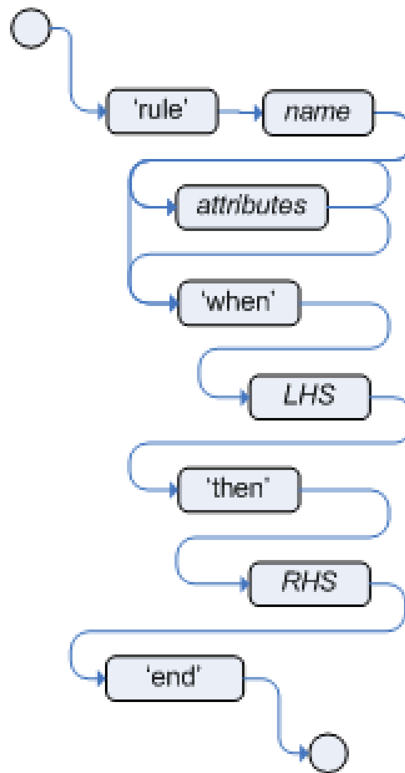
CHALMERS
UNIVERSITY OF TECHNOLOGY



Mathematical representation for the rule of the production rule system Drools

Master's thesis in engineering mathematics and computational science

AMIEL POUZAT

Department of mathematical sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

CHALMERS UNIVERSITY OF TECHNOLOGY

MASTER THESIS

Mathematical representation for rule of the business rule management system Drools

Author:
Pouzat AMIEL

Supervisor:
Casalino MATTEO
Examiner:
Sagitov SERIK

June 12, 2019

CHALMERS UNIVERSITY OF TECHNOLOGY

Abstract

Mathematical representation for rule of the business rule management system Drools

by Pouzat AMIEL

Computational science is a source of many technological paradigms as declarative programming. Declarative programming is used in industries that want to reach optimal processing. Drools is a rule engine allowing this declarative programming.

For classical programs (imperative programming), many verifying programs exist to check the behavior of these programs, depending only on the code. For a program based on declarative programming, the technics of verification are not common.

This thesis purposes a mathematical representation for the rule of the business rule management system: Drools. This representation is used to define two main errors for a set of rules: overlap and subsumption. Then, a presentation of the program developed is done.

Acknowledgements

I would like to thanks the BZR team for welcoming me for this 6 months. Special thanks to Matteo Casalino, which was my supervior, and showed me what is expected from an engineer.

Thanks to Florian and Matthieu, who helped me throughout the thesis to find and learn the technical specifics of java. I would like to thanks Florian Janel, without him, I couldn't make this thesis.

Moreover, I thanks a lot Serik Sagitov, my examiner, I owe him a big-time, and I enjoy taking his courses at Chalmers. I would like to thanks him for the time he took for me, to try to understand my project and to prepare my oral presentation. Thanks you Serik.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 2
1.1 Background . . . 2
1.2 Problem and aim . . . 2
1.3 Related work : result . . . 3
1.4 Thesis outline . . . 3

2 Business Rules and Drools 4
2.1 Production Rule Systems . . . 4
2.2 Drools Rule language (DRL) . . . 5
2.2.1 File . . . 5
2.2.2 Basic syntax . . . 6
Rule . . . 6
Facts . . . 6
2.2.3 Left Hand Side (LHS) . . . 6
2.3 Business example . . . 7

3 Mathematics representation 10
3.1 Set and Space . . . 10
3.1.1 Attribute definition set . . . 10
3.1.2 Totally ordered set . . . 10
3.1.3 Comparable attribute . . . 10
3.1.4 Interval . . . 11
3.1.5 Attribute set . . . 11
3.1.6 Operator : Attribute . . . 12
3.1.7 Pattern (effective) set . . . 12
3.1.8 Operator : Pattern and attribute . . . 14
3.1.9 Rule (effective) set . . . 15
3.1.10 Rule set : conjunctive form . . . 16
3.1.11 Important remarks . . . 17
3.1.12 Operator : Conjunctive rule . . . 18
3.2 Overlap and Subsumption . . . 20
3.2.1 Subsumption . . . 21
3.2.2 Overlap . . . 22

4 Implementation 24
4.1 Goals . . . 24
4.2 Field : Set . . . 26
4.3 Pattern : SetWrapper . . . 28
4.4 SubRule : SubRuleSet . . . 29

4.5 Verify Rules . . . 32
4.5.1 Drools-Verifier to mathematics representation . . . 32
4.5.2 Formal verification . . . 32

5 Conclusion **33**
5.1 Discussion . . . 33
5.2 Future works . . . 33

Chapter 1

Introduction

1.1 Background

IT has emerged in the industry as an essential tool for the development of tomorrow's world. Many programming languages were created, allowing the development of innovative applications.

"Business Rules" is a very intuitive programming technique. A business rule, individually, can be seen as a method that takes into variable a set of objects which respect certain conditions inherent to the rule. The rule returns another set of object resulting from the action of the rule on the set taken as variable. Very simply, a business rule represents an action to be executed for a given situation.
This principle is extremely general. The implementation of this principle will set a context in which the rules will be evaluated. This context is the rule engine.
Several rule engine implementations exist, based on pattern-matching algorithms, or from scratch. This thesis is based on the Business Rule Management System, Drools
A Business Rule Management System, BRMS, is a software allowing to manage business rules outside the application in which they apply. This externalization of the rules with respect to the app makes it possible to modify, add and remove rules according to the business knowledge. A BRMS provides an implementation of a rule engine.
The rule engine is the implementation of the Business Rules approach. For Drools, it corresponds to the implementation of PHREAK, a pattern-matching algorithm derived from RETE, another algorithm. This implementation was done in Java, by RED HAT teams, an open source company. The principle of Drools is, therefore, the principle of pattern-matching. The user defines the 'Facts' which are Java objects and confronts them to the conditions of his set of rules. If the facts' attributes respect conditions of a rule, and that rule has priority on other rules, then this rule 'fire,' which triggers its actions. In Drools, actions can be any java code.
Drools also provide a programming language to define rules and facts. (i.e. Chap.2).

1.2 Problem and aim

Drools is a pretty intuitive rule engine, but the size of the set of rules can lead to difficulty in ensuring its behavior. Drools does not provide tools to analyze the rules produced, especially to compare them. However, Drools is used in sensitive industries, where the slightest mistake can lead to significant economic losses.

Problem : For Drools, how to ensure the behavior of a set of rule ; to know, to compare and to check the situations in which a rule can fire or not.

Remark. "Fire" a rule is an action from the rule engine. A rule "fires" when the state of the working memory respects the condition related to these rules, and these rules have the priority. (i.e. Chap.2).

Aim: The objective of this thesis is to purpose a mathematical representation of business rules. This formalization, based on the Drools rule language, will be used to define semantic's errors of a set of rules by focusing on Overlaps and Subsumptions, two specific semantic's errors. Then, an implementation in Java will be carried out.

1.3 Related work : result

The work done is divided into two parts.

On the one hand, the mathematical representation of Business rules.
For the mathematical representation, the work focuses on the conditions of the rules and the interactions between the rules. Based on sets of set, the intersections and unions between rules will be defined. Then, this representation will be used to formalize overlaps and subsumptions.

On the other hand, the implementation of this representation.
The implementation makes it possible to test the mathematical representation in real cases. The implementation is done in Java, the same language used for the development of Drools and for its use.

1.4 Thesis outline

At first, we will introduce the different concepts related to Business Rules, based on Drools.
Then, we will see the proposed mathematical representation, and define the semantic errors using this formalization.
Finally, we will present the implementation done.

Chapter 2

Business Rules and Drools

Drools is a production rules system for writing "DRL" files that define the rules of our business model. These files are written using Drools' own programming language, the Drools Rule Language. A presentation of this language will be made focusing on the Left Hand Side of the Drools Business Rules. The Left Hand Side (LHS) of the rules corresponds to the conditions to be respected to be able to "fire" the rule, either to execute the actions which are associated with the rule. The LHS of a rule defines the situation in which a rule can "fire". The Left Hand Side is the main subject of this thesis.
This chapter is based on the documentation of **[1] Drools**

2.1 Production Rule Systems

In any business model, whatever it may be, there are explicitly or implicitly business rules. These business rules are the rules that manage the business, which regulates it. More generally, in a given situation, the business rules represent the desired behavior and characterize the evolution of the studied system. Each rule reacts to a certain situation, but the interaction between them can produce a multitude of results.

Drools is a rule engine called **"Production Rule Systems"**. A production rule systems is a computer program giving access to a form of artificial intelligence . This artificial intelligence consists of the behavior of a set of rules confronted with a set of facts called "working memory".
The rules of this system are called Production Rules, or simply Rules. As part of Drools, we also talk about Business Rules.

A production rule is basically composed of two parts.

- **Event(s)**: On one hand, the conditions of the rule, grouping the conditions to respect to be able to "fire" this rule. The conditions of the rules are associated with the modeling of the objects of our sample space. Thus, in the example presented at the end of this chapter, the conditions of the two rules are related to the attributes of the class "Personne".

- **Action(s)**: On the other hand, the actions to be performed when the rule "fire". In Drools, the actions performed can be any java code. However, the evolution of the state of working memory is not done under any circumstances. Thus, to notify the Drools rule engine of the state of the working memory, the rules include key-words that aim to insert, modify or delete facts in the working memory.

Working Memory The working memory is the set of objects used to react with the rules. Once placed in the working memory, the objects are then called facts.
Thus, two sets are fundamental to use Business Rules, on the one hand the working memory, on the other hand the set of rules.

2.2 Drools Rule language (DRL)

Drools provides a programming language: Drools Rule Language (DRL).

2.2.1 File

The rules are declaring in their own files. These files have a ".drl" extension. The "DRL" files are written using the programming language provided by Drools, the DRL.

A DRL file consists of several distinct components:

- package: the package name
- import: Class and functions necessary for the rules
- global: Global variables, variables outside the working memory
- queries: functions equivalent to a Left Hand Side (conditions of a rule).
- rule: the rules defined in the file

It is possible to declare a class in DRL files via the keyword: declare. Similarly, it is possible to define functions directly in a DRL, with the keyword: function.

Example : a DRL file

```
package rules.verifier.compute

import java.util.TreeMap;
import java.util.Map.Entry;
import java.util.ArrayList;

import com.amadeus.drools.verifier.ruleEngine.EngineReport;
global EngineReport engineReport;

import com.amadeus.drools.verifier.set.SubRuleSet;

declare wrap1
        key : String
        set : SubRuleSet
end

rule "Insert SubRuleSet"
salience -20
    when

    then
```

```
                TreeMap<String, Object> map = (TreeMap)
                   engineReport.getObject("Rules");
                engineReport.log("The map is :\n" + map);
                for(Entry<String,Object> i : map.entrySet()){
                        insert(new wrap1(i.getKey(),
                           (SubRuleSet) i.getValue()));
                }
end
```

Here, there is no condition (LHS empty), for the rule "Insert SubRuleSet". This is equivalent to a LHS always true, whatever the facts present in working memory.
The file starts with the package, then, the imports, globals and declarations necessary to the rules and finally the rules of the file, for this example there is only one rule, "Insert SubRuleSet".

2.2.2 Basic syntax

Rule

A Drools rules is composed of three main parts:

- The attributes of the rule: whose name
- The Left Hand Side (LHS) of the rule, the conditions.
- The Right Hand Side (RHS) of the rule, the actions.

The name of the rule must be unique per package. Thus, a rule is identifying by its name, and the package with it is associated. NewL The LHS of a rule is the set of conditions to respect to be able to "fire" the rule, to be able to trigger the associated RHS.

The RHS of a rule is the set of actions to be performed when the LHS is respected. These actions can be related to the objects of the working memory that triggered the rule, or to the entire working memory itself.
For Drools, the RHS of a rule is java code.

Facts

The facts are the objects present in the working memory.
Each fact is then characterizing by its Java class, where its attributes are defined. Attributes are used in the LHS of the rules. Then, if necessary, can be modified in the RHS.

2.2.3 Left Hand Side (LHS)

The LHS of a rule corresponds to the set of conditions that must be respected to be able to trigger the associated actions (RHS). These conditions are related to the facts of the working memory.
Example :

```
rule "example patterns"
when
    Pattern1( attribut1 == 0, attribut2)
    Pattern2( attribut1 == "String")
then
end
```

Here, two facts are necessary to trigger the rule :

- "Pattern1": a fact whose class is "Pattern1", whose "attribute1" is a number equal to 1, and whose "attribute2" is a true boolean.
- "Pattern2" : a fact whose Class is "Pattern2", whose "attribute1" is a String (a word) equal to "String".

By default, the LHS is pattern conjunction. Here, the notion of pattern refers to the PHREAK algorithm, derived from RETE and used to match the working memory with the rules.
Each pattern has a set of literals that apply to the attributes of the class in question. Like the conjunctive form between the different patterns, the conditions inherent to the attributes of a fact is conjunction (of literals). It is possible to use disjunctive forms in the LHS. Either using the keyword "or" and correctly partitioning the patterns and the literals. Either, using the negative conjunctive form, using the keyword "not".

2.3 Business example

For example : An airline company, on the advice of its engineers, wants to set up a rule engine for its business. Initially, the role of this rule engine will be to issue promotions according to the customer and his previous flights with the company. The company will ask its business experts to issue two rules to give promotions based on the number of kilometers traveled on the company, and the total amount of purchases made by the customer.
The business expert will issue two simple rules:

- When the customer has traveled more than 500,000 km then offer a discount of 50 % for the rest of the year
- When the customer has made more than 500 000 $ purchase, then offer a discount of 50% for the rest of the year

Once the rules wrote, the business expert sends them to the development team to implement them. The team will implement a 'Fact' person with four attributes.

- unique identifier of the customer
- number of kilometers traveled
- total amount of purchases
- discount allowed between 0 and 1.

In Java, a developer creates a class 'Personne' with four corresponding attributes.

```
    public static class Personne{
        public Personne(int id){
            this.id = id;
        }

        int id;
        Double km = 0.0;
        Double spent = 0.0;
        Double promotion = 0.0;

        public int getId(){return id;}

        public void setId(int id){ this.id = id;}

        public Double getKm(){return km;}

        public void setKm(Double km){this.km += km};

        public Double getSpent(){ return spent;}

        public void setSpent(){this.spent += spent;}

        public Double getPromo(){ return promotion;}

        public void setPromo(Double promo){promotion +=
           promo;}
    }
}
```

This class will serve as a fact type. Rule conditions will relate to the attributes of this class. Then, two rules will be written using the proper Drools rule language, using the class created.

```
rule "Promo km"
when
    $personne : Personne(km >= 500000)
then
    modify($personne){setPromo(0.5);};
end

rule "Promo spent"
when
    $personne : Personne(spent >= 500000)
then
    modify($personne){setPromo(0.5);};
end
```

In this example, the Business Rules interest will be to manage these rules independently from other applications. Therefore it is simple to change a rule or to add one. In that case, after a while, the company realizes that some customers travel for free. This because the promotion given to some clients is 100%.
For customers who have traveled more than 500 000 km and have spent more than

500 000 $, the promotion will accumulate, and therefore reach 100%. This error happens when the conditions of the two issued rules are satisfied for the same customer. A fact "Person" will respect both of rules if it attributes:

$$\{km \in [500000; +\infty)\} \text{ and } \{\text{ spent} \in [500000; +\infty)\}$$

Once the situation identified, the purposed solution is to add a condition to get a promotion. The rules will be modifying: if the client has a higher or equals promotion from 0.5, then the rules can not fire. The rules then become:

```
rule "Promo km"
when
    $personne : Personne(km >= 500000, promo < 0.5)
then
    modify($personne){setPromo(0.5);};
end

rule "Promo spent"
when
    $personne : Personne(spent >= 500000, promo < 0.5)
then
    modify($personne){setPromo(0.5);};
end
```

Here, this set of rules has not anymore the error giving a promotion of 100% to some clients. If one of the two rules fire, then the second one will not be able to fire since the promoted discount is already superior to that proposed.

Chapter 3

Mathematics representation

The purpose of this chapter is to give a mathematical representation to the LHS of a rule. The LHS of a rule represents the event that must occur to trigger rule actions (RHS). This event is a conjunction of pattern and may be represented as a mathematical set. Then, the intersection of two of these sets is a conjunction, and the union: a disjunction. NewL

3.1 Set and Space

3.1.1 Attribute definition set

The definition set of an attribute is all possible value for this attribute. We denote F this set. An attribute is also referred as a field. Then, element in F are denoted f.

Definition 3.1.

$$\{f, f \in F\}$$

3.1.2 Totally ordered set

A set provided with a binary relation is a partially ordered set if the provided relation satisfies the reflexivity, antisymmetry and transitivity properties on this set. A totally ordered set is a partially ordered set if, and only if, the codomain of the binary relation is equal to this set it-self. All elements on this set, all attribute instances are comparable between each other.

3.1.3 Comparable attribute

An attribute is comparable if and only if the definition set of this attribute is a totally ordered set. In Java, we define the binary relation of this set as `compareTo` with the fundamental axioms :

- Reflexivity : $\forall f \in F, \quad f.\texttt{compareTo}(f) = 0$
- Antisymmetry : $\forall f_1, f_2 \in F, \quad f_1.\texttt{compareTo}(f_2) \leq 0 \Leftrightarrow f_2.\texttt{compareTo}(f_1) \geq 0$

  if $\exists f_1, f_2 \in F, \{f_1.\texttt{compareTo}(f_2) \leq 0\} \cap \{f_2.\texttt{compareTo}(f_1) \leq 0\}$
  then $f_1.\texttt{compareTo}(f_2) = 0$

- Transitivity $\forall f_1, f_2, f_3 \in F,$

  $f_1.\texttt{compareTo}(f_2) \leq 0$ and $f_2.\texttt{compareTo}(f_3) \leq 0 \Rightarrow f_1.\texttt{compareTo}(f_3) \leq 0$

Notation f_1.`compareTo`$(f_2) \leq 0$ will be note $f_1 \leq f_2$

3.1.4 Interval

An interval is a subset of a comparable attribute definition set. We denote I this set. I has the properties :

- $I \subset F$

Definition 3.2.
$\exists i_{\max}, i_{\min} \in F$, such that

$$\forall f \in I, \quad f \geq i_{\min} \quad \text{and} \quad f \leq i_{\max} \quad \text{and}$$
$$\forall g \notin I, \quad g \leq i_{\min} \quad \text{or} \quad g \geq i_{\max}$$

In this case, we can note I as $I = \left[i_{\min}; i_{\max}\right]$

Remark. If $\forall f \in I, \quad \{f > i_{\min}\}$ then
$I = \left(i_{\min}; i_{\max}\right]$

Remark. If $\forall f \in I, \quad \{f < i_{\max}\}$ then
$I = \left[i_{\min}; i_{\max}\right)$

We denote I_F all possible interval on F.

Disjoint interval Two intervals I, J are disjoint if and only if

$$I \cap J = \{\emptyset\}$$

Set of disjoint interval For a set a disjoint interval, we can associate to this set a binary relation respecting the axioms: reflexivity, antisymmetry, and transitivity.

3.1.5 Attribute set

An attribute set represents a restriction on F, the definition set of the attribute. We denote E the effective set associated. If this attribute is comparable, then E is a countable union of disjoint interval I on F.

Definition 3.3.

$$E = \cup_{i=0}^{n} I_i \tag{3.1}$$
$$\forall i, j, i \neq j \Rightarrow I_i \cap I_j = \{\emptyset\} \tag{3.2}$$

The effective attribute space will represent all possible value for an attribute to match a given rule. For example : Given an object 'Fact' with an integer attribute 'value'. Let's have a rule:

```
declare Fact
    value : int
end

rule "Your really First rule"
when
    $fact : Fact( value > 10
    Fact( value != $fact.value
then
    delete($fact);
end
```

FIGURE 3.1: Attribute space target

```
declare Fact
    value : int
end

rule "Rule sup to 0"
when
    Fact( value > 0)
then
    System.out.println("The value is superior to 0");
end
```

Then for 'value' the definiton set without user's constraint is : $F = \mathbb{Z}$
The attribute effective set associated with "Rule sup to 0" is : $E = \mathbb{N}^{+*}$

So, the attribute set represent the literal "value > 0" as :

$$\{\text{value} \in \mathbb{Z}, \text{such that} : \text{value} \in \mathbb{N}^{+*}\}$$

3.1.6 Operator : Attribute

An attribute set is a one-dimensional space. An attribute set is associated with the standard operator of intersection ($\cap$) and union ($\cup$).

3.1.7 Pattern (effective) set

A pattern effective set is a set of attribute set. This set regroups the different attribute space given by the related pattern. It is a conjunction of attribute space.

```
declare Fact
    value : int
end

rule "Your really First rule"
when
    $fact : Fact( value > 10)
    Fact( value != $fact.value)
then
    delete($fact);
end
```

FIGURE 3.2: Pattern set target

We denote P this set. For example :

```
declare Fact
    value : int
    name : String
end

rule "Sample pattern"
when
    Fact( value == 42, name == "Doe")
then
    System.out.println("Where is Jhon ?");
end
```

The rule "Sample pattern" gives one pattern set. This set is :

$$P_0 = \{\{\text{value} = 42\}\} \text{ and } \{name = "Doe"\}\}$$

The imprecisely notation $'\text{value} = 42'$ is related to the precisely one : $'\ \text{value} \in [42; 42]'$.

Remark. A pattern set represent a conjunction. It is a conjunction of literal applying on each attribute into the fact.

Definition 3.4.

A pattern set P is a conjunction of literal, representing by a composition of attribute set, with an index 'n'.

$$P = \prod_{n \in \mathbb{N}} A_n$$

The index is essential here. The attribute space index is, in reality, the attribute name. Then, we should note $A_{\phi(n)}$ and not A_n, with ϕ a function returning the attribute name associated.

A pattern set is conjunction but with an index constraint (pattern index is different from the attribute index). Then, two pattern sets, into the same rule, are comparable if and only if their index is the same, so their 'position' into the rule. This possibility can happen when rules include a disjunction (a or). For example :

```
declare Fact
    value : int
    name : String
end

rule "Sample disjunction"
when
    Fact( value == 42 || value == 0, name == "Doe")
then
    System.out.println("Where is Jhon ?");
end
```

In this situation, rule produces two pattern set, one with 'value == 42' and another with 'value == 0'.

$P_0 = \{\{\text{value} = 42\}\} \cap \{name = "Doe"\}\}$
$P_1 = \{\{\text{value} = 0\}\} \cap \{name = "Doe"\}\}$

Comparable pattern set

Definition 3.5. Two pattern set P_1 and P_2 are comparable if and only if their definition set are equals, $\mathcal{P}_1 = \mathcal{P}_2$

This definition is saying two pattern sets are comparable if and only if the related fact type is the same. This definition is general and doesn't concern a pattern set inside the same rule. In this case, we have to take care of the index position.

3.1.8 Operator : Pattern and attribute

A pattern set is a conjunction of attribute effective space. An intersection or union of pattern set not related to the same fact type has no meaning. We have the property $P = \prod_{i \in \mathbb{N}} A_i$. Index here is, in reality, the attribute name.

Intersection For two pattern set related two the same fact type target ($P_1, P_2 \in \mathcal{P}_{\text{target}}$)

$$P_1 \cap P_2 = (\prod_{i=0}^{m} A_{(\text{target},1,i)}) \cap (\prod_{i=0}^{m} A_{(\text{target},2,i)})$$

Definition 3.6.

$$P_1 \cap P_2 = \prod_{i=0}^{m} (A_{(\text{target},1,i)} \cap A_{(\text{target},2,i)})$$

If one attribute set into the resulting pattern set is empty, then the pattern set is considered as empty, and so the operator returns an empty pattern set. This can happen when an intersection of two attribute sets is empty.

Respect

Definition 3.7. Respect
A pattern set P_1 respect an other pattern set P_2 if and only if

$$P_1 \cap P_2 \neq \{\emptyset\}$$

Union For two pattern set related two the same fact ($P_1, P_2 \in \mathcal{P}_{\text{target}}$)

$$P_1 \cup P_2 = (\prod_{i=0}^{m} A_{(\text{target},1,i)}) \cup (\prod_{i=0}^{m} A_{(\text{target},2,i)})$$

$$P_1 \cup P_2 = \prod_{i=0}^{m} (A_{(\text{target},1,i)} \cup A_{(\text{target},2,i)})$$

However, a union may also respect the conjunctive property. So if an intersection of the two patterns set is empty, then, union operator returns an empty pattern set.

Contains A pattern set P_1 contains an other pattern set P_2 if

Definition 3.8. Contains

- P_1 and P_2 are comparable
- $\forall i, A_{(\text{target},1,i)})$ contains $A_{(\text{target},2,i)})$

3.1.9 Rule (effective) set

As for the pattern set, the rule set is a set of pattern set given by the rule. We denote R this set.

```
declare Fact
    value : int
    name : String
end

rule "Or clause"
when
    or(
      Fact( name == "Doe"),
      Fact( value == 42)
    )
then
    System.out.println("Where is Jhon ?");
end
```

The set regroups the both pattern set of the last example. The set is :

$R = \{\{P_0\} \cup \{P_1\}\}$

3.1.10 Rule set : conjunctive form

```
declare Fact
    value : int
end

rule "Your really First rule"
when
    $fact : Fact( value > 10)
    Fact( value != $fact.value)
then
    delete($fact);
end
```

FIGURE 3.3: Conjunctive Rule set target

The previously defined rule set R is subject to a disjunction. However, Drools is oriented conjunction, the default operator between two patterns are an "and". From a given disjunction, we will create an element corresponding to each conjunction inside the disjunction. We denote CR this set, for conjunctive rule.
From the rule 'Or clause', we can create two subrules corresponding to:

$CR_0 = \{P_0\}$

$CR_1 = \{P_1\}$

Such that we can note $R = \{\ CR_0 \cup CR_1\ \}$
And $\forall R \in \mathcal{R}, R = \cup_{i=0}^{n} CR_i$, $n+1$ number of conjunction forming the rule.

Definition 3.9.

A conjunctive rule set CR is a conjunction of pattern, representing by pattern set, with an index of order 'n'.

$$CR = \prod_{n \in \mathbb{N}} P_n$$

Also here, the index is essential. Two patterns are comparable if the fact type related to is the same. Then we compare attribute space into pattern set. However, a common error can be to compare only literal without taking care of the index. (Even if to fact include an attribute id, this doesn't imply their fact types are equals)

```
declare Fact
    value : int
    name : String
end

rule "And clause"
when
      Fact( name == "Doe")
      Fact( value == 42)
then
    System.out.println("Where is Jhon ?");
end
```

In this rule, if the working memory includes a fact 'Fact', with an attribute 'name' equals to "Doe", and an attribute 'value' equals to 42

(Fact["Doe", 42])

, then the rule will fire, even with only one fact.
But, the rule includes two different pattern, and in this situation, two fact :

(Fact["Doe", 0] and Fact["Serik", 42])

may fire the rule also. A conjunctive rule set is a composition of pattern set, keeping index order.

3.1.11 Important remarks

We will define an operator of intersection and union, which will be different from a composition of attribute or pattern set. The operator $\cap$ mean "and". For example :

```
rule "I- Remarks"
when
      Fact( name == "Doe")
      Fact( value == 42)
then
    System.out.println("Where is Jhon ?");
end
```

This rule give a conjunctive rule set

$$CR = \text{Fact}_0(\text{ name } \in [\text{"Doe"}]) \text{ and } \text{Fact}_1(\text{ value } \in [42])$$

Which is different from

$$\text{Fact}(\text{ name } \in [\text{"Doe"}]) \cap \text{Fact}(\text{ value } \in [42]) = \text{Fact}(\text{ name } \in [\text{"Doe"}] \text{ and } \text{ value } \in [42])$$

This second equation is equivalent to

$$\text{Fact}_0 \cap \text{Fact}_0$$

3.1.12 Operator : Conjunctive rule

A conjunctive rule set is a conjunction of pattern effective set. The difficulty in operating on this set is to take care of fact type and pattern positions. Pattern position may be different for two comparable patterns when working on two different rules,.

First, we will define an intersection operator. An intersection between two sets is the common element of these sets. However, for two conjunctive rule sets, many conjunctive rule set may respect this definition, more than one. So we should return a rule set (under the disjunctive shape) for an intersection of two conjunctive rule set.

$$. \cap . : \mathcal{CR}x\mathcal{CR} \to \mathcal{R}$$

But to stay under a conjunctive shape, intersection and union operator should return set into the same definition set.

$$. \cap . : \mathcal{CR}x\mathcal{CR} \to \mathcal{CR}$$

To keep the conjunctive properties, we return the first conjunctive rule set respecting intersection properties ('common set').

Let's start by an intersection and union with a pattern set.

Weak/Strong intersection For a conjunctive rule set (CR_1) with a pattern set (P_{fact}). Denoting P_f the first pattern set into CR_1, such that P_f and P_{fact} are comparable.

Definition 3.10. Weak/Strong intersection

$$CR_1 \cap P_{\text{fact}} = \prod_{n \geq 0, n \neq f} P_n \quad * (P_{\text{fact}} \cap P_f)$$

If $P_{\text{fact}} \cap P_f = \{\emptyset\}$, operator will search for the next comparable pattern into CR_1.

Strong If there exists no more comparable set into CR_1, then operator return an empty set.

Weak If there exists no more comparable set into CR_1, then operator add pattern set into it.Weak intersection is used to construct conjunctive rule set.

Strong Respect

Definition 3.11. Strong respect
A pattern effective set P_{fact} strong respects a conjunctive rule CR_1 if

$$CR_1 \cap P_{\text{fact}} \neq \{\emptyset\}$$

with strong intersection

Then, we define respect between pattern set and conjunctive rule set. Under a weak intersection, the operator returns an empty set if he finds no comparable pattern which respects the given pattern set.

Weak Respect

Definition 3.12. Weak respect
A pattern effective set P_{fact} weak respect a conjunctive rule CR_1 if

$$\exists P \in CR_1 \text{ such that } P \text{ and } P_{\text{fact}} \text{ are comparable}$$

Moreover, if a pattern strong respectsa conjunctive rule, then this pattern weak respect this conjunctive rule.

Strong Union For union between a conjunctive rule set and pattern set, same idea, find the first comparable and try to make a union with this given pattern set, else find the next comparable pattern. As union operator should return a conjunctive rule set and not a rule set, we may be to take care than a union returns a conjunctive rule set or an empty set if it is not possible.

Definition 3.13. Strong union

$$CR_1 \cup P_{\text{fact}} = \prod_{n \geq 0, n \neq f} P_n \quad * (P_{\text{fact}} \cup P_f)$$

If there exist no comparable set into CR_1, then operator returns an empty set.
If $P_{\text{fact}} \cap P_f = \{\emptyset\}$, operator will search for the next comparable pattern into CR_1.
If intersection is empty for each comparable pattern into CR_1, the conjunctive property is not longer respected, and so union may return an empty set.

The weak intersection is used to construct the set. We continue with the strong operator to operate on two conjunctive rule set.

Intersection Now we got an operator to operate on a conjunctive rule set and a pattern set, let's construct operator for two conjunctive rule set. We got the constraint to stay on the same definition set. An intersection of two conjunctive rule set, which returns the common set between both, return a conjunctive rule set, even if more than one conjunctive rule set respects definition.
For two conjunctive rule sets CR_1 and CR_2

Definition 3.14. Intersection between two conjunctive rule set, $CR_1 \cap CR_2$
$CR_c = CR_2$
for each pattern $P_{1,i}$ in CR_1 :
- $CR_c = CR_c \cap P_{1,i}$
return CR_c

This operator is not symmetric. It returns the "first result" found, the first conjunctive rule set which is common to CR_1 and CR_2, starting by CR_2. This definition can be confusing. $CR_1 \cap CR_2$ will be note CR_1 'under' CR_2. This because we start from the CR_2 set, and then we try to compute an intersection with each CR_1's patterns. If CR_1 under CR_2 is empty, then intersection is empty starting by CR_2. But operator is not symmetric. Then, CR_2 under CR_1 may be not empty. This is related to "subsumption".

Strong Respect

Definition 3.15. Strong Respect
A conjunctive rule set CR_2 strong respects a conjunctive rule CR_1 if

$$CR_2 \cap CR_1 \neq \{\emptyset\}$$

Remark. Strong respect, or simply respect between two conjunctive rule set is not symmetric, because intersection between two conjunctive rule set is not symmetric.

Union Union operator is defined into a similar way. With same notation.

Definition 3.16. Union between two conjunctive rule set $CR_1 \cup CR_2$
$CR_c = CR_2$
for each pattern $P_{1,i}$ in CR_1 :
- $CR_c = CR_c \cup P_{1,i}$
return CR_c

3.2 Overlap and Subsumption

Let's define overlap and subsumption.

3.2.1 Subsumption

Definition 3.17. For two rule "rule 1" and "rule 2" which we associate two rule sets R_1 and R_2.

"rule 1" subsumes a rule "rule 2" if and only if R_2 strongly respects R_1.

Example

```
declare Fact
    value : int
end

rule "Rule 1"
when
    Fact(value > 0)
then
end

rule "Rule 2"
when
    Fact()
then
end
```

In this simple example, the rule "Rule 1" subsumes the rules "Rule 2" on the pattern set $P = \text{Fact}(\text{value } \in \mathbb{N}^{+*})$

To compute this set, first we compute the conjunctive rule set for "Rule 1" :

$$P_1 = \text{Fact}(\text{value } \in \mathbb{N}^{+*})$$

$$R_1 = \{P_1\}$$

Then, for the second rule "Rule 2", there is no condition about attribute, so :

$$P_2 = \text{Fact}(\text{value } \in \mathbb{Z})$$

$$R_2 = \{P_2\}$$

We will check if "Rule 2" subsumes "Rule 1". Here, let find a counterexample : $-1 \notin \mathbb{Z}$, so "Rule 2" does not subsume "Rule 1".
Then, regarding if "Rule 1" subsumes "Rule 2". We have to check if P_2 contains P_1.

$$\forall \text{value } \in \mathbb{N}^{+*}, \text{ value } \in \mathbb{Z}$$

3.2.2 Overlap

Definition 3.18. For two rule "rule 1" and "rule 2" which we associate two rule set R_1 and R_2.

There is an overlap between "rule 1" associated to the rule set R_1 and "rule 2" associated to R_2 if and only if

$$\exists P \in R_2 \text{ such that } P \text{ strong respect } R_1$$

This characterizes the definition of an intersection between two sets. There is an overlap if there is a common set, no empty, between both rules. So for these common sets, rules can both fire and create a no desired situation.

Remark. An overlap is symmetric.

Example

```
declare Fact
    value : int
    id : int
end

rule "Rule 1"
when
    Fact(value > 0, id == 0)
    Fact( id == 11)
then
end

rule "Rule 2"
when
    Fact(value > 0, id == 0)
    Fact( id == 10)
then
end
```

First, there is no subsumption between this two rules. The second pattern inside both rule does not respect the other rule.

$$R_1 = \{\text{Fact(value } \in \mathbb{N}^{+*} \text{ and id } = 0) \text{ and Fact(value } \in \mathbb{Z} \text{ and id } = 11)\}$$

$$R_2 = \{\text{Fact(value } \in \mathbb{N}^{+*} \text{ and id } = 0) \text{ and Fact(value } \in \mathbb{Z} \text{ and id } = 10)\}$$

Trying to match Fact(value $\in \mathbb{Z}$ and id $= 11)$ with R_2 is not possible. Same, Fact(value $\in \mathbb{Z}$ and id $= 10)$ does not match R_1.
But, there is an overlap for

$$P = \text{Fact(value } \in \mathbb{N}^{+*} \text{ and id } = 0)$$

which represent the common facts between both rules.

Remark. An subsumption is not symmetric.

Chapter 4

Implementation

4.1 Goals

Goals of implementation are to provide a java program, to process to static analysis of DRL files.
To start, we have to create a wrapper for Drools. This wrapper is a Java package, regrouping some class to manipulate Drools and Maven easily.
For example, by defining a DRL "autoFocus.drl", which is put into the resources folder of the maven project (see Maven Documentation for more). Then, the software calls the 'engine' and fires the session exactly as it fires a KIE session.

```
RuleEngine session = new RuleEngine();

session.addEngineResources("autoFocus.drl");

session.insert(new AnInt(0));

session.fire();
```

The "insert method" insert to working memory a fact 'AnInt', which is an object with one integer attribute, 'value'. The "fire" method matches working memory and rules.

With this wrapper, we extend it to create the 'verifier engine'. This verifier engine is composing in two part :

- Parse DRLs into some facts
- Check DRLs with Drools rules

Using Drools-Verifier, we recover the resulting objects from its parser (facts), and we put them into working memory. Then, we apply the rules to create the different sets.

The verifying rules will check for two given conjunctive rule sets, if one subsumes the other, and if they overlap each other.

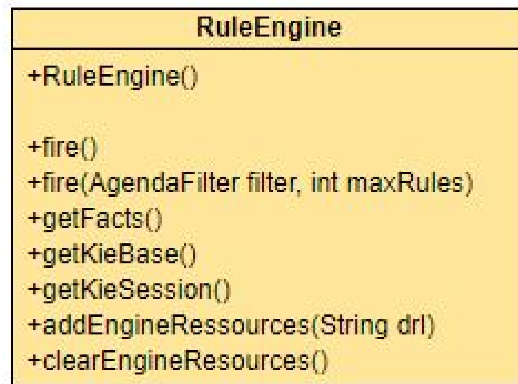


FIGURE 4.1: Drools wrapper : basic UML

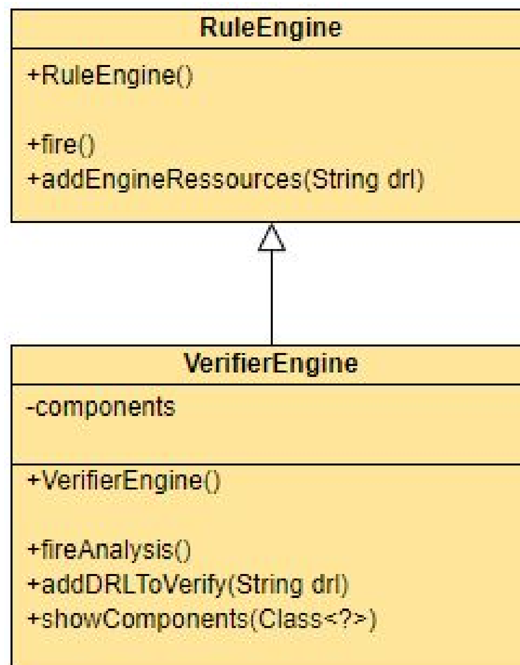


FIGURE 4.2: Basic UML : Drools wrapper

The method "addDRLToVerify" parses the given DRL into components. The verifier uses theses components as facts. You can have some console logs for these components using "showComponents". For example, using the DRL file "autoFocus.drl" :

```
VerifierEngine session = new VerifierEngine();

session.addDRLToVerify("autoFocus.drl");

session.showComponents();
```

Resulting to the log :

```
[Engine] INFO Components number -  52

[Engine] INFO TextConsequence: {
engineReport.log("");

[Engine] INFO Field 'bool' from object type
   'com.amadeus.drools.verifier.ruleEngine.EngineReport'

[Engine] INFO Field 'child' from object type
   'com.amadeus.drools.verifier.ruleEngine.EngineReport'

[Engine] INFO Field 'classId' from object type
   'com.amadeus.drools.verifier.ruleEngine.EngineReport'

    ... etc
```

Moreover, you can use the method "addEngineRessources" to add a verifying set of rule.
For now, there is two main DRL file for the project, into folder "src/main/resources/rules/verifier/compute/" :

- "compute.drl"
- "overlap3.drl"

The first file is the set of rules which computes the rule sets ('SubRuleSet') associated with each conjunctive rule. Drools-Verifier parses DRL into some java object. For each rule, Drools-Verifier creates some 'SubRule' objects, a Java object corresponding to a conjunctive rule. The file associate one 'SubRuleSet' for each 'SubRule' created by Drools-Verifier.
The second file checks if there are some subsumptions on a given set of rules, using the 'SubRuleSet' computed by the previous file.

4.2 Field : Set

We associate an attribute set for each literal into rules we analyze. Space is programming as 'Set'. 'Set' can be directly construct using a literal. The operator and value have to be separated into two variables. The supported operators are :

- Equals "=="
- Different "$\neq$"
- Inferior "<"
- Inferior or equals "$\leq$"
- Superior ">"
- Superior or equals "$\geq$"

For example :

```java
Set<Double> S9 = new Set<>(">", 9.0);

Set<Double> I20 = new Set<>("<", 20.0);

Set<Double> E15 = new Set<>("==", 15.0);

log.log("The set Superior to 9 is : " +
    S9.toString());

log.log("The set Inferior to 20 is : " +
    I20.toString());

log.log("The set Equals to 15 is : " +
    E15.toString());

log.log(" x > 9 and x < 20 : " + S9.cap(I20));

log.log("x <= 9 or x == 15 or x >= 20" +
    S9.complemetaire().cup(E15).cup(I20.complemetaire()));

log.log(" Empty : " +
    E15.complemetaire().cap(E15).set);

log.log(" Full : " +
    E15.cup(E15.complemetaire()).getInterval(0));
```

result to :

```
[main] / The set Superior to 9 is : Set : ]9.0 ; +Inf[

[main] / The set Inferior to 20 is : Set : ]-Inf ; 20.0[

[main] / The set Equals to 15 is : Set : [15.0]

[main] /  x > 9 and x < 20 : Set : ]9.0 ; 20.0[

[main] / x <= 9 or x == 15 or x >= 20Set : {]-Inf ; 9.0] or
   [15.0] or [20.0 ; +Inf[}

[main] /  Empty : []

[main] /  Full : Interval : ]-Inf ; +Inf[
```

Methods 'cap' represents an intersection operator, and methods 'cup' a union operator. The method 'isUndefined' return true is the 'Set' is undefined. There is more method to manipulate a 'Set' or an 'Interval'. For example, 'Set' can be used as a collection, with the method 'add, remove, size, contains, ...'.

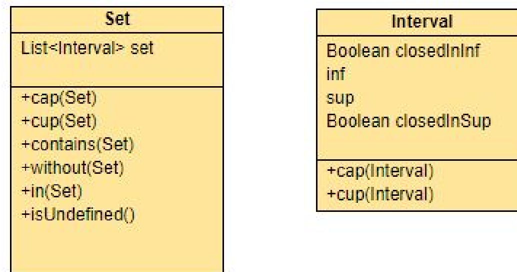


FIGURE 4.3: Basic UML : Set and Interval

Moreover, some different indexes are managing with this class. For example, to manage pattern orders inside a conjunctive rule, we use a set of String ('Set<String>') to know the fact name, and an integer to see the order.

4.3 Pattern : SetWrapper

SetWrapper represents a pattern effective set. A pattern set is a conjunction of literal on the attributes associated to the pattern. A 'SetWrapper' regroups this literals into one object, but doesn't keep information about pattern fact type. This role is assumed by 'SubRuleSet'.

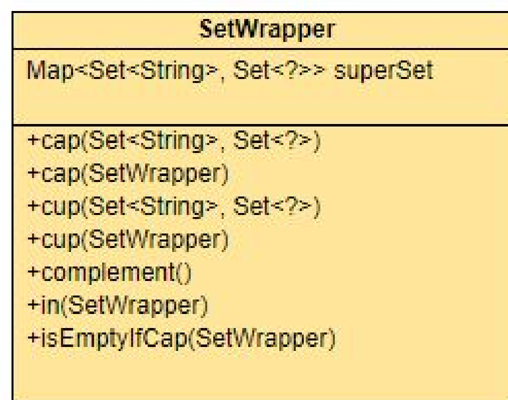


FIGURE 4.4: Basic UML : SetWrapper (pattern set)

'SetWrapper' is using a map associating a key 'Set<String>' to a value 'Set'. The key represents the attribute name.
For example :

```
Set<Double> S9 = new Set<>(">", 9.0);

Set<Double> I20 = new Set<>("<", 20.0);

Set<Double> E15 = new Set<>("==", 15.0);

Set<Double> C0 =
    S9.complemetaire().cup(E15).cup(I20.complemetaire());
```

```
SetWrapper P0 = new SetWrapper();

P0.addOrCap("v1", S9);
P0.addOrCap("v2", I20);

SetWrapper P1 = P0.copy();

P1 = P1.cap("v1", I20).cap("v2", C0);

SetWrapper P2 = P1.cup(P0);

log.log("P0 : " + P0);

log.log("P1 : " + P1);

log.log("P2 equals P0 ? " + P2.equals(P0));
```

Resulting to the log :

```
       [main] / P0 : Map-Set :
{ {["v1"] in ]9.0 ; +Inf[} and {["v2"] in ]-Inf ; 20.0[} }

[main] / P1 : Map-Set :
{ {["v2"] in ]-Inf ; 9.0]} and {["v1"] in ]9.0 ; 20.0[} }

[main] / P2 equals P0 ? true
```

Morever, 'SetWrapper' can be use as a 'Map'. 'SetWrapper' doesn't implements 'Map' but contains methods "put, remove, contains". You can get a value associating to a key using "getValue" and the key.

4.4 SubRule : SubRuleSet

SubRuleSet' represents a conjunctive rule set. It was named like this because Drools-Verifier used a 'SubRule' object to describe conjunctive rule set. However, goals are to identify overlap and subsumption and gives back a corresponding conjunctive rule set.
To check if a SubRuleSet A subsumes a SubRuleSet B, we have to use "under" method.' A "under" B ' return a conjunctive rule set, which matches A and B. With specificity to try to fire the rule B using all pattern in A.

To check if a SubRuleSet A and B overlap each other, you have to find all pattern which strong respect the other conjunctive rule. This verification is done by, first, checking pattern index (fact name) and then if the intersection is not empty. (Strong respect).

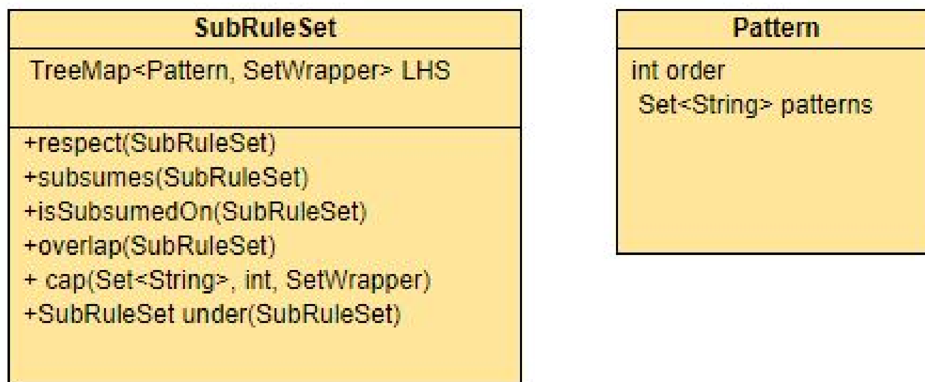


FIGURE 4.5: Basic UML : SubRuleSet

For example :

```
Set<Double> S9 = new Set<>(">", 9.0);

Set<Double> I20 = new Set<>("<", 20.0);

Set<Double> E15 = new Set<>("==", 15.0);

Set<Double> C0 =
    S9.complemetaire().cup(E15).cup(I20.complemetaire());

SetWrapper P0 = new SetWrapper();

P0.addOrCap("v1", S9);
P0.addOrCap("v2", I20);

SetWrapper P1 = P0.copy();

P1 = P1.cap("v1", I20).cap("v2", C0);

SetWrapper P2 = P1.cup(P0);

SubRuleSet R1 = new SubRuleSet();
SubRuleSet R2 = new SubRuleSet();

R1.cap("Pat", 0, P0).cap("Pat", 1, P1);
R2.cap("Pat",0,  P2).cap("Pat",1,  P0);

log.log("LHS 1 : " + R1.asLHS());

log.log("LHS 2 : " + R2.asLHS());

log.log("1 subsumes 2 ? " + !R1.under(R2).isEmpty());
log.log("on ? " + R1.under(R2).reduce().asLHS());
```

```
        log.log("2 subsumes 1 ? " + !R2.under(R1).isEmpty());
        log.log("on ? " + R2.under(R1).reduce().asLHS());
```

which gives back the log :

```
[main] / LHS 1 : ...
when
        Pat( v1 in ]9.0 ; +Inf[, v2 in ]-Inf ; 20.0[)
        Pat( v2 in ]-Inf ; 9.0], v1 in ]9.0 ; 20.0[)
then
...

[main] / LHS 2 : ...
when
        Pat( v2 in ]-Inf ; 20.0[, v1 in ]9.0 ; +Inf[)
        Pat( v1 in ]9.0 ; +Inf[, v2 in ]-Inf ; 20.0[)
then
...

[main] / 1 subsumes 2 ? true

[main] / on ? ...
when
        Pat( v2 in ]-Inf ; 9.0], v1 in ]9.0 ; 20.0[)
then
...

[main] / 2 subsumes 1 ? true

[main] / on ? ...
when
        Pat( v2 in ]-Inf ; 9.0], v1 in ]9.0 ; 20.0[)
then
...

[main] /  -- end --
```

The method 'subsumes' check if a 'SubRuleSet' under another one isn't empty. The method 'reduce' is the method 'under' applying to the same 'SubRuleSet'.
To build a 'SubRuleSet', we use the method 'cap'. The method 'cap' corresponds to an intersection. So when adding a pattern into the conjunctive rule set is the same to make: "this conjunctive rule set AND (∩) this pattern.", so: "this 'SubRuleSet' AND (∩) this 'SetWrapper'".

4.5 Verify Rules

4.5.1 Drools-Verifier to mathematics representation

Drools-Verifier provides us some java objects. These objects are not the set we implemented. To compute to this set, we use Drools rules. Into file "compute.drl", we associate different object to 'SubRuleSet'.

4.5.2 Formal verification

We have to use a method to check overlap and subsumption. This is done into 'overlap3.drl'. We recover the computed 'SubRuleSet', and we compare them together using rules.
Then, if we find overlap or subsumption, we create an engine message and keep it into a global variable. Users can get this message using the rule engine (verifier engine).

```
session.logResult();
SubRuleSet A0 = ((SubRuleSet) ((TreeMap<String, Object>)
    session.getReport().getObject("Rules")).get("A(0)"))
```

Moreover, user can recover the computed 'SubRuleSet' using 'getReport().getObject("Rules"))' and the rule name.
In "A(0)", the number "0" correspond to the first conjunctive rule set computed for the rule "A", the first "SubRule" computed by Drools-Verifier.

Chapter 5

Conclusion

5.1 Discussion

Declarative languages, as Drools provides with the DRL, can handle large amounts of data, basing their complexity on the number of rules in the system, and not on the number of objects, or facts. The use of computer programs based on its declarative languages saves time and money by allowing quick processing and easy maintenance. However, there is almost no solution to directly manipulate the rules of a program and thus ensure the behavior of it.
The mathematical representation proposed in this thesis is then a formalization of the computer definition of a Drools rules. This formalization is both simple, for the conditions of rules, and complex, for the actions of rules. However, when we want to check a set of rules, the actions of rules are much less important than the conditions of them. The actions of rules are summarily a change into the working memory. Considering only this, to ensure that two rules do not subsume, we only must compare the conditions of these rules, and not their actions.
An advantage in the program is the use of DRL file for the verification. The user of the program can use the model and the method to create his errors and check them with a simple DRL file.

5.2 Future works

Drools-Verifier uses the package descriptor to create its owns objects. However, the Package Descriptor is already a java representation of the DRL file. But this is not sufficient for many cases like to check the behavior of a whole session. Start from a Kie Session can be more exact.
Then, define generics semantic's errors and use the mathematics representation to solve it. By it, I mean to create a UI to create verifying rules. User should not write a DRL but should be able to verify his rules using the verifier.
Rule attributes are not managed. But attribute can add special behavior. Into this project, rules are comparing two by two. Manage attribute need to manage rule n by n, because each rule can infer it.
Moreover, this is also because we don't manage the working memory. We manage rules, and if there an object into the working memory related to zero rule, we can not see it. But how to? This problem is related to the completeness: each fact should participate in firing at least one rule and was not a goal of these project
The developed program doesn't handle the actions of rules (RHS). However, even if to check all possible actions is too complicated, the parts particular to Drools is verifiable. Drools provide three main key-words for the RHS: insert, modify, and delete. Parsing the actions correctly, using the general shape of this key-word is

possible. Then, we can check some other errors, as the coherence of the rule, for example. Moreover, the program is base on Drools-Verifier, a Drools-components. Unfortunately, this component has been abandoned for many years. Many upgrades of these components are possible, as the addition of the notion of 'template' in Java, which will simplify many facts created by Drools-Verifier.

Bibliography

[1] Red Hat (2018), Drools documentation 7.7

[2] A. Awad, G. Decker, M. Weske (2008),
Efficient Compliance Checking Using BPMN-Q and Temporal Logic

[3] V. Y. Lee, Y. Liu (June 2012),
ACARP: Auto Correct Activity Recognition Rules Using Process Analysis Toolkit (PAT)

[4] G. J. Nalepa, S. Bobek, A. Ligeza, K. Kaczor (2011),
HalVA - Rule Analysis Framework for XTT2 Rules

[5] P. Wolper (1983),
Temporal Logic Can Be More Expressive

[6] S. Lukichev (2014),
The Declarative Approach for Anomaly Detection in Production Rule Bases with Semantic Constraints

[7] M. Dohring, S. Heublein (2012),
Anomalies in Rule-Adapted Workflows - A Taxonomy and Solutions for vBPMN

[8] Tony Rikkola (2012),
Drools-Verifier (Drools components)

[9] Bruno Berstel-Da Silva (2014),
Verification of Business Rules Programs

[10] Lucas Amador (2012),
Drools Developer's Cookbook

[11] Esteban Aliverti,
Mariano De Maio, Mauricio Salatino (2016), Mastering JBoss Drools 6

[12] Von Halle, Barbara. Wiley (2001),
Business Rules Applied [electronic resource]: Building Better Systems Using the Business Rules Approach.