listing 13 illustrates how such a class is defined internally in the decision engine. Mapping each member variable of the class to the name tags of the JSON format is done by using the "Key" annotation.

Listing 13: Definition of the RuleJson Java Class

```
1 public class RuleJson {
2     @Key("rule-id")
3     private String ruleID;
4
5     @Key("input-domains")
6     private List<String> inputDomains;
```

1 https://github.com/google/google-http-java-client

```
7
8       @Key("actors")
9       private List<String> actors;
10
11      @Key("condition")
12      private String condition;
13
14      @Key("actions")
15      private List<String> actions;
16  }
```

Parsing the condition field of the rule definition and converting it to an Esper expression proved to be a bit difficult. We decided to design our own algorithm to manage the conversion. The algorithm implies that the string passed as the rule condition be converted to postfix format (also known as reverse polish notation) and then create the Esper expression while passing through each token of the postfix string.

Listing 14 illustrates at a high level the algorithm in Java for converting the condition string to an Esper expression.

Listing 14: Rule Condition Parsing Algorithm

```
1  Expression covertToEsperExpression(String condition){
2      String postFixCondition = convertToPostFix(condition);
3      Stack<Object> expressionStack = new Stack<Object>();
4      String[] tokens = postFixCondition.split(" ");
5
6      for (String token : tokens){
7          if (token.equals("&&") || token.equals("||")){
8              Expression expr1 =
9                  (Expression) expressionStack.pop();
10             Expression expr2 =
11                 (Expression) expressionStack.pop();
12             expressionStack.push(
13                 new Expression(token,expr1,expr2));
14             continue;
15         }
16         if (token.equals("=") || token.equals(">") ||
17                 token.equals("<") || token.equals(">=") ||
18                 token.equals("<=")){
19             String op1 = (String) expressionStack.pop();
20             String op2 = (String) expressionStack.pop();
21             expressionStack.push(new Expression(token,op1,op2));
22             continue;
23         }
24
25         // If no match until now, it means token it's a simple
               operand;
26         // We push it to the stack as it is.
27         expressionStack.push(token);
28     }
```

```
29
30      // After processing all tokens, the stack should only have
            one expression.
31      // If not, then postFixCondition is invalid.
32      if (expressionStack.size() != 1){
33          System.err.println("Failed to convert condition string to
                proper format.");
34      }
35
36      return (Expression) expressionStack.pop();
37  }
```

4.2 OUTPUT MODULE

Every built-in action are actually child classes of the abstract class "Action" which implements an interface of Esper called "StatementAwareUpdateListener". The interface permits Esper to use the child classes as listeners for any defined rule. Therefore once a rule is evaluated as true, Esper calls the listener's "update()" method which will in turn call the abstract method "executeAction()" included in the "Action" class.

Each child class of "Action" is required to implement the "executeAction()" method in order to properly be used as an action by the decision engine.

One of the most special built-in actions is the ReturnAction. The main purpose of it is to return in a programmatic manner a list of actors for each rule that has been evaluated as being true. Because multiple rules can contain this type of action, a map is used in order to distinguish the actors of each rule. The special method "insertInputDataWithReturnValue()" actually returns an object of type Map< String, Map< String,Object > > where the key of the first Map object is the Rule's ID and the key of the second Map object is the name of the actors which is associated with its value. We will refer to the returnable result as being the values map. Once the input data is inserted into the decision engine the method calls "getReturnValue()" method of ReturnValue object.

Implementing this action proved to be challenging because "insertInputDataWithReturnValue()" method had to wait until all the actors were retrieved before being returned. Therefore a CountDownLatch[2] object was used as a signal to notify when the actors are ready to be returned. When attaching a ReturnAction to a rule, the CountDownLatch would be incremented and it would be decremented each time the action executed. During execution, the actors of the rule would be fetched and inserted into the values map.

Listing 15 describes how ReturnAction works internally.

2 http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html

Listing 15: The Implementation of Return-Value Action

```
public class ReturnAction extends Action {
    @Override
    public void executeAction(String ruleID, Map<String, Object>
        actorMap) {
        // If nobody is waiting for a return value then there is
            no need to execute action.
        // ReturnAction can also be triggered by inserting data
            not in a programmatic manner.
        if (doneSignal.getCount() == 0){
            return;
        }
        // Retrieve the actors and insert values in the map
        for (String actorID : actorsToReturn){
            if (!actorMap.containsKey(actorID)){
                // Actor can not be returned.
                // It is not defined in the actors field of the
                    rule.
                return;
            }
            else{
                if (valuesMap.containsKey(ruleID)){
                    valuesMap.get(ruleID).put(actorID,actorMap.
                        get(actorID));
                }
                else{
                    Map<String,Object> newValue =
                        new HashMap<>();
                    newValue.put(actorID,actorMap.get(actorID));
                    valuesMap.put(ruleID,newValue);
                }
            }
        }
        // Decrease CountDownLatch signal
        doneSignal.countDown();
    }
    /* .... */
    public Map<String,Map<String,Object>> getReturnValue() {
        try {
            // Waiting until all returnAction are executed or
                timeout expires
            doneSignal.await(waitTimeout, TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Map<String,Map<String,Object>> aux =
            new HashMap<>(valuesMap);
        valuesMap.clear();
        return aux;
    }
}
```

4.3 WEB SERVER MODULE

We wanted the decision engine to be easily deployable as a web server. For this reason, we have chosen to use Spring Boot [5] which makes it easy to create Spring based applications that the user can "just run".

The framework already embeds Tomcat,Jetty or Undertow directly which means that there is no need to deploy any WAR files. The user only has to build a single executable JAR file that contains all the necessary resources and dependencies. This feature makes it very easy to deploy the service as a web application through the whole development lifecycle across multiple environments.

A Java class called "EngineController" is responsible for handling GET requests using the "RequestMapping" annotation provided by Spring Boot.

Listing 16 illustrates how input data insertion through an URL's query is implemented in the decision engine. It maps "/insert_input" to the "insertInput()" method which will process the data passed as request parameters and insert it into the decision engine.

Listing 16: Implementation of URL Query Data Insertion

```
@RestController
public class EngineController {
    private DecisionEngine engine = FactoryDecisionEngine.
        getSingletonInstance();
    @RequestMapping("/insert_input")
    public ResponseEntity<String> insertInput(
        @RequestParam(value = "inputId") String inputId,
        @RequestParam Map<String, String> dataMap)
    throws ControllerException {
        try {
            // DataMap includes the inputId because it is
                contained in the URL query.
            // We remove it in order to pass the map forward to
                the decision engine.
            dataMap.remove("inputId");
            Map<String,Object> newDataMap = engine.
                castToInputDataType(inputId,dataMap);
            engine.addInputData(inputId, newDataMap);
        }
        catch (EngineException e){
            throw new ControllerException(e.getMessage());
        }
        ResponseEntity<String> responseEntity =
            new ResponseEntity<String>(HttpStatus.ACCEPTED);
        return responseEntity;
    }
}
```

Another mapping is implemented for "/monitor" in order to serve the web page monitor with which the user can view the state of the engine.

4.4 FETCHER MANAGER MODULE

The module is responsible for managing all fetchers which are used in order to retrieve data from an external endpoint. The retrieval of the data is implemented using the "Google HTTP Client Library for Java" library.[3] After obtaining the data, the agent checks to see if a parser was defined by the user. If it was defined then it is used to parse the retrieved data. Otherwise, the agent has to convert the data into a format accepted by Esper. For parsing JSON strings, we use JSON.simple library which is a simple Java toolkit for encoding and decoding JSON.[4] It is mandatory that the field names and values retrieved from the external endpoint must match with the ones defined in the configuration of the engine.For example, if the retrieved data contains a field name which is not defined in the input domain or the type of the field's value is not the same then the decision engine toggles an error and rejects the data.

4.5 DECISION ENGINE MODULE

This module is represented by the "DecisionEngine" Java class and it is the main entity in configuring the decision engine programmatically. It contains many public methods which act as an API for the user. A few examples of methods can be

- public void setConfigurationFile(String filePath)
- public void initializeEngine()
- public void addNewRule(RuleModel ruleModel)
- public void addRuleListener(String ruleID, RuleListener listener)
- public Map<String,Map<String,Object» addInputDataWithReturnValue(String inputID, String actionID,Map<String, Object> dataMap)
- public void startWebServer()
- public void shutdown()

The class is practically a wrapper around an instance of Esper. Every configuration made or data inserted is converted and passed to Esper which acts as the main component for rule evaluation and input data processing. Every configuration component is associated to a Java

3 https://github.com/google/google-http-java-client
4 https://code.google.com/p/json-simple/

class and all instances are contained in a Map object in which the key is represented by the ID of the component. Each of these Map objects are obtained from the Configuration Parser module and can be edited by the DecisionEngine class. Other modules require that the objects are passed to them in order to properly run (example: Fetcher Manager).

Listing 17 represents the logic of initializing the decision engine. It first checks if a configuration file has been specified. If not, it initializes the engine with an empty configuration. Otherwise, it sets up the Configuration Parser module and creates the four Map objects which are associated to the configuration components. The next is to initialize Esper and configuring it based on the definition retrieved from the file.

Listing 17: Initialization Logic of the Decision Engine

```
1  public class DecisionEngine {
2      private final String ENGINE_ID = "esperEngine";
3      private String configurationFilePath;
4      private ConfigurationParser confParser;
5      private EPServiceProvider epService;
6      private EPRuntime epRuntime;
7      private Map<String, InputData> inputMap;
8      private Map<String, FetcherData> fetcherMap;
9      private Map<String, RuleData> ruleMap;
10     private Map<String, Action> actionMap;
11
12     /* .... */
13
14     public void initializeEngine(){
15         if (configurationFilePath == null){
16             // Setting up engine with empty configuration
17             Configuration cepConfig = new Configuration();
18             epService = EPServiceProviderManager.getProvider(
                   ENGINE_ID, cepConfig);
19             epRuntime= epService.getEPRuntime();
20             return;
21         }
22
23         // Setting up engine from configuration file
24         confParser = new ConfigurationParser(this,
               configurationFilePath);
25         confParser.parseJsonAndValidate();
26
27         inputMap = confParser.getInputMap();
28         fetcherMap = confParser.getFetcherMap();
29         ruleMap = confParser.getRuleMap();
30         actionMap = confParser.getActionMap();
31
32         // Use the configuration components to setup ESPER
33         Configuration cepConfig = new Configuration();
```

```
34
35          // Add input data types
36          for (String inputID : inputMap.keySet()){
37              InputData input = inputMap.get(inputID);
38              cepConfig.addEventType(input.getInputID(),input.
                    getTypeMap());
39          }
40
41          // Setup the rule engine
42          epService = EPServiceProviderManager.getProvider(
                ENGINE_ID, cepConfig);
43
44          // Define rules
45          RuleManager ruleManager = new RuleManager(ruleMap,
                actionMap,this);
46          ruleManager.addRulesToEngine(epService);
47          addLogToHistory("[CONFIG] - Rules and actions defined
                successfully.");
48
49          epRuntime= epService.getEPRuntime();
50
51          // Define fetchers
52          fetcherManager = new FetcherManager(fetcherMap,inputMap,
                this);
53          fetcherManager.startFetchers();
54      }
55
56      /* .... */
57  }
```

5 DECISION ENGINE RESULTS

Correctness is clearly the prime quality.
If a system does not do what it is supposed to do,
then everything else about it matters little.

— Bertrand Meyer

In the following chapter we talk about the achievements of the decision engine and the current features that are fully implemented. We compare our current version of the engine with other rule-based software so we can highlight the main advantages of our project. We also analyze how the Adobe prototype is configured and how can our own solution offer a better way of configuration.

5.1 TESTING

When it comes to testing the decision engine, we have designed a test plan which covers multiple test scenarios, each scenario having a number of test cases. Most of the test cases are done using the black box testing technique. That means that we have not considered the internal system design of the engine and the tests were based on requirements and functionality. To be more exact, we focused on checking if the decision engine is capable of covering the three use cases and the basic requirements of a rule engine.

The following tables describes the testing scenarios that were chosen and the test cases for each scenario.All passed successfully.

Table 1: Test Scenarios

ID	Test scenario	Test cases
TS001	Validate the initialization functionality of the engine when using configuration file.	4
TS002	Validate the functionality of a data fetcher	2
TS003	Verify the user can configure the application programmatically.	3
TS004	Verify the capability of the engine to be deployed as a web application server.	2
TS005	Validate the decision capabilities of the engine	2
TS006	Validate the rule definition functionality.	2
TS007	Verify integration of Esper.	3

Table 2: Test Cases

Scenario ID	Case ID	Test case	Actual result
TS001	TC001	Validate parsing of input definition	✓
TS001	TC002	Validate parsing of data-fetcher definition	✓
TS001	TC003	Validate parsing of rule definition	✓
TS001	TC004	Validate parsing of decision definition	✓
TS002	TC001	Validate the retrieving functionality.	✓
TS002	TC002	Validate the insertion of data in the engine.	✓
TS003	TC001	Verify the insertion of input data programmatically	✓
TS003	TC002	Verify the capability of defining rules programmatically	✓
TS003	TC003	Validate shutdown capability.	✓
TS004	TC001	Validate web server initialization.	✓
TS004	TC002	Verify the capability of inserting data through the URL's query.	✓
TS005	TC001	Validate built-in actions	✓
TS005	TC002	Validate custom action	✓
TS006	TC001	Validate the insertion of a rule which uses a comparison operator	✓
TS006	TC002	Validate the insertion of a rule which uses a comparison operator and logical operator.	✓
TS007	TC001	Validate insertion of data into Esper	✓
TS007	TC002	Validate definition of rule into Esper	✓
TS007	TC003	Verify if Esper triggers a rule when data is inserted.	✓

5.2 COMPARISON WITH OTHER RULE ENGINES

Based on the features that we have implemented in the decision engine, we can easily create a comparison with other existing rule engine software. The reason for comparing with rule engines is because there are no similar implementations of an engine which offer a support for decision making. Therefore we only focus on data processing, configuration, executing decisions and rule evaluation when doing the comparison.

The following table illustrates the difference between the decision engine and other rule engines in terms of features.

Table 3: Engine Comparison Based on Features

Feature	Decision Engine	Jess	Esper	Drools	CLIPS	Nools
Can define rules from a specialized file	✓	✗	✓	✓	✗	✓
Can define rules in a programmatic manner	✓	✓	✓	✓	✓	✓
Can define types of input data from a specialized file	✓	✗	✗	✗	✗	✓
Can define types of input data in a programmatic manner	✓	✓	✓	✓	✓	✓
Deployable as a web application	✓	✗	✗	✗	✗	✗
Web page to view the state of engine	✓	✗	✗	✗	✗	✗
Can insert input data using URL parameters	✓	✗	✗	✗	✗	✗
Can define own custom actions	✓	✓	✓	✓	✓	✓
Contains a built-in set of actions	✓	✗	✗	✗	✗	✗
Can automatically retrieve data from an external source	✓	✗	✗	✗	✗	✗

We can observe that the decision engine offers many more ways of configuration and it is easy to manipulate it so that it fits the user's needs. Dynamic is one of the key words that we have focused on when developing the engine and looking at the table above, we can confirm with confidence that we have achieved our goals.

5.3 CONFIGURATION ANALYSIS OF THE ADOBE PROTOTYPE AND THE DECISION ENGINE.

Both the decision engine and the prototype developed by Adobe can be initialized using a configuration file written in JSON format.As an example, we consider that both software components need to be configured to retrieve a float number every 30 seconds from a website, process it and display a certain message if the number is between 0.60 and 0.80.

Listing 18 shows the configuration of the prototype for the above situation.

Listing 18: Adobe Monitor Prototype Configuration

```
{
  "fetchers": {
    "client-fetcher": {
      "class": "com.example.ReportFetcher",
      "args": ["https://example.com/endpoint/client", 30]
    }
  },
  "triggers":  {
    "authn-conversion": {
      "description": "Authentication Conversion Status",
      "class": "com.example.RatioThresholdTrigger",
      "args":  [0.80, 0.60]
    }
  },
  "message-templates": {
    "authn-conversion": {
      "warning":"[Authentication][WARNING] Low conversion.",
    }
  }
}
```

We notice that a fetcher called "client-fetcher" is configured. A Java class is specified and two hard-coded values are passed as arguments to it. The first argument represent the target URL and the second one is the interval for retrieving data.

The trigger is where the user specifies the condition for triggering the printing of the message. Again, a Java class is specified which takes as arguments the two numbers that represent the trigger condition. The actual condition of the trigger is not displayed in the configuration file but rather is implemented inside the "RatioThresholdTrigger" class. This is a big disadvantage because modifying the condition implies modifying the Java code.

The last field is message-templates in which the actual message is defined. The association between a message and a trigger is done by name. Both fields need to have the same name in order for the trigger to know which message-template to call.

Overall, the configuration of the prototype is quite small in terms of size but it is very static. The user has very little freedom in defining more complex rules. Input data is not even defined in the configuration file. The user is responsible of writing the logic for "ReportFetcher" class and the "RatioThresholdTrigger" class in order to successfully retrieve the float number and evaluate if the value is between the two numbers provided as arguments.

Listing 19 illustrates how the decision engine is configured for the situation described at the beginning of the section.

Listing 19: Adobe Decision Engine Configuration

```
{
  "input":[
    {
      "input-id":"auth-input-data",
      "data":[
        {"name":"auth-rate",    "type":"float"},
      ]
    }
  ],
  "rules":[
    {
      "rule-id":"auth-rule-warning",
      "input-domains":["auth-input-data"],
      "actors":[ "auth-rate"],
      "condition": "auth_rate < 0.80 && auth_rate > 0.60",
      "actions" : ["print-warning-msg"]
    },
  ],
  "actions":[
    {
      "action-id":"print-warning-msg",
      "action-type":"built-in",
      "class":"com.adobe.primetime.adde.output.PrintMessageAction
          ",
      "arguments": {
        "message": "[Authentication][WARNING] Low conversion",
        "target": "STDOUT",
      }
    }],
  "fetchers":[
    {
      "fetcher-id":"dataFetcher",
      "receiver-input-id":"auth-input-data",
      "url":"https://example.com/endpoint/client",
      "interval":"30s"
    }
  ]}
```

The whole configuration of the engine is done totally in the JSON file. No Java classes are needed to be written in order to achieve our goal.

Compared with the prototype's configuration, the decision engine offers the user a more flexible way of interaction. It also makes the whole process of configuring the software very straight-forward and easy to understand.

5.4 WEB-MONITOR

The Web-Monitor is a great feature for the user who wants to always view and manage the state of the decision engine when it is deployed as a web application. In this particular use case, the purpose of the Web-Monitor was to offer an easy to use web page in which the user can easily interact with the engine anytime and anywhere.

Figure 2 represents a screenshot of the web page. It is consisted of two main components "Engine Configuration" and "Engine Monitor".

The first component offers four tabs in which the user can view the current configuration and one last tab with which data can be inserted into the engine by completing an auto generated web form.

The second component displays different messages which describes what is the current state of the engine and what is it currently doing. The monitor component updates periodically in real time. On the bottom of the monitor we can find a button for turning off the logging and another one for shutting down the engine. Finally, a message about the status of the engine is displayed between the two buttons. The message can either be "Running" or "Not running".

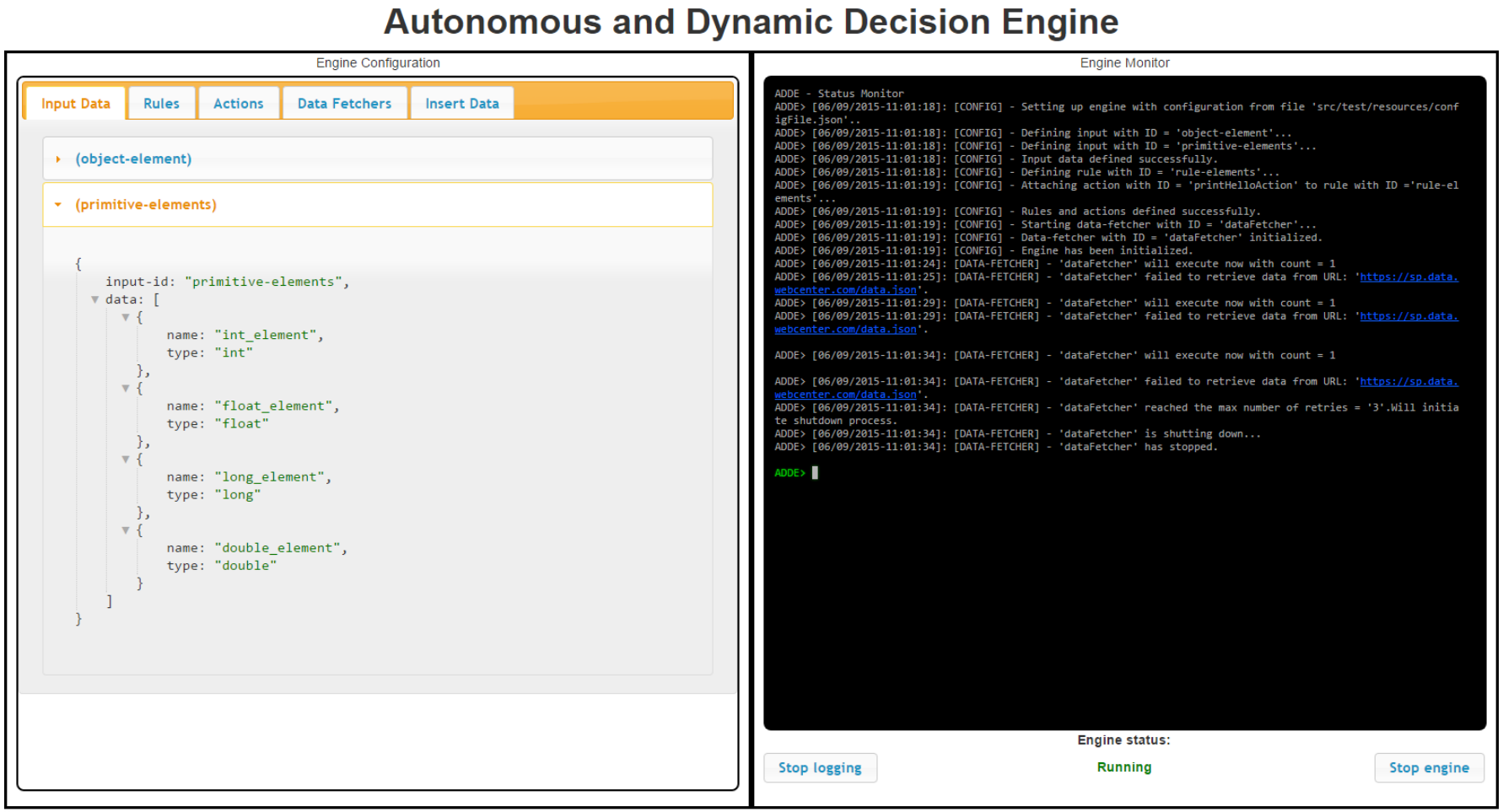


Figure 2: Decision Engine Web Monitor

Figure 3 shows how data can be inserted using the "Insert Data" tab from the "Engine Configuration" component. On the top of the tab there is a menu which can be used in order to select the input-domain. The decision engine requires the user to specify to which input-domain is the data meant for. Once the selection is made, a web form will be generated below the menu and the user will be able to complete it with data. When all fields are completed, the data is sent to the decision engine by pressing the "Send Data" button.

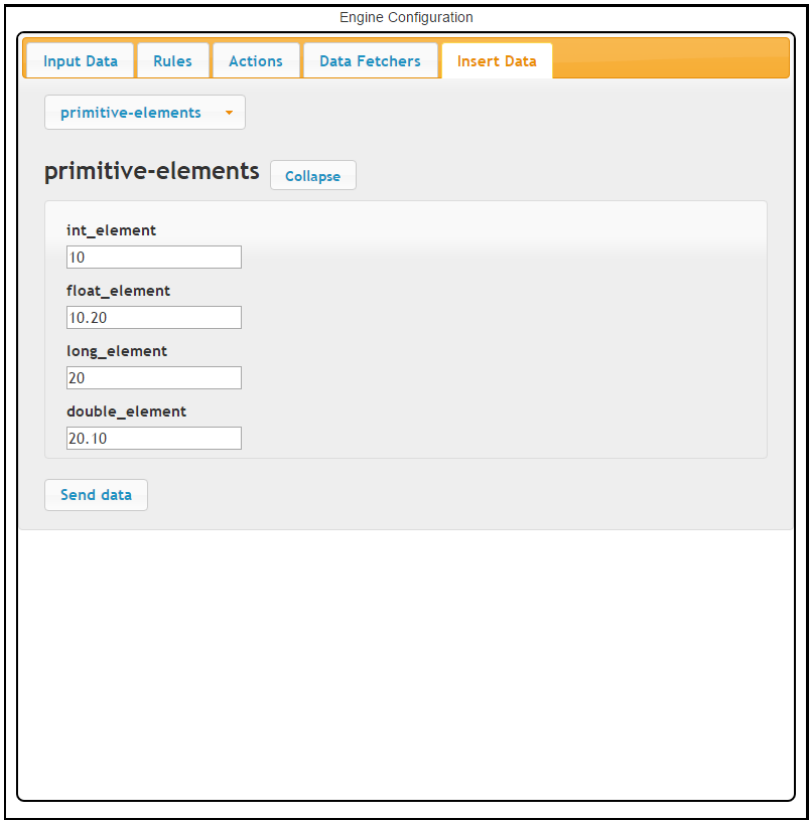


Figure 3: Data Insertion Through Web Monitor

5.5 USAGE AS A LIBRARY

One of the use cases for the decision engine is to be used as library in any of the user's projects. Therefore, the "DecisionEngine" class acts as an API for configuring and initializing the engine. Every modification of the engine can be done programmatically through that class.

In the following listing, we highlight how one can use the decision engine as a library through its API. We set up an instance of the decision engine with an empty configuration. After that, we define an a simple input data and a rule which has a built-in Print-Message action attached to it. Once everything is configured, we start the engine and begin inserting data into it. Finally, we shut it down so that all resources are freed.

Listing 20: Using the Decision Engine Java API

```
public void usageAsALibrary() {
    // Get decision engine instance and initialize with an empty
        configuration
    DecisionEngine decisionEngine = FactoryDecisionEngine.
        getSingletonInstance();
    decisionEngine.initializeEngine();

    // Define primitive elements
    InputDefinitionModel inputDefModel = new InputDefinitionModel
        ();
    inputDefModel.setInputID("primitive-elements");

    DataDefinitionModel dataDefModel = new DataDefinitionModel();
    dataDefModel.addDefinition("int_element", "int");
    dataDefModel.addDefinition("float_element", "float");
    dataDefModel.addDefinition("long_element", "long");
    dataDefModel.addDefinition("double_element", "double");

    inputDefModel.setDataDefinition(dataDefModel);
    decisionEngine.addInputDefinition(inputDefModel);

    // Define rule
    RuleModel ruleModel = new RuleModel();
    ruleModel.setRuleID("rule-elements");
    ruleModel.addInputDomain("primitive-elements");
    ruleModel.addActor("int_element");
    ruleModel.setCondition("int_element > 0");
    ruleModel.addAction("printHelloAction");

    decisionEngine.addNewRule(ruleModel);

    // We use a HashMap for associating the values with the
        proper names of the input data
    Map<String, Object> elementMap = new HashMap<>();
    elementMap.put("int_element", 10);
    elementMap.put("long_element", 20);
    elementMap.put("float_element", 10.20);
    elementMap.put("double_element", 20.10);

    // Insert data
    decisionEngine.addInputData("primitive-elements", elementMap)
        ;

    // Shutting down engine
    decisionEngine.shutdown();
}
```

We can observe that using the decision engine as a library is very easy and straightforward. The DecisionEngine class was specially designed so that it wraps around Esper covering the complicated SQL-

type statements which were required to define rules for the complex event processing component. We believe that we have managed to offer a Java component that is simple to integrate in any projects.

6 CONCLUSION AND FUTURE DEVELOPMENT

... with proper design, the features come cheaply.
This approach is arduous, but continues to succeed.

— Dennis Ritchie

In this chapter we present the end conclusion of this thesis and we describe what possible improvements can be added to the decision engine.

6.1 CONCLUSION

The project described how to build a rule engine that focuses more on decision-making and aims to have a flexible way of configuration. By using Java as the programming language, JSON format as the configuration syntax and Esper as the component responsible for rule validation, we managed to implement a component which can no longer be considered a rule-based solution but rather a decision-based one.

The comparison between the decision engine and the Adobe prototype highlights the fact that we have achieved our objective defined at the beginning this thesis. The JSON format and the structure of the configuration make it very simple for the user to understand what is the current state of engine and what fields need to be modified in order for the state to be changed.

We have showed in the previous chapters that the engine is dynamic by offering multiple ways of configuration and by being very easy to modify in order to cover multiple use cases. We consider that our decision-based solution is also autonomous by being capable of running independently as a standalone component.

6.2 FUTURE DEVELOPMENT

At the moment, the decision engine can be considered to be a minimal working product which is enough to be used as a proof a concept. It contains some limitations regarding parsing the configuration file, defining more complex rules and input data. Listing all of these limitations would not interest the reader too much and therefore we talk about what possible features can be added to the decision engine. We have narrowed it down to four major improvements that can be developed in the future.

6.2.1 *Using ANTLR for Parsing Rule Condition*

Right now, we use our own algorithm for parsing the condition string of a rule. But the algorithm has some limitations. For example, it fails to parse the condition if an opened or closed bracket is not delimited by at least one space character: "(x > 2)" - Fail; "(x > 2)" - Success.

A proper parsing mechanism must be implemented so the user can have total freedom when writing a rule condition. We came to the conclusion that ANTLR (ANother Tool for Language Recognition) might be a good answer to this problem. It is a powerful parser generator for reading, processing or translating any type of structured text. It is widely used to build languages.From a grammar, ANTLR generates a parser that can build and walk parse trees. [1]

6.2.2 *Including New Properties to Rules*

At the moment, the template of a rule is quite minimal which means that it can be evaluated only based on the condition specified by the user. However, Esper includes multiple features which permits the user to easily create more complex rules. Those features can be easily wrapped by the decision engine so the user can take advantage of them when writing the configuration file or when using the API.

One of the features is Time Window which is a moving window extending to a specified time interval into the past based on the system time. It enables the engine to limit the number of input data considered by a rule when it is evaluated. It can be very useful for the user if he wants a rule to take in consideration input data that were inserted five seconds ago, for example.

Esper also support SQL-type group functions which can be also included in the decision engine. Example of group functions: AVG, COUNT, MIN, MAX, SUM.

6.2.3 *Enhanced Web-Monitor*

Right now, the user can only view the configuration of the decision engine through the Web-Monitor. An interesting add-on would be to have the possibility to edit the input data, rules and actions. Therefore, the user can have complete control over the decision engine.

6.2.4 *Class Based Input Data*

An input domain can only have primitive types of data or a string type of data. We are considering modifying the decision engine so that it accepts Java classes as input data. The mechanism would be

1 http://www.antlr.org/

the same as in the definition of an action which is actually a Java class instantiated and loaded at runtime.

This improvement will make the decision engine more dynamic and gives the user the possibility to user their own data structures as input data.

BIBLIOGRAPHY

[1] The Rete Algorithm URL : http://www.jessrules.com/docs/71/rete.html [Online; accessed 19-August-2015].

[2] CLIPS - A Tool for Building Expert Systems URL : http://clipsrules.sourceforge.net/WhatIsCLIPS.html [Online; accessed 23-August-2015].

[3] CLIPS User's Guide - Chapter 1 : Just the Facts URL : http://clipsrules.sourceforge.net/documentation/v630/ug.pdf [Online; accessed 23-August-2015].

[4] Yuxin Ding, Qing Wang, Jiahua Huang; "The Performance Optimization of CLIPS", Hybrid Intelligent Systems, 2009. HIS '09. Ninth International Conference on , vol.1, no., pp.417,421, 12-14 Aug. 2009

[5] Spring Boot URL : http://projects.spring.io/spring-boot/ [Online; accessed 3-September-2015].

[6] Minsu Jang, Joo-Chan Sohn; "Bossam: An Extended Rule Engine for OWL Inferencing", Rules and Rule Markup Languages for the Semantic Web, pp 128-138, 2004-01-01