INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Platforma průmyslové spolupráce

CZ.1.07/2.4.00/17.0041

Název

Drools Fusion and Utilization of Complex Event Processing in Web Applications

Popis a využití

- studijní materiál pro platformu Drools, komplexní zpracování událostí (CEP)
- use case pro využití CEP v oblasti webových aplikací
- výuka: pokročilá Java

Jazyk textu

- anglický

Autor (autoři)

- Iva Žáková

Oficiální stránka projektu:

- http://lasaris.fi.muni.cz/pps

Dostupnost výukových materiálů a nástrojů online:

- http://lasaris.fi.muni.cz/pps/study-materials-and-tools

Contents

1 **Introduction** 1
2 **Drools Expert and Drools Fusion** 3
2.1 *Overview of Drools* 3
2.1.1 Introduction and history 3
2.1.2 Drools projects 3
2.1.3 Alternatives to Drools 5
2.2 *Rule engine* 6
2.2.1 Production and Hybrid Rule Systems 6
2.2.2 Working memory and agenda 8
2.2.3 Advantages and disadvantages of using a rule engine 9
2.3 *Rete Algorithm* 10
2.3.1 Introduction 10
2.3.2 Rete network 11
2.3.3 Optimization and effectiveness 15
2.4 *Constructing the rules* 16
2.4.1 Assembling resources 16
2.4.2 Drools Rule Language 18
2.5 *Drools Fusion* 21
2.5.1 Complex event processing 21
2.5.2 Event definition 22
2.5.3 Clocks 23
2.5.4 Temporal operators 24
2.5.5 Sliding windows 25
2.5.6 Entry points 26
2.5.7 Event processing modes 27
2.6 *Drools 6* 27
3 **Motivation to include CEP in web applications** 29
3.1 *The role of the user in the aims of websites* 29
3.2 *Benefits of incorporating Complex Event Processing* . . 30
3.3 *Existing solutions* 32
4 **The application** 33
4.1 *Overview* 33
4.2 *Analysis and design* 34
4.2.1 Roles 34

4.2.2 Data model 36
4.2.3 Plugging Drools in 37
4.3 *Implementation* 38
4.3.1 Java EE 38
4.3.2 Drools assembly at the start of the application . 40
4.3.3 Overview of generating and processing events . 42
4.3.4 Firing the rules 44
4.3.5 Use cases of Complex Event Processing utilization 46
4.3.6 Testing of the rules 51
4.4 *Deployment* 52
4.4.1 OpenShift 52
4.4.2 An application server 53
5 **Conclusion** 54
Bibliography 56
A **Events diagram** 61
B **Deploying the application to JBoss Application Server 7** . 62

1 Introduction

As the impact of information technologies on society increases every year, constructing more and more complex and sophisticated systems is inevitable. The automation of human processes or appropriate computer assistance combined with monitoring and accountability is one of contemporary concerns. There are also frequent requirements not to only enhance an already established system, but to extend it in ways that enable the utilization of present logic and data for different purposes such as additional processing, data mining or fraud detection.

The difficulties with imperative languages and the traditional approaches to building such a system and augmenting it in the way previously mentioned reside in the middle layer. A complex system is usually comprised of at least persistence, logic and presentation layers, whereas each can represent more layers, views or components. The logic layer is the critical scope of several matters such as processes in general, decision making, current aspects evolving or reusing, and adding additional features. Using an imperative language in a classical way in a complicated system might eventually lead to unnecessary long and chaotic code, performance decrease and the introduction of new bugs [1].

One of the possible solutions is to use a rule engine, which can separate the appointed logic into more understandable, maintainable and reusable rules, which can be also maintained and verified by domain experts. This goal of this thesis is to describe the core of the Drools engine, as well as its module called Drools Fusion enabling Complex Events Processing capabilities, and explore the possible utilization of Complex Event Processing within web applications. The practical conclusions are demonstrated in a Java web application using Drools, which handles the user actions as events and processes them in rules.

The structure of the thesis text is separated into several chapters, where the first part covers the description of technologies and motivation concerning the topic, and the second part describes the composed application. The first chapter represents the introduction and the last one the conclusion.

The second chapter focuses on the Drools engine and its essential components utilized to develop this thesis. This chapter commences with an overview of the Drools project and the platform. After the introduction to rule engines, it pursues the explication of the Drools core by describing the employed algorithm. Following sections target the composition of rules and the module for event processing. The chapter is concluded with a description of the upcoming version of Drools.

The third chapter examines the motivation of including Complex Event Processing into the scope of web applications. It discuss the importance and role of a user in this area, the benefits that the inclusion might provide and the existing solutions.

The fourth chapter describes the developed application. It discusses the analysis and design of the application, the implementation and deployment. The characterization and description of the implementation is focused on the components regarding rules and event processing, and applied use cases.

2 Drools Expert and Drools Fusion

2.1 Overview of Drools

2.1.1 Introduction and history

Drools is an open source project written in Java, licensed under the Apache License, Version 2.0[1]. The current fifth version of Drools released in May 2009 introduced the Business Logic integration Platform (BLiP), which provides a unified and integrated platform for rules, workflow, event processing and automated planning optimization. The project as such is maintained by the community, providing a new release every couple of months that includes new features and bug fixes. A productized version involving sanitized community releases and support called JBoss Enterprise Business Rules Management System (BRMS) is offered by Red Hat, Inc [2].

First work on the rule engine of Drools began in 2001; this first version was never released due to constraints engendered by the brute force linear search approach. In Drools 2.0, the concept was changed to be loosely based on the Rete algorithm, and the federation into JBoss in 2005 enabled the development of enhanced Rete implementation [3]. The performance was significantly improved and the community started to emerge. As the development of Drools gradually progressed through the versions, Drools specific aspects were introduced such as Drools Rule Language, more performance improvements of the algorithm were accomplished and several other projects were incorporated into Drools [1]. The latest version is currently 5.5, released in November 2012; it is this version which is explicated following. Drools 6 is currently being developed, which will present not only new features, but also a new algorithm as well.

2.1.2 Drools projects

The platform is currently composed of five main modules:

Drools Expert represents the rule engine itself. It is the declarative, rule based coding environment, which is used to define, execute

---

1. See `http://www.apache.org/licenses/LICENSE-2.0.html`.

and maintain the rules. Thus it can be considered as the core of the platform [4]. In a simplified way, it is not possible to use, for instance, only Drools Fusion per se. To develop correct and efficient rules, it is also advisable to comprehend how the rule engine works.

Drools Fusion represents the module enabling the event modelling capabilities. Data processed by the rule engine can consequently be perceived as events and not only as the simple facts. The ability to identify temporal relationships between events, to compose and aggregate them, and other aspects discussed further, make it possible to implement standard Complex Event Processing scenarios such as fraud detection or automatic trading. However, the application of a rule engine can facilitate creating them, produce another perspective on the subject, or assist in tackling entirely new problems.

jBPM engine enables the modelling and design of the business processes by means of BPMN 2.0 specification. Furthermore it provides the executional environment for processes and makes it possible to describe them in a formal language [5]. The current jBPM5 resulted from the merger of the original jBPM with the project Drools Flow. That enabled easy integration of rules into business processes. Moreover, there are multiple tools available in the form of Eclipse plugin or web editor to easily model and visualize the processes.

Drools Guvnor represents a centralized repository for keeping various knowledge assets such as business processes, models or rules, with a web graphical user interface also able to be used by domain experts, analysts or users with no programming experience. It contains a set of process-oriented tools and also tools for the authoring phase of created assets. The storing of rules and other assets is managed by a Java Content Repository (JCR) [6], which supports version and access control.

Drools Planner, renamed OptaPlanner in the upcoming version, represents the module for optimization and solving discrete optimization problems. It uses a system of efficient score calculation combined with optimization heuristics and metaheuristics. Because of this, it can explore solutions during an established amount of time and solve any NP-complete problem in a reasonable period of time [7]. Frequent use cases are, for instance, resource scheduling, queue planning or traveling salesman problem.

Another valuable component is the Drools and jBPM plugin for

Eclipse IDE. It facilitates writing and debugging rules and processes with syntax assistance or visualization.

To conclude, the platform provides the means to model complex behaviour of a broad spectrum of problem situations. The flow of data, information or logic can be decomposed into processes, perceived as - or transformed into - events, solved and optimized, or only processed by the rule engine. This thesis focuses on the core of the platform - the rule engine itself (Drools Expert) and the module enabling the event modelling capabilities (Drools Fusion).

2.1.3 Alternatives to Drools

Owing to the multiple area specializations of the platform, alternatives to Drools vary according to the purpose of use.

The noted paid option for a rule engine is WebSphere ILOG JRules BRMS from IBM. In comparison with Drools Expert, it supports Java, .NET and COBOL environments and contains a richer feature set for business analysts to be included more in the rule development and management lifecycle [8]. Other rule engine options are, for instance, Microsoft InRule, FICO Blaze Advisor Business Rules Management or Decision Management System [1].

A great deal of both paid and open source workflow and process engines can be found, the most popular open source options are as follows: Activiti, Bonita Open Solution, or ActiveBPEL [9].

The range of Complex Event Processing (CEP) solutions is also quite large. The Forrester Research[2] evaluation of CEP platforms from 2009 [10] established 114 criteria to compare ten vendors offering commercial solutions and Progress Software and Aleri came out as the strongest. The only vendor in this evaluation also offering an open source version of the product was EsperTech with Esper. Esper in comparison with Drools Fusion uses SQL-like syntax, which can be easier for programmers to learn, contains fewer temporal operators and may have better performance results within a short period of time [11].

Overall, there are a great deal of other options, which can be more effective or appropriate for certain problems where only one Drools

2. See `http://www.forrester.com/`.

module is required or the financial aspect is not an issue. The key factors of Drools are the facts that it is an open source solution and a unified platform, which enables it to combine several modules to create complex behavioural modelling.

2.2 Rule engine

2.2.1 Production and Hybrid Rule Systems

The term *rule engine* can cover a broad variety of systems, from ones using rules in the most simple form for instance for validation, to the complex systems where the logic processed by the rule engine is quite essential [12].

The Drools rule engine originated as a special type of rule engine known as a *Production Rule System* (PRS). A Production Rule System is often considered to be at the centre of an expert system, or is directly interchanged with the term. Since an expert system is a system that applies knowledge in the form of ontological model and declared rules defined by domain experts for the purpose of solving problem in a particular field [13], it would be misleading to describe the Drools rule engine as an expert system. To be accurate, it is a device to build expert systems.

The computational model of Product Rule System specifies the set of rules, where each rule is comprised of a condition and a consequential action. During the run of the system, the conditions are evaluated following the processed data and the consequential actions are executed. Therefore, each rule is in the following format:

```
when
  // condition
then
  // action
```

The important aspect to comprehend is the difference between the imperative *if... then* in the sequential code and the declarative *when... then* in rules. The rules do not specify a detailed sequence of steps. The rules declare the conditions specifying when the actions should be performed; the rule engine takes the rules, the data and

evaluates them in the most efficient way. Those rules with the condition evaluated to true have their action executed.

The data matched against the rules reside in memory in the form of facts. The matching itself is called *pattern matching* and is performed by an inference engine. The pattern matching can result in scheduling the executions - called the *firing* - of the rule actions by the agenda, and as a consequence, firing of rules can conclude in matching against other rules and engender firing those. This mechanism is referred to as *forward chaining* [12].

Figure 2.1 indicates the high-level view of a Production Rule System. The rules stored in Production Memory are matched against the facts that reside in Working Memory by the Inference Engine via pattern matching and can result in executions administered by the Agenda.

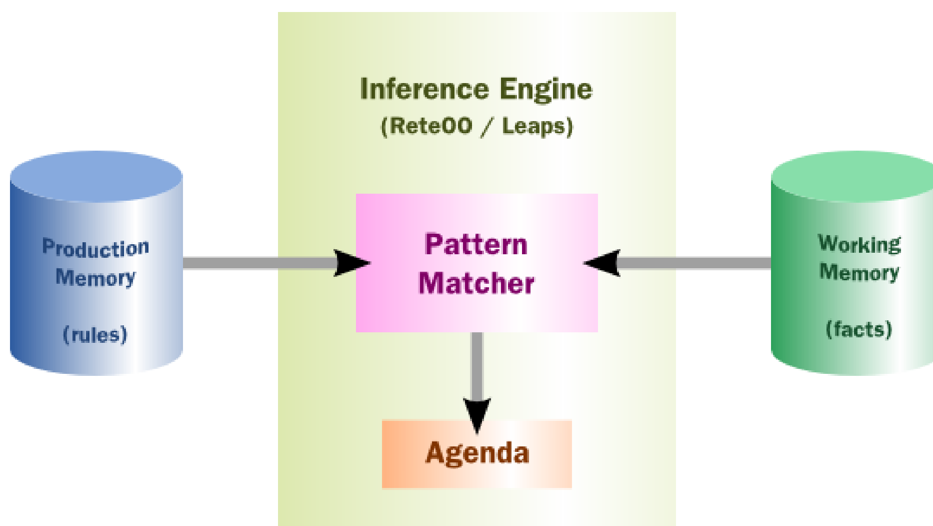

Figure 2.1: High-level view of a Production Rule System [12]

The described forward chaining could be characterized as reactionary and data-driven, as the asserting of facts into working memory results in executions. Another approach is called the *backward chaining*, which is goal-driven. The engine starts with a conclusion, which it tries to satisfy. If it cannot, it attempts to satisfy a subgoal conclusion, which would assist in satisfying an unknown part of the current goal. This activity then can continue recursively and ends

when the initial conclusion is proved or no other subgoals exist. An example of backward chaining system is, for instance, Prolog.

More reasoning techniques such as imperfect reasoning or defeasible logic exist [12]. Systems combining more types of these capabilities are referred to as *Hybrid Reasoning Systems* (HRS). Since Drools 5 also introduced backward chaining into its engine in the form of derivation queries, the current preferences incline to describe it as the Hybrid Reasoning System.

2.2.2 Working memory and agenda

The data and information in the object-oriented environment are represented by the object instances. When these instances are inserted into the Working Memory of the engine, in Drools terminology they become facts and henceforth can be evaluated against the declared conditions of the rules. In consequence of the insertion, a wrapper to the fact is created, which can be onwards used as the reference to update it or retract it. If a fact is modified, the update is required to be called tor the engine to re-evaluate the rules against the altered fact. A retract should be called provided that the information is no longer needed or would cause undesired behaviour.

When facts are inserted, updated or retracted from the Working Memory, the engine identifies the corresponding rules by means of pattern matching and creates the activations for the matching rules. The Agenda represents the list of all these activations, in other words of all the rules prepared to fire. It is important to note that the act of generating the activations does not execute the rules action consequences, nor that all the rules prepared to fire must necessarily do so. The activation list does not start firing until the appropriate method is called.

When this method is called and the agenda contains more than one activation, the engine needs to determine which activation should fire first. This decision is based on the conflict resolution strategy of the engine. The result of conflict resolution strategy can be based on many factors such as the attributes of the rule, the rule recency, which declares how many times has the rule fired, the complexity of the rule or the rules load order [14]. The selected activation is subsequently executed, which can result in altering some facts, re-evaluating the

rules against the new Working Memory state and as consequence creating new activations or removing the present ones.

After the process of the execution is finished, if the Agenda still holds some activations, the procedure continues with determining new rule activations to fire. This cycle carries on until there are no more activations. The whole process is outlined in the following Figure 2.2.

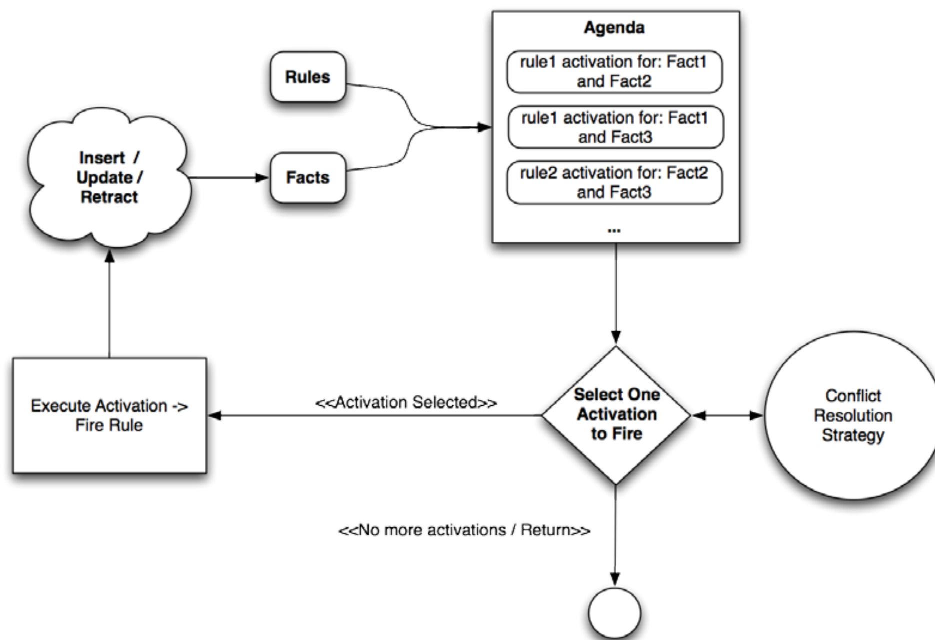

Figure 2.2: The outline of rules execution cycle [6]

The rule attributes provide additional impact on the Agenda behaviour such as avoidance of circular reactivations or assembling the rules into activation groups.

2.2.3 Advantages and disadvantages of using a rule engine

Many advantages result from the declarative approach. The maintainability is improved as the time can be allocated to what needs to be accomplished instead of how [15]. Furthermore, decomposing

the logic into rules may decrease the subsequent complexity and increase the flexibility owing to easy modification of the rules. The rule language can also enable the domain experts to be originators of the written logic and control its evolution.

However, switching to rule engine does not always have to be the right decision. To shift the thinking paradigm to a declarative rules style and to learn how to write the rules properly and effectively can be time consuming, which is not suitable for small projects where logic is not going to change in the future. Apart from this, the complex rules are more difficult to debug than traditional lines of codes, in the absence of highly advanced debuggers. In addition, it is not the best solution for situations where memory is the crucial factor, since a lot of calculations are stored for the engine to be efficient [1].

2.3 Rete Algorithm

2.3.1 Introduction

The speed efficiency and scalability of the Drools rule engine is engendered by the Rete algorithm on which it is based. The Rete algorithm [16], the name of which originates from the Latin word for 'net', represents a pattern matching algorithm for implementing production systems. The algorithm was designed by Dr Charles L. Forgy and first published in 1974. Since then, it has been modified and augmented by Dr Forgy himself and also many rule engines established on its basis [17].

The Drools rule engine uses the enhanced Rete algorithm called Rete-OO. It is an optimized implementation for object oriented systems including several types of schemes for reasoning with imperfect information and enables the architecture to be influenced by configuration parameters [18]. Whereas it has been mathematically proved to be faster and more scalable than the traditional *if... then* solution [14], the price for efficiency is in the higher memory consumption and usage due to a high amount of caching in order not to evaluate the conditions multiple times.

The inference engine applies the algorithm for pattern matching. In other words, the algorithm is responsible for matching the facts against rules and determining for which rules activations should be

created.

2.3.2 Rete network

The algorithm transforms the conditions of the rules into the Rete network. This discrimination network is represented by a rooted, acyclic and directed graph and is modified whenever a rule is altered, added or removed from the knowledge base. Its purpose is to efficiently filter data passing down the network.

The network is comprised of various node types. Suppose that the rule base consists only of the subsequent rule:

```
rule "Person older than 18."
  when
      Person(age > 18)
  then
      // consequence
end
```

This rule executes its consequence for a Person fact, where the fact has the attribute age greater than number 18 (the rules and their description are covered in section 2.4.2). The Rete network would be composed as depicted in Figure 2.3.

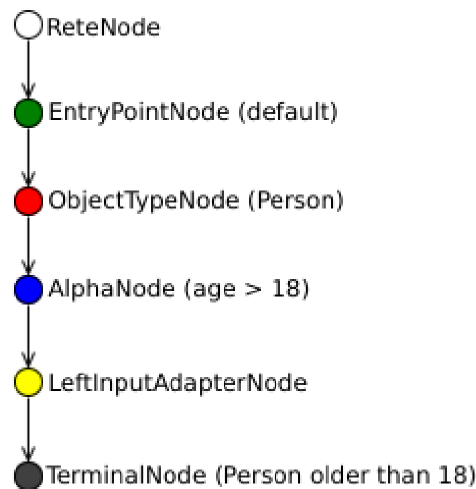

Figure 2.3: The Rete network for a simple rule.

A visualization graph representing the Rete network comprising of coloured nodes determined by their type can be generated by the Drools Eclipse plugin.

The root of each Rete network is represented by the node called the **`ReteNode`**. It can be perceived as the entrance to the network, and all the objects inserted to the memory must enter it.

Then follows the first level of discrimination, portrayed by one or many nodes of **`EntryPointNode`** type. The entry point mechanism enables the abstraction in the form of partitioning the working memory for facts and more importantly for event streams, which is covered in 2.5.6. In other words, objects entering the memory can be inserted either into the default entry point, or into a named one, which is reflected in the Rete network.

The ensuing level of discrimination is provided by at least one node of type **`ObjectTypeNode`**. This node performs filtering based on the fact type, which means that nothing but the facts of corresponding type are allowed to pass through. In this instance only objects of the type Person continue further down the network. Furthermore, following nodes will apply the constraints regarding the Person type.

The first level of matching is represented by the nodes of **`AlphaNode`** type. They have one input and define intra-element conditions, meaning that they evaluate constraints on single facts. The constraints may have various forms, such as literals, variables, inline evaluations or return values. Multiple constraints on the same type are expressed by more alpha nodes, each representing a single constraint. The order of the constraints in the rule is important.

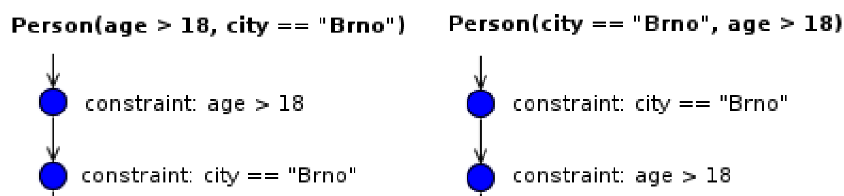

Figure 2.4: Rete network of transposing the constraints in the rule.

Transposing the constraints in the rule influences the concluding appearance of the Rete network, see Figure 2.4. For one thing, the more restrictive constraint should precede the less restrictive for the

rules to be efficient. More importantly, the order can affect re-usability of the rules in the more complex rule set.

The penultimate node in the Figure 2.3 is **LeftInputAdapterNode**. These nodes are used to create a tuple from single fact in order to enter some type of nodes.

The leaves of Rete networks are represented by nodes of **TerminalNode** type. They correspond to the action parts of rules, or to be more precise, to the activations of these rules, which should be assigned to agenda for future execution. Each rule introduces at least one `TerminalNode`, whereas a complex rule using the conditional disjunctive 'or' can have more than one of these nodes.

To conclude, if a Person fact was inserted into a default entry point, it would enter the Rete network in `ReteNode`, then it would continue through `EntryPointNode` and `ObjectNodeType` to `AlphaNode`. The `AlphaNode` would evaluate its constraint. Thereafter the situation would depend on the age attribute of the fact. If the age was less or equal to 18, the fact would not be propagated further and the insertion method would be finished. Otherwise the fact would pass through to the `LeftInputAdaptedNode`, where it would be transformed into a tuple and then reach the TerminalNode, which would result in placing the "Person older than 18." rule on the agenda. If the fact was inserted into another entry point or was of another type, the advancement would end at the relevant node. The introduction of more convoluted aspects uses a rule base comprising of two the following rules:

```
rule "Person living in Brno."
  when
    Person(city == "Brno")
  then
    // consequence
end

rule "Young person owns an accommodation in Brno."
  when
    $person: Person(city == "Brno", age < 30)
    Accommodation(owner == $person)
  then
```

```
    // consequence
end
```

The first rule executes its action for Person fact, where attribute city is equal to literal 'Brno'. The second rule requires a couple of Person facts with the city attribute equal to 'Brno', age less than 30 and an Accommodation fact with the owner attribute equal to the prior Person fact. The Rete network of this rule base is depicted in Figure 2.5.

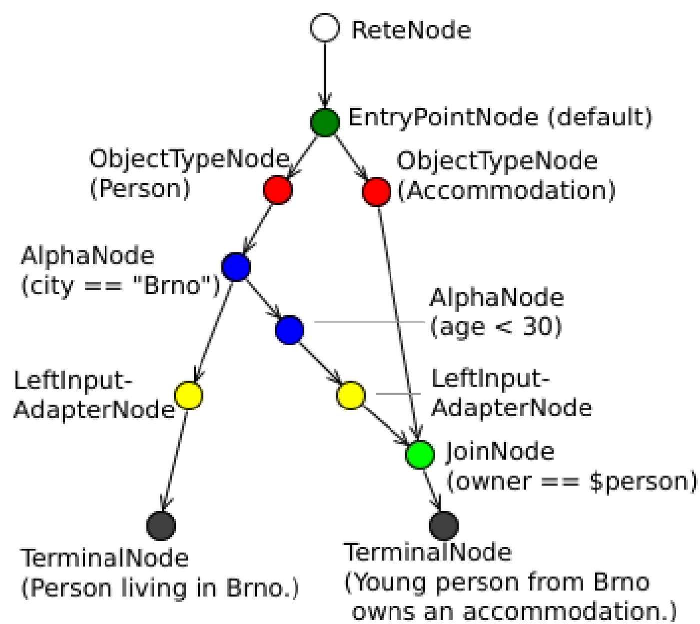

Figure 2.5: More complex Rete network.

Another important node type is **`BetaNode`**. These nodes have two inputs: one for tuples, one for facts, and define inter-element conditions. They enable comparison of facts or their attributes with each other.

The mentioned inputs are associated with memories which keep information about facts that have arrived. Later the facts can be evaluated against facts coming to the other input and the result propagated further.

There are various types of beta nodes. In the Figure 2.5 **`JoinNode`** is used, which is responsible for joining and evaluating a tuple and a

fact. The other types of BetaNode represent various rule constructs, the most used are NotNode, ExistsNode, AccumulateNode and CollectNode. More complex rules or rules using different constructs can also apply other types of nodes such as EvalNode or FromNode.

If a Person fact is inserted into the default entry point of the afore described rule base, it will pass through ReteNode, EntryPoint-Node, ObjectTypeNode for Person type to the first AlphaNode. If the constraint evaluation is successful, it will propagate to both the successor nodes. Within the left flow (in Figure 2.5) it continues through LeftInputAdapterNode to TerminalNode,which results in placing the activation of the first rule to agenda. Along the right flow it is evaluated in another AlphaNode. Supposing the evaluation of the constraint is successful, the fact is wrapped into a tuple in LeftInputAdapterNode and reaches the JoinNode. This tuple is added to the memory for tuple inputs of JoinNode and its fact memory is examined in order to find a match. Provided that there is no match, the proceeding is finished. If an Accommodation fact is inserted then, it passes through the ReteNode, EntryPointNode and ObjectTypeNode for Accommodation type to the JoinNode. There it is added to the memory for facts and the tuple memory is examined to find a match and evaluate constraints. If the process is successful, a tuple consisting of both the Person and Accommodation fact is constructed and continues to the TerminalNode, where it creates an activation of the second rule.

2.3.3 Optimization and effectiveness

It have been established and demonstrated in Figure 2.4 that the structure of the Rete network eminently depends on the order of conditions and constraints in the rule. As a consequence, the composition of the rules has a direct impact on the node sharing in the Rete network. The rules from previous example having their Rete network depicted in Figure 2.5 have except ReteNode, EntryPoint-Node and ObjectTypeNode also in common the AlphaNode. If the constraints for Person fact in the second rule were transposed, the network would contain an AlphaNode for each rule instead of one shared alpha node for both the rules. The node sharing enables the minimization of the size of the Rete network. For the algorithm to

be truly effective, the rules should share as many nodes as possible, making it possible to minimize the evaluations of the same conditions. Also in general, the condition and constraints of the rules should start with the more restrictive ones. Overall, the phase of designing the rules should not be rushed.

The aforementioned cases focus on inserting the facts. The activations of the rules are also re-evaluated in the case of modifying and retracting facts. The modification of a fact is achieved by a combination of retracting and inserting the fact. The technique called the asymmetrical Rete [1], which uses the tuple related to the fact to retrace the steps and retracts it from corresponding nodes, has been used for retraction since Version 5.

For the correct and effective functioning of the application it is important to emphasize the difference between insertion (or modifying and retraction) and firing the rules. Unless the firing the rules engenders in insertion, modifying or retraction, the utmost conditions processing occurs before the rules firing is even executed. The firing is responsible only for executing the actions of the activated rules in the agenda.

Whereas the original Rete algorithm started with only four basic types of nodes, the advanced Drools algorithm employs many node categories. To achieve a commendable performance the implementation uses techniques such as hashmap lookups for objects, caching results or indexing [12].

In conclusion, the algorithm provides an efficient way of evaluating the rules. As a real complex application can contain hundreds or thousands of rules, the network would be indubitably wide, but comparatively flat.

2.4 Constructing the rules

2.4.1 Assembling resources

In order to commence composing the rules and building the basis of Drools application, it is essential to comprehend the assembling of the resources. That is associated with three conceptions: Knowledge Builder, Knowledge Base and Knowledge session.

Knowledge Builder comprises parsers and semantic modules.

These elements enable the building of the binary representation of defined knowledge models. To put it in another way, the builder responsible for forming the knowledge packages from given resources. At present, it is possible to apply about a dozen of different resource types, such as types supporting processes, xml based documents or decision tables, which enable managing rules in a spreadsheet format. The discussed aspects focus on Drools Rule Language (DRL), where rules are usually stored in the *.drl* files.

Knowledge Base represents the container for compiled knowledge assets, such as rules, internal types or processes, which are grouped into knowledge packages. It allows not only their storage, but also their reuse in order to save computing time, since their creation is expensive [1]. To be more precise, it enables to establish the sessions, where their creation is in comparison inexpensive.

Knowledge Session represents the place where the rules run. By its means happens the interacting with the engine, such as inserting, updating or retracting facts and firing the rules. There are two types of session: stateless and stateful. The stateless session does not maintain the state between interactions. It is used for situations where a particular result needs to be computed. The stateful session maintains the state and is required where some previous information is necessary.

The detailed creation of the Knowledge Session is depicted in Figure 2.6.

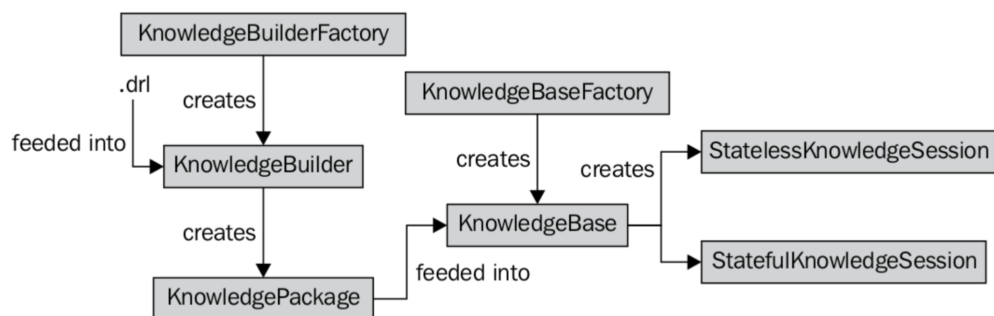

Figure 2.6: Thorough illustration of creating a Knowledge Session [1].

There are two possible ways of firing the rules. The first is by calling a `fireAllRules` method on a session, which results in im-

mediate firing of the activations placed on the agenda and potential activations created by them, followed by further continuation of the program. The second possibility is through the `fireUntilHalt` method, which fires an activation when it is created. The possible disadvantages of this solution are that it needs to be executed in a separate thread and is not recommended for certain scenarios.

2.4.2 Drools Rule Language

Rule files are represented by text files with *.drl* extension and they have usually following structure [12]:

```
package package-name

imports

globals

type declarations

functions

queries

rules
```

The structure of the file is not completely fixed. The file should start with package and imports; type declarations ought to precede their use, but functions, queries and rules can permeate. The elements are also not required to be in only one file. They can be spread across multiple files, for instance one holding only imports and declarations and one rules, or be separated into multiple rule files depending on logic. The important aspect is the order of their feeding into the Knowledge Builder.

Package represents the namespaces that are able to group elements from multiple files. In other words, all rule files appertaining to the same package are required to share the same package name.

```
package cz.example.pkg.first;
```

Import statements operate on the same basis as standard Java imports. Objects within the declared package are imported automatically. Objects from other packages must be imported or accessed by their fully qualified names.

```
import cz.example.pkg.second.Person;
import java.util.List;
```

Global variables enable objects to be reached within rules without insertion into session as facts. They are used in rule consequences for techniques such as logging, reporting or returning values. Their use in rule conditions is discouraged due to the fact that the engine is not notified about their modification.

```
global List results;
```

Type declaration has dual purpose. The first is declaration of metadata for already existing types. These metadata are used for reasoning process of the engine regarding the type. Metadata concerning Complex Event Processing are discussed in 2.5.2. The second purpose is to define new types. Metadata can be applied there as well, especially for achieving certain characteristics, such as the equality principle. Defining new types can be desirable in situations where there are reasons not to interfere with an already created domain model, or for types used only in rules and not much elsewhere.

```
declare Cat
  name : String
  owner : Person
  registrationNumber : int @key
end
```

Functions are used either in order to avoid code duplication or to separate some logic to a certain place. They can be used in both condition and consequence of the rule. The static methods of classes can be also imported to serve as functions.

```
function String greeting(String name) {
  return "Welcome "+ name + "!";
}
```

Query represents a simple way to obtain facts complying with defined conditions from code. Furthermore, a combination of queries can be used to solve tasks.

```
query "Adults."
  person : Person( age > 18 )
end
```

Rules possess the following structure:

```
rule "name"
  // attributes
  when
    // condition
  then
    // action
end
```

There is approximately a dozen rule attributes. They influence the position of the rule on the agenda, assemble rules into groups, enable activation of the timer for regular rule execution or prevent the recursive firing.

The conditional part of the rule is sometimes referred to as LHS (Left Hand Side) and can be composed of multiple conditions. These conditions are represented by patterns which can match facts inserted into the session. Patterns can contain constraints related to the inspected type, such as:

```
Person (age > 18)
```

The attributes are retrieved through the getters. The facts and their attributes can be bound to a variable and used later in the rule. The conditions support logical operators such as *or*, *not* or *exists*. The facts can be gathered into collections, processed and evaluated by means of great deal of operators. Their thorough description can be found in Drools Expert documentation [12].

The action part of the rule is sometimes referred to as the consequence or RHS (Right Hand Side) and can contain any Java code. The amount of code should not be large. It is possible to insert, modify

and retract the facts here and also to use a technique called *logical insert*, which ensures the automatic retraction of the inserted fact, in the case that the condition of the rule is no longer true.

To conclude, Drools Rule Language provides a wide variety of components to express the desired logic. The best practices [19] determine that each rule should describe one scenario and nested conditional logic and loops in the action part should be avoided.

2.5 Drools Fusion

2.5.1 Complex event processing

The matter of processing events in intuitive form is not recent. Weather prediction, network monitoring or information gathering have been based on assembling and processing events for decades. However, the forming of event processing theory and principles did not commence until the late nineties. The first book presenting concepts and applications in this area was The Power of Events by David Luckham, published in 2002 [20]. Since then, a great deal of both theoretical research and commercial applications has been produced and presented.

Even though the literature offers various definitions of the term *event*, it can be generally described as an occurrence within a particular system or domain [21]. To put it in other words, it signifies a change of state that happened. This abstraction enables the modelling of a broad variety of situations by means of events. The event can represent a human action, such as a door opening or order placement, an environment change like a rise in sea level or temperature, or any occurring issue in a system. Furthermore, the processing of several events can result in the composition of one or many high-level events.

The Complex Event Processing allows operating with events in a meaningful and sophisticated way, including their creation, reading, transformation, abstraction, and discarding [22]. The utilization of Complex Event Processing can be applied in various areas. Common scenarios are: monitoring, detection, dynamic behaviour and predictive processing, performed individually or combined. An example of monitoring and detection in the public sector is patient

monitoring, where the nurse is alerted when a certain combination of events is recognized; as for commercial scenarios, one frequent application is fraud detection, where a combination of seemingly unrelated events signify suspicious behaviour. Another important requirement in many systems is to perform a problem analysis which can be achieved by a combination of searching through logged data and observing the events. One application of dynamic behaviour and predictive processing is, for instance, the stock trading system.

Drools fusion is the module enabling the event modelling capabilities to the Drools rule engine, such as defining facts as events, event cycle management, temporal reasoning and sliding windows. Since the traditional Rete algorithm does not support temporal restriction on nodes, the Drools algorithm had to be enhanced [23] to provide features such as aggregation of values in time-based windows.

2.5.2 Event definition

Event represent a special type of fact. An important requirement for events is that they are supposed to be immutable. They represent a change of state that happened, and therefore should not be redefined. The engine applies this fact in event lifecycle definition and optimizations. However, this does not determine the immutability of the event object. Event data enrichment is one of the possible use cases. To put it in another way, it is possible to enrich the event with additional data, but already established attributes should not be modified.

The declaration of facts as events is performed by using metadata tag **@role**. The events can be created from imported types, as well as new ones defined in drl:

```
import cz.example.Person;

declare Person
  @role( event )
end

declare InnerPerson
```

```
    @role( event )
    name: String
    age: int
end
```

The `@role` tag is necessary for the fact to be considered as an event, moreover the event can employ additional metadata tags:

```
declare PowerBlackout
    @role( event )
    @timestamp( blackoutStart )
    @duration( blackoutDuration )
    @expires( 3h )
end
```

Each event is automatically assigned with the start timestamp corresponding with the time of insertion into the session. The tag **`@timestamp`** determines, that for this timestamp is used the class attribute `blackoutStart` instead of the time of the insertion.

There are two types of events: point-in-time and interval-based. The point-in-time can be perceived as interval-based with duration equal to zero. All events are implicitly considered as point-in-time. The **`@duration`** tag labels the event as interval-based, where the duration is read from the `blackoutDuration` class attribute. As a matter of fact, each event has two timestamps, determining start and end of the event, where the end timestamp is constructed from the start timestamp and the duration.

The automatic life cycle of events enables the removal of events that are no longer relevant to rules from the session. This expiration is calculated from temporal constraints and sliding windows. The **`@expires`** tag should override this calculation.

2.5.3 Clocks

The assertion of the timestamp and temporal reasoning over the facts require the system to acquire the time from somewhere. The desired time can differentiate due to environment the system is running in. The time conditions for standard system operation differ from rules testing and simulations, or running in a special environment, such as

a cluster. As a consequence, the engine is enabled to employ different clock implementations.

The engine provides two clock implementations. The first is real time clock, which uses the Java Virtual Machine implementation and represents the option for standard use and flow. The second is pseudo clock, where the flow of time is easily manipulated. Without this option, scenarios using long time intervals would be difficult to simulate. The clock can be set for each session creation.

2.5.4 Temporal operators

The events in Complex Event Processing based systems are required to have temporal attributes which enable the definition of their temporal relationships and temporal operation amongst them. Drools Fusion defines 13 temporal operators outlined by James F. Allen in [24] to determine all possible relationships between two events. The temporal operators are as follows: after, before, coincides, during, finishes, finished by, includes, meets, met by, overlaps, overlapped by, starts and started by. Some of them require interval-based events to be applicable. An example of after operator application in a rule condition:

```
$sea1: SeaReport( $shoreId : shoreId,
                  $temp : temperature )
       SeaReport( shoreId = $shoreId,
                  temperature > ($temp + 2),
                  this after[1h, 2h] $sea1 )
```

A rule with such a condition will be activated if two events with same identification, and difference in temperature of at least 3, also meet the after operator.

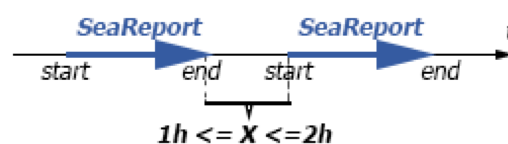

Figure 2.7: Illustration of using after operator in the rule.

The operator determines that the difference between end timestamp of the first event and start timestamp of the second one is between one and two hours as it is depicted in Figure 2.7.

This operator can be also defined using negative parameters for negative distance, or fewer parameters, where only one value would be coupled with positive infinity and no value generates an interval with 1ms and positive infinity.

The other operators allow the determination of different temporal relationships between events and are thoroughly described in Drools Fusion documentation [25].

2.5.5 Sliding windows

The concept of sliding windows enables restriction of the stream of events to a defined and moving interval. There are two types of sliding windows: time based window and length based window.

The **time sliding window** is a constantly moving time interval that takes into consideration only the events within the defined interval in terms of time.

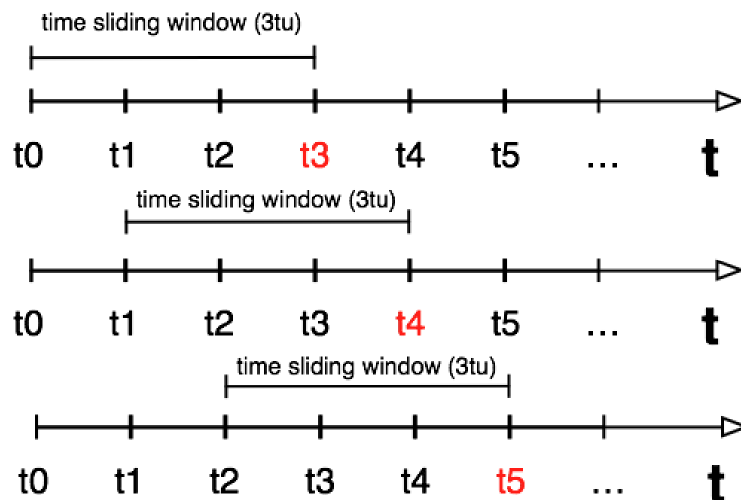

Figure 2.8: Time sliding window of three time units [6].

The Figure 2.8 illustrates an example of a time sliding window of

three time units. Time units support milliseconds, seconds, minutes, hours and days. The following condition would apply only to events happening in the last thirty minutes.

```
WarningEvent() over window : time (30m)
```

The essential variable for **length sliding window** is the order of events insertion. The window takes into account not more than the defined number of inserted events. The following example applies the window for the last fifty inserted events.

```
WarningEvent() over window : length (50)
```

2.5.6 Entry points

The entry points introduce abstraction over streams. The system can have various event streams, for example each for differen source of events. The engine enables objects to be inserted either into the default entry point or to a named one. To put it another way, each entry point represents a different stream of events.

There can be various motivations for the separation of events into streams. The streams can be homogeneous and contain only one type of events to increase the abstraction or the same types of events arriving from different sources might require being processed in a different way. In general, multiple entry points are used for partitioning the memory by the character and purpose of facts, and for decreasing cross products that can degrade the pattern matching performance.

A simple example of entry points utilization in rules:

```
rule "Passenger checks in at airport counter."
  when
    CheckIn( $passenger : passenger )
    from entry-point "Airport Counter Stream"
  then
    // the passenger's luggage is processed
end
```

```
rule "Passenger checks in from home"
  when
    CheckIn( $passenger : passenger )
    from entry-point "Online System Stream"
  then
    // the passenger's luggage is added
    // to automatic drop-off list
end
```

The checking in of the passenger is handled differently depending on the entry point where the checking event was inserted. Each event stores the information of the entry point it was inserted into. The Rete network containing these rules would have the `ReteNode` root followed by multiple nodes of `EntryPointNode` type.

2.5.7 Event processing modes

The engine provides two modes of processing: cloud and stream mode. The processing mode defines the behaviour including time and temporal aspects.

The default processing mode is **cloud mode**. There is no concept of time and its flow in this mode. As a consequence, it does not provide sliding windows, event life cycle, event ordering and clocks. In other words, all events are perceived as an unordered cloud. This mode is mostly targeted to be used with objects defined as facts, but it can also be employed for some scenarios including events that have no temporal relationships.

The **stream mode** enables the processing of streams of events with the concept of time. Since it is not the default processing mode, it has to be established in the Knowledge Base configuration. The requirements for it to work correctly are the time ordering of events within a stream and time synchronization of streams through the session clock.

2.6 Drools 6

The next version of Drools predicted to be released in the second quarter of 2013 is going to introduce a great deal of changes. It is go-

ing to be based on a new algorithm which merges concepts of multiple algorithms such as Leaps, Collection Oriented Match and Left and Right Unlinking. It targets the better use of parallelism and running in more constrained environments, for instance mobile devices [26].

Drools 6 is going to present context dependency injection and annotation driven development enabling sessions and bases to be injected. Furthermore, it allows the publishing of rules and processes as maven artifacts. Other innovations include the extension of Drools Knowledge Language in order to increase its expressiveness and power [27].

3 Motivation to include Complex Event Processing in web applications

3.1 The role of the user in the aims of websites

The growth and progress of the Internet is inseparably associated with evolution and advancement of websites. There are many types of them. Content-based websites profit from subscribed customers, sold pieces of content, or advertising. The scope for product-based and service-based websites is also quite large, from e-commerce and auction sites to cloud solutions. The common objective for these and a great deal of other types is represented by the user.

The potential capabilities of websites increase every year. The first websites were static and offered the same content for every visitor. As the technology progressed, the content commenced to differ for logged users. Following advancements introduced techniques using identification of the user by storing information in the cache of the browser, dynamic web pages processing user data or storing and analysing the actions of the user. Further improvements are constantly being developed to ensure better attracting, profiling and predicting of the user and his or her actions.

One of the companies pursuing the goal of the exceptional customer tracking is Amazon. With servers storing terabytes of data and millions of back-end operations every day it attempts to analyse the user data and recommend suitable products based on information in cache, searched products or reviews. The company also owns several patents regarding content personalization and use of product viewing histories [28].

However, aspects concerning the content adaptation are not the only ones to follow. Another appreciable aspect is detection of unsolicited behaviour. This can include various fields of interest where the apparent one is security. Each decade since the inception of networks, a new security focus has been introduced, namely risk assessment, securing of data centres, enterprise networks and the beginning of 21st century electronic commerce [29]. A great deal of present threats relate to the user. Misusing the sensitive data gathered from

social networks or exploitation of computational power to create bots to impersonate the user are just examples of current concerns. Detecting the suspicious behaviour commence to be necessary.

3.2 Benefits of incorporating Complex Event Processing

As has been established in the introduction to Complex Event Processing 2.5.1, an event can be described as an occurrence within a particular system or domain. This enables the perception and definition of any incident that occurred or action that was performed as an event. Even though the scope of the utilization of Complex Event Processing incessantly widens, so far there have not been many employments of it in web applications. This thesis considers the actions of users in web applications as events and explores their potential processing in order to enrich the web applications.

One of the most important augmentation that Complex Event Processing provides is the temporal reasoning. The definition of a temporal relationship over the events and the means to accumulate them easily in a meaningful way enable new or more easily performed use cases in various areas to be embraced. It enables the owner of the web application, or the application itself, not only to monitor the behaviour of visitors, but also to react quickly to specific situations. The unanticipated attendance of an article published long ago in the last ten minutes could indicate some affairs just happening. The accumulation of negative rating in the previous hour should be further investigated and engender appropriate reaction, such as retracting the rated element.

Due to the pace of nowaday life and the amount of competition on the Internet, the ability to recognize such situations and to respond to them promptly may account for the success or the failure of the website, in other words the profit or loss. For instance, recent considerable unsuccessful search on the website of an unknown Apple product might indicate a fresh announcement of it by the company and should result in immediate incorporation of the product and launching its pre-orders.

The automation of the reaction can vary from the one handled by

the person responsible responding to the created notification to fully automatic. An example of latter one can be the composition of main page. A disinterested or dissatisfied visitor is said to leave a page in 10-20 seconds, while the subconscious impression can be created in even shorter time [30]. The layout and content of the page can be established on various aspects, for example of what is currently happening, such as viewed, bought or discussed elements. Another example is creating heuristics for recommending products. The streams of events represented by such as the website itself, social networks, or review and benchmark magazines, could influence the recommendation list, for instance in terms of what other users of the same age bought in last hour.

Fraud detection is one of the common scenarios of Complex Event Processing utilization. Seemingly unrelated events are identified in patterns, which can uncover a sophisticated fraud or another type of deception occurring in the system. The same techniques are needed in websites where the exact application depends on the nature of the website. It can be particularly used to detect the suspicious behaviour and then further analysed to determine the origin and character of it, whether it is deliberate action, an error or false alarm. This can be specially combined with temporal reasoning in consequence of the fact that the events can be relevant or appealing for only certain period of time.

There are many advantages of using Complex Event Processing instead of performing code with same purpose over data in a standard way. Not all events are of interest to the system. As a matter of fact, the system might actually use only a small amount of events. As the events are represented by objects and expire after the time window employing them passes or by another expiration policy, they do not require being stored in a database of another storage to be processed. The traditional process would have to store all the data that events represent and then apply some enquiries over them, not to mention the management of their deleting or keeping. This has even greater impact in situations where the storage access is one of the restricting aspects.

The advantage of employing Drools Fusion to be the place for Complex Event Processing capabilities is in the rule engine. The logic represented by rules can be easily modified and broadened. Further-

more, it can be managed by the domain experts and not only by the programmers.

3.3 Existing solutions

Although the number of vendors offering solutions based on Complex Event Processing is increasing, the quantity of areas they have successfully penetrated is quite limited so far. The most frequent utilizations can be found in fields of algorithmic trading, financial fraud detection or network monitoring.

The only company found acknowledging the use of CEP in the scope of web applications is SeeWhy, which specializes in e-commerce analysis. They use this technology to ameliorate shopping cart abandonments. According to their statement, up to 70% of customers abandon their shopping cart before the completion and their service can assist in resolving 30% of it [31].

4 The application

4.1 Overview

As a consequence of the Drools being a project written in Java, the application exploring and demonstrating the possible utilization of Complex Event Processing encompassing actions of the user in web environment is a Java EE application. Generally, an application desiring the same functionality might employ different technologies and be based for example on C# or PHP, as event processing engines for these languages can be found.

The type and specialization of the application has been selected as product-based, as e-commerce is one of the significant areas of user processing. Owing to the fact that the purpose of the application is exploratory and demonstrational, the feature set is not required to be as broad as of a real one. However, it should provide a considerable amount of functions to develop various scenarios.

Basic functionality is represented by browsing and buying products which reside in separate categories. This is associated with managing shopping cart and checkout proceedings. Additional features include functions such as searching amongst the products or simple discussion of each product.

Most of these features are available to any visitor. Apart from this type of user, the application distinguishes users logged in for customers and administrators, each with different possibilities and rights.

On the whole, as unregistered visitors and registered customers browse the website, put products into the shopping cart, buy products or perform different actions, events are generated and inserted into the engine. The events are processed by the rule engine, which may result in activating rules. After the rules are fired, their consequences are performed and that results in influencing the web application itself or presenting the outcome to the administrator.

The application possess the prosaic name FusionWeb. It is composed as a Maven project and its sources are available at GitHub[3].

3. See `https://github.com/IxiCZ/fusionweb`.

As it is a Java EE application, it requires an application server to run, nevertheless it has been deployed to a Platform as a Service solution[4] as is described in chapter 4.4.1, where it can be easily explored as it contains a description in the *About* page and direct logging links.

Figure 4.1: Application screenshot of browsing products.

4.2 Analysis and design

4.2.1 Roles

Users are classified into three different groups. Each group is represented by a separate role and each role is associated with diverse

4. See `https://fusionweb-ixi.rhcloud.com`.

feature set and security realm. The use case diagram depicting the interaction of the users in roles with the application is delineated in Figure 4.2.

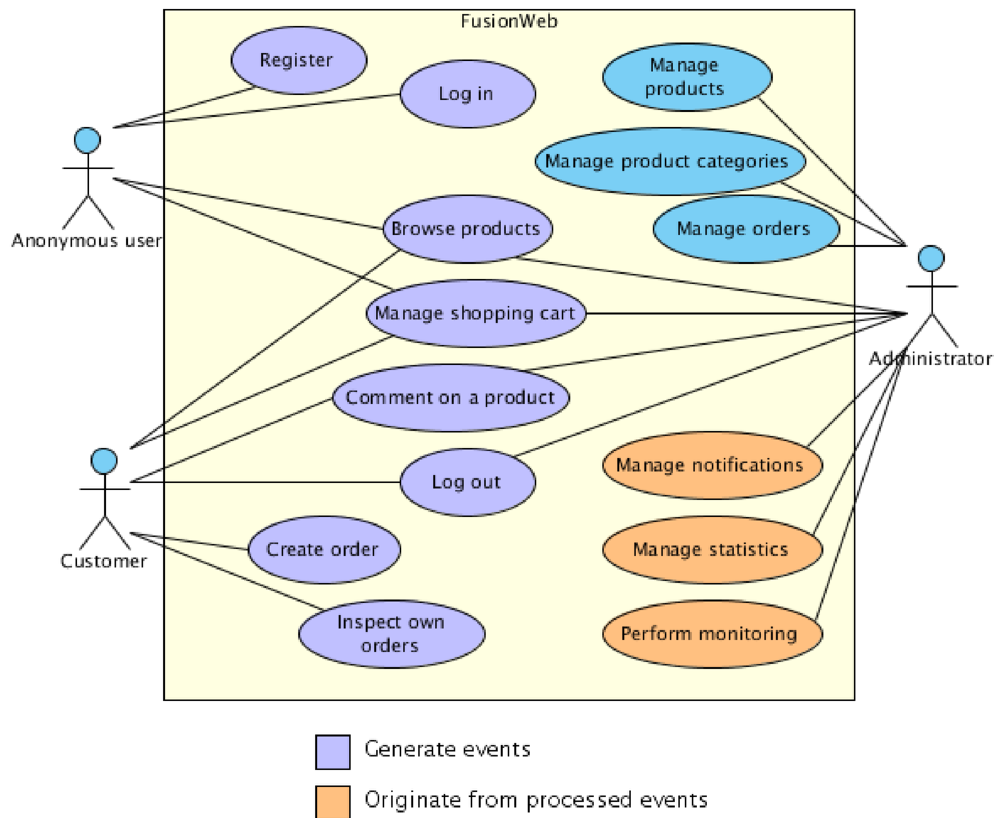

Figure 4.2: Use case diagram of roles interacting with the application.

Any user not logged in is represented by the **Anonymous user** role. They are allowed to performed actions associated with browsing products, such as paging through the categories, displaying details or searching the product by keywords, and also managing their shopping cart. However to complete the checkout and create the order the user is required to log in as a **Customer**. Customers can perform additional actions such as commenting on products and inspecting their orders. All the steps carried out by these two roles are transformed to events and inserted to the Drools engine.

Administrator represents the user responsible for managing the application content and responding to any issues happening. In other words, an administrator can operate with the application in a standard custodian way and perform for instance managing orders, but can also benefit from the outcome of the event processing engine.

4.2.2 Data model

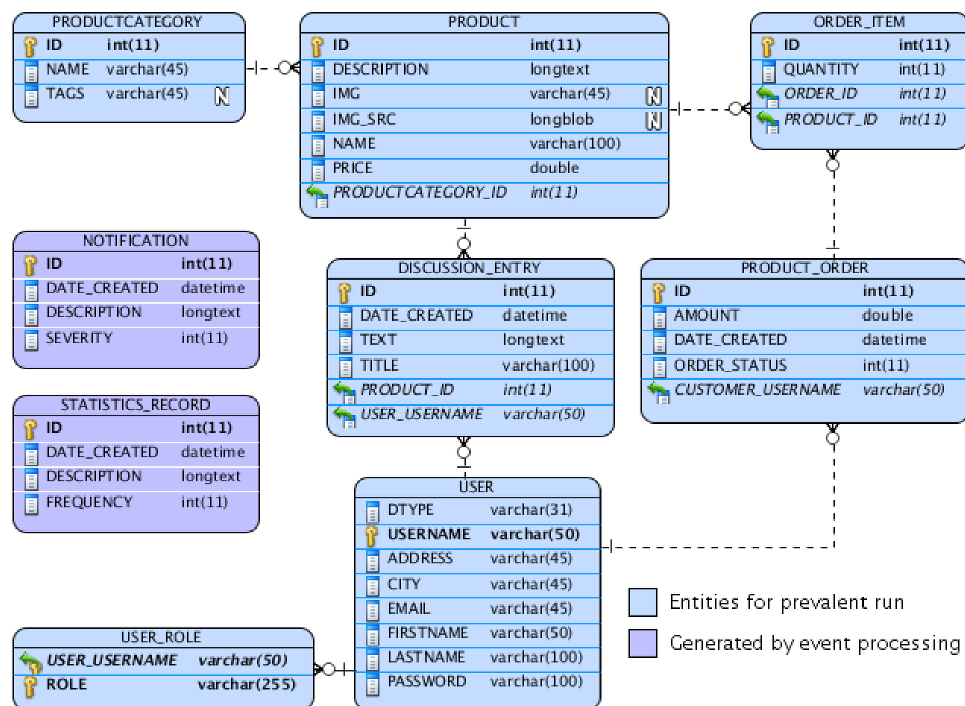

Figure 4.3: Entity-relationship diagram.

Figure 4.3 depicts the entity-relationship model employed by the application. Most entities are used for describing products, orders and users. However, there are two entities which store results from event processing. This does not imply the necessity of the outcome to be in a form of stored records. The rule consequence may have various forms, from sending a notification email to influencing the application.

The design of events represents different levels of data modelling. Drools rule language enables the definition of events either as standard Java classes or directly in drl files. The latter option has been selected for facts and events assisting in the correct rule firing, but not representing the user actions in the application itself. Useful cases of those, contributing to explored scenarios have been depicted as java classes. Their simple class diagram is in Appendix A.

4.2.3 Plugging Drools in

One of the essential decision at the beginning of constructing the application was generating events in a manner of tier location, to put it in another way, where the standard Java EE components will interact with Drools and feed it with events. This resolution influences not only the possible ways of implementation, but also the amount and character of potential actions that can be processed and transformed into events, not to mention other aspects, such as simplicity of future modifications, further reusability or greater adjustments.

As one might expect, the business and persistence tiers weren't taken into bigger consideration, as they considerably reduce the information regarding the action origin, in other words, they wouldn't be easily able to determine by whom and how is a specific operation initiated.

Another possibility was to link it by means of backing beans for the user interface. This could be realized not only by ponderous placement of the code within methods, but for example by annotating the methods with their own appropriate annotations. This approach brings several drawbacks. The purpose of beans is to execute and control the operation through reasonable amount of methods and attributes. Merging them with the Drools part would result in more complicated code handling, as any refactoring, modification or extension could project also into the necessary adjustments in Drools components. In a more important manner, beans would lack some interesting information such as which part of the page initiated the action.

Therefore, the final conclusion determined to realize the connection as far "up" in the tier terminology as possible, to be more precise, in the user interface components. This has not only enabled the fine

distinction between different situations, but also easier plugging in, with the meaning that any not yet considered link can be enriched with the apposite listener. The type of the listener and passed on parameters subsequently determine the events that should be generated.

To conclude, the overall majority of the elements generating the events listens to actions in user interface components. The outcome of rule consequences and other mechanisms such as monitoring and layout configuration communicate with various tiers. Figure 4.4 outlines this in a simple view.

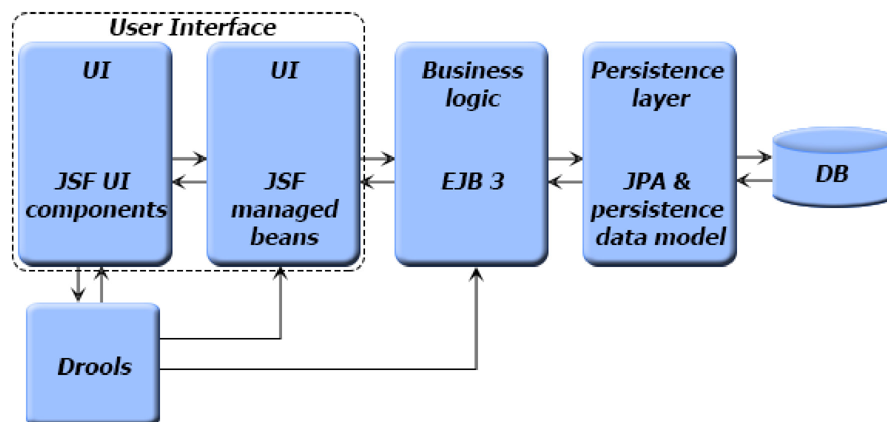

Figure 4.4: Simple illustration of situating Drools into architecture.

4.3 Implementation

4.3.1 Java EE

The prerequisite for composing and building the Drools components was to develop a simple Java EE application representing the place of origin for generating events. The application employs the technologies of Java Enterprise Edition platform version 6 and follows the guiding techniques for developing enterprise applications from Oracle [32].

The web tier is composed by means of JavaServer Faces (JSF) technology. It a is server-side component framework which is primarily composed of API for constructing components and tag libraries for integrating the components into web pages. The tag elements in web pages are essential for generating the events as that is the place where events originate. To be more precise, the JSF part consists of descriptor configuration files, tagged web pages, custom components such as converters or validators and managed beans that represent container-managed objects applying techniques such as dependency injection or lifecycle callbacks.

The business tier and logic are encapsulated by Enterprise JavaBeans (EJB) technology. Beans represent server-side components which facilitate transaction management and scaling. The application employs for the most part two types of them: stateless and singleton. The hierarchy of stateless beans is used for enquiring and handling the data model stored in the database. Singletons with a combination of application post construction are applied for creating and filling up the tables in database, or establishing the Drools session.

The persistence tier is provided by Java Persistence API (JPA). It enables the management of relational data by means of object/relational mapping capabilities, where class entities correspond to the tables in the database and their instances to rows in the tables. Apart from the annotations to determine the mapping of entities and entity relationships to relational data, the entity classes in the application also employ mechanisms for validating data called JavaBeans Validation, and Named Queries. Managing entities is administered through Java Persistence Query Language (JPQL) and Criteria API. The targeted database is set in the descriptor file.

The security of the application is achieved by means of both declarative and programmatic security aspects. The scope of user capabilities is determined by his or her role. Administrator and Customer roles are assigned to users at the end of the authorization mechanism of logging in with username and password. The resources are declaratively labelled in a descriptor file to be available only to a certain role. That means, that if users not logged in attempt to reach the resource they are redirected to the login page and if users in an incorrect role try to reach it, they are prevented to do so.

To conclude, applying the afore described technologies forms a fundamental point for further enriching and adapting for the purpose of incorporating event processing. The application was developed also considering other aspects such as design employing CSS or internationalization using a resource bundle for messages.

4.3.2 Drools assembly at the start of the application

At the start of the application it is necessary to assemble the Drools session, in other words, to create a proper Rete network and environment from resources, configuration and various components for it to be able to receive events and commence firing rules. This is achieved in an init method of EJB bean, heading of which is in Figure 4.5.

```
@Singleton
@Startup
@Lock(LockType.READ)
@DependsOn({ "StartupDBConfigBean" })
public class DroolsResourcesBean {

  @PostConstruct
  public void init() {
...
```

Figure 4.5: Heading of the bean responsible for Drools assembly.

The `@Singleton` annotation determines the bean type to be a singleton session bean. Singleton session beans are instantiated only once and exist for the lifecycle of the application. The `@Startup` annotation dictates this creation to occur upon application deployment. Subsequently the dependency injection (not depicted in the example) is performed and then the `@PostConstruct` method is invoked.

The `@DependsOn` annotation is used in situations where more singleton session beans require being initialized in a specific order. The usage in the example specifies that another bean needs to be constructed first. The purpose of this initial bean is to establish the tables in the database and to fill them with starting data such as a

considerable amount of products or users in roles. The bean outlined in the example does not only assemble the Drools session, but also afterwards serves as an access point to it. The `@LockOn` annotation is used for influencing the container-managed concurrency, as the EJB container controls the access to the methods. As a whole, the bean is labelled by the read type, which means that its methods can be concurrently accessed or shared by various clients. However, to preserve the consistency, some methods modifying the state are annotated with the `LockType.WRITE`, which locks the bean while the method is called.

The detailed diagram of the Drools session assembly is depicted in Figure 4.6.

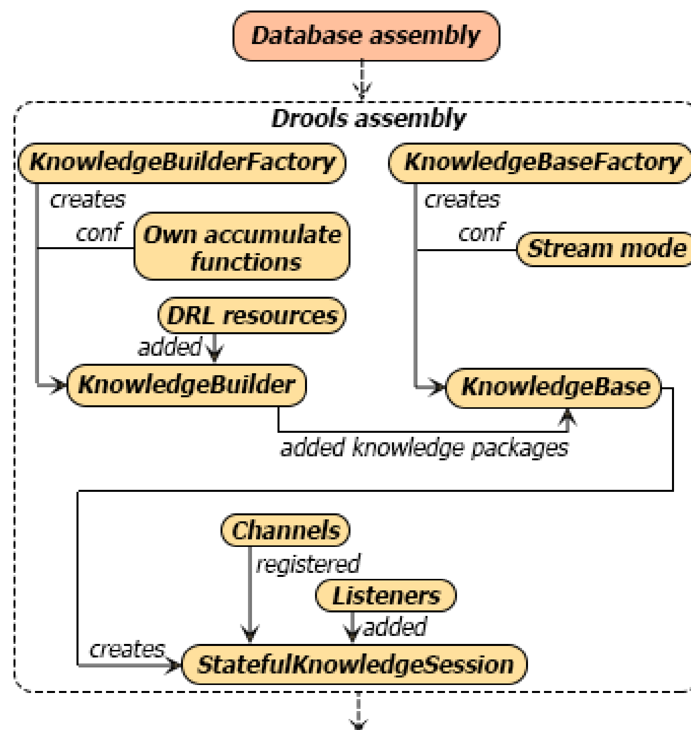

Figure 4.6: Detailed Drools session assembly.

The KnowledgeBuilder is configured to also be constructed with integration of own accumulate functions. Accumulate functions enable the rule to easily iterate over collection of objects, apply them to

achieve the result and return the result object. The engine contains predefined functions such as sum or count, and enable to compose own by implementing appropriate interface. The example of its own accumulate function is the function for iterating over set of events representing visited products and returning the most frequent one.

After KnowledgeBuilder is created, the resources represented by drl files are fed into it. The rules and other elements are divided into multiple files to facilitate the process of their maintaining and testing.

To be aware of the concept of time, KnowledgeBase requires to be constructed in stream mode. Subsequently, the knowledge packages from KnowledgeBuilder are added into it and it creates the StatefulKnowledgeSession. The session is then enriched by channels and listeners. Employment of channels is one of the possible ways to propagate the information from rule consequence. The listeners are used for logging as they invoke methods relating to the activations and rule firing, such as after each fired rule.

4.3.3 Overview of generating and processing events

The generation of the majority of events originates in the user interface JSF command links and buttons. In other words, they result from the user navigating through the website and performing actions. An example of such component is illustrated in Figure 4.7.

```
<h:commandButton
 action="#{shoppingCart.addItem(
 productController.selected)}"
 actionListener=
   "#{shoppingCartListener.productAddedIntoCart(
      productController.selected,
      userController.user)}"
 value="#{bundle.AddToCart}"
 styleClass="addToCartDetail"
/>
```

Figure 4.7: Example of a command button with listener producing events.

The attributes `value` and `styleClass` determine the appearance of the button. The attribute `action` specifies the bean and its method processing the input, and resulting in navigation. The element applied for generating events is `actionListener`. It defines a bean method which processes an action event (not to confuse the term with the discussed events) invoked by the component tag. To put it in another way, this attribute registers when a component is activated and executes appropriate method, which creates an event and inserts it into the session. There are two arguments in this example from which the event is constructed, the user and product, nevertheless, more of them can be applied to compose even more detailed events.

After composing the event, the listener inserts it into the session. There it passes through the Rete network and might create activations of one or more rules. When the rules are fired, their consequences are executed. The form of rule consequences can be diverse. Some of them are applied to compose different types of events or perform other actions considering only the engine and its elements, such as updates or retraction of events. Other rules need to propagate an acquired result or piece of information outside the session. There are various ways to achieve this. A component capable of invoking some desired method can be inserted into the session as a fact; however, this brings a disadvantage of blending these components with processed data. Therefore, a more appropriate option is to employ globals or channels. Globals are named objects which require being set on the session and declared in the rule file to be used. Channels are instances of classes implementing the appropriate Drools interface for receiving objects, and also needs to be registered to the session.

The general overview of the process resulting in the storing of notification is outlined in Figure 4.8. When a user uses the relevant component in the page, the listener generates an event and inserts it into the session, where it might activate a rule. After the rule is fired, its consequence employing a channel is executed and it creates an apposite notification and stores it into the database.

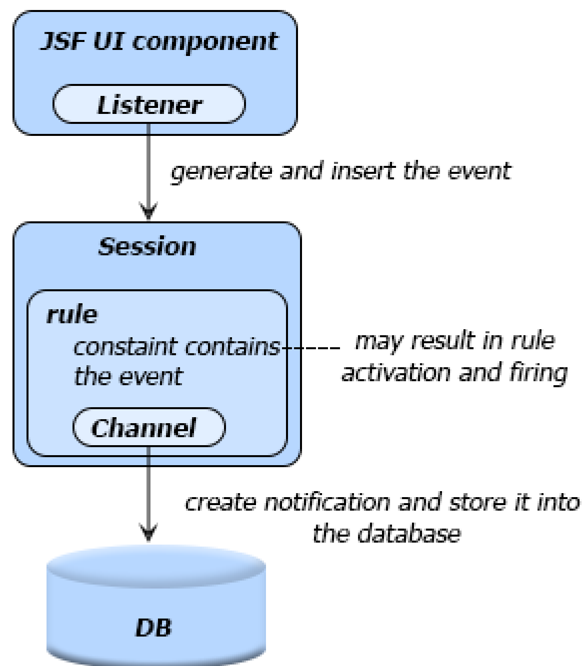

Figure 4.8: Overview of user action resulting in notification.

4.3.4 Firing the rules

As it has been established, the insertion of the fact into the session only creates the activations of the rules. To execute the consequential parts of rules, the rules need to be fired.

Drools enable the firing of rules in two general ways. The first is one-time firing by means of `fireAllRules` method on the session and the second is managing of different threads with `fireUntil-Halt` method execution on the session, which fires a rule when it is activated. Due to the fact that managing a own threads in applications is strongly discouraged within Java EE container [33], the first way is employed.

The appropriate approach to this depends on the character of the rules. If the rules can be activated only by event insertion and it is important to execute the rule consequence as soon as possible after the activation, the firing method can directly follow the act of insertion. However, if rules contain constraints dependent not only on event

insertions, for example when the application should evaluate the absence of certain facts over last hour, or if the reaction rate is not essential, an automatic timer can be applied. The application currently applies the combination of both approaches.

The EJB container constructs the automatic timer when a properly annotated enterprise bean is deployed. The Figure 4.9 depicts a bean used for firing rules.

```
@Startup
@Singleton
@DependsOn({"StartupDBConfigBean",
            "DroolsResourcesBean"})
  public class FireRulesScheduler {

  @Inject
  private DroolsResourcesBean drools;

  @PostConstruct
  @Schedule(hour="*", minute="*/5", second="0",
  persistent=false)
  public void init(){
    drools.fireAllRules();
  }
}
```

Figure 4.9: Bean used for firing rules every five minutes.

The bean is created at startup after the completion of table creation and constructing the session. The bean representing the access point to the Drools session is injected and an automatic timer is constructed by the container. The character of the timer is determined by the `@Schedule` annotation, where the frequency is defined by an expression with similar syntax to the unix cron utility. This example specifies the rules to fire every five minutes.

As the use of the timer is combined with the firing rule after every event insertion, this method requires annotation with the `LockType.WRITE` to prevent concurrency problems.

4.3.5 Use cases of Complex Event Processing utilization

The application illustrates four general use cases of Complex Event Processing utilization implemented by means of Drools. They are as follows: notifications, statistics, layout adjustment and monitoring.

Notifications represent immediate reaction when an application reaches a state defined in rules and reports it, in other words, when something is happening that administrator should know about and presumably react to. This does not necessarily imply an undesired state, as the administrator might for instance want to be aware of the situation where price of a product can be raised.

The application differentiates four types of notification: severe, warning, info and good. The example of a severe notification is the one composed when there is a product visited a lot, but not bought much in the last hour. The condition of the rule responsible for this calculates the difference between the sum of all visiting events for each visited product event from the last hour and sum of all events representing the buying of the product from the last hour, and if the result is greater that the set number, the rule is activated and the consequence of the rule creates the notification. An example of a good notification is one reporting a greatly bought product in the last hour.

Although some notifications can be composed with the application of a single rule, some requires the cooperation of more rules to achieve this result. Following example depicts three rules employed for creating notifications when there is a lot of unsuccessful product search events with the same searched text in the last hour.

```
rule "Insert searched text event"
  when
    ProductSearchUnsuccessfulEvent($searchedText:
    searchedText)
    not SearchedText(text == $searchedText)
  then
    insert(new SearchedText($searchedText));
end

rule "Retract searched text event"
  when
```

```
    $searched : SearchedText($text : text)
    not ProductSearchUnsuccessfulEvent(
      searchedText == $text)
  then
    retract($searched);
end

rule "Create notification if too many unsuccessful
seach events"
no-loop
  when
    $searchedText: SearchedText($text: text)
    not SearchedTextReported(text == $text)
      over window:time(1h)
    ArrayList(size >= 10) from collect(
      ProductSearchUnsuccessfulEvent(searchedText
        == $text) over window:time(1h))
  then
    channels["productSearchUnsuccessful"]
      .send($text);
    insert(new SearchedTextReported($text));
end
```

This collection of rules employs not only the events generated by the UI component listener, but also a supporting fact and event declared in rules. The first two rules manage insertion and retraction of the supporting fact `SearchedText` used for easy grouping of events in the rule creating the notification by searched text. The first rule inserts this fact if it does not exist, and the second rule retracts it when there is not a corresponding event anymore, in other words, if all events of this kind with this searched text have expired. The third rule calculates the count of corresponding events for each such searched text and compares it to the set number. This rule also uses a supporting event to report this kind of behaviour not more than once per hour.

The channel applied in the third rule in the example is the specific channel created for this situation, expecting the searched text. The channel creates the notification object and sets a proper mes-

sage to it. This has the advantages of detailed channel differentiation and uncomplicated rule consequences, however the notification attributes such as severity level or text cannot be easily altered in rules. Folowing example depicts a simple rule employing another channel of a general character which expects notifications.

```
rule "Create notification when discussion entry
contains help"
  when
    $event: DiscussionEntryEvent(
      text.toLowerCase().contains("help"))
  then
    channels["notificationsGeneral"].send(new
    Notification(NotificationSeverity.WARNING,
    new Date(), "The discussion entry created
    by user '" + $event.getUsername() +
    "' by '" + $event.getProductName() +
    "' product contains 'help'." ));
end
```

As the channel expects the already composed notification, it has to be created in the rule. This can impede the transparency of the rule, but facilitate its customization. Also a hybrid approach could be applied, where only a text, severity level, or their tuple could be sent. Namely this rule could be easily implemented using the standard techniques in the code as its condition could be rewritten to a simple if statement. However, the separation of this logic from the rule enables the easy construction of dozens of similar conditions without the necessity of altering the code and it can be achieved by even a non-programmer familiar with the rules.

Statistics are generated results from rules every established period of time. The application possesses two types of them, hourly and daily, where the difference is only in the frequency of their execution and therefore the amount of time they take into consideration. The example of a rule generating a statistic:

```
rule "Report how many customers logged in
in the last day"
timer (cron:0 0 0 */1 * ?)
```

```
no-loop
  when
    $total: Number() from accumulate(
    $e: CustomerLogInEvent() over
    window:time(24h), count($e))
  then
    channels["statisticsDaily"].send("In the
    last day logged in "+ $total +
    " customers." );
end
```

The rule applies the accumulated count function to all events from the last day and sends the result message through the appropriate channel. The timer element is employed to achieve regular execution. It can be defined either using the standard Unix cron expression, or by the interval expression described in the Drools Expert documentation [12]. Only initial firing of the rules is required for the timer to be activated.

The purpose of **layout adjustment** can be various, from attracting new visitors to recommending products to familiar users. However, to achieve that properly, the presentation components have to be designed with this purpose in mind. The example of this is that the most visited product in the last hour is established as the main product on the home page, where this alternation is executed every hour. The rule responsible for this is in the folowing example:

```
rule "Change main product if there is a better
one."
timer (cron:0 0 */1 * * ?)
no-loop
when
  exists(ProductNavigationEvent()
    over window:time(1h))
  $current: CurrentMainProduct()
  $mostVisitedProductEvent:
    ProductNavigationEvent() from accumulate
    ($PNEventMain: ProductNavigationEvent()
    over window:time(1h),
```

```
      mostVisitedProduct($PNEventMain))
    eval ($current.getId() !=
      $mostVisitedProductEvent.getProductId())
  then
    modify ($current) {
      setId($mostVisitedProductEvent.
        getProductId());
    }
    channels["defaultLayout"].send(
      $mostVisitedProductEvent.getProductId());
  end
```

This rule employs a supporting fact representing the current main product. This fact is inserted in another rule possessing a simple condition based on non-existence of this fact. The rule also utilizes its own accumulate function `mostVisitedProduct`, which iterates over collection of events and returns the most frequent one. The condition of the rules also requires the new main product to be different from the last one. In other words, if the same product is the most visited again, the rule is not activated. If the rule is activated, the current main product fact is altered and the id is sent through the channel responsible for setting of the main product.

Furthermore, it would not be difficult to modify the rule to inspect other elements to set the main product, so it could be for instance based on the most bought or the most visited product. The more sophisticated version could alternate these options in a defined pattern and then automatically set the most successful one depending on the time of day, user category or different aspects.

Monitoring in the application represents a scenario where events in the session can be examined to determine what is currently happening. Administrators can choose a user recently logged in and see what the user is presently doing. This is implemented by the means of Drools object filters. The session is capable of returning the collection of events meeting the defined condition. This condition is represented by the implementation of the specific interface. The administrator can utilize such information for example to inspect the navigation through the website or series of steps leading to buying a product.

4.3.6 Testing of the rules

Testing of the rules is achieved by means of unit testing each rule or a couple of rules representing a certain scenario. Each such test case is covered by several unit tests attempting to evaluate a different situation. The tests comprising these unit tests are grouped in compliance with the separation of the rules into drl files, in other words, a test may add into Knowledge Builder only the drl file holding the declarations and the drl file representing the tested logic.

Each test is composed of unit tests and a before and after method. The before method is responsible for composing the session before each unit test and the after method for correct disposal of it. The constructing of the session is performed in the similar way as in the init phase of the application with the exception of resources, channels and clock. Only the resources necessary for the scenario are used for creating knowledge packages. The mock objects are substituted for actual channels as they require the application server to operate.

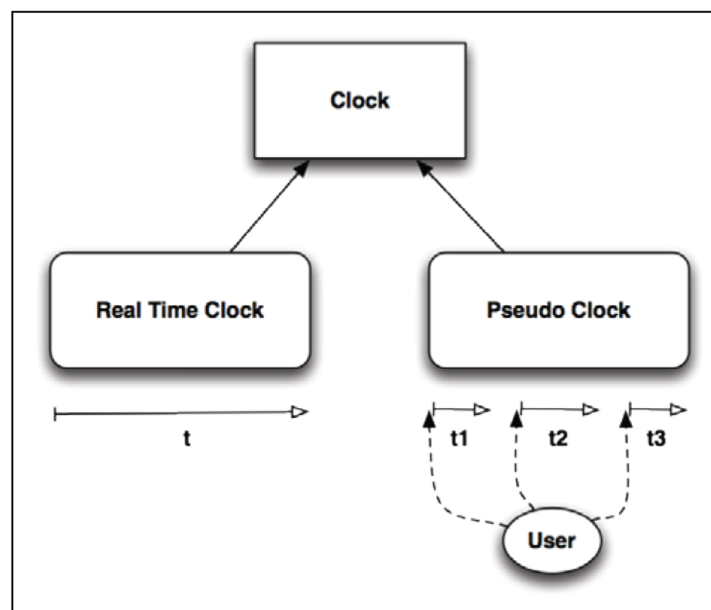

Figure 4.10: The two Drools clock implementations [6].

Due to the fact that many of the rules employs the temporal rea-

soning aspects, the clock implementation in the tests must be different. Drools provides two clock implementations, demonstrated in Figure 4.10, where the *Real Time Clock* applies the JVM clock and *Pseudo Clock* enables the application to control the flow of time in all different ways. In order for the real-time clock to be the default option, the setting of the pseudo-clock has to be configured during creating the session.

4.4 Deployment

4.4.1 OpenShift

Openshift is a Cloud Computing Platform as a Service (PaaS) solution built on open source technologies managed by RedHat. It offers built-in support for Java, Ruby, Python, PHP, Perl, and Node.js and customizable cartridge functionality for adding any other language, for instance Cobol. Furthermore, it provides both automatic and manual scaling of the resources. Deploying the application on Openshift is associated with selecting and applying present cartridges such as server or database, or building one's own [34].

Managing the application on Openshift can be achieved either through the command line or Eclipse IDE with the appropriate plugin installed. The security is provided by means of SSH as the public SSH key needs to be provided by the user to use the account.

The application has been deployed to Openshift in order to be easily demonstrated and explored. It uses the following cartridges: *JBoss Application Server 7.1* to be the server, *MySQL Database 5.1* for the relational database and *phpMyAdmin 3.4* to easily inspect the database. The configuration for the openshift profile is similar to the one of local server and can be found at the github repository for the application[3]. The url through which the application can be reached is:

`https://fusionweb-ixi.rhcloud.com/`

3. See `https://github.com/IxiCZ/fusionweb`.

4.4.2 An application server

For deployment to an application server the required elements are a properly configured Java application server and a database system capable of communicating with it, in other words, for which the appropriate driver exists.

The database system only needs to contain a corresponding database to the one defined in server configuration. Tables are created and filled during the deployment of the application.

The configuration of the application server must set the database as data source and establish the security domain. The guide how to install the application to the JBoss Application Server is in Appendix B.

5 Conclusion

This thesis presents the introduction to Drools project and focuses on explicating the core of the engine, the employed algorithm, rule language and module for event processing capabilities. It also discuss the motivation of incorporating Complex Event Processing into the scope of web applications and demonstrates the conclusions in practical solution applying Drools.

The application was designed and implemented as a multi-tier Java EE application capable of generating events as users browse, navigates or perform various actions by means of it, where different ways of realizing the event generation had been considered. To provide the sufficient base for the event processing, the application was developed to contain the appropriate features such as user logging in, product ordering or administration, and to offer more interesting types of events, features such as product discussion or searching amongst the products. The uppermost endeavour was given to linking the Drools with the application and processing the events by apposite rules.

The application presents various scenarios of utilization of Complex Event Processing in this scope. For ease of demonstration the application has been deployed to Platform as a Service solution, where it can be explored and the instances of use cases are documented.

The application is a prototype of a possible solution. As a consequence, it does not cover all possible use cases which could be applied in this area. For instance, the distinguishing of events could contain more attributes such as exact element and position on the page generating it or detailed information about the user. In addition, many additional types of events might be defined and a great deal of rules considering the events could be composed. A real complex solution would also require to consider other aspects, such as caching or clustering.

To conclude, the area of Complex Event Processing can be utilized in the scope of web applications providing various benefits of analysing the events and reactions of their processing. The incorporation of such a mechanism is currently required to be handled individually and for the proliferation of this approach, advanced tech-

niques and frameworks should be developed.

Bibliography

[1] Bali, M. (2009). *Drools JBoss Rules 5.0 Developer's Guide*. Birmingham: Packt Publishing.

[2] *Drools - The Business Logic integration Platform*. (n.d.). Retrieved March 6, 2013, from http://www.jboss.org/drools/

[3] Meeks, K. (2008, January 26). *Introduction to Drools*. Retrieved from http://www.readbag.com/intltechventures-presentations-2008-01-26-introduction-to-drools

[4] Amador, L. (2012). *Drools Developer's Cookbook*. Birmingham: Packt Publishing.

[5] Salatino, M. (2009). *JBPM Developer Guide*. Birmingham: Packt Publishing.

[6] Aliverti, E. & Salatino, M. (2012). *jBPM5 Developer Guide*. Birmingham: Packt Publishing.

[7] *Drools Planner*. (n.d.). Retrieved March 8, 2013, from http://www.jboss.org/drools/drools-planner.html

[8] Sandeep, J. (2012, August 6). *Comparison between Websphere ILOG JRules and JBoss Drools BRMS*. Retrieved March 8, 2013, from http://www.javagenious.com/ilog/WebSphere_ILOG_JRules_BRMS_vs_JBoss_Drools.html

[9] *Open Source Workflow Engines in Java*. (n. d.). Retrieved March 8, 2013, from http://java-source.net/open-source/workflow-engines

[10] Gualtieri, M. & Rymer, R. R.. (2009, August 4). *The Forrester WaveTM: Complex Event Processing (CEP) Platforms, Q3 2009*.

Retrieved from
ftp://ftp.software.ibm.com/itsolutions/SOA/wave_complex_event_processing_cep_platforms.pdf

[11] Mijović, V. & Vraneš, S. (2011). *A survey and evaluation of CEP tools*.
Retrieved from
http://www.e-drustvo.org/proceedings/YuInfo2011/html/pdf/165.pdf

[12] The JBoss Drools team. *Drools Expert User Guide Version 5.5.0.Final*. (2012, November 13).
Retrieved March 9, 2013, from
http://docs.jboss.org/drools/release/5.5.0.Final/drools-expert-docs/pdf/drools-expert-docs.pdf

[13] Agüero, M., Esperon, G., Lopez De Luise, D. & Madou, F. (2010). *Enhancing Source Code Metrics Scope Through Artificial Intelligence*.
Retrieved from
http://www.iiis.org/cds2010/cd2010sci/citsa_2010/paperspdf/ia292nw.pdf

[14] Browne, P. (2009). *JBoss Drools Business Rules*. Birmingham: Packt Publishing.

[15] Lloyd, J. W. (1994). *Practical Advantages of Declarative Programming*. University of Bristol.

[16] Forgy, Ch. (1974). *A network match routine for production systems*. Working paper.

[17] Doorenbos, R. B. (1995, January 31). *Production Matching for Large Learning Systems*.
Retrieved from
http://reports-archive.adm.cs.cmu.edu/anon/1995/CMU-CS-95-113.pdf

[18] Mello, P., Proctor, M. & Sottara D. (2010, September 16). *A Configurable Rete-OO Engine for Reasoningwith Different Types of*

Imperfect Information. IEEE Transactions on Knowledge and Data Engineering, Vol. 22.

[19] Tirelli, E. (2011, October). *BRMS: Best (and worst) Practices and Real World Examples*.
Retrieved March 24, 2013, from
`http://www.slideshare.net/etirelli/brms-best-practices2011octfinal`

[20] Luckham, D. (2002). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston: Addison-Wesley Longman Publishing.

[21] Etzion, O. & Niblett, P. (2011). *Event Processing in Action*. Greenwich: Manning Publications.

[22] Luckam, D. (2012). *Event Processing for Business*. New Jersey: John Wiley & Sons.

[23] Breddin, T., Groch, M. & Walzer, K. (2008). *Supporting Temporal Reasoning in the Rete Algorithm for Complex Event Processing*.
Retrieved from
`http://www.rn.inf.tu-dresden.de/uploads/dexaPaper.pdf`

[24] Allen, F. J. (1981). *An Interval-Based Representation of Temporal Knowledge*.
Retrieved from
`http://www.ijcai.org/PastProceedings/IJCAI-81-VOL1/PDF/045.pdf`

[25] The JBoss Drools team. *Drools Fusion User Guide Version 5.5.0.Final*. (2012, November 13).
Retrieved March 25, 2013, from
`http://docs.jboss.org/drools/release/5.5.0.Final/drools-fusion-docs/pdf/drools-fusion-docs.pdf`

[26] Proctor, M. *Life Beyond Rete - R.I.P Rete 2013 :)*. (2013, January 02).

Retrieved March 30, 2013, from
http://blog.athico.com/2013/01/life-beyond-rete-rip-rete-2013.html

[27] Fusco, M. *Drools 6: News & Noteworthy*. (n. d.).
Retrieved March 30, 2013, from
http://2013.33degree.org/talk/show/20

[28] Layton, J. *How Amazon Works*. (n. d.).
Retrieved April 2, 2013,
http://money.howstuffworks.com/amazon.htm

[29] Lacey, D. (2009). *Managing the Human Factor in Information Security*. New Jersey: John Wiley & Sons.

[30] Nielsen, J. *How Long Do Users Stay on Web Pages?* (2011, September 11).
Retrieved April 4, 2013, from
http://www.nngroup.com/articles/how-long-do-users-stay-on-web-pages/

[31] Gonsalves, A. *SeeWhy Takes Complex Event Processing To Web Analytics* (2009, May 8).
Retrieved April 6, 2013, from
http://www.informationweek.com/software/business-intelligence/seewhy-takes-complex-event-processing-to/217400025

[32] Cervera-Navarro, R., Evans, I., Haase, K., Gollapudi, D., Jendrock, E., Markito, W. & Srivathsa, C. *The Java EE 6 Tutorial* (2013, Jabuary).
Retrieved April 10, 2013, from
http://docs.oracle.com/javaee/6/tutorial/doc/

[33] EJB 3.0 Expert Group. *JSR-000220 Enterprise JavaBeans 3.0* (n.d.).
Retrieved April 20, 2013, from
http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html

[34] Red Hat, Inc. *OpenShift Platform as a Service* (n.d.). Retrieved April 30, 2013, from https://www.openshift.com/paas

A Events diagram

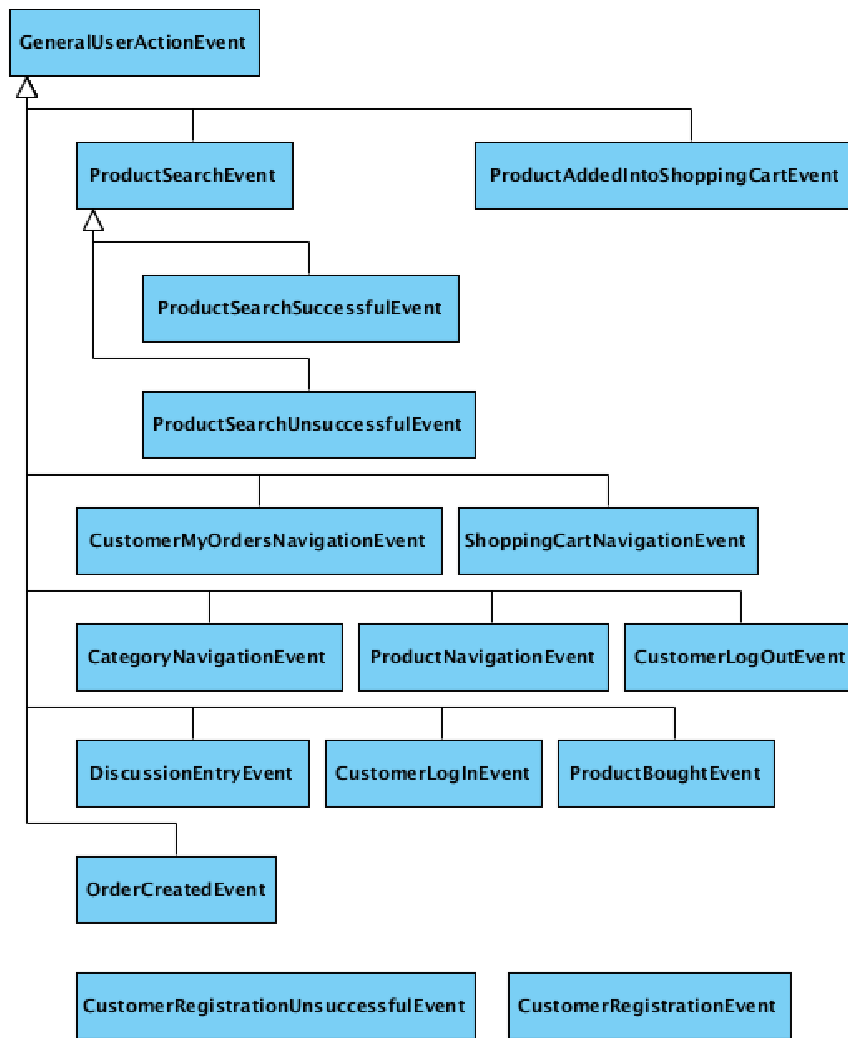

Figure A.1: Simple class diagram of significant events.

B Deploying the application to JBoss Application Server 7

1. Download the latest final version of JBoss Application Server 7 (currently it is 7.1.1 from `http://www.jboss.org/jbossas/downloads`).

2. Extract the downloaded archive to the destined folder.

3. Create the database for the application in the database system.

4. Copy the driver for the database to the standalone/deployments folder (for example the apropriate JDBC Driver for MySQL is called Connector/J and can be obtained at `http://www.mysql.com/products/connector/`. The file `mysql-connector-java-5.1.22-bin.jar` would then be copied to `server-folder/standalone/deployments`)

5. Configure the server:

   (a) Open file `server-folder/standalone/configuration/standalone.xml`

   (b) Add the appropriate data source (inside the `datasources` xml element), example:

```
<datasource jta="false"
  jndi-name="java:jboss/datasources/
             mysqlfusionweb"
  pool-name="FusionWeb" enabled="true"
    use-ccm="false">
  <connection-url>jdbc:mysql://localhost:
    3306/fusionweb?zeroDateTimeBehavior=
    convertToNull
  </connection-url>
  <driver-class>
    com.mysql.jdbc.Driver
  </driver-class>
  <driver>
```

```
      mysql-connector-java-5.1.22-bin.jar
    </driver>
    <security>
      <user-name>root</user-name>
      <password>1234</password>
    </security>
    <validation>
      <validate-on-match>
        false
      </validate-on-match>
      <background-validation>
        false
      </background-validation>
    </validation>
    <statement>
      <share-prepared-statements>
        false
      </share-prepared-statements>
    </statement>
  </datasource>
```

The **jndi-name** must correspond to the one in `persistence.xml` file (default is this). The **connection-url** must correspond to the relevant url for connecting to the database. The **driver** must be the same as the defined driver (in the next step). The **security** element represents the credentials for logging into the database.

(c) Add the corresponding driver (inside the `drivers` xml element), for example:

```
<driver
 name="mysql-connector-java-5.1.22-bin.jar"
 module="com.mysql">
 <driver-class>
    com.mysql.jdbc.Driver
 </driver-class>
</driver>
```

(d) Add the security domain (inside the `security-domains` xml element). Example:

```
<security-domain name="fusionweb">
  <authentication>
  <login-module code="Database"
    flag="required">
  <module-option name="dsJndiName"
    value="java:jboss/datasources/
    mysqlfusionweb"/>
  <module-option name="principalsQuery"
    value="select password from User where
    username=?"/>
  <module-option name="rolesQuery"
    value="select role,'Roles'
    from UserRoles where User_username=?"/>
  <module-option name="hashAlgorithm"
    value="SHA-256"/>
  <module-option name="hashEncoding"
    value="base64"/>
  <module-option
    name="unauthenticatedIdentity"
    value="guest"/>
  <module-option name="allowEmptyPasswords"
    value="false"/>
  </login-module>
  </authentication>
</security-domain>
```

The security domain specifies the authentication of the users. The value of **`dsJndiName`** must be same as the data-source.

(e) Increase the timeout for the deployment - as during the time the deployment is created, the tables filled and the session constructed, it might not meet the default timeout for the deployment. To the `deployment-scanner` xml element add attribute `deployment-timeout="600"`.

(f) Save and close the file. The configuration can also be man-

aged through the administration console of the server.

6. Start the server. It can be achieved through IDE or the execution of the appropriate file in the `server-folder/bin` folder (for Linux based operating systems standalone.sh).

7. Deploy the application - either through IDE, console or copy the FusionWeb.war file into `server-folder/standalone/deployments`. Explore the application on: `http://localhost:8080/FusionWeb/`