Logic Formalization and Automated Deductive Analysis of Business Rules

Johannes Schramm
Diplomarbeit an der Technischen Universität Darmstadt – 25. November 2014
1. Gutachter: Prof. Dr. Reiner Hähnle
2. Gutachter: Prof. Dr. Martin Ziegler
Betreuer: Richard Bubel

TECHNISCHE UNIVERSITÄT DARMSTADT

Software Engineering Group

Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmitteln verwendet zu haben.

Johannes Schramm
Darmstadt, den 25.11.2014

Abstract

Automated formal verification of certain properties of business rule management systems (BRMS) is demanded by companies using such systems in productive environments. We implement this process for certain termination properties of the BRMS Drools. Syntax and structural operational semantics for fragments of the Drools Rule Language (DRL) are defined and used to proof a termination criterion for DRL. The program verifying these fragments is available to the public.

Acknowledgements

I would like to thank Prof. Dr. Reiner Hähnle and Prof. Dr. Martin Ziegler who made this thesis possible. Furthermore, I am indebted to Maik Weinard from Capgemini who supplied me with the real-world examples of business rules.

Special thanks go to Richard Bubel for giving me constant advice and support while writing this thesis and helping me with many organizational issues.

Contents

1. Introduction

Business rule management systems (BRMS) play a crucial role in the organizational processes of many companies, public agencies and other enterprises. BRMS provide an abstraction layer to existing IT infrastructure which allows to capture the business logic regarding the data and functions provided by the underlying systems. For a better understanding of what business rules are, we would like to quote [5, p. 4-5]:

> *"A business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business. (...)*
>
> *From the information system perspective, it pertains to the facts that are recorded as data and constraints on changes to the values of those facts. That is, the concern is what data may, or may not, be recorded in the information system. (...) Accordingly, a business rule expresses specific constraints on the creation, updating, and removal of persistent data in an information system."*

In most BRMS one can combine collections of business rules that circle around a common topic in some kind of storage. Such storage is called *rule base* (RB). The properties of RBs and especially the interaction of the rules within a given RB are the central theme of this thesis. More formally, one could imagine a RB as a structure of the following form:

$$\begin{array}{rccc} rule_1: & \langle preconditions_1 \rangle & \longrightarrow & \langle conclusions_1 \rangle \\ rule_2: & \langle preconditions_2 \rangle & \longrightarrow & \langle conclusions_2 \rangle \\ rule_3: & \langle preconditions_3 \rangle & \longrightarrow & \langle conclusions_3 \rangle \\ \vdots & \vdots & \vdots & \vdots \\ rule_n: & \langle preconditions_n \rangle & \longrightarrow & \langle conclusions_n \rangle \end{array}$$

Here the preconditions of a rule define when it can be applied and the conclusions define the effect of this application. Now one might ask how these preconditions and conclusions influence each other and what happens when we repeatedly evaluate and apply rules? In some cases, the answer could be that we enter a vicious circle. The effect of one rule possibly makes the precondition of some other rule true; and the conclusion is an effect which causes the next precondition to be true and so forth, thus creating a never-ending cascade of effects.

In this thesis we introduce a way to detect some of these infinite loops and present an implementation that is capable to execute this detection for certain RBs of the open-source BRMS Drools. This implementation heavily relies on the software verification tool AProVE [8]. Like this tool our approaches can be categorized in the field of formal software verification where one is interested in the automated verification of certain properties of programs. From this perspective, we try to verify the termination property for RBs, which can be considered as programs for the rule interpreter of Drools.

The demand for the formal verification of RBs of Drools came from Capgemini, an IT company which provides consulting and custom software solutions. The software engineers at Capgemini use Drools to develop register applications for public agencies in Germany. From companies' point of view, formal software verification can be seen under two aspects: On the one hand, it provides useful help when developing and debugging programs; on the other hand, it can help to assure certain software quality measures, which are especially of interest when the provided programs need to be in accordance with certain laws.

The RBs of Drools, which we cover in the theoretical part of this thesis, are restricted and provide only a fraction of the features defined by the rule language of Drools. The RBs of Drools, which we can practically analyze with our implementation, are even more restricted and far away from the RBs used in productive environments. However, we are able to cover many core concepts of Drools. In our case study we show that this is sufficient to produce useful results with practical importance. We prepare a RB developed at Capgemini and analyze this RB with our implementation. In this process we gather valuable information about the RB and show that the restrictions of our implementation are not of fundamental nature and could in principle be overcome, provided one is willing to invest the necessary software development efforts. Hence our work can be seen as the proof-of-concept and a first step towards the goal of an automated verification process for Drools RBs that are relevant in real-world scenarios.

In Chapter 2 we present the basic concepts which are necessary for the rest of the thesis. A brief overview of the Drools rule engine and the Drools Rule Language (DRL) [10] is given. We shortly explain the Rete algorithm [6], which is the basis for Drools and many other BRMS. Finally, we introduce the required formal framework, the so-called *term rewriting systems*.

We start Chapter 3 with an introduction of syntax and structural operational semantics of a fragment of DRL. These semantics are then used to define certain properties of DRL; among them the most prominent is the termination property. In the last section of the chapter we interrelate this termination property of rule bases with the termination property of the term rewriting systems introduced earlier, which leads to a termination criterion for DRL.

Chapter 4 gives a short guide on how to install and use the implementation. Here we also present some details regarding the internal structure of the program and how it reuses existing classes of Drools and utilizes AProVE [8].

In Chapter 5 we present Drools RBs that are anonymized versions of excerpts of RBs used at Capgemini and explain the steps necessary to translate central aspects of these RBs into the previously defined fragment of DRL. We use these translations to illustrate the results and performance of the implementation.

The last chapter gives a brief summary of the results and final conclusions.

2. Preliminaries

This chapter introduces the basic concepts, notations, and terminology that are used throughout the rest of the text. Most of its content is based on [2, 6, 7, 10].

We begin the first section with a short description of the *rule engine* (or *runtime*) of Drools [10, p. 107]. The mechanics of this part of Drools are of great importance to us, since it is responsible for the relationship of RBs and data provided by external systems. In this context, the entities of data are called *facts*. We end this section with a brief discussion of the shape and behavior the rule engine expects from such facts.

The next section presents an overview of the *Drools Rule Language* (DRL) [10, p. 187], which is used to formulate the RBs for Drools. DRL is a feature-rich language with a close relationship to the programming language Java [9]. Since DRL essentially possesses the full expressive power of Java, it is far too comprehensive to be presented in detail. Instead, we try to expose its general concepts and give an example of how Drools RBs might look like.

The third section briefly introduces the *Rete algorithm* [6], which forms the basis of Drools and many other BRMSs. This pattern-matching algorithm, allows efficient handling of large numbers of facts and rules. A basic understanding of the ideas behind this algorithm helps us to explain our later formalization.

Finally, we introduce *term rewriting systems* (TRS) [2] and *conditional integer term rewriting systems* (ITRS) [7]. These systems have a mature theory of *termination properties*, which we utilize later. In the next chapter we show how to extract certain ITRS from a given DRL, such that the termination of the ITRS guarantees the termination of the respective DRL.

2.1. Drools Rule Engine

Drools Expert is the rule engine of the BRMS Drools and the primary objective of our formalisation approaches. It is part of the JBoss Developer program organized by the company Red Hat. Like all JBoss projects, it is written completely in the programming language Java and is available as an open source software. At the moment of publication the latest stable release of Drools Expert is version 6.1, which we from now on refer to as simply *Drools*. For a complete documentation, see [10, p. 107].

Since the term *rule engine* can be rather ambiguous, we state more precisely that Drools is a *production rule system* and based on the *Rete algorithm* [6]. This algorithm is the core of most production rule systems and allows efficient handling of large numbers of facts and rules by implementing a sophisticated caching strategy for intermediate results. We give more details on this algorithm in Section 2.3. A production rule system consists of two parts: the *working memory* and the *inference engine*. The working memory maintains a list of facts and other data, which represent the current state of knowledge in the system. The inference engine holds the current RB and tries to match its rules against the facts in the working memory. If a match is found, we say that the matching rule is *triggered*. Once all matches are found, the triggered rules are prioritized in a so-called *agenda*. The order of this agenda can be influenced directly through the design of the RB. However, this order might also depend on other factors like the time at which facts were asserted to the working memory, or the complexity of a rule and many other criteria. After this prioritization step, which is called *conflict resolution*, the *actions* defined by the rules on the agenda are executed in a batch process. We say the rules *fire*. The whole process of pattern matching, conflict resolution, and rule firing is called *match-resolve-act cycle*. The actions executed when firing a rule might change the working memory. In this case, the current agenda is dismissed and the inference engine returns to matching facts and rules. Thus, the triggering of a rule can result in a cascade of other actions and conclusions. This is called *forward chaining*. A sequence of match-resolve-act cycles caused by forward chaining, is called *evaluation cycle*. To emphasize this dynamic behavior of rules and the possible manifestation of conclusions as facts in the working memory, rules are sometimes called *productions* in this context.

Drools provides access to its working memory and inference engine through two Java interfaces: `StatelessKieSession` and `KieSession`. The former does not maintain the working memory after an evaluation cycle and is intended for short-lived tasks like validation or calculation. The later maintains the working memory between evaluation cycles and is intended for long-lived processes like real-time monitoring or real-time diagnostics. Since we mainly focus on what happens inside a single evaluation cycle, this difference is not of importance to us; and we only briefly describe the `KieSession` interface. This interface exposes the working memory through the method `insert(Object o)`, which inserts an object into the working memory. After all desired objects are inserted into the working memory, one can start the evaluation cycle of the inference engine by calling the method `fireAllRules()`.

From this point of view, we can narrow the topic of this thesis and say that we are interested in the following question: Does a call of the method `fireAllRules()` terminate for an arbitrary working memory and the given RB?

While we do not want to go into more details about the implementation of Drools, we need a basic understanding of the nature of the objects used as facts in the working memory. We discuss this in the next paragraphs.

Facts in Drools

Since Drools is written in Java and mostly used in Java environments it is natural to represent facts as Java objects. Indeed, Drools accepts *any* Java object as a fact. However, some issues have to be considered before passing objects to Drools or when designing classes that are meant to represent facts.

Drools uses a Java feature called *introspection* or *reflection* to analyze the objects inserted into working memory. Java introspection allows the analysis of the class of an

Listing 2.1: Example of a Java class used to represent facts

```
1  public class Flower {
2
3    private String color;
4    private String name;
5
6    public String getColor() {
7      return color;
8    }
9
10   public String getName() {
11     return name;
12   }
13
14   public void setColor(String color) {
15     this.color = color;
16   }
17
18   public void setName(String name) {
19     this.name = name;
20   }
21 }
```

object at the *runtime*. It reveals the public methods and fields, implemented interfaces, and other valuable information about the object. It can also be used to call the received methods, thus allowing to work with objects, whose classes are not available at compile time. When introspecting an object, Drools assumes that certain characteristics of the object represent so called *attributes*. We illustrate that with an example:

Consider the Java class in Listing 2.1. After introspection of an instance of the class `Flower`, Drools assumes that the object has the attributes `color` and `name`. The methods in Lines 6 to 12 are used to receive their respective values; they are called *getters*. The methods in Lines 14 to 20 are used to set their respective values; these are called *setters*.

Since Drools has only access to the signature of these methods it relies on their correct implementation and expects a certain behavior. That is, the value of an attribute must not change when calling a getter; and the call of a setter changes only the value of the respective attribute. Furthermore, it is required to inform Drools about the change of an attribute of a fact when it is done outside of Drools. This can be achieved by calling the `update(Object o)` method of the `KieSession` interface.

The example in Listing 2.1 shows a simple class that acts well-behaved and like expected by Drools. The values of the attributes are stored in private fields and everything is easily understood. However, this is not necessarily the case and it might not be trivial to guarantee the expected behavior in the case of a more complex class. For such classes software verification tools like KeY [3] can be used to test and verify the stated requirements. Since our thesis is about the analysis of RBs and not the analysis of Java code, we do not discuss this topic further.

Instead, we choose another approach, which makes sure that the facts in our RBs behave like expected. In Drools it is also possible to define the structure of a fact directly in a RB. For those facts a consistent behavior is guaranteed. We come back to this feature at the end of the next section about the rule language of Drools.

2.2. Drools Rule Language

The *Drools Rule Language* (DRL) is used to formulate RBs for Drools. DRL has a close relationship to the programming language Java [9] and incorporates some of its notions and reuses Java syntax directly in numerous cases. However, there are some unique features which facilitate the declaration of facts and rules, which we want to present in this section. DRL possesses many other interesting features and is far too comprehensive to be discussed in detail. For a complete documentation, refer to [10, p. 187]. In this section we illustrate some of the core features of DRL with a concrete example of a simple RB and explain the intent behind the used constructs.

Listing 2.2 shows a simple Drools RB written in DRL. Line 1 shows what some readers might identify as a Java *namespace declaration*. We come back to this topic, when we discuss facts in DRL. For now it suffices to imagine that it defines the name of the RB. Lines 3 to 8 show the first rule of the RB. Line 3 indicates the beginning of a rule declaration and also defines the name of the rule. Line 4 indicates the start of

Listing 2.2: Example of a rule base written in DRL

```
1  package mother.goose.rhymes;
2
3  rule "Roses are red"
4    when
5      Flower(color == "red", name == "Rose")
6    then
7      System.out.println("We found a red rose.");
8  end
9
10 rule "Violets are blue?"
11   when
12     $f : Flower(color != "blue", name == "Violet")
13   then
14     System.out.println("We need to fix some violet.");
15     modify ($f) { setColor("blue") }
16 end
17
18 rule "Violets are blue!"
19   when
20     Flower(color == "blue", name == "Violet")
21   then
22     System.out.println("We found a blue violet.");
23 end
24
25 rule "Sugar is sweet and so are you"
26   when
27     Sugar($sweetness : sweetness) and $p : Person(sweetness == $sweetness)
28   then
29     System.out.format("Maybe this is you: %s.\n", $p);
30 end
```

the conditional part of the rule which defines when a rule can be applied. This part is also called the *left-hand side* (LHS) of the rule. In this case, the LHS consists of the single Line 5 which shows what is called a *pattern* in the idiom of Drools. Patterns are the most important conditional constructs of Drools, since they allow to refer to the facts in the working memory. We explain the meaning of the pattern in Line 5 as follows: The rule `"Roses are red"` matches every fact in the working memory such that its type is `Flower`, when considered as a Java object. Furthermore, each matched fact needs to have the attributes `color` and `name` with the values `"red"` respectively `"Rose"`. The part of the pattern between the parentheses defines what is called the *constraints* of the pattern. Note that here the operator == does not have the usual Java semantics, that is the constraint `color == "Red"` has the meaning of the Java statement `color.equals("Red")`. Line 6 indicates the start of the consequence part of the rule which defines what happens when a rule is fired. This part is also called the *right-hand side* (RHS) of the rule. Generally, the RHS of a rule can be an arbitrary sequence of Java statements and essentially defines a Java method. In this instance, the RHS consists of the single Line 7 which prints the string `"We found a red rose."` to the current standard output.

The first rule of this RB is very simple, in the sense that it just tests the existence of certain facts in the working memory and the RHS does not even depend on those facts. Typically, one wants to refer to the facts matched on the LHS of a rule in the RHS of that rule. This can be achieved using the so-called *pattern bindings*. An example of such a pattern binding can be found in Line 12. Here we have a pattern very similar to the one in Line 5, however, it is preceded with the variable `$f` followed by a colon. This statement binds a fact matched by the respective pattern to the variable `$f` which is then also available on the RHS of the rule. For example, Line 15 modifies the fact bound to the variable `$f` by changing the attribute `color` to the value `"blue"`. The statement in Line 15 is Drools specific and not found in standard Java. There are other Drools specific statements similar to `modify` which insert or retract facts from the working memory. Despite the obvious meaning of `modify`, an important aspect of this statement is that it also informs Drools about a change of the working memory. Thus, firing the second rule of our example would cause Drools to skip the current agenda and return to matching mode.

To discuss this behavior in more detail, imagine a working memory which contains a single fact with type `Flower` and attributes with values `"yellow"` respectively `"Violet"`. Only the second rule is applicable, hence Drools fires this rule. Now Drools returns to the matching mode and finds that only the third rule is applicable and thus fires this rule. After this the agenda is empty and Drools stops the evaluation cycle, which illustrated an example of forward chaining. It is also noteworthy in this example that firing the second rule changes the matched fact in such a way that this rule does not match the same fact in the next match-resolve-act cycle. Without the constraint `color != "blue"` our supposed working memory would cause a never-ending loop. Drools would modify the same fact over and over again even though the value of the attribute `color` is already `"blue"`. In most cases this is not the desired behavior and one is interested in rules which are not repeatedly applicable to the same facts. This behavior is related to the so-called *self-deactivation* property of rules, which we formally define in Section 3.3.

So far all considered rules referred to single facts on their respective LHS, since they contained only one pattern. Another important feature of Drools is the join of multiple facts in the working memory. An example of this shows the last rule of our RB. The intended meaning of its LHS is: This rule is applicable, when there is a fact of type `Sugar` and a fact of type `Person` in the working memory, such that both have the same value of their attribute `sweetness`. This is achieved using the operator `and` and the so-called *attribute bindings*. The statement between the first parentheses of Line 27 binds the value of the attribute `sweetness` of the currently considered fact with the type `Sugar` to the variable `$sweetness`. This variable is then used as a part of the constraint of the second pattern of the LHS.

In general, we need to understand that Drools creates a match for each fact in the working memory to which a rule is applicable. If a rule refers to multiple facts, it creates a match for every n-tuple of facts which satisfies the stated constraints. This means the number of possible matches of a single rule is generally in a polynomial relationship to the number of facts in the working memory where the leading exponent is determined by the number of patterns of that rule.

Facts in DRL

As mentioned in Section 2.1, Drools uses Java introspection or reflection to analyze the structure of facts. Here the Java namespace declaration at the beginning of a RB plays an important role. Drools searches in the current Java class path for the class corresponding to a fact and assumes that the fully qualified name of this class begins with the defined package. That is, in the case of Listing 2.2, Drools assumes that patterns matching the type `Flower` refer to facts which are instances of a class with the fully qualified name `mother.goose.rhymes.Flower`. In case that the classes of the facts are defined in different packages, we can use the DRL statement `import`, which defines the fully qualified name of each class and works quite similar to its Java counterpart.

These mechanisms are commonly used in productive environments, since here the model of the facts is most likely not only used in DRL, but also in other contexts and should thus be independent of Drools. For our theoretical considerations, the use of these constructs has the major drawback: they break the self-sustenance of RBs and we need to view RBs using these features in the context of a complete Java environment. Luckily, DRL has a feature which allows us to directly define the structure of facts inside RBs.

Listing 2.3 shows a possible declaration of the facts used in Listing 2.2. The syntax is self-explanatory. Drools translates such type declarations to Java classes, which look very similar to the one in Listing 2.1 and uses them in the background. Another benefit of the direct declaration of types, is that it also guarantees the expected behavior of facts, which we discussed in Section 2.1. Yet, in real-world scenarios this feature is mainly used to define the structure of facts, which represent intermediate results, created inside the evaluation cycle of a rule, and are not accessed outside of Drools. This has also to do with the complicated procedure necessary to instantiate and handle such facts outside of

Listing 2.3: Example of a type declaration written in DRL

```
declare Flower
  color : String
  name : String
end

declare Sugar
  sweetness : Integer
end

declare Person
  firstName : String
  lastName : String
  sweetness : Integer
end
```

Drools. Nevertheless, these issues are irrelevant for our theoretical analysis and we use this feature to define the structure of facts in Section 3.1 to make the considered RBs self-sustained objects.

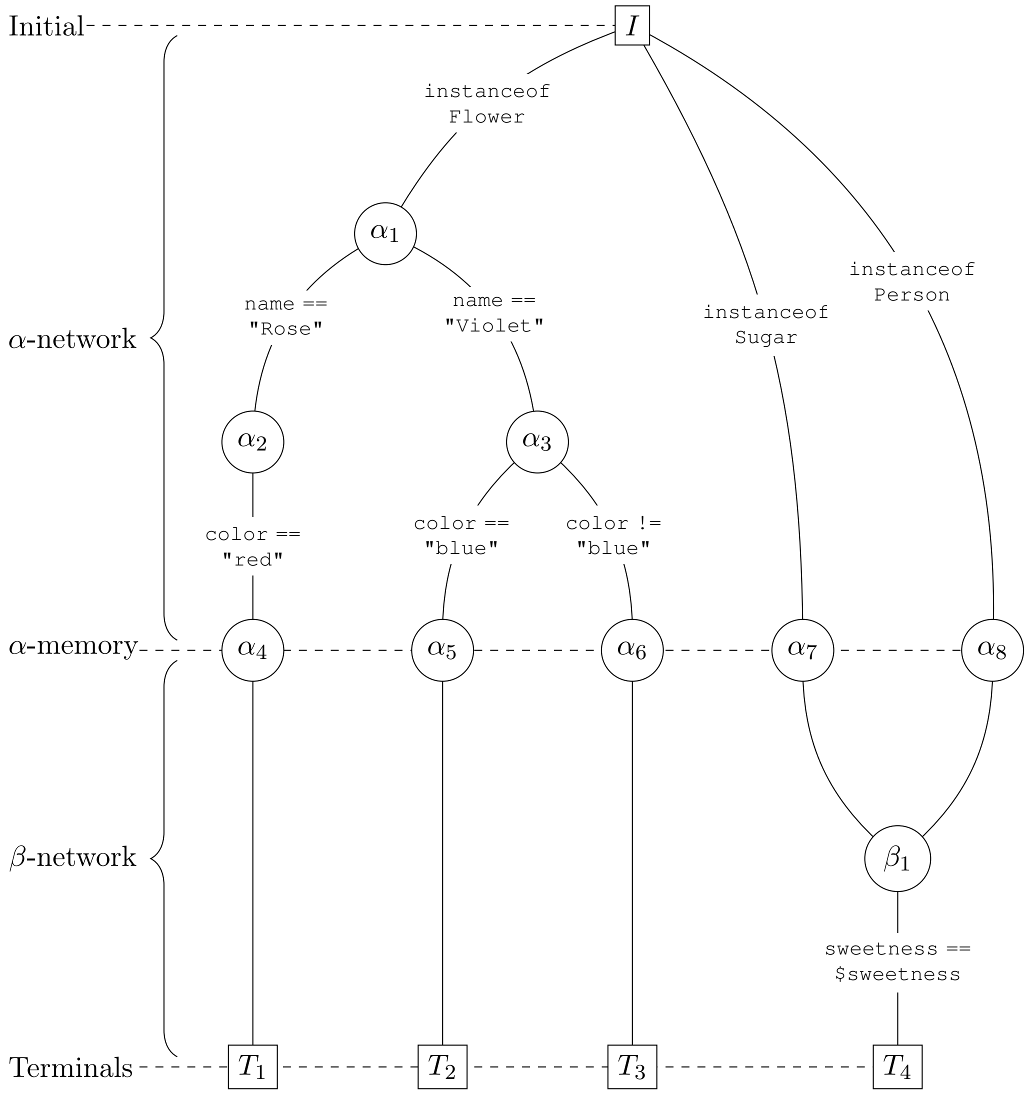
2.3. The Rete Algorithm

Drools is a production rule system and this type of rule engine is in most cases based on the *Rete algorithm* [6] or on some of its variations. This algorithm, developed by Charles L. Forgy in 1974, describes the implementation of an sophisticated pattern matching strategy which performs well for large numbers of facts and rules. The main idea behind this algorithm is the caching of the intermediate results that occur in the pattern matching process. Hence this algorithm is a typical example for a programming strategy which trades working memory for processor time in an efficient and beneficial way.

The caching of the intermediate results happens in a so called Rete which is the Greek term for network; and indeed, a Rete has an in-memory representation of a directed acyclic graph, which can be visualized as a network. Such network represents all rules of the given RB and is generated at run-time by the inference engine. In the matching process, facts from the working memory traverse this network following the directions of the edges. Each Rete has a single initial node which represents the entry of the network and, in most cases, many terminal nodes which represent the exits of the network. The terminal nodes relate to the matches for the rules in the compiled RB. This means, if a fact traverses the Rete from the initial node to a terminal node, we find a match for the associated rule.

Figure 2.1 shows a possible Rete for the Drools RB from Listing 2.2. We use this graphic to illustrate more details of the structure of these networks. A Rete can be further divided into the so called α*-network* and β*-network*. The α-network is connected to the initial node and its last layer forms the so called α*-memory*. The β-network is located between the α-memory and the terminal nodes.

Figure 2.1.: Rete of the rule base in Listing 2.2



Alpha Network

The α-network consists of α-*nodes* and forms a discrimination network, which divides the facts using simple criteria, which can be tested for an *individual* fact. An α-node has a single input and possibly many outputs. It represents a single test criterion for a fact. Such a test typically compares an attribute of a fact to a constant value or compares two attributes of the same fact. Another important test is the identification of the object type, which is usually performed in the first layer of α-nodes, which are directly connected to the initial node. If a fact passes the test in an α-node it is moved to the succeeding α-nodes until it eventually reaches the α-memory which forms the last layer of the α-network. The α-memory caches the arriving facts for further use. All facts stored in a node of the α-memory have passed the tests in the connected branch of α-nodes. Hence they have passed all respective test conditions.

For example the node α_1 in Figure 2.1 is a gateway for all facts of type `Flower`; and the node α_3 lets pass all facts of type `Flower`, which have an attribute `name` whose value is `"Violet"`. The node α_6 is part of the α-memory and contains all facts coming from node α_3 which have an attribute `color` whose value is different from `"blue"`.

Beta Network

The β-network consists of *β-nodes* and is responsible for the join of facts from the α-memory. It is optional and only created, when at least one rule refers to at least two facts on its LHS. A β-node has two inputs, which are called *left* and *right*, and possibly many outputs. The left input receives tuples of facts and the right input receives a single fact. Of course, the β-nodes directly connected to the α-memory receive on both sides single facts. A β-node, typically has its own β-memory, which stores all tuples from the left input and represents partial matches. If a fact enters the right side of the β-node it is tested against the tuples in the β-memory and added to those tuples for which it passes the test. Those tuples are then sent to the left side of the next β-node or directly to a terminal node. The tests performed in β-nodes typically refer to the attributes of *two* facts. However, there are more complex types of β-nodes, which might depend on *all* facts coming from the inputs. For example, the DRL join operator `not`, which we discuss in the next chapter, leads to a special kind of β-node.

Figure 2.1 contains the single β-node β_1 which performs the join of facts from node α_7 and α_8, that is facts of type `Sugar`, respectively, `Person`. It creates all possible pairs, for which the first component is of type `Sugar` and the second component of type `Person`. Next, the β-node tests for each pair if the value of the attribute `sweetness` of both components is equal. Pairs, which pass this test, are send to the terminal node T_4.

Note the important difference between certain constraints of patterns, which look quite similar in plain DRL. That is, in terms of our example from Listing 2.2, the handling of a constraint, like for `color == "Red"` invokes a completely different mechanism than the handling of a constraint like `sweetness == $sweetness`. This difference plays a crucial role in our later formalization.

2.4. Term Rewriting Systems

A *term rewriting system* (TRS) is an *abstract rewriting system* $(\mathcal{A}, \rightarrow)$, where the *object set* $\mathcal{A}$ is a set of terms and the *rewrite relation* $\rightarrow$ is a binary relation over $\mathcal{A}$. Since we want a convenient notation to define such rewrite relations, a more sophisticated definition is needed. This is achieved using so-called *rewrite rules*, which also cover the replacement of *subterms*. Next, we introduce a special signature for terms handling arithmetic integer expressions and a way to restrict the applicability of rewrite rules using certain conditional elements. This is necessary for the formulation of the *conditional integer term rewriting systems* (ITRS), which we need for the analysis of termination properties of Drools RB.

Most content of this section is taken from [2], which provides a comprehensive overview of the field of rewriting systems. The definition of ITRSs and the related theorems come from [7].

Now we begin with our formal introduction by defining the concept of *signatures*:

Definition 2.4.1 A *TRS-signature* Σ is a tuple $\Sigma = (\mathcal{V}, \mathcal{F}, \alpha)$, where:

(1) $\mathcal{V} = \{v_0, v_1, v_2, \ldots\}$ is a countable set of *variable symbols.*

(2) $\mathcal{F} = \{f_0, f_1, f_2, \ldots\}$ is a countable set of *function symbols.*

(3) $\alpha : \mathcal{F} \to \mathbb{N}$.

(4) $\mathcal{V} \cap \mathcal{F} = \varnothing$.

Sometimes we write x, y, z instead of v_0, v_1, v_2 and f, g, h instead of f_0, f_1, f_2. We call $\alpha(f) = n$ the *arity* of f. In this case, we call f an *n-ary* function symbol. The 0-ary function symbols are called *constant symbols.*

Note that this definition of a signature is a little different from the one commonly used when stating other formal systems, since we have no need for predicate symbols. However, our signature gives rise to a set of terms in a well-known way:

Definition 2.4.2 The set of *terms* $\mathcal{T}_\Sigma$ over a signature Σ is the smallest set such that:

(1) $\mathcal{V} \subseteq \mathcal{T}_\Sigma$.

(2) If $f \in \mathcal{F}$, $\alpha(f) = n$, and $t_0, \ldots, t_{n-1} \in \mathcal{T}_\Sigma$ then $f(t_0, \ldots, t_{n-1}) \in \mathcal{T}_\Sigma$.

Sometimes we write $\mathcal{T}$ instead of $\mathcal{T}_\Sigma$ when the signature is clear from the context.

In this section, we assume from now on the presence of an arbitrary given signature Σ and most of the following definitions are relative to this Σ. As we want to be able to probably define the rewriting of subterms of terms, we need the notion of *positions*:

Definition 2.4.3 Let $t \in \mathcal{T}$ be a term. The set $\mathrm{Pos}(t) \subset \mathbb{N}^*$ consists of words over the alphabet $\mathbb{N}$ and is recursively defined as follows:

(1) If t is a constant or variable symbol then $\mathrm{Pos}(t) = \{\varepsilon\}$.

(2) If $t = f(t_0, \ldots, t_{n-1})$ then $\mathrm{Pos}(t) = \bigcup_{i=0}^{n-1} \{i\pi | \pi \in \mathrm{Pos}(t_i)\}$.

We call the elements $\pi \in \mathrm{Pos}(t)$ the *positions* of t.

Pay attention that here and generally in the context of strings, we use the symbol ε to refer to the empty word. Now we can precisely define the subterms of a given term and how to replace them:

Definition 2.4.4 Let $t \in \mathcal{T}$ be a term and $\pi \in \mathrm{Pos}(t)$ a position. The term $t|_\pi$ is recursively defined as follows:

(1) $t|_\varepsilon = t$.

(2) $f(t_0, \ldots, t_{n-1})|_{i\pi} = t_i|_\pi$.

We call $t|_\pi$ the *subterm* of t at position π.

Definition 2.4.5 Let $t, s \in \mathcal{T}$ be terms and $\pi \in \mathrm{Pos}(t)$ a position. The term $t[s]_\pi$ denotes the result of the *replacement* of $t|_\pi$ in t with s.

Before we can define rewrite rules, we need some last ingredients that help us to clarify, when we are allowed to make such replacements. These are *substitutions* and *matching terms*:

Definition 2.4.6 Let $\sigma : \mathcal{V} \to \mathcal{T}$ be a function and $t, s \in \mathcal{T}$ terms. The function σ is called *substitution* iff $\sigma(x) \neq x$ for only finitely many $x \in \mathcal{V}$. Then, we call the set $D(\sigma) = \{x \in \mathcal{V} | \sigma(x) \neq x\}$ the *domain* of σ and say that, s *matches* t iff $\sigma(s) = t$. In this case, we call $D(\sigma)$ the *necessary instantiation* for the matching.

Here we can finally define what a rewrite rule is and how to translate a given set of rewrite rules into a term rewrite relation:

Definition 2.4.7 Let $l, r \in \mathcal{T}$ be terms, such that $l \notin \mathcal{V}$. A *rewrite rule* is an expression of the form

$$l \to r.$$

We use the symbol $\mathcal{R}$ to denote sets of rewrite rules.

Definition 2.4.8 Let $\mathcal{R}$ be a set of rewrite rules. The *term rewrite relation* $\to_\mathcal{R}$ is a binary relation over $\mathcal{T}$ and defined as follows:

$$s \to_\mathcal{R} t \equiv \begin{cases} \text{There exist } l \to r \in \mathcal{R}, \pi \in \mathrm{Pos}(s), \text{ and } \sigma : \mathcal{V} \to \mathcal{T} \\ \text{such that } s|_\pi = \sigma(l) \text{ and } t = s[\sigma(r)]_\pi. \end{cases}$$

In abuse of notation, we define the resulting *term rewriting system* $\mathcal{R} = (\mathcal{T}, \to_\mathcal{R})$. This is not a problem, since it should always be clear from the context, whether we refer to the set of rewrite rules or the actual term rewriting system.

In the context of term rewriting systems, one is generally interested in the transitive closure $\to_\mathcal{R}^+$ of the rewrite relation $\to_\mathcal{R}$. The termination property states that this transitive closure exists for all terms:

Definition 2.4.9 Let $\mathcal{R}$ be a term rewriting system. We say that $\mathcal{R}$ is *terminating* iff there is no infinite sequence $t_n : \mathbb{N} \to \mathcal{T}$ of terms such that

$$t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \dots$$

Since rewrite relations are generally infinite objects, one is interested in possibilities to trace back their termination property to properties of the finite set of rewrite rules from which they result. This can be achieved through the so-called reduction order:

Definition 2.4.10 Let $>$ be a strict order on $\mathcal{T}$. We call $>$ a *rewrite order* iff it is compatible with Σ-operations and closed under substitutions. That is:

(1) For all $s_1, s_2 \in \mathcal{T}$, $n \in \mathbb{N}$ and $f \in F$ with $\alpha(f) = n$:

$$s_1 > s_2 \text{ implies } f(t_1, \dots, t_{i-1}, s_1, t_{i+1}, \dots, t_n) > f(t_1, \dots, t_{i-1}, s_2, t_{i+1}, \dots, t_n)$$

for all $t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n \in \mathcal{T}$ and all i with $1 \leq i \leq n$.

(2) For all $s_1, s_2 \in \mathcal{T}$ and all substitutions σ working on $\mathcal{T}$:

$$s_1 > s_2 \text{ implies } \sigma(s_1) > \sigma(s_2)$$

A *reduction order* is a well-founded rewrite order.

Theorem 2.4.11 *The term rewriting system $\mathcal{R}$ terminates iff there exists a reduction order $>$ such that:*

$$l > r \text{ for all } l \to r \in \mathcal{R}$$

.

Proof: See [2, p. 103]. □

Next, we define the integer term rewriting systems which we utilize later. Theoretically, these systems have the same expressive power as plain term rewriting systems. However, in practical applications, where performance becomes more of an issue, ITRSs are often a more adequate choice. The following definitions are taken from [7].

Definition 2.4.12 An *ITRS-signature* $\Sigma_{\mathbb{Z}}$ is a TRS-signature $\Sigma_{\mathbb{Z}} = (\mathcal{V}, \mathcal{F}, \alpha)$, such that:

(1) $\mathbb{Z} = \{0, 1, -1, 2, -2, \dots\} \subseteq \mathcal{F}$ contains constant *integer symbols.*

(2) $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\} \subseteq \mathcal{F}$ contains constant *boolean symbols.*

(3) $\mathcal{F}_A = \{+, -, *, /, \%\} \subseteq \mathcal{F}$ contains binary *arithmetic operation symbols.*

(4) $\mathcal{F}_B = \{\land, \to\} \subseteq \mathcal{F}$ contains binary *boolean operation symbols.*

(5) $\mathcal{F}_R = \{>, \geq, =, \neq, \leq, <\} \subseteq \mathcal{F}$ contains binary *relational operation symbols.*

We write $\mathcal{T}_{\mathbb{Z}}$ instead of $\mathcal{T}_{\Sigma_{\mathbb{Z}}}$.

Definition 2.4.13 The set of *pre-defined ITRS-rules* $\mathcal{D}_\mathbb{Z}$ is defined as follows:

$$\begin{array}{rcl}\mathcal{D}_\mathbb{Z} & = & \{\circ(n,m) \to l \mid n,m,l \in \mathbb{Z}, \circ \in \mathcal{F}_A, n \circ m = l\} \\ & \cup & \{\circ(a,b) \to c \mid a,b,c \in \mathbb{B}, \circ \in \mathcal{F}_B, a \circ b = c\} \\ & \cup & \{\circ(n,m) \to \mathbf{true} \mid n,m \in \mathbb{Z}, \circ \in \mathcal{F}_R, n \circ m\} \\ & \cup & \{\circ(n,m) \to \mathbf{false} \mid n,m \in \mathbb{Z}, \circ \in \mathcal{F}_R, \neg n \circ m\}\end{array}$$

For example, we have $+(1,2) \to 3, <(4,5) \to \mathbf{true}, \wedge(\mathbf{true},\mathbf{false}) \to \mathbf{false} \in \mathcal{D}_\mathbb{Z}$. These pre-defined rules are used to define the desired conditional integer term rewriting systems.

Definition 2.4.14 Let $l,r,c \in \mathcal{T}_\mathbb{Z}$, such that $l \notin \mathbb{B} \cup \mathbb{Z} \cup \mathcal{V}$ and l does not contain symbols from $\mathcal{F}_A \cup \mathcal{F}_B \cup \mathcal{F}_R$. A *conditional rewrite rule* is an expression of the form:

$$l \to r \mid c$$

We consider the rewrite rules $l \to r$ from Definition 2.4.7 to be conditional rewrite rules of the form $l \to r \mid \mathbf{true}$ and write $l \to r$ instead of $l \to r \mid c$ when $c = \mathbf{true}$.

Definition 2.4.15 Let $\mathcal{R}_\mathbb{Z}$ be a finite set of conditional rewrite rules. The *conditional integer term rewrite relation* $\to_\mathbb{Z}$ is a binary relation over $\mathcal{T}_\mathbb{Z}$ and defined as follows:

$$s \to_{\mathcal{R}_\mathbb{Z}} t \equiv \left\{ \begin{array}{l} \text{There exist } l \to r \mid c \in \mathcal{R}_\mathbb{Z} \cup \mathcal{D}_\mathbb{Z}, \pi \in \mathrm{Pos}(s), \text{ and } \sigma : \mathcal{V} \to \mathcal{T}_\mathbb{Z} \\ \text{such that } s|_\pi = \sigma(l), t = s[\sigma(r)]_\pi \text{ and } \sigma(c) \to^+_{\mathcal{R}_\mathbb{Z}} \mathbf{true}. \end{array} \right.$$

In abuse of notation, we define the resulting *conditional integer term rewriting system* $\mathcal{R}_\mathbb{Z} = (\mathcal{T}_\mathbb{Z}, \to_{\mathcal{R}_\mathbb{Z}})$. This is not problem, since it should always be clear from the context, whether we refer to the set of conditional integer rewrite rules or the actual conditional integer term rewriting system.

3. Theory

This chapter describes the theoretical background of the termination criterion on which we rely in our implementation. We start by formally defining the syntax of a fragment of DRL which we call DRL$_{\mathbb{Z}}$. This name is chosen to emphasize that this fragment handles facts whose attributes represent integer values. The introduced syntax is not abstract and the resulting expressions are valid DRL: that is, Drools would accept them as actual rule bases.

Next, we define *structural operational semantics* for the previously defined fragment. We introduce a so-called *abstract rule engine*. This theoretical device allows us to simulate certain properties of the inference process executed by Drools. An interesting aspect of this formalism is the exposure of the non-deterministic choices made during the inference process.

In the third section we define and show some useful properties of the previously defined syntax and semantics. The most important among them is the termination property. We also briefly discuss the *Turing completeness* of our abstract rule engine, which shows that its termination property is generally not decidable.

In the last section we define and prove a sufficient termination criterion for our abstract rule engine. We show how to extract certain ITRSs from the considered DRL expressions. Then we prove that the termination of such ITRS guarantees that the abstract rule engine terminates for an arbitrary working memory when executing the respective DRL expression.

3.1. Syntax of DRL$_{\mathbb{Z}}$

In this section we introduce the syntax of the fragment DRL$_{\mathbb{Z}}$, which we examine in the rest of this chapter. As mentioned in Section 2.2, DRL is a feature-rich language which is used in productive environments and mostly business related scenarios. As such, it presents certain obstacles when made the objective of a theoretical analysis.

Hence we are forced to skip many interesting features of DRL. However, we try to preserve its core concepts and philosophy in our fragment DRL$_{\mathbb{Z}}$. One aspect of this goal is the waiver of a syntactical abstraction layer: thus all expression of DRL$_{\mathbb{Z}}$ are actually valid DRL, which could be executed in Drools.

We formally define the syntax and some syntactical properties of DRL$_{\mathbb{Z}}$. This rather technical task is executed by employing so called *syntax-* or *railroad diagrams* [11]. Finally, we compare features of full DRL like defined in [10, p. 187] and DRL$_{\mathbb{Z}}$. In this process we argue why we think that DRL$_{\mathbb{Z}}$ covers the core concepts of DRL.

Syntax Diagrams of $\mathrm{DRL}_{\mathbb{Z}}$

Like already mentioned in Section 2.2, DRL has a close relation to the programming language Java [9]. Many concepts of Java are only included in DRL for the convenience of the programmer and can be considered, what is sometimes called, *syntactical sugar*.

Nevertheless, we require some Java related notions, which we briefly discuss now. In our syntax diagrams, we assume the existence of well-defined nonterminals ⟨*Identifer*⟩, ⟨*Variable*⟩, ⟨*IntegerComparison*⟩, and ⟨*IntegerExpression*⟩. An instance of ⟨*Identifer*⟩ is an alphanumeric string which must not be equal certain *keywords*. For example `MyType`, `MyRule`, and `Atz2X1` are valid instances of ⟨*Identifier*⟩, while `rule`, `4fg+`, and `white space` are not. An instance of ⟨*Variable*⟩ is an ⟨*Identifier*⟩ with a direct prefix of the symbol `$`. That is, `$x`, `$var1`, and `$qweA34` are valid instances of ⟨*Variable*⟩, while `ger$ws` is not. An instance of ⟨*IntegerComparison*⟩ is one of the Java operators used to compare integer values. The expressions `==`, `!=`, `<`, `<=`, `>`, and `>=` are valid instances of ⟨*IntegerComparison*⟩. Finally, ⟨*IntegerExpression*⟩ refers to certain expressions which are composed of arithmetic integer operators, integer literals, parentheses, and instances of ⟨*Variable*⟩ in a common way. For example, `21 * ($v + 7)` and `42` are valid instances of ⟨*IntegerExpression*⟩. Furthermore, we silently assume that most literals and nonterminals in our syntax diagrams are separated by a finite sequence of so-called *whitespace characters* and ignore this topic below. Now we begin with the formal introduction of the syntax of $\mathrm{DRL}_{\mathbb{Z}}$ by introducing the concept of a *package*.

Definition 3.1.1 The nonterminal ⟨*Package*⟩ is defined by the following syntax diagram:

⟨*Package*⟩ ::= `dialect` – `"mvel"` – ⟨*Type*⟩ – ⟨*Rule*⟩

⟨*Type*⟩ ::= `declare` – ⟨*Identifier*⟩ – ⟨*Attribute*⟩ – `end`

⟨*Attribute*⟩ ::= ⟨*Identifier*⟩ – `:` – `Integer`

⟨*Rule*⟩ ::= `rule` – ⟨*Identifier*⟩ – `when` – ⟨*LHS*⟩ – `then` – ⟨*RHS*⟩ – `end`

The first instance of ⟨*Identifier*⟩ in an instance of ⟨*Rule*⟩, ⟨*Type*⟩, or ⟨*Attribute*⟩ is called *rule identifier*, *type identifier*, respectively *attribute identifier*. We restrict ⟨*Package*⟩ such that the following conditions hold:

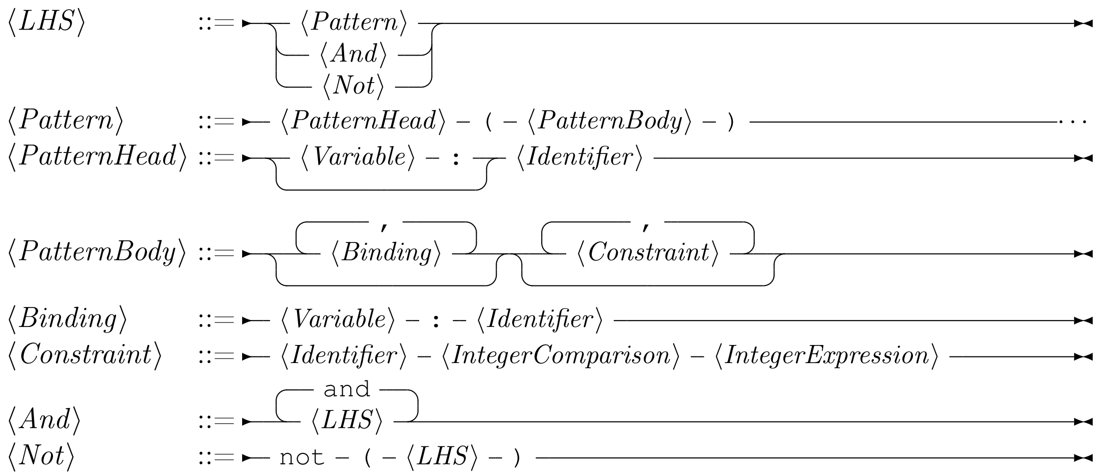
(1) Rule identifiers are unique.

(2) Type identifier are unique.

(3) Attribute identifiers are unique in each instance of ⟨*Type*⟩.

Next, we introduce the nonterminals ⟨*LHS*⟩ and ⟨*RHS*⟩, which we left undefined for the moment, to allow a neat arrangement of syntax diagrams and the related notations and restrictions.

Definition 3.1.2 The nonterminal ⟨*LHS*⟩ is defined by the following syntax diagram:



An instance of ⟨*Identifier*⟩ which appears in a ⟨*PatternHead*⟩ is called *pattern type identifier*. An instance of ⟨*Variable*⟩ which appears in a ⟨*PatternHead*⟩ is called *pattern binding*; and an instance of ⟨*Variable*⟩ which appears at the beginning of a ⟨*Binding*⟩ is called *attribute binding*. An instance of ⟨*Constraint*⟩ is called *β-constraint* iff it contains ⟨*Variable*⟩ which appear in attribute bindings outside the current ⟨*Pattern*⟩. All other instances of ⟨*Constraint*⟩ are called *α-constraints*. In particular, an α-constraint which contains no ⟨*Variable*⟩ at all is called *constant constraint*. Instances of ⟨*Not*⟩ are called *scope-delimiter*. Furthermore, we restrict ⟨*LHS*⟩ such that the following conditions hold:

(1) Each pattern type identifier is equal to some type identifier.

(2) Instances of ⟨*Variable*⟩ which appear in a pattern or attribute binding are unique.

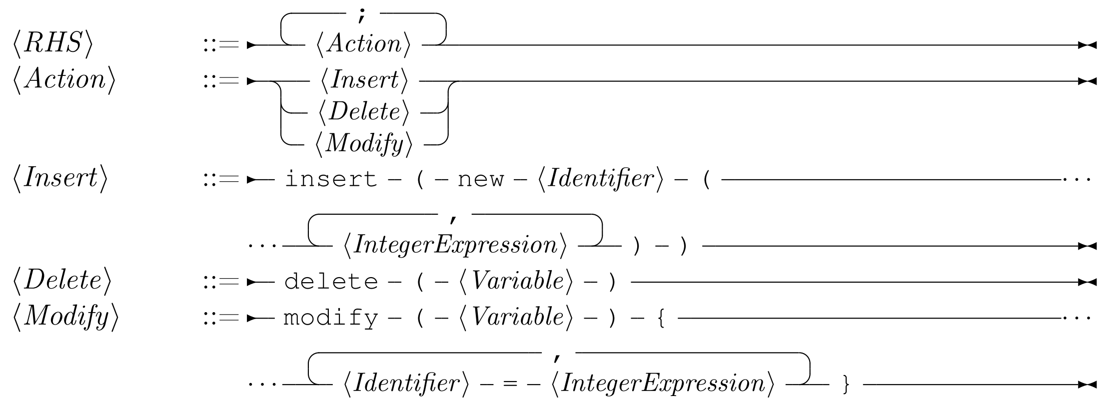
(3) Instances of ⟨*Variable*⟩ which appear inside of an ⟨*IntegerExpression*⟩ must also appear in an attribute binding on the *left side* of the ⟨*IntegerExpression*⟩.

(4) Variables used in attribute bindings inside a scope-delimiter *must not* appear to the *right side* of that scope-delimiter.

(5) Instances of ⟨*Constraint*⟩ are ordered such that the α-constraints appear before the β-constraints.

The concepts of α- and β-constraints is in direct relation to the concepts of α- and β-nodes from Section 2.3. The requirements on the order of these constraints is stated to facilitate the introduction of the semantics of $\mathrm{DRL}_{\mathbb{Z}}$ in the next section, since they are evaluated at different stages of the matching process.

Definition 3.1.3 The nonterminal ⟨*RHS*⟩ is defined by the following syntax diagram:



Furthermore, we restrict ⟨*RHS*⟩ such that the following conditions hold:

(1) The first instance of ⟨*Identifier*⟩ in an ⟨*Insert*⟩ equals some type identifier and the number of integer expressions equals the number of attributes related to that type identifier.

(2) The first ⟨*Variable*⟩ in a ⟨*Delete*⟩ or ⟨*Modify*⟩ equals some pattern binding in the current instance of ⟨*Rule*⟩.

(3) Instances of ⟨*Identifier*⟩ which appear in ⟨*Modify*⟩ are unique and equal some attribute identifier in the instance of ⟨*Type*⟩ which is identified by the pattern type identifier that corresponds to the pattern binding in the current ⟨*Modify*⟩.

(4) Instances of ⟨*Variable*⟩ which appear in an ⟨*IntegerExpression*⟩ equal attribute bindings in the current instance of ⟨*Rule*⟩.

An instance of ⟨*Modify*⟩ is called *β-modification* if it contains a ⟨*Variable*⟩, which appears in an attribute binding outside the ⟨*Pattern*⟩ which corresponds to its pattern binding. All other instances of ⟨*Modify*⟩ are called *α-modifications*. In particular, an α-modification which contains no ⟨*Variable*⟩ at all is called *constant modification*.

We illustrate the previous definitions of the syntax of $\mathrm{DRL}_{\mathbb{Z}}$ with an example of an instance of ⟨*Package*⟩, which is presented in Listing 3.1. The definition of the Java dialect MVEL in Line 1 is necessary to facilitate a syntax for the modification of facts, which allows the use of attribute names instead of the related setters. For a complete documentation of MVEL, see [4]. In Lines 14 to 23 we find an instance of ⟨*LHS*⟩, and specifically an instance of ⟨*And*⟩. In Lines 14 to 18 and Lines 20 to 23 we find instances of ⟨*Pattern*⟩. In Lines 14 and 20 we find pattern bindings, and in Lines 15 and 21 – attribute bindings. In Lines 16 and 17 we find α-constraints, where Line 16 contains a constant constraint. In Line 22 we find a β-constraint. In Lines 25 to 32 we find an instance of ⟨*RHS*⟩. In Lines 25 to 28 we find an α-modification. Line 29 gives us an example of an

Listing 3.1: Example of a rule base written in $\mathrm{DRL}_{\mathbb{Z}}$

```
dialect "mvel"

declare A
  x : Integer
  y : Integer
end

declare B
  z : Integer
end

rule R
  when
    $a : A(
      $x : x,
      y > 4,
      y != $x
    )
    and
    B(
      $z : z,
      z < $x
    )
  then
    modify ($a) {
      y = 10,
      x = $x * 5
    }
    insert(new A(20, $x * $z));
end
```

instance of ⟨*Insert*⟩. This example shows that the concepts and notions introduced in this section are not really hard to grasp; and the main challenge we are facing here is the establishment of a clear and precise notation for these very concepts and notions, which we need for the definition of the semantics of $\mathrm{DRL}_{\mathbb{Z}}$ in the next section.

Next, we compare our fragment to the complete language specification of DRL. In this process we argue why we think that $\mathrm{DRL}_{\mathbb{Z}}$ covers the core concepts and philosophy of DRL.

Comparison between $\mathrm{DRL}_{\mathbb{Z}}$ and DRL

Let us take a look at the language specification of DRL found in [10, p. 187]. If we compare the ⟨*Package*⟩ which we find there to our definition of ⟨*Package*⟩, we see that we omitted the features *functions*, *queries*, *globals*, and *imports*. Functions allow the definition of a Java helper class inside a DRL file. While this might be convenient in some cases, this is a clear case of syntactic sugar which we already mentioned earlier. Queries are simply speaking instances of ⟨*LHS*⟩ and allow the programmer to employ the power of the pattern matching algorithm of Drools to receive filtered lists of objects from working memory. This has also an convenience feature and has no direct relation to the evaluation cycle of Drools.

Globals allow the definition of variables to which one might refer throughout an RB. The values of these variables must be initialized outside of Drools before starting the inference process. This is clearly an important feature of DRL, which is used in most productive RBs. For example, if one wants to reason about dates and time, one might be interested if a date lies in the future or in the past. This could be achieved by introducing a global variable `$now` and supplying a proper initialization of its value. However, if we take a closer look at the intended use of globals, we find the following sentence [10, p. 199]: *"It is strongly discouraged to set or change a global value from inside your rules."* This basically means one should consider globals as immutable constant values throughout the inference process. In the case of $\text{DRL}_{\mathbb{Z}}$, we can use integer literals to emulate concrete instances of globals.

We already discussed certain aspects of import statements versus type declarations at the end of Section 2.2; and that we use type declarations, since this makes the considered RBs self-contained objects. In full DRL the expressive power of type declarations is almost equal to the one of Java classes, especially in regards of the properties which are relevant for the inference process. In comparison, the type declarations in $\text{DRL}_{\mathbb{Z}}$ are very restricted since they only allow attributes of type `Integer`. This is obviously one of the biggest restrictions of $\text{DRL}_{\mathbb{Z}}$ and we are not able to cover attributes which represent collections of objects or complex structures. Basic data types, like for example `Boolean`, `Date`, or `String`, however, can be encoded using integers and we exemplify this process for the type `String` in Chapter 5.

Next, we compare our $\langle \mathit{Rule} \rangle$ to the one found in the full language specification of DRL. We note that we omitted the so-called *rule attributes*. These are properties which are mostly used to influence the conflict resolution of the inference engine. That is, they control the order of rule execution when multiple rules are matching. In practice this might be necessary to solve specific problems. Nevertheless, it is considered best practice to assume an arbitrary rule order as stated in [10, p. 152]: *"As a general rule, it is a good idea not to count on rules firing in any particular order, and to author the rules without worrying about a 'flow'. However when a flow is needed a number of possibilities exist beyond salience: agenda groups, rule flow groups, activation groups and control/semaphore facts."* This means that on the one hand we lose expressiveness by not supporting rule attributes and on the other hand we encourage the intended use of DRL. We further discuss this topic in the next section.

The $\langle \mathit{LHS} \rangle$ of rules in full DRL introduces additional operators which are used to combine patterns. There we have the operators `or`, `exists`, and `forall` besides `and` and `not`. Obviously, these operators are convenient to have, but they do not add expressive power and can be reduced to `and` and `not`. The documentation makes this explicit for the operator `forall` [10, p. 254]: *"As a side note,* `forall(p1 p2 p3...)` *is equivalent to writing:* `not(p1 and not(and p2 p3...))`*."* The same is true for `exists` since `exists(p1 p2 p3...)` is equivalent to `not(not(p1 p2 p3...))`. The operator `or` can be eliminated through DNF transformations and rule splitting. In fact, the rule engine of Drools does exactly this in a preprocessing step when compiling the Rete for a RB. The same is true for Boolean operators, like for example `||`, which are generally allowed in the constraints of patterns in DRL.

In full DRL the ⟨*RHS*⟩ of a rule basically defines a Java methods. From this perspective, our definition of ⟨*RHS*⟩ introduces a massive restriction to the expressive power of $\mathrm{DRL}_{\mathbb{Z}}$. However, like in the aforementioned cases, we find recommendations for the intended use of ⟨*RHS*⟩; we quote [10, p. 294]: *"It is bad practice to use imperative or conditional code in the RHS of a rule; as a rule should be atomic in nature (...). The RHS part of a rule should also be kept small, thus keeping it declarative and readable. (...) The main purpose of the RHS is to insert, delete or modify working memory data."* The definition of ⟨*RHS*⟩ in $\mathrm{DRL}_{\mathbb{Z}}$ enforces these recommendations.

3.2. Semantics of $\mathrm{DRL}_{\mathbb{Z}}$

In this section we introduce structural operational semantics for the fragment $\mathrm{DRL}_{\mathbb{Z}}$ which we have defined in the previous section. This task is necessary to define the termination property for $\mathrm{DRL}_{\mathbb{Z}}$ in the next section. Furthermore, it gives us a better understanding of the match-resolve-act cycle of Drools. An interesting aspect is the exposure of the non-deterministic choice points which occur in the resolve stage of this cycle.

We start with the definition of *abstract working memories* and *abstracts matches* which serve as models for their concrete counterparts described in Section 2.2. Next, these concepts are used to define semantics of instances of ⟨*LHS*⟩ and ⟨*RHS*⟩. This allows us to capture the matching process, respectively the firing of rules. Finally, we introduce the concept of an *abstract rule engine*, which is basically a relation that describes the valid transitions between abstract working memories for packages of $\mathrm{DRL}_{\mathbb{Z}}$. This step interrelates the previously defined semantics of ⟨*LHS*⟩ and ⟨*RHS*⟩. At this point, we further investigate the conflict resolution strategies of Drools which prioritize matches and rules when multiple matches occur.

Abstract Working Memory

We model the abstract working memory of our abstract rule engine using elements of $\mathbb{N}$ and $\mathbb{Z}$. Finite subsets of $\mathbb{N}$ are used to represent the facts in the working memory and their elements can be considered as *abstract object pointers*. Furthermore, we define a function which maps such pointers to type identifiers, thus defining the types of our abstract facts. Finally, we introduce a partial function which maps abstract object pointers and attribute identifiers to elements of $\mathbb{Z}$. This function represents the attribute values of the facts in the working memory.

Definition 3.2.1 Let $\mathcal{P}$ be a package of $\mathrm{DRL}_{\mathbb{Z}}$ and $\mathcal{I}$ the set of instances of ⟨*Identifier*⟩ in $\mathcal{P}$. An *abstract working memory* $\mathcal{W}$ for $\mathcal{P}$ is a tuple $\mathcal{W} = (\mathcal{O}, \Gamma, \mathcal{A})$ such that:

(1) $\mathcal{O} \subset \mathbb{N}$ is a finite set of *abstract object pointers*.

(2) $\Gamma : \mathcal{O} \to \mathcal{I}$ is a function such that $\Gamma(o)$ is a type identifier in $\mathcal{P}$ for all $o \in \mathcal{O}$.

(3) $\mathcal{A} : \mathcal{O} \times \mathcal{I} \rightharpoonup \mathbb{Z}$ is a partial function such that $\mathcal{A}(o, a)$ is defined iff $o \in \mathcal{O}$ and a is an attribute identifier in the instance of ⟨*Type*⟩ identified by $\Gamma(o)$.

We illustrate this definition with an example of an abstract working memory for the package shown in Listing 3.1.

Example 3.2.2 Suppose, we initialize the previously empty working memory of Drools for the RB shown in Listing 3.1 with the following rule:

```
rule Initialize
  then
    insert(new A(6, 6));
    insert(new A(5, 6));
    insert(new B(3));
    insert(new B(7));
    insert(new B(2));
end
```

This rule is not part of $\mathrm{DRL}_{\mathbb{Z}}$ since $\langle LHS \rangle$ is empty. Nevertheless, we show the related abstract abstract working memory $\mathcal{W}_0 = (\mathcal{O}_0, \Gamma_0, \mathcal{A}_0)$:

$$\begin{aligned} \mathcal{O}_0 &= \{0, 1, 2, 3, 4\} \\ \Gamma_0 &= \{0 \mapsto \mathtt{A}, 1 \mapsto \mathtt{A}, 2 \mapsto \mathtt{B}, 3 \mapsto \mathtt{B}, 4 \mapsto \mathtt{B}\} \\ \mathcal{A}_0 &= \{(0, \mathtt{x}) \mapsto 6, (0, \mathtt{y}) \mapsto 6, (1, \mathtt{x}) \mapsto 5, (1, \mathtt{y}) \mapsto 6, (2, \mathtt{z}) \mapsto 3, (3, \mathtt{z}) \mapsto 7, (4, \mathtt{z}) \mapsto 2\} \end{aligned}$$

Next, we need a structure which represents the matches produced by the LHS of a rule. This structure is composed of a pointer to the most recently matched object and a partial function which represents variable bindings. While the pointer is only of intermediate relevance for the matching process in $\langle LHS \rangle$, the binding functions plays a crucial role for the evaluation of $\langle RHS \rangle$ and represents the relevant data for the actual match.

Definition 3.2.3 Let $\mathcal{P}$ be a package of $\mathrm{DRL}_{\mathbb{Z}}$, $\mathcal{W}$ an abstract working memory for $\mathcal{P}$, and $\mathcal{U}$ the set of instances of $\langle Variable \rangle$ in $\mathcal{P}$. An *abstract match* m in $\mathcal{W}$ is a tuple $m = (o, b_v)$ such that:

(1) $o \in \mathcal{O}$ is an abstract object pointer representing the most recently matched fact.

(2) $b_v : \mathcal{U} \rightharpoonup \mathbb{Z}$ is a partial function representing variable bindings.

We write $\mathcal{M}$ to denote a set of abstract matches.

The semantics for $\langle LHS \rangle$ and $\langle RHS \rangle$ which we define next, depend on the semantics of instances of $\langle IntegerExpression \rangle$. It is well-known how to define such semantics in the context of a function, which represents the values of variables. In our case, this function is b_v. Hence we assume a well-defined semantics $\langle e, b_v \rangle \Rightarrow z \in \mathbb{Z}$ such that $[\![e]\!]$ represents the element of $\mathbb{Z}$ to which the integer expression e evaluates for b_v. In the next section, we show that the restrictions to the syntax of $\mathrm{DRL}_{\mathbb{Z}}$ and the definition of the semantics of $\langle LHS \rangle$ ensure that the function b_v is always defined properly and facilitates the evaluation of integer expressions.

Semantics of LHS

We define a relation $\langle L, \mathcal{W} \rangle \Rightarrow \mathcal{M}$, such that $\mathcal{M}$ represents the matches of the instance L of $\langle LHS \rangle$ in the abstract working memory $\mathcal{W}$. We define this relation inductively over the structure of the syntax of $\mathrm{DRL}_{\mathbb{Z}}$. We start with the empty pattern which matches all facts in the abstract working memory such that their type is the respective pattern type identifier. Afterwards, we introduce semantics for pattern bindings and attribute bindings which populate the variable binding function b_v. At this point we are able to evaluate α-constraints. Finally, we introduce semantics for the operators `and` and `not` and define rules which allow the evaluation of β-constraints.

Definition 3.2.4 Let $\mathcal{P}$ be a package of $\mathrm{DRL}_{\mathbb{Z}}$, $\mathcal{W}$ an abstract working memory for $\mathcal{P}$, $\mathcal{M}$ a set of abstract matches in $\mathcal{W}$, and L an instance of $\langle LHS \rangle$ in $\mathcal{P}$. We define the operational semantics $\langle L, \mathcal{W} \rangle \Rightarrow \mathcal{M}$ inductively over the syntactic structure of L, like defined in Definition 3.1.2:

(Pattern)

$$\frac{\mathcal{M} = \{o \in \mathcal{O} | \Gamma(o) = T\} \times \{\varnothing\}}{\langle T\texttt{()}, \mathcal{W} \rangle \Rightarrow \mathcal{M}}$$

where T is a type identifier.

(BindP)

$$\frac{\langle T\texttt{()}, \mathcal{W} \rangle \Rightarrow \mathcal{M} \qquad \mathcal{M}' = \{(o, \{v \mapsto o\}) | (o, \varnothing) \in \mathcal{M}\}}{\langle v\texttt{:}T\texttt{()}, \mathcal{W} \rangle \Rightarrow \mathcal{M}'}$$

where v is an instance of $\langle \mathit{Variable} \rangle$.

(BindA)

$$\frac{\langle P\texttt{(}B\texttt{)}, \mathcal{W} \rangle \Rightarrow \mathcal{M} \qquad \mathcal{M}' = \{(o, b_v \cup \{v \mapsto \mathcal{A}(o, a)\}) | (o, b_v) \in \mathcal{M}\}}{\langle P\texttt{(}B\texttt{,}\, v\texttt{:}a\texttt{)}, \mathcal{W} \rangle \Rightarrow \mathcal{M}'}$$

where P is an instance of $\langle \mathit{PatternHead} \rangle$, B a possibly empty finite sequence of instances of $\langle \mathit{Binding} \rangle$, and a an attribute identifier related to the pattern type identifier in P.

($\mathrm{Cons}_{\alpha,==}$)

$$\frac{\langle P\texttt{(}B\texttt{,}\, C_\alpha\texttt{)}, \mathcal{W} \rangle \Rightarrow \mathcal{M} \qquad \mathcal{M}' = \{(o, b_v) \in \mathcal{M} | \mathcal{A}(o, a) = [\![e]\!]\}}{\langle P\texttt{(}B\texttt{,}\, C_\alpha\texttt{,}\, a\texttt{==}e\texttt{)}, \mathcal{W} \rangle \Rightarrow \mathcal{M}}$$

where C_α is a possibly empty finite sequence of α-constraints and e is an instance of $\langle \mathit{IntegerExpression} \rangle$ which contains only variables appearing in B.

(And)

$$\frac{\langle L, \mathcal{W} \rangle \Rightarrow \mathcal{M}_1 \quad \langle P\texttt{(}B\texttt{,}\, C_\alpha\texttt{)}, \mathcal{W} \rangle \Rightarrow \mathcal{M}_2 \quad \mathcal{M}' = \{m_1 \circ m_2 | (m_1, m_2) \in \mathcal{M}_1 \times \mathcal{M}_2\}}{\langle L \;\texttt{and}\; P\texttt{(}B\texttt{,}\, C_\alpha\texttt{)}, \mathcal{W} \rangle \Rightarrow \mathcal{M}'}$$

where $m_1 \circ m_2 = (o_1, b_{v,1}) \circ (o_2, b_{v,2}) = (o_2, b_{v,1} \cup b_{v,2})$.

($\text{Cons}_{\beta,==}$)

$$\frac{\langle L \ \texttt{and} \ P\texttt{(}B\texttt{,}C_\alpha\texttt{,}C_\beta\texttt{)}\texttt{,}\mathcal{W}\rangle \Rightarrow \mathcal{M} \qquad \mathcal{M}' = \{(o,b_v)\in\mathcal{M}|\mathcal{A}(o,a)=[\![e]\!]\}}{\langle L \ \texttt{and} \ P\texttt{(}B\texttt{,}C_\alpha\texttt{,}C_\beta\texttt{,}a\texttt{==}e\texttt{)}\texttt{,}\mathcal{W}\rangle \Rightarrow \mathcal{M}'}$$

where C_β is a possibly empty finite sequence of β-constraints and e contains at least one variables appearing in attribute bindings in L.

($\text{Not}_\top$)

$$\frac{\langle L,\mathcal{W}\rangle \Rightarrow \mathcal{M} \qquad \langle L \ \texttt{and} \ K,\mathcal{W}\rangle \Rightarrow \varnothing}{\langle L \ \texttt{and} \ \texttt{not(}K\texttt{)}\texttt{,}\mathcal{W}\rangle \Rightarrow \mathcal{M}}$$

where K denotes an instance of $\langle LHS\rangle$.

($\text{Not}_\perp$)

$$\frac{\langle L \ \texttt{and} \ K,\mathcal{W}\rangle \Rightarrow \mathcal{M} \qquad \mathcal{M}\neq\varnothing}{\langle L \ \texttt{and} \ \texttt{not(}K\texttt{)}\texttt{,}\mathcal{W}\rangle \Rightarrow \varnothing}$$

Furthermore, we have the rules ($\text{Cons}_{\alpha,\texttt{!=}}$), ($\text{Cons}_{\beta,\texttt{!=}}$), ($\text{Cons}_{\alpha,\texttt{<}}$), ($\text{Cons}_{\beta,\texttt{<}}$), ($\text{Cons}_{\alpha,\texttt{<=}}$), ($\text{Cons}_{\beta,\texttt{<=}}$), ($\text{Cons}_{\alpha,\texttt{>}}$), ($\text{Cons}_{\beta,\texttt{>}}$), ($\text{Cons}_{\alpha,\texttt{>=}}$) and ($\text{Cons}_{\beta,\texttt{>=}}$) analogue to ($\text{Cons}_{\alpha,==}$) respectively ($\text{Cons}_{\beta,==}$). Finally, there are special cases for the rules ($\text{Not}_\top$) and ($\text{Not}_\perp$) iff $\langle LHS\rangle$ starts with `not`:

($\text{Not}_{\top,\alpha}$)

$$\frac{\langle K,\mathcal{W}\rangle \Rightarrow \varnothing \qquad \mathcal{M}=(\varnothing,\varnothing,\varnothing)}{\langle \texttt{not(}K\texttt{)}\texttt{,}\mathcal{W}\rangle \Rightarrow \mathcal{M}}$$

($\text{Not}_{\perp,\alpha}$)

$$\frac{\langle K,\mathcal{W}\rangle \Rightarrow \mathcal{M} \qquad \mathcal{M}\neq\varnothing}{\langle \texttt{not(}K\texttt{)}\texttt{,}\mathcal{W}\rangle \Rightarrow \varnothing}$$

We illustrate this definition with an example by deriving the abstract matches for the instance of $\langle LHS\rangle$ in Listing 3.1 in the abstract working memory $\mathcal{W}_0$ from Example 3.2.2.

Example 3.2.5 We derive $\langle L,\mathcal{W}_0\rangle \Rightarrow \mathcal{M}_0$, where L is the instance of $\langle LHS\rangle$ in Listing 3.1, i.e. `$a : A($x : x, y > 4, y != $x) and $b : B($z : z, z < $x)` and $\mathcal{W}_0$ the abstract working memory from Example 3.2.2. First, we show the related derivation for $L_A =$ `$a : A($x : x, y > 4, y != $x)`:

$$\frac{\dfrac{\mathcal{M}_1=\{o\in\mathcal{O}|\Gamma(o)=\texttt{A}\}\times\{\varnothing\}}{\langle\texttt{A()},\mathcal{W}_0\rangle \Rightarrow \mathcal{M}_1} \qquad \mathcal{M}_2=\{(o,\{\texttt{\$a}\mapsto o\})|(o,\varnothing)\in\mathcal{M}_1\}}{\langle\texttt{\$a : A()},\mathcal{W}_0\rangle \Rightarrow \mathcal{M}_2}$$

$$\frac{\langle\texttt{\$a : A()},\mathcal{W}_0\rangle \Rightarrow \mathcal{M}_2 \qquad \mathcal{M}_3=\{(o,b_v\cup\{\texttt{\$x}\mapsto\mathcal{A}(o,\texttt{x})\})|(o,b_v)\in\mathcal{M}_2\}}{\langle\texttt{\$a : A(\$x : x)},\mathcal{W}_0\rangle \Rightarrow \mathcal{M}_3}$$

$$\frac{\langle \texttt{\$a : A(\$x : x)}, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_3 \qquad \mathcal{M}_4 = \{(o, b_v) \in \mathcal{M}_3 | \mathcal{A}(o, \texttt{y}) > [\![\texttt{4}]\!]\}}{\langle \texttt{\$a : A(\$x : x, y > 4)}, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_4}$$

$$\frac{\langle \texttt{\$a : A(\$x : x, y > 4)}, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_4 \qquad \mathcal{M}_5 = \{(o, b_v) \in \mathcal{M}_4 | \mathcal{A}(o, \texttt{y}) \neq [\![\texttt{\$x}]\!]\}}{\langle \texttt{\$a : A(\$x : x, y > 4, y != \$x)}, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_5}$$

Here we applied the rules (Pattern), (BindP), (BindA), (Cons$_{\alpha,>}$), (Cons$_{\alpha,\texttt{!=}}$) and produced the following sets of abstract matches:

$$\begin{array}{lcl}
\mathcal{M}_1 & = & \{(0, \varnothing), (1, \varnothing)\} \\
\mathcal{M}_2 & = & \{(0, \{\texttt{\$a} \mapsto 0\}), (1, \{\texttt{\$a} \mapsto 1\})\} \\
\mathcal{M}_3 & = & \{(0, \{\texttt{\$a} \mapsto 0, \texttt{\$x} \mapsto 6\}), (1, \{\texttt{\$a} \mapsto 1, \texttt{\$x} \mapsto 5\})\} \\
\mathcal{M}_4 & = & \mathcal{M}_3 \\
\mathcal{M}_5 & = & \{(1, \{\texttt{\$a} \mapsto 1, \texttt{\$x} \mapsto 5\})\}
\end{array}$$

Next, we show the respective derivation for $L_B = \texttt{B(\$z : z)}$:

$$\frac{\mathcal{M}_6 = \{o \in \mathcal{O} | \Gamma(o) = \texttt{B}\} \times \{\varnothing\}}{\langle \texttt{B()}, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_6}$$

$$\frac{\langle \texttt{B()}, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_6 \qquad \mathcal{M}_7 = \{(o, b_v \cup \{\texttt{\$z} \mapsto \mathcal{A}(o, \texttt{z})\}) | (o, b_v) \in \mathcal{M}_6\}}{\langle \texttt{B(\$z : z)}, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_7}$$

Here we applied the rules (Pattern), (BindA) and produced the following sets of abstract matches:

$$\begin{array}{lcl}
\mathcal{M}_6 & = & \{(2, \varnothing), (3, \varnothing), (4, \varnothing)\} \\
\mathcal{M}_7 & = & \{(2, \{\texttt{\$z} \mapsto 3\}), (3, \{\texttt{\$z} \mapsto 7\}), (4, \{\texttt{\$z} \mapsto 2\})\}
\end{array}$$

Finally, we can derive the matches for $L = L_A$ `and B($z : z, z < $x)`:

$$\frac{\langle L_A, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_5 \qquad \langle L_B, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_7 \qquad \mathcal{M}_8 = \{m_1 \circ m_2 | (m_1, m_2) \in \mathcal{M}_5 \times \mathcal{M}_7\}}{\langle L_A \texttt{ and } L_B, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_8}$$

$$\frac{\langle L_A \texttt{ and } L_B, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_8 \qquad \mathcal{M}_0 = \{(o, b_v) \in \mathcal{M}_8 | \mathcal{A}(o, \texttt{z}) < [\![\texttt{\$x}]\!]\}}{\langle L, \mathcal{W}_0\rangle \Rightarrow \mathcal{M}_0}$$

Here we applied the rules (And), (Cons$_{\beta,<}$), and produced the following sets of abstract matches:

$$\begin{array}{lcl}
\mathcal{M}_8 & = & \{(2, \{\texttt{\$a} \mapsto 1, \texttt{\$x} \mapsto 5, \texttt{\$z} \mapsto 3\}), (3, \{\texttt{\$a} \mapsto 1, \texttt{\$x} \mapsto 5, \texttt{\$z} \mapsto 7\}), \\
 & & (4, \{\texttt{\$a} \mapsto 1, \texttt{\$x} \mapsto 5, \texttt{\$z} \mapsto 2\})\} \\
\mathcal{M}_0 & = & \{(2, \{\texttt{\$a} \mapsto 1, \texttt{\$x} \mapsto 5, \texttt{\$z} \mapsto 3\}), (4, \{\texttt{\$a} \mapsto 1, \texttt{\$x} \mapsto 5, \texttt{\$z} \mapsto 2\})\}
\end{array}$$

Notice, that we were able to resolve variables in instances of $\langle$*IntegerExpression*$\rangle$ using the variable binding functions b_v. This is always possible due to the restrictions to the syntax and since instances of $\langle$*Binding*$\rangle$ are evaluated before α- and β-constraints.

Semantics of RHS

We define a relation $\langle U, \mathcal{W}, b_v \rangle \Rightarrow \mathcal{W}'$ such that $\mathcal{W}'$ represents the new abstract working memory after applying the actions defined in the instance U of $\langle RHS \rangle$, based on the abstract working memory $\mathcal{W}$ and the variable binding function b_v.

The semantics of $\langle RHS \rangle$ are canonical and follow common patterns known from structural operational semantics for imperative programming languages. We define how a given working memory is changed for each action in the RHS of a rule when a certain variable binding function from the matches of the LHS of that rule is selected. Finally, we define the sequential composition of such actions.

Definition 3.2.6 Let $\mathcal{P}$ be a package of $\mathrm{DRL}_{\mathbb{Z}}$, $\mathcal{W}$ and $\mathcal{W}'$ abstract working memories for $\mathcal{P}$, R an instance of $\langle Rule \rangle$ in $\mathcal{P}$, L the instance of $\langle LHS \rangle$ in R, and U the instance of $\langle RHS \rangle$ in R. Furthermore, let $\langle L, \mathcal{W} \rangle \Rightarrow \mathcal{M}$ and $(o, b_v) \in \mathcal{M}$. We define the operational semantics $\langle U, \mathcal{W}, b_v \rangle \Rightarrow \mathcal{W}'$ inductively over the syntactic structure of U, like defined in Definition 3.1.3:

(Delete)

$$\frac{\mathcal{O}' = \mathcal{O} \setminus \{b_v(v)\} \qquad \Gamma' = \Gamma|\mathcal{O}' \qquad \mathcal{A}' = \mathcal{A}|(\mathcal{O}' \times \mathcal{I})}{\langle \texttt{delete(}v\texttt{)}, \mathcal{W}, b_v \rangle \Rightarrow \mathcal{W}'}$$

where v is an instance of $\langle Variable \rangle$ appearing in a pattern binding in L.

(Insert)

$$\frac{\mathcal{O}' = \mathcal{O} \cup \{o\} \qquad \Gamma' = \Gamma \cup \{o \mapsto T\} \qquad \mathcal{A}' = \mathcal{A} \cup \bigcup_i \{(o, a_i) \mapsto [\![e_i]\!]\}}{\langle \texttt{insert(new }T\texttt{(}e_1\texttt{, ..., }e_n\texttt{))}, \mathcal{W}, b_v \rangle \Rightarrow \mathcal{W}'}$$

where $o = \min(\mathbb{N} \setminus \mathcal{O})$, (e_n) a sequence of integer expressions, and (a_n) is the sequence of all attribute identifiers appearing in the instance of $\langle Type \rangle$ which is identified by the type identifier T.

(Modify)

$$\frac{\mathcal{A}' = \{(o, a, z) \in \mathcal{A} | o \neq b_v(v) \vee \bigwedge_i a \neq a_i\} \cup \bigcup_i \{(b_v(v), a_i) \mapsto [\![e_i]\!]\}}{\langle \texttt{modify(}v\texttt{)\{}a_1\texttt{=}e_1\texttt{, ..., }a_k\texttt{=}e_k\texttt{\}}, \mathcal{W}, b_v \rangle \Rightarrow \mathcal{W}'}$$

where $\mathcal{O}' = \mathcal{O}$, $\Gamma' = \Gamma$, (e_n) a sequence of integer expressions, and (a_k) is some sequence of attribute identifiers appearing in the instance of $\langle Type \rangle$ which is identified by the pattern type identifier related to the pattern binding v.

(Sequence)

$$\frac{\langle U, \mathcal{W}, b_v \rangle \Rightarrow \mathcal{W}' \qquad \langle A, \mathcal{W}', b_v \rangle \Rightarrow \mathcal{W}''}{\langle U\texttt{;}A, \mathcal{W}, b_v \rangle \Rightarrow \mathcal{W}''}$$

where A is an instance of $\langle Action \rangle$.

Example 3.2.7 We derive $\langle U, \mathcal{W}_0, b_v \rangle \Rightarrow \mathcal{W}_2$, where U is the instance of $\langle RHS \rangle$ in Listing 3.1, $\mathcal{W}_0$ the abstract working memory from Example 3.2.2, and b_v the variable binding function associated with the abstract object identifer 2 in $\mathcal{M}_0$ from Example 3.2.5, that is $b_v = \{\texttt{\$a} \mapsto 1, \texttt{\$x} \mapsto 5, \texttt{\$z} \mapsto 3\}$.
First, we show the derivation for $A_M = \texttt{modify (\$a) \{y = 10, x = \$x*5 \}}$:

$$\frac{\mathcal{A}_1 = \{(o,a,z) \in \mathcal{A}_0 | o \neq 1\} \cup \{(1,\texttt{y}) \mapsto [\![\texttt{10}]\!], (1,\texttt{x}) \mapsto [\![\texttt{\$x*5}]\!]\}}{\langle \texttt{modify (\$a) \{y = 10, x = \$x*5 \}}, \mathcal{W}_0, b_v \rangle \Rightarrow \mathcal{W}_1}$$

Here we applied the rule (Modify) and produced the following abstract working memory $\mathcal{W}_1 = (\mathcal{O}_1, \Gamma_1, \mathcal{A}_1)$:

$$\begin{aligned} \mathcal{O}_1 &= \mathcal{O}_0 \\ \Gamma_1 &= \Gamma_0 \\ \mathcal{A}_1 &= \{(0,\texttt{x}) \mapsto 6, (0,\texttt{y}) \mapsto 6, (1,\texttt{x}) \mapsto 25, (1,\texttt{y}) \mapsto 10, (2,\texttt{z}) \mapsto 3, \\ &\quad (3,\texttt{z}) \mapsto 7, (4,\texttt{z}) \mapsto 2\} \end{aligned}$$

Next, we show the derivation for $A_I = \texttt{insert(new A(20, \$x*\$z))}$:

$$\frac{\mathcal{O}_2 = \mathcal{O}_1 \cup \{5\} \qquad \Gamma_2 = \Gamma_1 \cup \{5 \mapsto \texttt{A}\} \qquad \mathcal{A}_2 = \mathcal{A}_1 \cup \{(5,\texttt{x}) \mapsto [\![\texttt{20}]\!], (5,\texttt{y}) \mapsto [\![\texttt{\$x*\$z}]\!]\}}{\langle \texttt{insert(new A(20, \$x*\$z))}, \mathcal{W}_1, b_v \rangle \Rightarrow \mathcal{W}_2}$$

Here we applied the rule (Insert) and produced the following abstract working memory $\mathcal{W}_2 = (\mathcal{O}_2, \Gamma_2, \mathcal{A}_2)$:

$$\begin{aligned} \mathcal{O}_2 &= \{0,1,2,3,4,5\} \\ \Gamma_2 &= \{0 \mapsto \texttt{A}, 1 \mapsto \texttt{A}, 2 \mapsto \texttt{B}, 3 \mapsto \texttt{B}, 4 \mapsto \texttt{B}, 5 \mapsto \texttt{A}\} \\ \mathcal{A}_2 &= \{(0,\texttt{x}) \mapsto 6, (0,\texttt{y}) \mapsto 6, (1,\texttt{x}) \mapsto 25, (1,\texttt{y}) \mapsto 10, (2,\texttt{z}) \mapsto 3, \\ &\quad (3,\texttt{z}) \mapsto 7, (4,\texttt{z}) \mapsto 2, (5,\texttt{x}) \mapsto 20, (5,\texttt{y}) \mapsto 15\} \end{aligned}$$

Finally, we can apply the rule (Sequence) and know that $\langle U, \mathcal{W}_0, b_v \rangle \Rightarrow \mathcal{W}_2$.

Abstract Rule Engine

Now that we have defined the semantics for $\langle LHS \rangle$ and $\langle RHS \rangle$, we can model the match-resolve-act cycle of Drools. Specifically, we need to decide how to handle conflict resolution. The relevant aspects of conflict resolution in our case are on the one hand the prioritization of multiple matches for one rule; and on the other hand the prioritization of rules, when multiple rules are triggered.

The order in which multiple matches for a given rule are processed by Drools depends on many factors. For example, the order in which the facts are inserted into the working memory is of great importance. Hereby, every update of a fact can influence the order. Moreover, certain caching strategy can alter the order as the working memory grows. In general, the realization of this order is rather opaque. Hence the correct functioning of RBs should never rely on a specific order for the processing of matches.

As a consequence for our theoretical considerations, it is reasonable to assume an arbitrary execution order for multiple matches. This translates to a non-deterministic choice point in our semantics. The semantics we have introduced so far are deterministic. That means that there is at most one $\mathcal{M}$ such that $\langle L, \mathcal{W}\rangle \Rightarrow \mathcal{M}$ for given L and $\mathcal{W}$; and at most one $\mathcal{W}'$ such that $\langle U, \mathcal{W}, b_v\rangle \Rightarrow \mathcal{W}'$ for given U, $\mathcal{W}$, and b_v. To interrelate these semantics, we need to choose a $(o, b_v) \in \mathcal{M}$ and we leave this choice arbitrary. We make this explicit in our next definition:

Definition 3.2.8 Let $\mathcal{P}$ be a package of $\mathrm{DRL}_{\mathbb{Z}}$, $\mathcal{W}$ and $\mathcal{W}'$ abstract working memories for $\mathcal{P}$, R an instance of $\langle \mathit{Rule}\rangle$ in $\mathcal{P}$, $\mathrm{LHS}(R)$ the instance of $\langle \mathit{LHS}\rangle$ in R, and $\mathrm{RHS}(R)$ the instance of $\langle \mathit{RHS}\rangle$ in R. We define the relation $\mathcal{W} \Rightarrow_R \mathcal{W}'$ which models the valid transitions between abstract working memories for the rule R:

(Rule)

$$\frac{\langle \mathrm{LHS}(R), \mathcal{W}\rangle \Rightarrow \mathcal{M} \qquad (o, b_v) \in \mathcal{M} \qquad \langle \mathrm{RHS}(R), \mathcal{W}, b_v\rangle \Rightarrow \mathcal{W}'}{\mathcal{W} \Rightarrow_R \mathcal{W}'}$$

Now we discuss the conflict resolution of Drools when multiple rules are triggered. In contrast to the previously discussed conflict resolution, this strategy is completely transparent. If there are no attributes which influence the control flow, the rules are prioritized based on their order in the DRL file. However, like already mentioned at the end of Section 3.1, it is considered bad practice to rely on a specific order for the execution of rules. Again, we quote [10, p. 152]: *"As a general rule, it is a good idea not to count on rules firing in any particular order, and to author the rules without worrying about a 'flow'."* Hence it is justified to take the same approach as before and assume an arbitrary conflict resolution strategy when multiple rules are triggered. This leads us to the final definition of this section: the semantics of packages in $\mathrm{DRL}_{\mathbb{Z}}$:

Definition 3.2.9 Let $\mathcal{P}$ be a package of $\mathrm{DRL}_{\mathbb{Z}}$, $\mathcal{W}$ and $\mathcal{W}'$ abstract working memories for $\mathcal{P}$. We define the relation $\mathcal{W} \Rightarrow_{\mathcal{P}} \mathcal{W}'$ which models the valid transitions between abstract working memories for the package $\mathcal{P}$:

(Package)

$$\frac{R \in \mathcal{P} \qquad \mathcal{W} \Rightarrow_R \mathcal{W}'}{\mathcal{W} \Rightarrow_{\mathcal{P}} \mathcal{W}'}$$

3.3. Termination Property for $\mathrm{DRL}_{\mathbb{Z}}$

In this section we define the termination property for packages in $\mathrm{DRL}_{\mathbb{Z}}$ using the semantics from the previous section. Furthermore, we discuss *Turing completeness* of $\mathrm{DRL}_{\mathbb{Z}}$ and show that the termination of $\mathrm{DRL}_{\mathbb{Z}}$ is generally undecidable.

The definition of our termination property resembles the termination property for term rewriting systems, as given in Definition 2.4.9:

Definition 3.3.1 Let $\mathcal{P}$ be a package of $\mathrm{DRL}_{\mathbb{Z}}$ and $\mathcal{W}$ an abstract working memory for $\mathcal{P}$. We say that $\mathcal{P}$ is *terminating for* $\mathcal{W}$ iff there is no infinite sequence $(\mathcal{W}_n)$ of abstract working memories such that $\mathcal{W} = \mathcal{W}_0$ and

$$\mathcal{W}_0 \Rightarrow_{\mathcal{P}} \mathcal{W}_1 \Rightarrow_{\mathcal{P}} \mathcal{W}_2 \Rightarrow_{\mathcal{P}} \cdots$$

We call $\mathcal{P}$ *terminating* iff $\mathcal{P}$ is terminating for all abstract working memories.

The termination of a package $\mathcal{P}$ implies that every derivation sequence $\mathcal{W}_0 \Rightarrow_{\mathcal{P}} \mathcal{W}_1 \Rightarrow_{\mathcal{P}} \mathcal{W}_2 \Rightarrow_{\mathcal{P}} \cdots$ eventually ends in an abstract working memory $\mathcal{W}_n$ such that all rules in $\mathcal{P}$ yield an empty set of matches in $\mathcal{W}_n$. Like in the case of TRSs, it is certainly interesting to investigate the related transitive closure $\Rightarrow_{\mathcal{P}}^{+}$ of $\Rightarrow_{\mathcal{P}}$ which could for example be used to define the confluence property for $\mathcal{P}$. However, due to time restrictions in this thesis we limit ourselves to the termination property. In this context the *self-deactivation* property is significant. That is, in a terminating package every rule is self-deactivating:

Definition 3.3.2 Let $\mathcal{P}$ be a package of $\mathrm{DRL}_{\mathbb{Z}}$, R an instance of $\langle \mathit{Rule} \rangle$ in $\mathcal{P}$ and $\mathcal{W}$ an abstract working memory for $\mathcal{P}$. We say that R is *self-deactivating* for $\mathcal{W}$ iff there is no infinite sequence $(\mathcal{W}_n)$ of abstract working memories such that $\mathcal{W} = \mathcal{W}_0$ and

$$\mathcal{W}_0 \Rightarrow_R \mathcal{W}_1 \Rightarrow_R \mathcal{W}_2 \Rightarrow_R \cdots$$

We call R *self-deactivating* iff R is self-deactivating for all abstract working memories.

There is a sequence of matches $(\mathcal{M}_i)$ for sequences of abstract working memories, like in Definition 3.3.2, such that $\langle \mathrm{LHS}(R), \mathcal{W}_i \rangle \Rightarrow \mathcal{M}_i$. The self-deactivation property for R guarantees that such sequences of matches are finite and eventually reach the empty set. Typically for self-deactivating rules are sequences of matches $(\mathcal{M}_i)$, such that $|\mathcal{M}_i|$ is (strictly) decreasing. Rules used in productive environments often define constraints which exclude the possibility to match the same fact twice. Listing 2.2 shows a related technique, which we find also in Chapter 5. However, this is generally not a characteristic for the self-deactivation of rules. Listing 3.2 presents an atypical example of self-deactivation. The considered rule is obviously self-deactivating, since in the presence of an fact of type `B`, eventually every fact of type `A` in the abstract working memory is modified such that the value of the related attribute `x` is greater than 5. Nevertheless, the related sequence $|\mathcal{M}_i|$ is increasing, since every application of this rule inserts another fact of type `B`, which results in a higher number of pairs produced by the operator `and` during the matching process. This example also exhibits the subtle considerations one needs to take into account when looking for a termination criterion for $\mathrm{DRL}_{\mathbb{Z}}$.

Next, we discuss the *Turing completeness* of our abstract rule engine, which shows that the termination property for packages is generally not decidable.

Listing 3.2: Atypical example of self-deactivation in $\mathrm{DRL}_{\mathbb{Z}}$

```
declare A x : Integer end
declare B x : Integer end

rule SD
  when
    $a : A($x : x, x < 5) and B()
  then
    insert(new B($x));
    modify($a) {
      x = $x + 1
    }
end
```

Turing Completeness of $\mathrm{DRL}_{\mathbb{Z}}$

Full DRL is obviously Turing complete since it incorporates the expressive power of Java. However, despite the removal of most Java related features our fragment $\mathrm{DRL}_{\mathbb{Z}}$ remains Turing complete. We consider two basic approaches to verify this, even though we do not carry out the required formal proofs. On the one hand, it is sufficient to show that $\mathrm{DRL}_{\mathbb{Z}}$ can be used to emulate reasoning via *Horn clauses* known from logic programming. On the other hand, one could argue that $\mathrm{DRL}_{\mathbb{Z}}$ has the expressive power to simulate *primitive* and *μ-recursive* functions.

Horn clauses are logical formulas used in logic programming, which can be written as implications

$$p_0 \wedge p_1 \wedge \cdots \wedge p_{n-1} \rightarrow q$$

where p_i and q are atomic formulas. Now one needs show that abstract working memories can serve as a model for such formulas and that the rules of $\mathrm{DRL}_{\mathbb{Z}}$ can emulate the reasoning with Horn clauses. It is intuitively clear that this is possible. Listing 3.3 shows a representation in $\mathrm{DRL}_{\mathbb{Z}}$ of the following classical example:

$$\begin{gathered} human(x) \rightarrow mortal(x) \\ socrates(x) \rightarrow human(x) \end{gathered}$$

The only surprise which one might find in Listing 3.3 is the use of the operator `not`. The related construction ensures the self-deactivation of the rules and guarantees the

Listing 3.3: Emulation of Horn clauses in $\mathrm{DRL}_{\mathbb{Z}}$

```
declare Human x : Integer end
declare Mortal x : Integer end
declare Socrates x : Integer end

rule HumansAreMortal
  when Human($x : x) and not(Mortal(x == $x))
  then insert(new Mortal($x))
end

rule SocratesIsHuman
  when Socrates($x : x) and not(Human(x == $x))
  then insert(new Human($x))
end
```

Listing 3.4: Emulation of functions in $\mathrm{DRL}_{\mathbb{Z}}$

```
declare F e : Integer n0 : Integer n1 : Integer f : Integer end

rule EvaluateF
  when $f : F($n0 : n0, $n1 : n1, e == 0)
  then modify($f) {e = 1, f = 2 * $n0 + $n1}
end
```

termination of this specific package. However, our primary concern here is that this construction also facilitates the intended behavior of the inference process.

The simulation of basic arithmetic functions in $\mathrm{DRL}_{\mathbb{Z}}$ is obviously not a problem. Listing 3.4 shows an example for the function $f(n_0, n_1) = 2n_0 + n_1$. Here we represent elements of functions $\mathbb{N}^n \to \mathbb{N}$ through facts with $n+2$ attributes. The first attributes indicates whether the function value is already evaluated and is crucial to suppress the repeated evaluation of functions. The other attributes represent the function arguments and the related function value.

The composition of functions can be realized through multiple rules. One rule for every inner function, which ensures that the necessary values are present in the abstract working memory; and another rule for the evaluation of the outer function. Listing 3.5 shows an example for the functions $g(n_0, n_1) = n_0 + n_1$ and $f(n) = g(n, n) + 42$.

The idea behind the construction used in Listing 3.5 can be extended to simulate primitive and μ-recursion. Listing 3.6 shows an example which realizes the following primitive recursive function:

$$f(n) = \Sigma_{i=1}^{n} i$$

Listing 3.7 implements the minimisation operator μ for an arbitrary function $f : \mathbb{N} \to \mathbb{N}$:

$$\mu(f) = \min f^{-1}\{0\}$$

Due to time restrictions for this thesis, we are not able to carry out the formal proofs related to the presented approaches. Yet, the rule bases exhibited in this section are not only of interest in regards to the Turing completeness of $\mathrm{DRL}_{\mathbb{Z}}$, but they also reveal certain issues which one should keep in mind when investigating the termination of packages.

Listing 3.5: Emulation of function composition in $\mathrm{DRL}_{\mathbb{Z}}$

```
declare F e : Integer n : Integer f : Integer end
declare G e : Integer n0 : Integer n1 : Integer g : Integer end

rule EvaluateF
  when $f : F($n : n, e == 0) and G($g : g, e == 1, n0 == $n, n1 == $n)
  then modify($f) { e = 1, f = $g + 42 }
end

rule EvaluateFInitializeG
  when F($n : n, e == 0) and not(G(n0 == $n, n1 == $n))
  then insert(new G(0, $n, $n, 0))
end

rule EvaluateG
  when $g : G($n0 : n0, $n1 : n1, e == 0)
  then modify($g) { e = 1, g = $n0 + $n1 }
end
```

Listing 3.6: Emulation of primitive recursion in $\mathrm{DRL}_{\mathbb{Z}}$

```
declare F e : Integer n : Integer f : Integer end

rule EvaluateF
  when $f : F($n : n, e == 0) and F($p : f, e == 1, n == $n - 1))
  then modify($f) { e = 1, f = $p + $n }
end

rule EvaluateFInitialize1
  when $f : F(e == 0, n == 1)
  then modify($f) { e = 1, f == 1 }
end

rule EvaluateFInitializeNMinus1
  when F($n : n, n > 1, e == 0) and not(F(n == $n - 1))
  then insert(new F(0, $n - 1, 0));
end
```

Listing 3.7: Emulation of μ-recursion in $\mathrm{DRL}_{\mathbb{Z}}$

```
declare F e : Integer n : Integer f : Integer end
declare MuF e : Integer m : Integer end

rule EvaluateMuF
  when $m : MuF(e == 0) and F($n : n, e == 1, f == 0)
  then modify($m) { e = 1, m = $n }
end

rule EvaluateMuFInitialize0
  when $m : MuF(e == 0) and not(F(n == 0))
  then insert(new F(0, 0, 0));
end

rule EvaluateMuFInitializeNPlus1
  when $m : MuF(e == 0) and F($n : n, e == 1, f > 0)
    and not(F(n == $n + 1)) and not(F(n == 0))
  then insert(new F(0, $n + 1, 0));
end
```

3.4. Termination Criterion for $\mathrm{DRL}_{\mathbb{Z}}$

In this section, we present a sufficient termination criterion for certain packages in $\mathrm{DRL}_{\mathbb{Z}}$. The related theorem provides the theoretical background for the implementation described in Chapter 4. The idea behind our termination criterion is to extract ITRSs from packages in $\mathrm{DRL}_{\mathbb{Z}}$, such that the termination of an ITRS guarantees the termination of the related package. The rewrite rules of the considered ITRSs correspond to instances of $\langle \mathit{Modify} \rangle$ and translate transitions between facts in abstract working memories to the rewriting of terms representing such facts.

We start with the definition of the necessary restrictions to the fragment $\mathrm{DRL}_{\mathbb{Z}}$:

Definition 3.4.1 Let $\mathrm{DRL}_{\mathbb{Z}}^{t}$ denote the set of packages $\mathcal{P}$ in $\mathrm{DRL}_{\mathbb{Z}}$, such that:

(1) $\mathcal{P}$ does not contain instances of $\langle \mathit{Delete} \rangle$, $\langle \mathit{Insert} \rangle$, or $\langle \mathit{Not} \rangle$.

(2) All instances of $\langle \mathit{Constraint} \rangle$ in $\mathcal{P}$ are α-constraints.

(3) All instances of $\langle \mathit{Modify} \rangle$ in $\mathcal{P}$ are α-modifications.

Next, we describe the procedure which extracts terms and rewrite rules from abstract working memories and packages.

Extraction of Terms and Rewrite Rules

We define an integer term rewriting system $\mathcal{R}_{\mathbb{Z}}(\mathcal{P})$ for packages $\mathcal{P} \in \mathrm{DRL}_{\mathbb{Z}}^{t}$, such that each instance of $\langle \mathit{Modify} \rangle$ in $\mathcal{P}$ corresponds to a conditional rewrite rule $s \to t \mid c$. The term s reflects the type of the fact which is modified, t expresses the modifications to attributes defined in the related instance of $\langle \mathit{Modify} \rangle$, and c is a result of the constraints of the pattern which relates to the pattern binding of the instance of $\langle \mathit{Modify} \rangle$.

Listing 3.8: Example of a rule base written in $\mathrm{DRL}_{\mathbb{Z}}^{t}$

```
declare A
  x : Integer
  y : Integer
  z : Integer
end

declare B
  p : Integer
  q : Integer
end

rule R1
  when
    $a : A($z : z, x == 1, y < $z)
  then
    modify ($a) {
      x = $z + 3
    }
end

rule R2
  when
    $a : A(z > 4) and $b : B(q == 5)
  then
    modify ($a) {
      y = 7,
      z = 5
    }
    modify ($b) {
      p = 6
    }
end
```

We use examples to illustrate the definitions which implement the translation of syntactical and semantic structures of $\mathrm{DRL}_{\mathbb{Z}}$ to notions of signatures, terms and rewrite rules. Listing 3.8 presents the rule base $\mathcal{P}_0$, which we use for this purpose. We start with the definition of an ITRS-signature for packages in $\mathrm{DRL}_{\mathbb{Z}}$:

Definition 3.4.2 Let $\mathcal{P}$ be a package of $\mathrm{DRL}_{\mathbb{Z}}$. The *package signature* $\Sigma_{\mathbb{Z}}(\mathcal{P})$ is an ITRS-signature $\Sigma_{\mathbb{Z}}(\mathcal{P}) = (\mathcal{V}, \mathcal{F}, \alpha)$, like specified in Definition 2.4.12, such that:

(1) $\mathcal{F}$ contains a function symbols f_T for all type identifiers T in $\mathcal{P}$.

(2) $\alpha(f_T) = n$, where n is the number of attributes associated with T.

For our example we know that the signature $\Sigma_{\mathbb{Z}}(\mathcal{P}_0)$ contains the function symbols $f_{\mathtt{A}}$ and $f_{\mathtt{B}}$ in addition to the pre-defined function symbols of ITRSs. Furthermore, we have $\alpha(f_{\mathtt{A}}) = 3$ and $\alpha(f_{\mathtt{B}}) = 2$. Next, we define terms which are used to represent types and integer expressions:

Definition 3.4.3 Let $\mathcal{P}$ be a package of $\mathrm{DRL}_{\mathbb{Z}}$. We define the following terms:

(1) The *term representation for types* $t_T = f_T(v_0, \ldots, v_{n-1})$, where T is a type identifier in $\mathcal{P}$, $v_0, \ldots, v_{n-1} \in \mathcal{V}$ are distinct variables, and $n = \alpha(f_T)$.

(2) The *term representation for integer expressions* t_e denotes the canonically defined term representing the integer expression e in $\mathcal{P}$.

For example, we can extract two term representations for types from Listing 3.8: Line 1 to 5 give $t_{\mathtt{A}} = f_{\mathtt{A}}(v_0, v_1, v_2)$ and Line 7 to 10 translate to $t_{\mathtt{B}} = f_{\mathtt{B}}(v_0, v_1)$. Then, we present some of the term representations for integer expression which we can extract from Listing 3.8: Line 14 gives $t_1 = 1$ and $t_{\mathtt{\$z}} = v_0$, Line 17 contains $t_{\mathtt{\$z + 3}} = v_0 + 3$, and the integer expression in Line 26 translates to $t_7 = 7$.

So far the variables in t_T and t_e are independent. Next, we define a relationship between the variables of t_T and t_e and introduce term representations for integer expressions which incorporate attribute bindings. In packages of $\mathrm{DRL}^t_{\mathbb{Z}}$, integer expressions either appear in α-constraints or α-modifications. Hence, every integer expression has an associated type identifier T, where T is either the pattern type identifier of the current instance of $\langle \textit{Pattern} \rangle$ or the pattern type identifier associated with the pattern binding of the current instance of $\langle \textit{Modify} \rangle$. It is obvious how to resolve attribute bindings such that variables in t_e correspond to variables t_T. We denote the result of the necessary replacement of variables in t_e with t^b_e.

The application of the described procedure to the representations for integer expression from our example yields the following terms: $t^b_1 = 1$, $t^b_{\mathtt{\$z}} = v_2$, $t^b_{\mathtt{\$z + 3}} = v_2 + 3$, and $t^b_7 = 7$. Let us introduce terms for instances of $\langle \textit{Modify} \rangle$ and $\langle \textit{Constraint} \rangle$:

Definition 3.4.4 Let $\mathcal{P}$ be a package of $\mathrm{DRL}^t_{\mathbb{Z}}$. We define the following terms:

(1) The *term representation for modifications* $t_M = f_T(t_0, \ldots, t_{n-1})$ where M is an instance of $\langle$*Modify*$\rangle$ in $\mathcal{P}$, T the pattern type identifier corresponding to the pattern binding in M, and $t_i = v_i$ iff $\langle$*Modify*$\rangle$ does not contain the attribute represented by v_i and $t_i = t^b_e$ otherwise, where e is the integer expression following the respective attribute identifier.

(2) The *term representation for constraints* t_C denotes the canonically defined term representing the α-constraint C in $\mathcal{P}$.

In Listing 3.8 we find instances of $\langle$*Modify*$\rangle$ in Line 16 to 18, Line 25 to 28, and Line 29 to 31, which we denote with M_0, M_1, and M_2. We have $t_{M_0} = f_{\mathtt{A}}(v_2 + 3, v_1, v_2)$, $t_{M_1} = f_{\mathtt{A}}(v_0, 7, 5)$ and $t_{M_2} = f_{\mathtt{B}}(6, v_1)$. Afterwards, we find four instances of $\langle$*Constraint*$\rangle$ in Line 14 and 23, which we denote with C_0, C_1, C_2, and C_3. This gives the terms $t_{C_0} = (v_0 = 1)$, $t_{C_1} = (v_1 < v_2)$, $t_{C_2} = (v_2 > 4)$, and $t_{C_3} = (v_1 = 5)$. Finally, we have the necessary tools to define rewrite rules for instances of $\langle$*Modify*$\rangle$, $\langle$*Rule*$\rangle$, and $\langle$*Package*$\rangle$:

Definition 3.4.5 Let $\mathcal{P}$ be a package of $\mathrm{DRL}^t_{\mathbb{Z}}$. We define the following conditional rewrite rules and term rewriting systems:

(1) $r_M = (t_T \to t_M \mid t_{C_0} \wedge \cdots \wedge t_{C_{n-1}})$, where M is an instance of $\langle$*Modify*$\rangle$ in $\mathcal{P}$, T the pattern type identifier corresponding to the pattern binding in M, and C_i are the instances of $\langle$*Constraint*$\rangle$ in the respective pattern.

(2) $\mathcal{R}_{\mathbb{Z}}(R) = \{r_M \mid M$ instance of $\langle$*Modify*$\rangle$ in $R\}$, where R is an instance of $\langle$*Rule*$\rangle$ in $\mathcal{P}$.

(3) $\mathcal{R}_{\mathbb{Z}}(\mathcal{P}) = \bigcup_R \mathcal{R}(R)$ is the term rewriting system for the package $\mathcal{P}$.

The integer term rewriting system $\mathcal{R}_{\mathbb{Z}}(\mathcal{P}_0)$ for our example from Listing 3.8 is given below:

$$
\begin{gathered}
r_{M_0} : f_{\mathtt{A}}(v_0, v_1, v_2) \to f_{\mathtt{A}}(v_2 + 3, v_1, v_2) \mid v_0 = 1 \wedge v_1 < v_2 \\
r_{M_1} : f_{\mathtt{A}}(v_0, v_1, v_2) \to f_A(v_0, 7, 5) \mid v_2 > 4 \\
r_{M_2} : f_{\mathtt{B}}(v_0, v_1) \to f_{\mathtt{B}}(6, v_1) \mid v_1 = 5
\end{gathered}
$$

Before we can state our termination criterion, we need one last definition which gives term representations for the facts in abstract working memories:

Definition 3.4.6 Let $\mathcal{P}$ be a package of $\mathrm{DRL}_{\mathbb{Z}}$ and $\mathcal{W}$ an abstract working memory for $\mathcal{P}$. We define the *term representation for facts* $t_{o,\mathcal{W}} = f_{\Gamma(o)}(\mathcal{A}(o, a_0), \ldots, \mathcal{A}(o, a_{n-1}))$, where $o \in \mathcal{O}$ and a_i is are the respective attribute identifiers associated with the type identified by $\Gamma(o)$.

Suppose we execute the actions `insert(new A(1, 2, 3))` and `insert(new B(4, 5))` in an empty abstract working memory $\mathcal{W}$ for $\mathcal{P}_0$. The resulting abstract working $\mathcal{W}'$ memory would yield the term representations for facts $t_{0,\mathcal{W}'} = f_{\mathtt{A}}(1, 2, 3)$ and $t_{1,\mathcal{W}'} = f_{\mathtt{B}}(4, 5)$.

The Termination Criterion

We use the definitions stated above to formulate a sufficient termination criterion for the packages of $\mathrm{DRL}_{\mathbb{Z}}^{t}$. We interrelate the termination property of packages $\mathcal{P}$ with the respective property of the integer term rewriting system $\mathcal{R}_{\mathbb{Z}}(\mathcal{P})$ such that the termination of $\mathcal{R}_{\mathbb{Z}}(\mathcal{P})$ guarantees the termination of $\mathcal{P}$. The prove of our termination criterion follows basically by construction and an argument about certain limit points in infinite sequences of abstract working memories.

We begin with the statement of a lemma, which allows the extraction of term rewriting steps from the semantics for rules in $\mathrm{DRL}_{\mathbb{Z}}^{t}$:

Lemma 3.4.7 *Let $\mathcal{P}$ be a package of $DRL_{\mathbb{Z}}^{t}$ and $\mathcal{W}$, $\mathcal{W}'$ abstract working memories for $\mathcal{P}$. Furthermore, let R be an instance of $\langle Rule \rangle$ in $\mathcal{P}$ such that*

$$\mathcal{W} \Rightarrow_R \mathcal{W}'.$$

Then, there exists an instance M of $\langle Modify \rangle$ in R and an abstract object identifier $o \in \mathcal{O}$ such that

$$t_{o,\mathcal{W}} \rightarrow_{r_M} t_{o,\mathcal{W}'}$$

is a valid rewrite step in $\mathcal{R}_{\mathbb{Z}}(\mathcal{P})$.

Proof: The statement follows by induction over the semantics of TRS and $\mathrm{DRL}_{\mathbb{Z}}$. $\square$

We do not carry out the cumbersome proof required to formally verify Lemma 3.4.7. However, it is intuitively clear that the above statement is valid. The $\langle RHS \rangle$ of rules in $\mathrm{DRL}_{\mathbb{Z}}^{t}$ allow only the modification of facts. Therefore, at least one fact o is modified in the transition $\mathcal{W} \Rightarrow_R \mathcal{W}'$. This fact o satisfied the α-constraints of the related pattern in the abstract working memory $\mathcal{W}$. The construction of the rewrite rule r_M guarantees the validity of the rewrite step $t_{o,\mathcal{W}} \rightarrow_{r_M} t_{o,\mathcal{W}'}$.

Finally, we present our desired termination criterion:

Theorem 3.4.8 *Let $\mathcal{P}$ be a package of $DRL_{\mathbb{Z}}^{t}$. The following statement holds:*

$$\mathcal{R}_{\mathbb{Z}}(\mathcal{P}) \text{ terminating} \longrightarrow \mathcal{P} \text{ terminating}.$$

Proof: We prove this by contraposition. Suppose $\mathcal{P}$ is non-terminating, then we have an infinite sequence $(\mathcal{W}_n)$ of abstract working memories and an infinite sequence (R_n) of rules in $\mathcal{P}$ such that:

$$\mathcal{W}_0 \Rightarrow_{R_0} \mathcal{W}_1 \Rightarrow_{R_1} \mathcal{W}_2 \Rightarrow_{R_2} \cdots$$

Since the rules of $\mathcal{P}$ do not contain instances of $\langle Delete \rangle$ or $\langle Insert \rangle$, we know that no facts are created or deleted in this process, that is $\mathcal{O}_i = \mathcal{O}_{i+1} =: \mathcal{O}$. We also know that each rule must contain at least one instance of $\langle Modify \rangle$ that is why at least one $o \in \mathcal{O}$ is matched by $\mathrm{LHS}(R_i)$ and modified in $\mathrm{RHS}(R_i)$.

Since set $\mathcal{O}$ is finite and the sequence $(\mathcal{W}_n)$ is infinite, we know that at least one abstract object pointer in $\mathcal{O}$ is matched infinitely often. Let (i_m) be a sequence of indices, such that $o \in \mathcal{O}$ is matched by $\text{LHS}(R_{i_j})$ in every step of the following sequence:

$$\mathcal{W}_{i_0} \Rightarrow_{R_{i_0}} \mathcal{W}_{i_1} \Rightarrow_{R_{i_1}} \mathcal{W}_{i_2} \Rightarrow_{R_{i_2}} \cdots$$

We denote the related instances of $\langle \mathit{Modify} \rangle$ with M_{i_j}. Lemma 3.4.7 guarantees that

$$t_{o,\mathcal{W}_{i_0}} \to_{M_{i_0}} t_{o,\mathcal{W}_{i_1}} \to_{M_{i_1}} t_{o,\mathcal{W}_{i_2}} \to_{M_{i_2}} \cdots$$

is an infinite application of rewrite rules of $\mathcal{R}_{\mathbb{Z}}(\mathcal{P})$. $\square$

For our example it is obvious that $\mathcal{R}_{\mathbb{Z}}(\mathcal{P}_0)$ is non-terminating. It is not a problem to find the following infinite term rewrite sequence:

$$f_{\mathrm{B}}(0,5) \to_{r_{M_2}} f_{\mathrm{B}}(6,5) \to_{r_{M_2}} f_{\mathrm{B}}(6,5) \to_{r_{M_2}} f_{\mathrm{B}}(6,5) \to_{r_{M_2}} \cdots$$

In this case, our termination criterion does not guarantee the termination of the package $\mathcal{P}_0$. Note, that our criterion is not a characterization of termination in the sense that the non-termination of $\mathcal{R}_{\mathbb{Z}}(\mathcal{P})$ implies the non-termination of $\mathcal{P}$. Still, our result might be an indicator for the non-termination of $\mathcal{P}_0$.

Indeed, $\mathcal{P}_0$ is non-terminating. Assume an abstract working memory $\mathcal{W}$ such that $f_A(0,7,5)$ and $f_B(6,5)$ are the term representations for the facts in $\mathcal{W}$. Let R_2 be the second rule of $\mathcal{P}_0$. We can derive the following infinite sequence:

$$\mathcal{W} \Rightarrow_{R_2} \mathcal{W} \Rightarrow_{R_2} \mathcal{W} \Rightarrow_{R_2} \cdots$$

In the next chapter we showcase our implementation which allows the automated extraction of the described integer term rewriting systems from packages of $\text{DRL}^t_{\mathbb{Z}}$. Then, we discuss how to use AProVE to test the termination property of the extracte ITRSs. In Chapter 5 we give practical applications of our termination criterion and show the proof of termination for a rule base which is used in productive environments.

4. Implementation

In this chapter we present our implementation and explain how to install and use it. Furthermore, we briefly describe the structure of the related source code and discuss the approach we chose to process DRL. At the moment of publication the implementation should be considered a prototype. The current version is 0.9.1.

The first section states the system requirements and dependencies which are necessary to setup and run the implementation. In the next section we show how to use the command line interface of the program and explain its core features. Afterwards, we instruct the reader how to use AProVE to evaluate the ITRSs which are produced by the implementation. Section 4.3 illustrates the overall program structure and describes the components of the implementation. Finally in the last section, we give some details on how the program parses DRL and which classes and libraries of the Drools project are utilized in this process.

4.1. Program Installation

The latest source code and binary version of the implementation is available online at `https://github.com/jss-de/drools-checker`. The implementation is written in Java and depends on the Java Runtime Environment (JRE) version 1.7 to be executed.

There are two possibilities to install the implementation itself and the required dependencies: compiling the implementation from source with the build automation tool Maven; or downloading the binary version of the implementation and its dependencies from their respective vendors. We recommend the first option, since Maven automatically downloads the required dependencies from remote repositories.

To build the implementation with Maven, install Maven and download the source code of the implementation. Open a terminal and navigate to the root directory of the downloaded source code. This directory should contain the file `pom.xml`. Here execute the following commands:

```
mvn package -Dmaven.test.skip=true
mvn install dependency:copy-dependencies -Dmaven.test.skip=true
```

During successful execution of these commands Maven creates the directory `target` in the current location, which then contains the file `drools-checker-0.9.1.jar` and the directory `dependency`. The file `drools-checker-0.9.1.jar` is a Java archive which contains the implementation itself. The directory `dependency` contains the required dependencies. At this point the installation is finished.

To manually install the program, download the file `drools-checker-0.9.1.jar` which can be found under the aforementioned URL. Acquire the following Java libraries, which should be available online and are provided by their respective vendors:

antlr-runtime-3.5.jar
commons-cli-1.2.jar
drools-compiler-6.1.0.Final.jar
drools-core-6.1.0.Final.jar
ecj-4.3.1.jar
junit-4.8.1.jar
kie-api-6.1.0.Final.jar
kie-internal-6.1.0.Final.jar
log4j-1.2.14.jar
mvel2-2.2.1.Final.jar
protobuf-java-2.5.0.jar
slf4j-api-1.7.2.jar
slf4j-log4j12-1.5.6.jar
slf4j-simple-1.6.2.jar
stax-utils-20070216.jar
xmlpull-1.1.3.1.jar
xpp3_min-1.1.4c.jar
xstream-1.4.7.jar

Create the directory `dependency` in the location of `drools-checker-0.9.1.jar` and store the aforementioned Java libraries in this directory. This completes the installation.

4.2. Program Operation

The main feature of the program is the implementation of the algorithm described in Section 3.4, which generates ITRSs for rule bases in $\mathrm{DRL}^t_{\mathbb{Z}}$. We ease some of the related restrictions of the syntax. For example, we can parse RB without the declaration of the MVEL dialect. Furthermore, the program can translate files in DRL format to a custom XML format which exposes the intermediate representation of DRL in the program. This feature is intended for debugging purposes.

The functionality of the program is accessible via a command line interface. The layout of this interface follows common standards. To display the documentation of this interface, open a terminal and navigate to the location of `drools-checker-0.9.1.jar`. Here execute the command `java -jar drools-checker-0.9.1.jar -H`. Listing 4.1 show the output of this command. For example, the following command generates the ITRS for the DRL file `test.drl` and stores it in the file `test.inttrs`: `java -jar drools-checker-0.9.1.jar -I test.drl -O test.inttrs -T INTTRS`.

Listing 4.1: Command line interface of the implementation

```
usage: java -jar drools-checker-0.9.1.jar [-H] [-I <arg>]
       [-O <arg>] [-T <arg>] [-V]
 -H,--help           display this help and exit
 -I,--input <arg>    specify input file
 -O,--output <arg>   specify output file
 -T,--type <arg>     specify output type:
                     INTTRS for integer term rewriting system
                     XML for extensible markup language
 -V,--version        output version information and exit
```

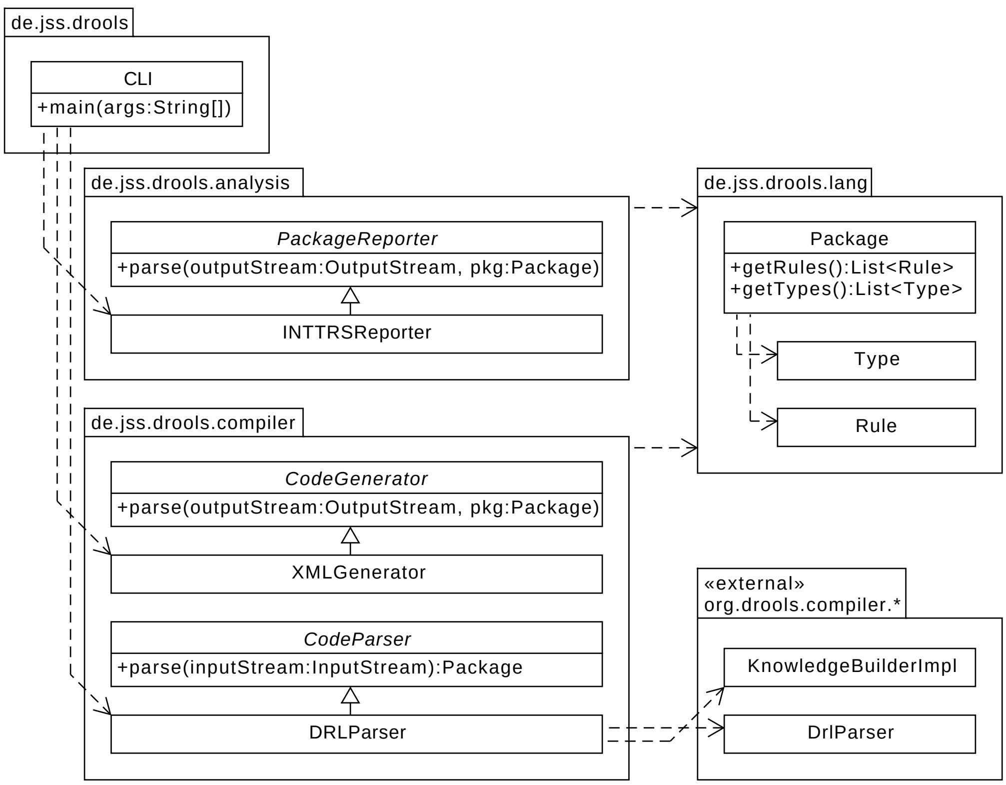
The termination property of the generated ITRS can then be evaluated with AProVE. It is possible to install and use AProVE locally, but in some cases it might be more convenient to use its web interface which can be found online at `http://aprove.informatik.rwth-aachen.de/`.

4.3. Program Structure

The source code of the implementation is organized in a Maven project and consists of four Java packages, 29 classes and 2174 lines of source code. Figure 4.1 provides an UML diagram which gives an overview of these packages, selected classes, and their dependencies. Furthermore, it showcases the dependency between our implementation and the Drools project. The complete documentation of the source code can be found in Appendix B.

The package `de.jss.drools` stores the command line interface and contains the main entry point of the application. The package `de.jss.drools.lang` represents

Figure 4.1.: UML diagram of program packages and selected classes



Listing 4.2: XML representation of Line 10 to 16 of Listing 2.2

```
<rule name="Violets are blue?">
  <conditions>
    <pattern binding="$f" type="mother.goose.rhymes.Flower">
      <constraint attribute="color" expression="&quot;blue&quot;" relation="!=" />
      <constraint attribute="name" expression="&quot;Violet&quot;" relation="==" />
    </pattern>
  </conditions>
  <consequences>
    <message value="System.out.println(&quot;We need to fix some violet.&quot;)" />
    <action binding="$f" type="Modification">
      <assignment attribute="color" expression="&quot;blue&quot;" />
    </action>
  </consequences>
</rule>
```

the model of the application and encapsulates classes which are used to build the abstract syntax tree (AST) that is used for the internal representation of DRL. The package `de.jss.drools.compiler` contains classes which parse DRL files into ASTs and generate XML files from ASTs. Finally, the package `de.jss.drools.analysis` contains the classes which implement the algorithm that generates ITRSs from ASTs.

The package `de.jss.drools` is easily understood, since it consists of the single class `CLI` which implements the command line interface and contains the main entry point of the application. Here we use the Apache Commons CLI library [1] to provide a standard conform command line interface.

The package `de.jss.drools.lang` contains classes used to build the abstract syntax tree of DRL files and represents the data model of the implementation. The root of the AST is represented through the class `Package`. This class instantiates list of the classes `Rule` and `Type` which represent the second level of the AST. This schema continues to cover different aspects of DRL, like, for example, attributes, conditions, consequences, patterns or bindings. As an example, Listing 4.2 displays the XML representation of the AST of the second rule of Listing 2.2.

The central classes of the package `de.jss.drools.compiler` are `DRLParser` and `XMLGenerator`. The class `DRLParser` is used to parse DRL files and creates the internal AST representation of their content. In the next section we give more details about the implementation of `DRLParser` and its dependencies to classes from the Drools project. The class `XMLGenerator` generates XML files from ASTs and can be used to expose their structure. Listing 4.2 showcases this functionality. The class `XMLGenerator` implements a common XML serialization pattern and uses the library provided by the StAX Utilities Project (`https://java.net/projects/stax-utils/`) to create standard conform XML documents.

Finally, the abstract base class `PackageReporter` and its concrete implementation `ITRSReporter` are stored in the package `de.jss.drools.analysis`. The class `ITRSReporter` implements the algorithm described in Section 3.4 and generates integer term rewriting systems for DRL rule bases represented by ASTs. Since AProVE does not have native support for the operators == and != it is necessary to add another translation step, which transforms these operators using >= and <= respectively < and >.

4.4. Parsing DRL

When confronted with the task to programmatically analyze DRL files, we investigated possibilities to reuse existing classes of the libraries provided by Drools. Here the class `InternalKnowledgePackage` is used for the internal representation of DRL and the class `KnowledgeBuilderImpl` creates the respective instances from DRL files. These instances provide most of the information we need for our evaluation. However, some necessary properties are not accessible since `KnowledgeBuilderImpl` compiles the RHS of rules to Java bytecode. In this process, it hides some required details, like for example pattern bindings. The class `KnowledgeBuilderImpl` itself depends on the classes `PackageDescr` and `org.drools.compiler.compiler.DrlParser`. The class `DrlParser` creates a low-level representation of the considered DRL in the form of `PackageDescr` instances. These instances expose the details which are hidden in the instances of `InternalKnowledgePackage`.

Our implementation of `DRLParser` utilizes all of the aforementioned classes to create high-level representations of DRL in the form of `InternalKnowledgePackage` instances and low-level representations of DRL in the form of `PackageDescr` instances. These objects are then traversed in parallel and relevant information is gathered from the `InternalKnowledgePackage` whenever possible and from the `PackageDescr` if necessary.

5. Case Study

In this chapter we show practical applications of our implementation by analyzing a rule base used in productive environments and present the related data and results. Next, we describe the preparations which are necessary to analyze this particular RB and discuss how to possibly automatize these preparations. At the end of the chapter we give some benchmarks, which are based on the data of this case study.

The considered RB is taken from an actual project at Capgemini. Due to a non-disclosure agreement we cannot expose the original data. Nevertheless, we are able to present here an anonymized version of this RB. We use the name *Capgemini rule base* (CRB) to refer to the investigated RB.

Like mentioned in Chapter 3 and 4, we can only analyze certain RBs which lie in the fragment $\mathrm{DRL}_{\mathbb{Z}}$. To be able to analyze the CRB, we are required to manually translate it to this fragment. Since this translation process is cumbersome, we limit ourselves to a selected set of three rules of the CRB. Most of this translation process can be automated.

Notice that there is one preparation step which can not be automated based on the sole information from the CRB. This preparation step is necessary, since the CRB does not terminate for an arbitrary working memory and relies on certain assumptions about the number of facts in the working memory. We discuss this in detail in Section 5.1 and 5.2 and present possible solution approaches. Here we also demonstrate that most of the restrictions of the implementation are not of fundamental nature and their overcoming is only a matter of software development efforts, which were not possible in this thesis due to time restrictions.

In the first section we give a rough outline of the CRB and describe its purpose staying in the limits of our non-disclosure agreement. In the next section we describe the necessary translation steps, which enabled us to analyze the CRB. In Section 5.3 we present the results of this analysis and show the benefits that an automated analysis could have for the development process of the CRB. In the last section we present some benchmarks and discuss the performance of our implementation and AProVE when confronted with the translated parts of the CRB.

5.1. Subject

As mentioned before, we are bound to a non-disclosure agreement and cannot expose too much details about the CRB. Hence the examples and figures we present in this chapter are taken from an anonymized version of the actual rule base used at Capgemini. Nevertheless, we are able to give a rough outline of its structure and purpose.

The CRB is used to decide about the visibility of certain data in some client-server context. Here the client sends a request for data to the server. The server loads the requested data from a database and stores it in intermediate objects. These objects together with an object representing information about the client and the request itself are then passed to the working memory of the CRB. Depending on these objects, the CRB divides the data in three categories: visible, partially visible, and invisible. In following steps, the server uses this categorization to compile a response to the client.

The CRB is used to process a single request at a time. Hence it is not confronted with an arbitrary working memory and thus relies on the assumption that the working memory contains exactly one instance of the object which represents the request of the client. This restriction of the working memory plays a crucial role in the preparations we describe in Section 5.2.

The CRB is stored in a collection of so-called *decision tables*. This format allows the compact representation of the Drools RBs which consist of many rules with similar structure. A decision table begins with a rule template followed by the data which is used to generate the actual rules from this template. The first two or three rows of a decision table define the rule template and each following row provides data which is combined with the template to define a rule. The first column of a decision table represents the rule name. The following columns either have the header *condition* or *action* and relate to the LHS respectively the RHS of rules. Figure 5.1 shows an example of a decision table which mirrors Rule 1 and 3 of Listing 2.2. For a complete documentation of decision tables in Drools, see [10, p. 164].

Figure 5.1.: Decision table for Rule 1 and 3 of Listing 2.2

Rule Name	Condition	Condition	Action
	Flower		
	color == "$param"	name == "$param"	System.out.println ("$param");
Roses are red.	red	Rose	We found a red rose.
Violets are blue.	blue	Violet	We found a blue violet.

Drools supports multiple file formats for decision tables, among them Excel spreadsheets, which are used for the development of RBs at Capgemini. The CRB is part of a collection of Excel spreadsheets which define 27 rule bases containing 815 decision tables and 3479 rules. The CRB itself contains an average of 320 rules which are grouped in 30 decision tables. We selected one of these decision tables for further investigation. The selected decision table has 40 columns and 51 rows, and defines 49 rules. Figure 5.2 gives an overview of the selected decision table. A complete and more readable version can be found in the appendix in Figure A.1, A.2, and A.3. The Columns 2 to 4 describe the LHS of rules and the Columns 5 to 40 the RHS of rules. The LHS of rules is used to select certain data objects and the request object from the working memory. The RHS defines whether the selected data object should be marked as visible, partially visible, or invisible in the context of the provided request. The Columns 5 to 38 are used to

Figure 5.2.: Investigated decision table – Overview

	CONDITION	CONDITION	CONDITION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION
			Request($senderGroup : senderGroup) and $param	String action="%";	if("01".equals($senderGroup)) { action = "$param"; }	if("02".equals($senderGroup)) { action = "$param"; }	if("03".equals($senderGroup)) { action = "$param"; }	if("04".equals($senderGroup)) { action = "$param"; }	if("05".equals($senderGroup)) { action = "$param"; }	if("06".equals($senderGroup)) { action = "$param"; }	if("07".equals($senderGroup)) { action = "$param"; }	if("08".equals($senderGroup)) { action = "$param"; }	if("09".equals($senderGroup)) { action = "$param"; }	if("10".equals($senderGroup)) { action = "$param"; }	if("11".equals($senderGroup)) { action = "$param"; }	if("12".equals($senderGroup)) { action = "$param"; }	if("13".equals($senderGroup)) { action = "$param"; }	if("14".equals($senderGroup)) { action = "$param"; }	if("15".equals($senderGroup)) { action = "$param"; }	if("16".equals($senderGroup)) { action = "$param"; }	if("17".equals($senderGroup)) { action = "$param"; }	if("18".equals($senderGroup)) { action = "$param"; }	if("19".equals($senderGroup)) { action = "$param"; }	if("20".equals($senderGroup)) { action = "$param"; }	if("21".equals($senderGroup)) { action = "$param"; }	if("22".equals($senderGroup)) { action = "$param"; }	if("23".equals($senderGroup)) { action = "$param"; }	if("24".equals($senderGroup)) { action = "$param"; }	if("25".equals($senderGroup)) { action = "$param"; }	if("26".equals($senderGroup)) { action = "$param"; }	if("27".equals($senderGroup)) { action = "$param"; }	if("28".equals($senderGroup)) { action = "$param"; }	if("29".equals($senderGroup)) { action = "$param"; }	if("30".equals($senderGroup)) { action = "$param"; }	if("31".equals($senderGroup)) { action = "$param"; }	if("32".equals($senderGroup)) { action = "$param"; }	if("33".equals($senderGroup)) { action = "$param"; }	if(action.equals("N") && !"0" .equals($ds.getVisibility())) { $ds.setVisibility("0"); update($ds); }	if(action.equals("H") && !"1" .equals($ds.getVisibility())) { $ds.setVisibility("1"); update($ds); }
Rule Name	Data Set	ID	LHS		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	N	H
Rule 1	DataSet1		$ds : DataSet1()		J	J	J	J	J	J	N	J	J	J	N	H	H	J	J	J	J	J	J	J	N	N	N	N	J	N	N	N	J	N	N	N	N		
Rule 2	DataSet2		$ds : DataSet2()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	N	N	J	N	N	J	J	J	J	J	N		
Rule 3	DataSet3		$ds : DataSet3()		J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	J	J	N	N		
Rule 4	DataSet4		$ds : DataSet4()		J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	J	J	J	J		
Rule 5	DataSet5		$ds : DataSet5()		J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	J	J	N	N		
Rule 6	DataSet6		$ds : DataSet6()		J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	J	J	N	N		
Rule 7	DataSet7		$ds : DataSet7()		J	J	J	J	J	J	N	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 8	DataSet8		$ds : DataSet8()		J	J	J	J	J	J	N	J	J	J	J	H	H	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 9	DataSet9	== "A"	$ds : DataSet9(id == "A")		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 10	DataSet10	== "A"	$ds : DataSet10(id == "A")		J	J	J	J	J	J	N	J	J	J	J	H	H	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 11	DataSet11	!= "A"	$ds : DataSet11(id != "A")		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 12	DataSet12		$ds : DataSet12()		J	J	J	J	J	J	N	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 13	DataSet13		$ds : DataSet13()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 14	DataSet14		$ds : DataSet14()		J	J	J	J	J	J	N	J	J	J	N	H	H	J	J	J	J	J	J	J	J	J	N	N	J	N	N	J	J	N	N	N	N		
Rule 15	DataSet15		$ds : DataSet15()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	N	N	N	N	J	N	N	N	J	J	J	J	N		
Rule 16	DataSet16		$ds : DataSet16()		J	J	J	J	J	J	N	J	J	J	J	H	H	J	J	J	J	J	J	J	J	J	N	N	J	N	N	J	J	J	J	N	N		
Rule 17	DataSet17		$ds : DataSet17()		J	J	J	J	J	J	N	J	J	J	N	H	H	J	J	J	J	J	J	J	N	N	N	N	J	J	N	N	J	N	N	N	N		
Rule 18	DataSet18	!= "B"	$ds : DataSet18(id != "B")		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	N	N	J	N	N	J	J	J	J	J	N		
Rule 19	DataSet19	== "B"	$ds : DataSet19(id == "B")		J	J	J	J	N	J	N	N	N	N	N	N	N	J	N	N	N	J	J	N	J	J	N	N	J	N	N	J	N	N	N	N	N		
Rule 20	DataSet20		$ds : DataSet20()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	N	N	J	N	N	J	J	J	J	J	N		
Rule 21	DataSet21		$ds : DataSet21()		J	J	J	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J	J	J	J	N	N	J	N	N	J	J	N	N	N	N		
Rule 22	DataSet22		$ds : DataSet22()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 23	DataSet23		$ds : DataSet23()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 24	DataSet24		$ds : DataSet24()		H	H	H	H	J	H	H	H	H	H	H	H	H	H	H	H	H	J	J	J	H	H	N	N	H	N	N	H	H	N	N	N	N		
Rule 25	DataSet25		$ds : DataSet25()	X	J	J	J	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	N	N	N	N	X	X
Rule 26	DataSet26		$ds : DataSet26()		J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	J	J	J	J		
Rule 27	DataSet27		$ds : DataSet27()		J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J		
Rule 28	DataSet28		$ds : DataSet28()		J	J	J	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	N	N	N	N		
Rule 29	DataSet29		$ds : DataSet29()		J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	J	J	J	J		
Rule 30	DataSet30		$ds : DataSet30()		J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J		
Rule 31	DataSet31		$ds : DataSet31()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 32	DataSet32		$ds : DataSet32()		J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	N		
Rule 33	DataSet33		$ds : DataSet33()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 34	DataSet34		$ds : DataSet34()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	J	J	N	N		
Rule 35	DataSet35		$ds : DataSet35()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	N	N	N	N	J	N	N	N	J	J	J	J	N		
Rule 36	DataSet36		$ds : DataSet36()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
Rule 37	DataSet37		$ds : DataSet37()		J	J	J	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	N	N	N	N		
Rule 38	DataSet38		$ds : DataSet38()		J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J		
Rule 39	DataSet39		$ds : DataSet39()		J	J	J	J	J	J	N	N	N	N	N	N	N	N	J	N	N	J	J	J	J	J	J	N	J	J	N	J	N	N	N	N	N		
Rule 40	DataSet40		$ds : DataSet40()		J	J	J	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	N	N	N	N		
Rule 41	DataSet41		$ds : DataSet41()		J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J		
Rule 42	DataSet42		$ds : DataSet42()		J	J	J	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J	J	J	J	N	N	J	N	N	J	J	N	N	N	N		
Rule 43	DataSet43		$ds : DataSet43()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	N	N	J	N	N	J	J	J	J	J	N		
Rule 44	DataSet44		$ds : DataSet44()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	N	N	J	N	N	J	J	J	J	J	N		
Rule 45	DataSet45		$ds : DataSet45()		J	J	J	J	J	J	N	N	N	N	N	N	N	N	J	N	N	J	J	J	J	J	J	N	J	J	N	J	N	N	N	N	N		
Rule 46	DataSet46		$ds : DataSet46()		J	J	J	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J	J	J	J	J	N	J	J	N	J	J	N	N	N	N		
Rule 47	DataSet47		$ds : DataSet47()		J	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	J	J	N	N	N	N	N	N	N	N	N	J	N	N	N	N		
Rule 48	DataSet48		$ds : DataSet48()		J	J	J	J	J	J	N	J	J	J	N	H	H	J	J	J	J	J	J	J	J	J	N	N	J	N	N	J	J	J	J	N	N		
Rule 49	DataSet49		$ds : DataSet49()		J	J	J	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J	J	J	J	N	J	N	N	J	J	J	J	J	N		

distinguish the request by one of its attributes and, depending on this attribute, the visibility is set, which happens in the Columns 39 and 40. The entries ‘J’ (green) define complete visibility, the entries ‘H’ (yellow) correspond to partial visibility, and the entries ‘N’ (magenta) translate to invisibility.

At this point is where the question arises: why the RHS of rules is used to further distinguish the request object as this could and should already be done on the LHS of rules? Especially, since it is considered bad practice to use conditional code in the RHS of a rule (see [10, p. 282]). The answer to this question is related to certain requirements of the customer of Capgemini. A design of the decision table, which is more conform to the philosophy of Drools, would lead to a different structure; and the current structure of the table gives a good overview of the relation between certain types of data and requests. This is a desired property which plays an important role in the communication process between Capgemini and their customer. Albeit, for our investigation it is necessary to shift these conditional constructs from the RHS to the LHS of the rules.

Listing 5.1: Overview of the structure of Rule 17

```
package com.capgemini.rulebase;

...

import com.capgemini.model.DataSet17;
import com.capgemini.model.Request;

...

rule "Rule 17"
  when
    Request($id : id, $senderGroup : senderGroup) and $ds : DataSet17()
  then
    String action="%";
    ...
    if("10".equals($senderGroup)) {
      action = "J";
    }
    if("11".equals($senderGroup)) {
      action = "N";
    }
    if("12".equals($senderGroup)) {
      action = "H";
    }
    ...
    if(action.equals("N") && !"Invisible".equals($ds.getVisibility())) {
      $ds.setVisibility("Invisible");
      update($ds);
    }
    if(action.equals("H") && !"Partial".equals($ds.getVisibility())) {
      $ds.setVisibility("Partial");
      upate($ds);
    }
end

...
```

The DRL representation of the selected decision table forms the basis for our investigation. It has 5686 lines of code and each rule is defined through 115 lines of code. Since we are required to manually prepare this code for further analysis, we limit ourselves to three selected rules, namely Rule 16, 17, and 18. Listing 5.1 shows essential parts of the DRL defining Rule 17 and gives the reader an overview of the considered DRL. The complete DRL representation of Rule 16, 17 and 18 can be found in the appendix in Listing A.1.

Keep in mind, that the LHS of these rules is relative sparse and only used to bind certain attributes of the request and the data object. The actual workload of the rule happens on the RHS. A local variable `action` is defined, whose values is set to either `"J"`, `"H"`, or `"N"`, depending on the attribute `senderGroup` of the request object. The values `"J"`, `"H"`, and `"N"` correspond to the respective entries in the decision table. Depending on the value of the variable `action`, the visibility of the data object is modified. It is also noteworthy that the visibility is only modified when the value of `action` is `"H"` or `"N"`, since the decision table relies on the assumption that all data objects are initially marked as visible. Therefore, in many cases the considered rules do not modify the working memory at all.

These rules are not in the required form that we discussed in Chapter 3. To facilitate an analysis regardless of this situation, a series of translation steps is necessary which is described in the next section.

5.2. Preparations

In this section we describe the preparations and translation steps which are necessary to analyze the previously selected rules. Most of these translations could be easily automated and are based solely on the information found in the considered DRL. However, like already mentioned, to function properly the CRB relies on certain assumptions about the working memory. To produce relevant results, it is necessary to incorporate these assumptions in our preparations.

The order in which we execute the translation steps is more or less arbitrary and many steps are interchangeable. The order we chose should promote a neat presentation and is maybe not optimal when one wants to automate the described process, since then the performance is more of an issue.

In the first translation step we eliminate the intermediate variable `action`, which is used to store results of comparisons of the attribute binding `$senderGroup` to certain string literals. Depending on these comparisons the working memory is modified or not touched at all. It is clear how to contract the related `if` statements to make the variable `action` dispensable. In this process, we also eliminate conditional constructs which do not lead to a modification of the working memory. Listing 5.2 shows this translation step for the previously considered parts of Rule 17.

Next, we make the RB a self-contained object by replacing the `import` statements with appropriate type declarations. Listing 5.3 shows an example of such type declarations for Rule 17 from Listing 5.2. In this translation step we also introduce shortcuts for

Listing 5.2: Elimination of variable `action` in the RHS of Rule 17

```
rule "Rule 17"
  when
    Request($id : id, $senderGroup : senderGroup) and $ds : DataSet17()
  then
    ...
    if("11".equals($senderGroup) && !"Invisible".equals($ds.getVisibility())) {
      $ds.setVisibility("Invisible");
      update($ds);
    }
    if("12".equals($senderGroup)) && !"Partial".equals($ds.getVisibility())) {
      $ds.setVisibility("Partial");
      upate($ds);
    }
    ...
end
```

attribute and type names to facilitate a more compact presentation. The creation of the introduced type declarations is a simple process in which one only needs to list all attributes of each type to which the RB refers. If one has access to the Java classes which are referenced via the `import` statements, it would also be feasible to use the same Java introspection features as Drools. This means one basically mimics the process described at the end of Section 2.1.

After changing the type declarations we need to adjust the rules, since we modified the value type of most attributes. This leads us to our next translation step. In the considered RB the value type of most attributes is `String`. In general, strings and integers behave quite differently and are not interchangeable, since they have different genuine operators with different semantics. However, in our RB we do not use many of these operators. We only test the equality, respectively, inequality of attributes and string literals. Then, we set the value of attributes to certain string literals. This is typical for most rules in the CRB and many other Drools RBs. In such case, a transition between integers and strings which preserves the core of the semantics of each rule is canonical. One simply needs to create an index of all distinct string literals in the considered RB and make appropriate replacements of string literals and related operators.

We assign well distinguished integer literals to each string literal. These integer literals are then used to replace the string literals in the RB. In the same step we replace the string

Listing 5.3: Appropriate type declarations for Rule 17

```
declare D17
  vi : Integer
end

declare R
  id : Integer
  sg : Integer
end
```

Listing 5.4: Replacement of string literals in Rule 17

```
rule Rule17
  when
    R($id : id, $senderGroup : sg) and $ds : D17($visibility : vi)
  then
    ...
    if($senderGroup == 11 && $visibility != 100) {
      modify($ds) {
        setVi(100)
      }
    }
    if($senderGroup == 12 && $visibility != 101) {
      modify($ds) {
        setVi(101)
      }
    }
    ...
end
```

operator `String.equals(String s)` and its negation with the integer operators `==` respectively `!=`. Listing 5.4 shows this translation step for the parts of Rule 17 which are shown in Listing 5.2. We choose to replace string literals like `"10"` through the obvious integer literals and use the values `100` and `101` to represent the literals `"Invisible"` respectively `"Partial"`. Other preparations shown in Listing 5.4 are the adjustment of the rule identifier and the use of the Drools statement `modify` instead of the `update` statement.

Afterwards, we begin with the deconstruction of the conditional constructs in the RHS of the rules. The remaining conditional constructs compare constant values with variables which are bound to attributes. It is obvious, how to shift one of those `if` statements

Listing 5.5: Splitting and shift of conditional constructs to LHS of Rule 17

```
...
rule Rule17G11N
  when
    R(sg == 11) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule17G12H
  when
    R(sg == 12) and $ds : D17(vi != 101)
  then
    modify($ds) {
      setVi(101)
    }
end
...
```

to the LHS of the rule. In order to incorporate all statements we need to create a copy of Rule 17 for each remaining `if` statement whose RHS contains only the respective conditional construct. Then we can shift the conditions in the RHS to the LHS of the newly created copies of Rule 17. Listing 5.5 shows this translation step.

The RB, which we obtain after the last translation step, lies in the fragment $\mathrm{DRL}^t_{\mathbb{Z}}$ and we can analyze it with our implementation. This analysis yields that the CRB does not terminate for an arbitrary working memory. One can already see this by looking at the code in Listing 5.5. Suppose, we have two request objects in working memory so that their attribute `sg` has the value `11`, respectively `12`. Next, assume the working memory contains at least one data object of type `D17`. In this case, Drools would repeatedly combine the data object with each request object and alter the visibility of the data object back and forth in an infinite loop.

While it is certainly and interesting result that we can expose the non-termination of the CRB for an arbitrary working memory, we already knew this before and want to capture the intended functionality of the CRB. In order to do this, we need to incorporate the aforementioned assumptions about the working memory into the DRL of the rule base. Our solution approach is to encode data and request as a single object. This preparation step is shown in Listing 5.6.

Let us take a closer look at what we just did here. In the context of the CRB, we know

Listing 5.6: Merge of request and data objects for Rule 17

```
...

declare DR17
  id : Integer
  sg : Integer
  vi : Integer
end

...

rule Rule17G11N
  when
    $dr : DR17(sg == 11, vi != 100)
  then
    modify($rd) {
      setVi(100)
    }
end

rule Rule17G12H
  when
    $dr : DR17(sg == 12, vi != 101)
  then
    modify($dr) {
      setVi(101)
    }
end

...
```

that the working memory should contain exactly one object of type `R`. We can express this more formally using the terms of Chapter 3. We should only consider abstract working memories $\mathcal{W}$ for which the following statement is true:

$$\forall o_1, o_2 \in O(\mathtt{R} = \Gamma(o_1) = \Gamma(o_2) \rightarrow o_1 = o_2) \tag{5.1}$$

Consequently, we know that LHSs like, for example, defined in Line 4 and 13 of Listing 5.5 can never simultaneously produce a match. Furthermore, we know that the CRB does not modify the request object. Thus, the pairings between objects of type `R` and `D17` matched by the DRL operator `and` are basically pairings between constant integer values and the values of the data object. Accordingly, the behavior of the CRB would not change if these constant integer values were provided as attributes of the data objects, which therefore justifies our last preparation step.

This explanation also shows that the automatization of the last step of our preparations is not a trivial task. On the one hand, we require access to formal specifications, which model the restrictions to the working memory, like exemplified in (5.1). On the other hand, we need to analyze the complete RB to confirm certain properties, like in our case that the request object is not modified.

The DRL$_\mathbb{Z}$ representation of Rule 16, 17, and 18 except the last preparation step can be found in Listing A.2. The representation which incorporates the last preparation step can be found in Listing A.3. We prepared a version of Listing A.3 which is used to demonstrate that we can identify certain error scenarios. We assume there is a typo in the rule template of the considered decision table. Here the value of the cell containing `if("11".equals($senderGroup)) { action = "$param" }` should be changed to `if("12".equals($senderGroup)) { action = "$param" }`. If we would carry out the same preparations as before for this defective version of the decision table, the typo would propagate to a change of Line 13 of Listing 5.6, respectively Line 111 of Listing A.3. In these lines the integer literal `11` would be replaced with `12`.

5.3. Results

In this section, we present the results of the analysis of the CRB based on the preparations made in the previous section. We show excerpts of the ITRSs that were generated using our implementation and discuss the related evaluations of AProVE.

Listing 5.7 shows the ITRS which results from the rules given in Listing 5.5. The complete ITRS for the Rule 16, 17, and 18 in that respective preparation step can be

Listing 5.7: Excerpt of the ITRS in Listing A.4

```
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(101) [vi > 101]
D17(vi) -> D17(101) [vi < 101]
```

found in Listing A.4. It is quite obvious that this ITRS does not terminate and AProVE has no trouble to generate the related proof which can be found in Listing A.6.

In this case, the termination criterion from Section 3.4 cannot guarantee the termination of the CRB for an arbitrary working memory. And indeed, we already discussed in the last section a concrete working memory for which the CRB does not terminate. We also discussed the assumptions about the working memory on which the CRB relies that are related to this result. In the case of the CRB, we already knew about these assumptions before and found a way to incorporate them properly. However, this might not generally be the case. This result shows that our implementation can help to reveal certain assumptions about the working memory on which a RB relies.

Listing 5.8 presents the ITRS which results from the rules in Listing 5.6. The complete ITRS can be found in Listing A.5. Listing 5.9 shows an excerpt of the related AProVE report. The complete AProVE report can be found in Listing A.7. This report gives us an example of a successful proof of termination of the related ITRS; thus we can apply our termination criterion and know that the termination of Rule 16, 17, and 18 from the investigated decision table of the CRB is guaranteed if one provides a working memory which respects the requirements of CRB.

Finally, we show that our implementation can theoretically help to detect certain error scenarios. Assume the typo described at the end of the last section. This typo would propagate to a change in Line 1 and 2 of Listing 5.8 respectively Line 21 and 22 of Listing A.5. These lines would contain the value `12` instead of `11`. Listing 5.10 shows an excerpt of the AProVE report for this scenario. The complete AProVE report can be found in Listing A.8.

Listing 5.8: Excerpt of the ITRS in Listing A.5

```
1  DR17(id, sg, vi) ->  DR17(id, sg, 100) [sg >= 11 && sg <= 11 && vi < 100]
2  DR17(id, sg, vi) ->  DR17(id, sg, 100) [sg >= 11 && sg <= 11 && vi > 100]
3  DR17(id, sg, vi) ->  DR17(id, sg, 101) [sg >= 12 && sg <= 12 && vi < 101]
4  DR17(id, sg, vi) ->  DR17(id, sg, 101) [sg >= 12 && sg <= 12 && vi > 101]
```

Listing 5.9: Excerpt of the AProVE report in Listing A.7

```
YES
proof of crb-2.inttrs
# AProVE Commit ID: 2e6638c59cfd6c865410a35d3360fc0074b41f84 ffrohn 20140725


Termination of the given IRSwT could be proven:

(0) IRSwT
(1) IRSwTTerminationDigraphProof [EQUIVALENT, 56.9 s]
(2) TRUE


...
```

Listing 5.10: Excerpt of the AProVE report in Listing A.8

```
1  NO
2  proof of crb-3.inttrs
3  # AProVE Commit ID: 2e6638c59cfd6c865410a35d3360fc0074b41f84 ffrohn 20140725
4
5
6  Termination of the given IRSwT could be disproven:
7
8  (0) IRSwT
9  (1) IRSwTTerminationDigraphProof [EQUIVALENT, 56.6 s]
10 (2) IRSwT
11 (3) IntTRSUnneededArgumentFilterProof [EQUIVALENT, 0 ms]
12 (4) IntTRS
13 (5) FilterProof [EQUIVALENT, 0 ms]
14 (6) IntTRS
15 (7) IntTRSPeriodicNontermProof [COMPLETE, 11 ms]
16 (8) NO
17
18 ...
19
20 (6)
21 Obligation:
22 Rules:
23 DR17(x58, x59) -> DR17(x58, 100) :|: x58 >= 12 && x58 <= 12 && x59 > 100
24 DR17(x67, x68) -> DR17(x67, 101) :|: x67 >= 12 && x67 <= 12 && x68 < 101
25
26 ----------------------------------------
27
28 (7) IntTRSPeriodicNontermProof (COMPLETE)
29 Normalized system to the following form:
30 f(pc, x58, x59) -> f(1, x58, 100) :|: pc = 1 && (x58 >= 12 && x58 <= 12 && x59 > 100)
31 f(pc, x67, x68) -> f(1, x67, 101) :|: pc = 1 && (x67 >= 12 && x67 <= 12 && x68 < 101)
32 Witness term starting non-terminating reduction: f(1, 12, 101)
33
34 ...
```

We see that our typo would lead to a non terminating ITRS. This result is a strong indicator of some kind of error in the considered RB, especially, if a previous analysis was able to guarantee the termination of this RB. A closer look at Listing 5.10 reveals a witness for a non-terminating reduction in Line 32 and the related rules in Line 23 and 24. This information could be used to trace back the typo in the original decision table.

5.4. Benchmarks

In this section we present some benchmarks created during our case study. These benchmarks were created on a personal computer with an Intel® Core™ 2 Quad processor running at 2.8 GHz frequency and 3.7 GiB working memory.

The runtime of our implementation, while generating the ITRSs presented in the previous section, was measured using the Linux command `time`. The result of this measurement can be found in Figure 5.3. This benchmark shows that the runtime of our implementation is almost neglectable and unproblematic.

Figure 5.3.: Benchmark of the runtime of the implementation

	Real	User	Sys
Generation of Listing A.4	0m1.415s	0m1.817s	0m0.040s
Generation of Listing A.5	0m1.414s	0m1.860s	0m0.033s

Figure 5.4.: Benchmark of the runtime of AProVE

	Time
Generation of Listing A.6	0m0.140s
Generation of Listing A.7	0m56.9s
Generation of Listing A.8	0m56.6s

Figure 5.5.: Benchmark of the runtime of AProVE – Different sample sizes

	Time
Input Line 1 - 10 of Listing A.6	0m0.403s
Input Line 1 - 20 of Listing A.6	0m1.083s
Input Line 1 - 30 of Listing A.6	0m2.248s
Input Line 1 - 40 of Listing A.6	0m5.119s
Input Line 1 - 50 of Listing A.6	0m9.906s
Input Line 1 - 60 of Listing A.6	0m20.3s
Input Line 1 - 70 of Listing A.6	0m36.5s
Input Line 1 - 78 of Listing A.6	0m56.6s

The reports of AProVE provide their own time measurement and Figure 5.4 summarizes the relevant data. Since the runtime of AProVE is significant, we conducted another test in which we tested the runtime of AProVE for differently sized ITRSs. The first sample contained the first 10 lines of Listing A.5; the second sample – the first 20 lines, and so on. Figure 5.5 shows the generated results. We can see the asymptotic exponential increase of the runtime which is typical for many fields of automated theorem proving.

However, the ITRS generated by our implementation could be optimized and one can eliminate certain redundancies, which would lead to better performance. This can be achieved using approaches like symmetric reduction or subsumption.

6. Conclusion

In this thesis we combined both theoretical and practical approaches towards the goal of an automated deductive analysis of the Business Rule Management System Drools. Furthermore, our case study showed that these approaches are applicable in real-world scenarios and yield useful results for the development process of Drools rule bases.

A central aspect of the theoretical work presented in Chapter 3 is the definition of a formalism, which allows us to capture the internal structure of the inference engine of Drools and give structural operational semantics to certain expressions in DRL. Nevertheless, Drools and DRL undergo rapid development; and DRL can not be considered a stable language. During the creation of this thesis we witnessed four stable releases of Drools, namely version 5.5, 5.6, 6.0, and 6.1. At the moment of publication version 6.2 is another release candidate. Most of these versions brought minor changes to syntax and semantics of DRL or introduced new language features. For example, an interesting new feature are so-called *fine grained property change listeners*, which has direct impact on the semantics of DRL. This feature introduces new requirements to the classes used for the representation of facts and needs to be explicitly activated. In this case, Drools will only reevaluate facts, if the value of an attribute is actually changed by a modification. This suppresses the repeated modification of facts, which might not change the values of attributes. Therefore, it is crucial to keep track with the current development of Drools.

Next, in Chapter 3 we presented and discussed the termination criterion. To be useful in practice, it was necessary to put side conditions on allowed working memories. The general nature of these side conditions is specific to applications. In our case study we incorporated them directly into the translation process. For a more general approach, it remains to formalize these side conditions as e.g. logical formulas and to apply theorem provers or solvers to support the termination analyzer.

The implementation presented in Chapter 4 is the first prototype which allows the automated extraction of integer term rewriting systems from certain Drools rule bases. It demonstrates the practical accessibility of Drools and DRL for formal software verification approaches and produces useful results in combination with AProVE. However, if one is interested in the goal of a fully automated analysis of Drools rule bases, which are used in productive environments, further development is needed.

In Chapter 5 we have shown how to bridge the gap between our theoretical considerations and problems which occur in productive environments. In this process we encountered and described certain obstacles. Hereby, we have shown that these are not fundamental in nature and can be overcome with adequate efforts. Furthermore, we have shown that our approaches lead to results with practical relevance.

Overall we presented the proof-of-concept for the automated deductive analysis of business rules in Drools.

Bibliography

[1] Apache Commons CLI Team — The Apache Software Foundation. Apache Commons CLI, 2014. `http://commons.apache.org/proper/commons-cli/`.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[4] Mike Brock et al. MVEL Language Guide for 2.0, 2014. `http://mvel.codehaus.org/Language+Guide+for+2.0`.

[5] Business Rules Group. Defining Business Rules ∼ What Are They Really?, 2000. `http://www.businessrulesgroup.org/first_paper/BRG-whatisBR_3ed.pdf`.

[6] Charles L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17 – 37, 1982.

[7] Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. Proving termination of integer term rewriting. In *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil*, volume 5595 of *Lecture Notes in Computer Science*, pages 32 – 47. Springer, 2009.

[8] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated Termination Proofs with AProVE. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany*, volume 3091 of *Lecture Notes in Computer Science*, pages 210 – 220. Springer, 2004.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, 3rd edition, 2005.

[10] JBoss Drools Team. Drools Documentation, Version 6.1.0.Final, 2014. `http://docs.jboss.org/drools/release/6.1.0.Final/drools-docs/pdf/drools-docs.pdf`.

[11] Wikipedia. Syntax diagram — Wikipedia, The Free Encyclopedia, 2014. `http://en.wikipedia.org/w/index.php?title=Syntax_diagram&oldid=627859901`, accessed 2014-10-07.

Figures

Listings

Appendices

A. Investigated Rules and Related Data

This appendix contains data, rules, and results, which form the basis for the case study presented in Chapter 5. The first section shows a detailed version of the investigated decision table. The next section exhibits the DRL and $\mathrm{DRL}_{\mathbb{Z}}$ representation of Rule 16, 17, and 18 from this decision table. The third section gives the ITRSs which were generated using our implementation. The last section portrays the related AProVE results.

A.1. Investigated Decision Table

Figure A.1, A.2, and A.3 show Column A to H, Column I to X, respectively, Column Y to AN of the investigated decision table.

A.2. Investigated Rules

Listing A.1 contains the DRL representation of Rule 16, 17, and 18 of the decision table presented in the previous section. Listing A.2 shows the $\mathrm{DRL}_{\mathbb{Z}}$ representation of Rule 16, 17, and 18 which results from Listing A.1 after the first four translation steps described in Section 5.2. Listing A.3 presents the $\mathrm{DRL}_{\mathbb{Z}}$ representation of Rule 16, 17, and 18 which results from Listing A.1 after all translation steps described in Section 5.2.

Listing A.1: DRL representation of Rule 16, 17, and 18

```
package com.capgemini.rulebase;

import com.capgemini.model.DataSet16;
import com.capgemini.model.DataSet17;
import com.capgemini.model.DataSet18;
import com.capgemini.model.Request;

rule "Rule 16"
  when
    Request($senderGroup : senderGroup) and $ds : DataSet16()
  then
    String action="%";
    if("01".equals($senderGroup)) {
      action = "J";
    }
    if("02".equals($senderGroup)) {
      action = "J";
    }
    if("03".equals($senderGroup)) {
      action = "J";
    }
    if("04".equals($senderGroup)) {
```

Figure A.1.: Investigated decision table – Column A to H

	A	B	C	D	E	F	G	H
1		CONDITION	CONDITION	CONDITION	ACTION	ACTION	ACTION	ACTION
2				Request($senderGroup : senderGroup) and $param	String action="%";	if("01".equals($senderGroup)) { action = "$param"; }	if("02".equals($senderGroup)) { action = "$param"; }	if("03".equals($senderGroup)) { action = "$param"; }
3	**Rule Name**	**Data Set**	**ID**	**LHS**		**1**	**2**	**3**
4	**Rule 1**	DataSet1		$ds : DataSet1()		J	J	J
5	**Rule 2**	DataSet2		$ds : DataSet2()		J	J	J
6	**Rule 3**	DataSet3		$ds : DataSet3()		J	J	J
7	**Rule 4**	DataSet4		$ds : DataSet4()		J	J	J
8	**Rule 5**	DataSet5		$ds : DataSet5()		J	J	J
9	**Rule 6**	DataSet6		$ds : DataSet6()		J	J	J
10	**Rule 7**	DataSet7		$ds : DataSet7()		J	J	J
11	**Rule 8**	DataSet8		$ds : DataSet8()		J	J	J
12	**Rule 9**	DataSet9	== "A"	$ds : DataSet9(id == "A")		J	J	J
13	**Rule 10**	DataSet10	== "A"	$ds : DataSet10(id == "A")		J	J	J
14	**Rule 11**	DataSet11	!= "A"	$ds : DataSet11(id != "A")		J	J	J
15	**Rule 12**	DataSet12		$ds : DataSet12()		J	J	J
16	**Rule 13**	DataSet13		$ds : DataSet13()		J	J	J
17	**Rule 14**	DataSet14		$ds : DataSet14()		J	J	J
18	**Rule 15**	DataSet15		$ds : DataSet15()		J	J	J
19	**Rule 16**	DataSet16		$ds : DataSet16()		J	J	J
20	**Rule 17**	DataSet17		$ds : DataSet17()		J	J	J
21	**Rule 18**	DataSet18	!= "B"	$ds : DataSet18(id != "B")		J	J	J
22	**Rule 19**	DataSet19	== "B"	$ds : DataSet19(id == "B")		J	J	J
23	**Rule 20**	DataSet20		$ds : DataSet20()		J	J	J
24	**Rule 21**	DataSet21		$ds : DataSet21()		J	J	J
25	**Rule 22**	DataSet22		$ds : DataSet22()		J	J	J
26	**Rule 23**	DataSet23		$ds : DataSet23()		J	J	J
27	**Rule 24**	DataSet24		$ds : DataSet24()		H	H	H
28	**Rule 25**	DataSet25		$ds : DataSet25()	X	J	J	J
29	**Rule 26**	DataSet26		$ds : DataSet26()		J	J	J
30	**Rule 27**	DataSet27		$ds : DataSet27()		J	J	J
31	**Rule 28**	DataSet28		$ds : DataSet28()		J	J	J
32	**Rule 29**	DataSet29		$ds : DataSet29()		J	J	J
33	**Rule 30**	DataSet30		$ds : DataSet30()		J	J	J
34	**Rule 31**	DataSet31		$ds : DataSet31()		J	J	J
35	**Rule 32**	DataSet32		$ds : DataSet32()		J	J	J
36	**Rule 33**	DataSet33		$ds : DataSet33()		J	J	J
37	**Rule 34**	DataSet34		$ds : DataSet34()		J	J	J
38	**Rule 35**	DataSet35		$ds : DataSet35()		J	J	J
39	**Rule 36**	DataSet36		$ds : DataSet36()		J	J	J
40	**Rule 37**	DataSet37		$ds : DataSet37()		J	J	J
41	**Rule 38**	DataSet38		$ds : DataSet38()		J	J	J
42	**Rule 39**	DataSet39		$ds : DataSet39()		J	J	J
43	**Rule 40**	DataSet40		$ds : DataSet40()		J	J	J
44	**Rule 41**	DataSet41		$ds : DataSet41()		J	J	J
45	**Rule 42**	DataSet42		$ds : DataSet42()		J	J	J
46	**Rule 43**	DataSet43		$ds : DataSet43()		J	J	J
47	**Rule 44**	DataSet44		$ds : DataSet44()		J	J	J
48	**Rule 45**	DataSet45		$ds : DataSet45()		J	J	J
49	**Rule 46**	DataSet46		$ds : DataSet46()		J	J	J
50	**Rule 47**	DataSet47		$ds : DataSet47()		J	N	N
51	**Rule 48**	DataSet48		$ds : DataSet48()		J	J	J
52	**Rule 49**	DataSet49		$ds : DataSet49()		J	J	J

Figure A.2.: Investigated decision table – Column I to X

	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
1	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION
2	if("04".equals($senderGroup)) { action = "$param"; }	if("05".equals($senderGroup)) { action = "$param"; }	if("06".equals($senderGroup)) { action = "$param"; }	if("07".equals($senderGroup)) { action = "$param"; }	if("08".equals($senderGroup)) { action = "$param"; }	if("09".equals($senderGroup)) { action = "$param"; }	if("10".equals($senderGroup)) { action = "$param"; }	if("11".equals($senderGroup)) { action = "$param"; }	if("12".equals($senderGroup)) { action = "$param"; }	if("13".equals($senderGroup)) { action = "$param"; }	if("14".equals($senderGroup)) { action = "$param"; }	if("15".equals($senderGroup)) { action = "$param"; }	if("16".equals($senderGroup)) { action = "$param"; }	if("17".equals($senderGroup)) { action = "$param"; }	if("18".equals($senderGroup)) { action = "$param"; }	if("19".equals($senderGroup)) { action = "$param"; }
3	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
4	J	J	J	N	J	J	J	N	H	H	J	J	J	J	J	J
5	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
6	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J
7	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J
8	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J
9	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J
10	J	J	J	N	J	J	J	J	J	J	J	J	J	J	J	J
11	J	J	J	N	J	J	J	J	H	H	J	J	J	J	J	J
12	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
13	J	J	J	N	J	J	J	J	H	H	J	J	J	J	J	J
14	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
15	J	J	J	N	J	J	J	J	J	J	J	J	J	J	J	J
16	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
17	J	J	J	N	J	J	J	N	H	H	J	J	J	J	J	J
18	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
19	J	J	J	N	J	J	J	J	H	H	J	J	J	J	J	J
20	J	J	J	N	J	J	J	N	H	H	J	J	J	J	J	J
21	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
22	J	N	J	N	N	N	N	N	N	N	J	N	N	N	J	J
23	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
24	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J
25	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
26	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
27	H	J	H	H	H	H	H	H	H	H	H	H	H	H	J	J
28	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J
29	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J
30	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J
31	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J
32	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J
33	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J
34	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
35	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J
36	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
37	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
38	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
39	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
40	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J
41	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J
42	J	J	J	N	N	N	N	N	N	N	N	J	N	N	J	J
43	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J
44	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J
45	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J
46	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
47	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J
48	J	J	J	N	N	N	N	N	N	N	N	J	N	N	J	J
49	J	J	J	N	J	J	J	N	N	N	J	J	J	J	J	J
50	N	N	N	N	N	N	N	N	N	N	N	N	N	N	J	J
51	J	J	J	N	J	J	J	N	H	H	J	J	J	J	J	J
52	J	J	J	N	J	J	J	J	N	N	J	J	J	J	J	J

Figure A.3.: Investigated decision table – Column Y to AN

	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM	AN
1	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION	ACTION
2	if("20".equals($senderGroup)) { action = "$param"; }	if("21".equals($senderGroup)) { action = "$param"; }	if("22".equals($senderGroup)) { action = "$param"; }	if("23".equals($senderGroup)) { action = "$param"; }	if("24".equals($senderGroup)) { action = "$param"; }	if("25".equals($senderGroup)) { action = "$param"; }	if("26".equals($senderGroup)) { action = "$param"; }	if("27".equals($senderGroup)) { action = "$param"; }	if("28".equals($senderGroup)) { action = "$param"; }	if("29".equals($senderGroup)) { action = "$param"; }	if("30".equals($senderGroup)) { action = "$param"; }	if("31".equals($senderGroup)) { action = "$param"; }	if("32".equals($senderGroup)) { action = "$param"; }	if("33".equals($senderGroup)) { action = "$param"; }	if(action.equals("N") && !"0".equals($ds.getVisibility())) { $ds.setVisibility("0"); update($ds); }	if(action.equals("H") && !"1".equals($ds.getVisibility())) { $ds.setVisibility("1"); update($ds); }
3	20	21	22	23	24	25	26	27	28	29	30	31	32	33	N	H
4	J	N	N	N	N	J	N	N	N	J	N	N	N	N		
5	J	J	J	N	N	J	N	N	J	J	J	J	J	N		
6	J	J	J	J	N	J	J	N	J	J	J	J	N	N		
7	J	J	J	J	N	J	J	N	J	J	J	J	J	J		
8	J	J	J	J	N	J	J	N	J	J	J	J	N	N		
9	J	J	J	J	N	J	J	N	J	J	J	J	N	N		
10	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
11	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
12	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
13	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
14	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
15	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
16	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
17	J	J	J	N	N	J	N	N	J	J	N	N	N	N		
18	J	N	N	N	N	J	N	N	N	J	J	J	J	N		
19	J	J	J	N	N	J	N	N	J	J	J	J	N	N		
20	J	N	N	N	N	J	J	N	N	J	N	N	N	N		
21	J	J	J	N	N	J	N	N	J	J	J	J	J	N		
22	N	J	J	N	N	J	N	N	J	N	N	N	N	N		
23	J	J	J	N	N	J	N	N	J	J	J	J	J	N		
24	J	J	J	N	N	J	N	N	J	J	N	N	N	N		
25	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
26	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
27	J	H	H	N	N	H	N	N	H	H	N	N	N	N		
28	J	J	J	J	N	J	J	N	J	J	N	N	N	N	X	X
29	J	J	J	J	N	J	J	N	J	J	J	J	J	J		
30	J	J	J	J	J	J	J	J	J	J	J	J	J	J		
31	J	J	J	J	N	J	J	N	J	J	N	N	N	N		
32	J	J	J	J	N	J	J	N	J	J	J	J	J	J		
33	J	J	J	J	J	J	J	J	J	J	J	J	J	J		
34	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
35	J	J	J	J	J	J	J	J	J	J	J	J	J	N		
36	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
37	J	J	J	J	N	J	J	N	J	J	J	J	N	N		
38	J	N	N	N	N	J	N	N	N	J	J	J	J	N		
39	J	J	J	J	N	J	N	N	J	J	J	J	J	N		
40	J	J	J	J	N	J	J	N	J	J	N	N	N	N		
41	J	J	J	J	J	J	J	J	J	J	J	J	J	J		
42	J	J	J	J	N	J	J	N	J	N	N	N	N	N		
43	J	J	J	J	N	J	J	N	J	J	N	N	N	N		
44	J	J	J	J	J	J	J	J	J	J	J	J	J	J		
45	J	J	J	N	N	J	N	N	J	J	N	N	N	N		
46	J	J	J	N	N	J	N	N	J	J	J	J	J	N		
47	J	J	J	N	N	J	N	N	J	J	J	J	J	N		
48	J	J	J	J	N	J	J	N	J	N	N	N	N	N		
49	J	J	J	J	N	J	J	N	J	J	N	N	N	N		
50	N	N	N	N	N	N	N	N	N	J	N	N	N	N		
51	J	J	J	N	N	J	N	N	J	J	J	J	N	N		
52	J	J	J	J	N	J	N	N	J	J	J	J	J	N		

```
      action = "J";
    }
    if("05".equals($senderGroup)) {
      action = "J";
    }
    if("06".equals($senderGroup)) {
      action = "J";
    }
    if("07".equals($senderGroup)) {
      action = "N";
    }
    if("08".equals($senderGroup)) {
      action = "J";
    }
    if("09".equals($senderGroup)) {
      action = "J";
    }
    if("10".equals($senderGroup)) {
      action = "J";
    }
    if("11".equals($senderGroup)) {
      action = "J";
    }
    if("12".equals($senderGroup)) {
      action = "H";
    }
    if("13".equals($senderGroup)) {
      action = "H";
    }
    if("14".equals($senderGroup)) {
      action = "J";
    }
    if("15".equals($senderGroup)) {
      action = "J";
    }
    if("16".equals($senderGroup)) {
      action = "J";
    }
    if("17".equals($senderGroup)) {
      action = "J";
    }
    if("18".equals($senderGroup)) {
      action = "J";
    }
    if("19".equals($senderGroup)) {
      action = "J";
    }
    if("20".equals($senderGroup)) {
      action = "J";
    }
    if("21".equals($senderGroup)) {
      action = "J";
    }
    if("22".equals($senderGroup)) {
      action = "J";
    }
    if("23".equals($senderGroup)) {
      action = "N";
    }
    if("24".equals($senderGroup)) {
      action = "N";
    }
```

```
    if("25".equals($senderGroup)) {
      action = "J";
    }
    if("26".equals($senderGroup)) {
      action = "N";
    }
    if("27".equals($senderGroup)) {
      action = "N";
    }
    if("28".equals($senderGroup)) {
      action = "J";
    }
    if("29".equals($senderGroup)) {
      action = "J";
    }
    if("30".equals($senderGroup)) {
      action = "J";
    }
    if("31".equals($senderGroup)) {
      action = "J";
    }
    if("32".equals($senderGroup)) {
      action = "N";
    }
    if("33".equals($senderGroup)) {
      action = "N";
    }
    if(action.equals("N") && !"0".equals($ds.getVisibility())) {
      $ds.setVisibility("0");
      update($ds);
    }
    if(action.equals("H") && !"1".equals($ds.getVisibility())) {
      $ds.setVisibility("1");
      upate($ds);
    }
end

rule "Rule 17"
  when
    Request($senderGroup : senderGroup) and $ds : DataSet17()
  then
    String action="%";
    if("01".equals($senderGroup)) {
      action = "J";
    }
    if("02".equals($senderGroup)) {
      action = "J";
    }
    if("03".equals($senderGroup)) {
      action = "J";
    }
    if("04".equals($senderGroup)) {
      action = "J";
    }
    if("05".equals($senderGroup)) {
      action = "J";
    }
    if("06".equals($senderGroup)) {
      action = "J";
    }
    if("07".equals($senderGroup)) {
      action = "N";
```

```
    }
    if("08".equals($senderGroup)) {
      action = "J";
    }
    if("09".equals($senderGroup)) {
      action = "J";
    }
    if("10".equals($senderGroup)) {
      action = "J";
    }
    if("11".equals($senderGroup)) {
      action = "N";
    }
    if("12".equals($senderGroup)) {
      action = "H";
    }
    if("13".equals($senderGroup)) {
      action = "H";
    }
    if("14".equals($senderGroup)) {
      action = "J";
    }
    if("15".equals($senderGroup)) {
      action = "J";
    }
    if("16".equals($senderGroup)) {
      action = "J";
    }
    if("17".equals($senderGroup)) {
      action = "J";
    }
    if("18".equals($senderGroup)) {
      action = "J";
    }
    if("19".equals($senderGroup)) {
      action = "J";
    }
    if("20".equals($senderGroup)) {
      action = "J";
    }
    if("21".equals($senderGroup)) {
      action = "N";
    }
    if("22".equals($senderGroup)) {
      action = "N";
    }
    if("23".equals($senderGroup)) {
      action = "N";
    }
    if("24".equals($senderGroup)) {
      action = "N";
    }
    if("25".equals($senderGroup)) {
      action = "J";
    }
    if("26".equals($senderGroup)) {
      action = "J";
    }
    if("27".equals($senderGroup)) {
      action = "N";
    }
    if("28".equals($senderGroup)) {
```

```
        action = "N";
      }
      if("29".equals($senderGroup)) {
        action = "J";
      }
      if("30".equals($senderGroup)) {
        action = "N";
      }
      if("31".equals($senderGroup)) {
        action = "N";
      }
      if("32".equals($senderGroup)) {
        action = "N";
      }
      if("33".equals($senderGroup)) {
        action = "N";
      }
      if(action.equals("N") && !"0".equals($ds.getVisibility())) {
        $ds.setVisibility("0");
        update($ds);
      }
      if(action.equals("H") && !"1".equals($ds.getVisibility())) {
        $ds.setVisibility("1");
        upate($ds);
      }
end

rule "Rule 18"
  when
    Request($senderGroup : senderGroup) and $ds : DataSet18(id != "B")
  then
    String action="%";
    if("01".equals($senderGroup)) {
      action = "J";
    }
    if("02".equals($senderGroup)) {
      action = "J";
    }
    if("03".equals($senderGroup)) {
      action = "J";
    }
    if("04".equals($senderGroup)) {
      action = "J";
    }
    if("05".equals($senderGroup)) {
      action = "J";
    }
    if("06".equals($senderGroup)) {
      action = "J";
    }
    if("07".equals($senderGroup)) {
      action = "N";
    }
    if("08".equals($senderGroup)) {
      action = "J";
    }
    if("09".equals($senderGroup)) {
      action = "J";
    }
    if("10".equals($senderGroup)) {
      action = "J";
    }
```

```
    if("11".equals($senderGroup)) {
      action = "J";
    }
    if("12".equals($senderGroup)) {
      action = "N";
    }
    if("13".equals($senderGroup)) {
      action = "N";
    }
    if("14".equals($senderGroup)) {
      action = "J";
    }
    if("15".equals($senderGroup)) {
      action = "J";
    }
    if("16".equals($senderGroup)) {
      action = "J";
    }
    if("17".equals($senderGroup)) {
      action = "J";
    }
    if("18".equals($senderGroup)) {
      action = "J";
    }
    if("19".equals($senderGroup)) {
      action = "J";
    }
    if("20".equals($senderGroup)) {
      action = "J";
    }
    if("21".equals($senderGroup)) {
      action = "J";
    }
    if("22".equals($senderGroup)) {
      action = "J";
    }
    if("23".equals($senderGroup)) {
      action = "N";
    }
    if("24".equals($senderGroup)) {
      action = "N";
    }
    if("25".equals($senderGroup)) {
      action = "J";
    }
    if("26".equals($senderGroup)) {
      action = "N";
    }
    if("27".equals($senderGroup)) {
      action = "N";
    }
    if("28".equals($senderGroup)) {
      action = "J";
    }
    if("29".equals($senderGroup)) {
      action = "J";
    }
    if("30".equals($senderGroup)) {
      action = "J";
    }
    if("31".equals($senderGroup)) {
      action = "J";
```

```
    }
    if("32".equals($senderGroup)) {
      action = "J";
    }
    if("33".equals($senderGroup)) {
      action = "N";
    }
    if(action.equals("N") && !"0".equals($ds.getVisibility())) {
      $ds.setVisibility("0");
      update($ds);
    }
    if(action.equals("H") && !"1".equals($ds.getVisibility())) {
      $ds.setVisibility("1");
      upate($ds);
    }
end
```

Listing A.2: $\mathrm{DRL}_{\mathbb{Z}}$ representation of Rule 16, 17, and 18 — Preparation steps 1 to 4

```
declare R
  id : Integer
  sg : Integer
end

declare D16
  vi : Integer
end

declare D17
  vi : Integer
end

declare D18
  vi : Integer
end

rule Rule16B1
  when
    R(sg == 7) and $ds : D16(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule16B2
  when
    R(sg == 12) and $ds : D16(vi != 101)
  then
    modify($ds) {
      setVi(101)
    }
end

rule Rule16B3
  when
    R(sg == 13) and $ds : D16(vi != 101)
  then
    modify($ds) {
      setVi(101)
    }
```

```
end

rule Rule16B4
  when
    R(sg == 23) and $ds : D16(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule16B5
  when
    R(sg == 24) and $ds : D16(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule16B6
  when
    R(sg == 26) and $ds : D16(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule16B7
  when
    R(sg == 27) and $ds : D16(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule16B8
  when
    R(sg == 32) and $ds : D16(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule16B9
  when
    R(sg == 33) and $ds : D16(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule17B1
  when
    R(sg == 7) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
```

```
    }
end

rule Rule17B2
  when
    R(sg == 11) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule17B3
  when
    R(sg == 12) and $ds : D17(vi != 101)
  then
    modify($ds) {
      setVi(101)
    }
end

rule Rule17B4
  when
    R(sg == 13) and $ds : D17(vi != 101)
  then
    modify($ds) {
      setVi(101)
    }
end

rule Rule17B5
  when
    R(sg == 21) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule17B6
  when
    R(sg == 22) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule17B7
  when
    R(sg == 23) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule17B8
  when
    R(sg == 24) and $ds : D17(vi != 100)
  then
    modify($ds) {
```

```
      setVi(100)
    }
end

rule Rule17B9
  when
    R(sg == 27) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule17B10
  when
    R(sg == 28) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule17B11
  when
    R(sg == 30) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule17B12
  when
    R(sg == 31) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule17B13
  when
    R(sg == 32) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule17B14
  when
    R(sg == 33) and $ds : D17(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule18B1
  when
    R(id != 66, sg == 7) and $ds : D18(vi != 100)
  then
```

```
    modify($ds) {
      setVi(100)
    }
end

rule Rule18B2
  when
    R(id != 66, sg == 12) and $ds : D18(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule18B3
  when
    R(id != 66, sg == 13) and $ds : D18(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule18B4
  when
    R(id != 66, sg == 23) and $ds : D18(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule18B5
  when
    R(id != 66, sg == 24) and $ds : D18(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule18B6
  when
    R(id != 66, sg == 26) and $ds : D18(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule18B7
  when
    R(id != 66, sg == 27) and $ds : D18(vi != 100)
  then
    modify($ds) {
      setVi(100)
    }
end

rule Rule18B8
  when
    R(id != 66, sg == 33) and $ds : D18(vi != 100)
```

```
  then
    modify($ds) {
      setVi(100)
    }
end
```

Listing A.3: $\mathrm{DRL}_{\mathbb{Z}}$ representation of Rule 16, 17, and 18 — Preparation steps 1 to 5

```
declare DR16
  id : Integer
  sg : Integer
  vi : Integer
end

declare DR17
  id : Integer
  sg : Integer
  vi : Integer
end

declare DR18
  id : Integer
  sg : Integer
  vi : Integer
end

rule Rule16B1
  when
    $dr : DR16(sg == 7, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule16B2
  when
    $dr : DR16(sg == 12, vi != 101)
  then
    modify($dr) {
      setVi(101)
    }
end

rule Rule16B3
  when
    $dr : DR16(sg == 13, vi != 101)
  then
    modify($dr) {
      setVi(101)
    }
end

rule Rule16B4
  when
    $dr : DR16(sg == 23, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end
```

```

rule Rule16B5
  when
    $dr : DR16(sg == 24, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule16B6
  when
    $dr : DR16(sg == 26, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule16B7
  when
    $dr : DR16(sg == 27, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule16B8
  when
    $dr : DR16(sg == 32, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule16B9
  when
    $dr : DR16(sg == 33, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule17B1
  when
    $dr : DR17(sg == 7, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule17B2
  when
    $dr : DR17(sg == 11, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
```

```
end

rule Rule17B3
  when
    $dr : DR17(sg == 12, vi != 101)
  then
    modify($dr) {
      setVi(101)
    }
end

rule Rule17B4
  when
    $dr : DR17(sg == 13, vi != 101)
  then
    modify($dr) {
      setVi(101)
    }
end

rule Rule17B5
  when
    $dr : DR17(sg == 21, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule17B6
  when
    $dr : DR17(sg == 22, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule17B7
  when
    $dr : DR17(sg == 23, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule17B8
  when
    $dr : DR17(sg == 24, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule17B9
  when
    $dr : DR17(sg == 27, vi != 100)
  then
    modify($dr) {
      setVi(100)
```

```
    }
end

rule Rule17B10
  when
    $dr : DR17(sg == 28, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule17B11
  when
    $dr : DR17(sg == 30, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule17B12
  when
    $dr : DR17(sg == 31, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule17B13
  when
    $dr : DR17(sg == 32, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule17B14
  when
    $dr : DR17(sg == 33, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule18B1
  when
    $dr : DR18(id != 66, sg == 7, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule18B2
  when
    $dr : DR18(id != 66, sg == 12, vi != 100)
  then
    modify($dr) {
```

```
      setVi(100)
    }
end

rule Rule18B3
  when
    $dr : DR18(id != 66, sg == 13, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule18B4
  when
    $dr : DR18(id != 66, sg == 23, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule18B5
  when
    $dr : DR18(id != 66, sg == 24, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule18B6
  when
    $dr : DR18(id != 66, sg == 26, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule18B7
  when
    $dr : DR18(id != 66, sg == 27, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end

rule Rule18B8
  when
    $dr : DR18(id != 66, sg == 33, vi != 100)
  then
    modify($dr) {
      setVi(100)
    }
end
```

A.3. Integer Term Rewriting Systems

Listing A.4 contains the ITRS which is the output of our implementation when applied to Listing A.2. Listing A.5 show the output of our implementation when applied to Listing A.3.

Listing A.4: ITRS for Listing A.2

```
D16(vi) -> D16(100) [vi > 100]
D16(vi) -> D16(100) [vi < 100]
D16(vi) -> D16(101) [vi > 101]
D16(vi) -> D16(101) [vi < 101]
D16(vi) -> D16(101) [vi > 101]
D16(vi) -> D16(101) [vi < 101]
D16(vi) -> D16(100) [vi > 100]
D16(vi) -> D16(100) [vi < 100]
D16(vi) -> D16(100) [vi > 100]
D16(vi) -> D16(100) [vi < 100]
D16(vi) -> D16(100) [vi > 100]
D16(vi) -> D16(100) [vi < 100]
D16(vi) -> D16(100) [vi > 100]
D16(vi) -> D16(100) [vi < 100]
D16(vi) -> D16(100) [vi > 100]
D16(vi) -> D16(100) [vi < 100]
D16(vi) -> D16(100) [vi > 100]
D16(vi) -> D16(100) [vi < 100]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(101) [vi > 101]
D17(vi) -> D17(101) [vi < 101]
D17(vi) -> D17(101) [vi > 101]
D17(vi) -> D17(101) [vi < 101]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D17(vi) -> D17(100) [vi > 100]
D17(vi) -> D17(100) [vi < 100]
D18(vi) -> D18(100) [vi > 100]
D18(vi) -> D18(100) [vi < 100]
D18(vi) -> D18(100) [vi > 100]
D18(vi) -> D18(100) [vi < 100]
D18(vi) -> D18(100) [vi > 100]
D18(vi) -> D18(100) [vi < 100]
D18(vi) -> D18(100) [vi > 100]
```

```
D18(vi) -> D18(100) [vi < 100]
D18(vi) -> D18(100) [vi > 100]
D18(vi) -> D18(100) [vi < 100]
D18(vi) -> D18(100) [vi > 100]
D18(vi) -> D18(100) [vi < 100]
D18(vi) -> D18(100) [vi > 100]
D18(vi) -> D18(100) [vi < 100]
D18(vi) -> D18(100) [vi > 100]
D18(vi) -> D18(100) [vi < 100]
```

Listing A.5: ITRS for Listing A.3

```
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 7 && sg <= 7 && vi > 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 7 && sg <= 7 && vi < 100]
DR16(id, sg, vi) -> DR16(id, sg, 101) [sg >= 12 && sg <= 12 && vi > 101]
DR16(id, sg, vi) -> DR16(id, sg, 101) [sg >= 12 && sg <= 12 && vi < 101]
DR16(id, sg, vi) -> DR16(id, sg, 101) [sg >= 13 && sg <= 13 && vi > 101]
DR16(id, sg, vi) -> DR16(id, sg, 101) [sg >= 13 && sg <= 13 && vi < 101]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 23 && sg <= 23 && vi > 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 23 && sg <= 23 && vi < 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 24 && sg <= 24 && vi > 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 24 && sg <= 24 && vi < 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 26 && sg <= 26 && vi > 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 26 && sg <= 26 && vi < 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 27 && sg <= 27 && vi > 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 27 && sg <= 27 && vi < 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 32 && sg <= 32 && vi > 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 32 && sg <= 32 && vi < 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 33 && sg <= 33 && vi > 100]
DR16(id, sg, vi) -> DR16(id, sg, 100) [sg >= 33 && sg <= 33 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 7 && sg <= 7 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 7 && sg <= 7 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 11 && sg <= 11 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 11 && sg <= 11 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 101) [sg >= 12 && sg <= 12 && vi > 101]
DR17(id, sg, vi) -> DR17(id, sg, 101) [sg >= 12 && sg <= 12 && vi < 101]
DR17(id, sg, vi) -> DR17(id, sg, 101) [sg >= 13 && sg <= 13 && vi > 101]
DR17(id, sg, vi) -> DR17(id, sg, 101) [sg >= 13 && sg <= 13 && vi < 101]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 21 && sg <= 21 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 21 && sg <= 21 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 22 && sg <= 22 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 22 && sg <= 22 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 23 && sg <= 23 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 23 && sg <= 23 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 24 && sg <= 24 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 24 && sg <= 24 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 27 && sg <= 27 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 27 && sg <= 27 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 28 && sg <= 28 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 28 && sg <= 28 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 30 && sg <= 30 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 30 && sg <= 30 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 31 && sg <= 31 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 31 && sg <= 31 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 32 && sg <= 32 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 32 && sg <= 32 && vi < 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 33 && sg <= 33 && vi > 100]
DR17(id, sg, vi) -> DR17(id, sg, 100) [sg >= 33 && sg <= 33 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 7 && sg <= 7 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 7 && sg <= 7 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 7 && sg <= 7 && vi > 100]
```

```
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 7 && sg <= 7 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 12 && sg <= 12 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 12 && sg <= 12 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 12 && sg <= 12 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 12 && sg <= 12 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 13 && sg <= 13 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 13 && sg <= 13 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 13 && sg <= 13 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 13 && sg <= 13 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 23 && sg <= 23 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 23 && sg <= 23 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 23 && sg <= 23 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 23 && sg <= 23 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 24 && sg <= 24 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 24 && sg <= 24 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 24 && sg <= 24 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 24 && sg <= 24 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 26 && sg <= 26 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 26 && sg <= 26 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 26 && sg <= 26 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 26 && sg <= 26 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 27 && sg <= 27 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 27 && sg <= 27 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 27 && sg <= 27 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 27 && sg <= 27 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 33 && sg <= 33 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id > 66 && sg >= 33 && sg <= 33 && vi < 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 33 && sg <= 33 && vi > 100]
DR18(id, sg, vi) -> DR18(id, sg, 100) [id < 66 && sg >= 33 && sg <= 33 && vi < 100]
```

A.4. AProVE Results

Listing A.6 shows the result of AProVE for the ITRS in Listing A.4. Listing A.7 gives the result of AProVE for the ITRS in Listing A.5. Listing A.8 presents the result of AProVE for the ITRS which results from Listing A.5 through replacing `11` with 12 in Line 21 and 22.

Listing A.6: AProVE report for Listing A.4

```
NO
proof of crb-1.inttrs
# AProVE Commit ID: 2e6638c59cfd6c865410a35d3360fc0074b41f84 ffrohn 20140725


Termination of the given IRSwT could be disproven:

(0) IRSwT
(1) IRSwTTerminationDigraphProof [EQUIVALENT, 136 ms]
(2) AND
    (3) IRSwT
        (4) FilterProof [EQUIVALENT, 0 ms]
        (5) IntTRS
        (6) IntTRSPeriodicNontermProof [COMPLETE, 0 ms]
        (7) NO
    (8) IRSwT
        (9) FilterProof [EQUIVALENT, 0 ms]
        (10) IntTRS
        (11) IntTRSNonPeriodicNontermProof [COMPLETE, 4 ms]
```

```
        (12) NO


----------------------------------------

(0)
Obligation:
Rules:
D16(vi) -> D16(100) :|: vi > 100
D16(x) -> D16(100) :|: x < 100
D16(x1) -> D16(101) :|: x1 > 101
D16(x2) -> D16(101) :|: x2 < 101
D17(x3) -> D17(100) :|: x3 > 100
D17(x4) -> D17(100) :|: x4 < 100
D17(x5) -> D17(101) :|: x5 > 101
D17(x6) -> D17(101) :|: x6 < 101
D18(x7) -> D18(100) :|: x7 > 100
D18(x8) -> D18(100) :|: x8 < 100

----------------------------------------

(1) IRSwTTerminationDigraphProof (EQUIVALENT)
Constructed termination digraph!
Nodes:
(1) D16(vi) -> D16(100) :|: vi > 100
(2) D16(x) -> D16(100) :|: x < 100
(3) D16(x1) -> D16(101) :|: x1 > 101
(4) D16(x2) -> D16(101) :|: x2 < 101
(5) D17(x3) -> D17(100) :|: x3 > 100
(6) D17(x4) -> D17(100) :|: x4 < 100
(7) D17(x5) -> D17(101) :|: x5 > 101
(8) D17(x6) -> D17(101) :|: x6 < 101
(9) D18(x7) -> D18(100) :|: x7 > 100
(10) D18(x8) -> D18(100) :|: x8 < 100

Arcs:
(1) -> (4)
(2) -> (4)
(3) -> (1)
(4) -> (1)
(5) -> (8)
(6) -> (8)
(7) -> (5)
(8) -> (5)

This digraph is fully evaluated!
----------------------------------------

(2)
Complex Obligation (AND)

----------------------------------------

(3)
Obligation:

Termination digraph:
Nodes:
(1) D17(x3) -> D17(100) :|: x3 > 100
(2) D17(x6) -> D17(101) :|: x6 < 101

Arcs:
```

```
(1) -> (2)
(2) -> (1)

This digraph is fully evaluated!

----------------------------------------

(4) FilterProof (EQUIVALENT)
Used the following sort dictionary for filtering:
D17(VARIABLE)
Replaced non-predefined constructor symbols by 0.
----------------------------------------

(5)
Obligation:
Rules:
D17(x3) -> D17(100) :|: x3 > 100
D17(x6) -> D17(101) :|: x6 < 101

----------------------------------------

(6) IntTRSPeriodicNontermProof (COMPLETE)
Normalized system to the following form:
f(pc, x3) -> f(1, 100) :|: pc = 1 && x3 > 100
f(pc, x6) -> f(1, 101) :|: pc = 1 && x6 < 101
Witness term starting non-terminating reduction: f(1, 100)
----------------------------------------

(7)
NO

----------------------------------------

(8)
Obligation:

Termination digraph:
Nodes:
(1) D16(vi) -> D16(100) :|: vi > 100
(2) D16(x2) -> D16(101) :|: x2 < 101

Arcs:
(1) -> (2)
(2) -> (1)

This digraph is fully evaluated!

----------------------------------------

(9) FilterProof (EQUIVALENT)
Used the following sort dictionary for filtering:
D16(VARIABLE)
Replaced non-predefined constructor symbols by 0.
----------------------------------------

(10)
Obligation:
Rules:
D16(vi) -> D16(100) :|: vi > 100
D16(x2) -> D16(101) :|: x2 < 101

----------------------------------------
```

```

(11) IntTRSNonPeriodicNontermProof (COMPLETE)
Normalized system to the following form:
f(pc, vi) -> f(1, 100) :|: pc = 1 && vi > 100
f(pc, x2) -> f(1, 101) :|: pc = 1 && x2 < 101
Proved unsatisfiability of the following formula, indicating that the system is never left after entering:
((((run2_0 = ((1 * 1)) and run2_1 = ((1 * 100))) and (((run1_0 * 1)) = ((1 * 1)) and ((run1_1 * 1)) > ((1 * 100)))) or ((run2_0 = ((1 * 1)) and run2_1 = ((1 * 101))) and (((run1_0 * 1)) = ((1 * 1)) and ((run1_1 * 1)) < ((1 * 101))))) and (!(((run2_0 * 1)) = ((1 * 1)) and ((run2_1 * 1)) > ((1 * 100))) and !(((run2_0 * 1)) = ((1 * 1)) and ((run2_1 * 1)) < ((1 * 101)))))
Proved satisfiability of the following formula, indicating that the system is entered at least once:
(((run2_0 = ((1 * 1)) and run2_1 = ((1 * 100))) and (((run1_0 * 1)) = ((1 * 1)) and ((run1_1 * 1)) > ((1 * 100)))) or ((run2_0 = ((1 * 1)) and run2_1 = ((1 * 101))) and (((run1_0 * 1)) = ((1 * 1)) and ((run1_1 * 1)) < ((1 * 101)))))

----------------------------------------

(12)
NO
```

Listing A.7: AProVE report for Listing A.5

```
YES
proof of crb-2.inttrs
# AProVE Commit ID: 2e6638c59cfd6c865410a35d3360fc0074b41f84 ffrohn 20140725


Termination of the given IRSwT could be proven:

(0) IRSwT
(1) IRSwTTerminationDigraphProof [EQUIVALENT, 56.9 s]
(2) TRUE


----------------------------------------

(0)
Obligation:
Rules:
DR16(id, sg, vi) -> DR16(id, sg, 100) :|: sg >= 7 && sg <= 7 && vi > 100
DR16(x, x1, x2) -> DR16(x, x1, 100) :|: x1 >= 7 && x1 <= 7 && x2 < 100
DR16(x3, x4, x5) -> DR16(x3, x4, 101) :|: x4 >= 12 && x4 <= 12 && x5 > 101
DR16(x6, x7, x8) -> DR16(x6, x7, 101) :|: x7 >= 12 && x7 <= 12 && x8 < 101
DR16(x9, x10, x11) -> DR16(x9, x10, 101) :|: x10 >= 13 && x10 <= 13 && x11 > 101
DR16(x12, x13, x14) -> DR16(x12, x13, 101) :|: x13 >= 13 && x13 <= 13 && x14 < 101
DR16(x15, x16, x17) -> DR16(x15, x16, 100) :|: x16 >= 23 && x16 <= 23 && x17 > 100
DR16(x18, x19, x20) -> DR16(x18, x19, 100) :|: x19 >= 23 && x19 <= 23 && x20 < 100
DR16(x21, x22, x23) -> DR16(x21, x22, 100) :|: x22 >= 24 && x22 <= 24 && x23 > 100
DR16(x24, x25, x26) -> DR16(x24, x25, 100) :|: x25 >= 24 && x25 <= 24 && x26 < 100
DR16(x27, x28, x29) -> DR16(x27, x28, 100) :|: x28 >= 26 && x28 <= 26 && x29 > 100
DR16(x30, x31, x32) -> DR16(x30, x31, 100) :|: x31 >= 26 && x31 <= 26 && x32 < 100
DR16(x33, x34, x35) -> DR16(x33, x34, 100) :|: x34 >= 27 && x34 <= 27 && x35 > 100
DR16(x36, x37, x38) -> DR16(x36, x37, 100) :|: x37 >= 27 && x37 <= 27 && x38 < 100
DR16(x39, x40, x41) -> DR16(x39, x40, 100) :|: x40 >= 32 && x40 <= 32 && x41 > 100
DR16(x42, x43, x44) -> DR16(x42, x43, 100) :|: x43 >= 32 && x43 <= 32 && x44 < 100
DR16(x45, x46, x47) -> DR16(x45, x46, 100) :|: x46 >= 33 && x46 <= 33 && x47 > 100
DR16(x48, x49, x50) -> DR16(x48, x49, 100) :|: x49 >= 33 && x49 <= 33 && x50 < 100
DR17(x51, x52, x53) -> DR17(x51, x52, 100) :|: x52 >= 7 && x52 <= 7 && x53 > 100
```

```
DR17(x54, x55, x56) -> DR17(x54, x55, 100) :|: x55 >= 7 && x55 <= 7 && x56 < 100
DR17(x57, x58, x59) -> DR17(x57, x58, 100) :|: x58 >= 11 && x58 <= 11 && x59 > 100
DR17(x60, x61, x62) -> DR17(x60, x61, 100) :|: x61 >= 11 && x61 <= 11 && x62 < 100
DR17(x63, x64, x65) -> DR17(x63, x64, 101) :|: x64 >= 12 && x64 <= 12 && x65 > 101
DR17(x66, x67, x68) -> DR17(x66, x67, 101) :|: x67 >= 12 && x67 <= 12 && x68 < 101
DR17(x69, x70, x71) -> DR17(x69, x70, 101) :|: x70 >= 13 && x70 <= 13 && x71 > 101
DR17(x72, x73, x74) -> DR17(x72, x73, 101) :|: x73 >= 13 && x73 <= 13 && x74 < 101
DR17(x75, x76, x77) -> DR17(x75, x76, 100) :|: x76 >= 21 && x76 <= 21 && x77 > 100
DR17(x78, x79, x80) -> DR17(x78, x79, 100) :|: x79 >= 21 && x79 <= 21 && x80 < 100
DR17(x81, x82, x83) -> DR17(x81, x82, 100) :|: x82 >= 22 && x82 <= 22 && x83 > 100
DR17(x84, x85, x86) -> DR17(x84, x85, 100) :|: x85 >= 22 && x85 <= 22 && x86 < 100
DR17(x87, x88, x89) -> DR17(x87, x88, 100) :|: x88 >= 23 && x88 <= 23 && x89 > 100
DR17(x90, x91, x92) -> DR17(x90, x91, 100) :|: x91 >= 23 && x91 <= 23 && x92 < 100
DR17(x93, x94, x95) -> DR17(x93, x94, 100) :|: x94 >= 24 && x94 <= 24 && x95 > 100
DR17(x96, x97, x98) -> DR17(x96, x97, 100) :|: x97 >= 24 && x97 <= 24 && x98 < 100
DR17(x99, x100, x101) -> DR17(x99, x100, 100) :|: x100 >= 27 && x100 <= 27 && x101 >
    100
DR17(x102, x103, x104) -> DR17(x102, x103, 100) :|: x103 >= 27 && x103 <= 27 && x104
    < 100
DR17(x105, x106, x107) -> DR17(x105, x106, 100) :|: x106 >= 28 && x106 <= 28 && x107
    > 100
DR17(x108, x109, x110) -> DR17(x108, x109, 100) :|: x109 >= 28 && x109 <= 28 && x110
    < 100
DR17(x111, x112, x113) -> DR17(x111, x112, 100) :|: x112 >= 30 && x112 <= 30 && x113
    > 100
DR17(x114, x115, x116) -> DR17(x114, x115, 100) :|: x115 >= 30 && x115 <= 30 && x116
    < 100
DR17(x117, x118, x119) -> DR17(x117, x118, 100) :|: x118 >= 31 && x118 <= 31 && x119
    > 100
DR17(x120, x121, x122) -> DR17(x120, x121, 100) :|: x121 >= 31 && x121 <= 31 && x122
    < 100
DR17(x123, x124, x125) -> DR17(x123, x124, 100) :|: x124 >= 32 && x124 <= 32 && x125
    > 100
DR17(x126, x127, x128) -> DR17(x126, x127, 100) :|: x127 >= 32 && x127 <= 32 && x128
    < 100
DR17(x129, x130, x131) -> DR17(x129, x130, 100) :|: x130 >= 33 && x130 <= 33 && x131
    > 100
DR17(x132, x133, x134) -> DR17(x132, x133, 100) :|: x133 >= 33 && x133 <= 33 && x134
    < 100
DR18(x135, x136, x137) -> DR18(x135, x136, 100) :|: x135 > 66 && x136 >= 7 && x136
    <= 7 && x137 > 100
DR18(x138, x139, x140) -> DR18(x138, x139, 100) :|: x138 > 66 && x139 >= 7 && x139
    <= 7 && x140 < 100
DR18(x141, x142, x143) -> DR18(x141, x142, 100) :|: x141 < 66 && x142 >= 7 && x142
    <= 7 && x143 > 100
DR18(x144, x145, x146) -> DR18(x144, x145, 100) :|: x144 < 66 && x145 >= 7 && x145
    <= 7 && x146 < 100
DR18(x147, x148, x149) -> DR18(x147, x148, 100) :|: x147 > 66 && x148 >= 12 && x148
    <= 12 && x149 > 100
DR18(x150, x151, x152) -> DR18(x150, x151, 100) :|: x150 > 66 && x151 >= 12 && x151
    <= 12 && x152 < 100
DR18(x153, x154, x155) -> DR18(x153, x154, 100) :|: x153 < 66 && x154 >= 12 && x154
    <= 12 && x155 > 100
DR18(x156, x157, x158) -> DR18(x156, x157, 100) :|: x156 < 66 && x157 >= 12 && x157
    <= 12 && x158 < 100
DR18(x159, x160, x161) -> DR18(x159, x160, 100) :|: x159 > 66 && x160 >= 13 && x160
    <= 13 && x161 > 100
DR18(x162, x163, x164) -> DR18(x162, x163, 100) :|: x162 > 66 && x163 >= 13 && x163
    <= 13 && x164 < 100
DR18(x165, x166, x167) -> DR18(x165, x166, 100) :|: x165 < 66 && x166 >= 13 && x166
    <= 13 && x167 > 100
```

```
DR18(x168, x169, x170) -> DR18(x168, x169, 100) :|: x168 < 66 && x169 >= 13 && x169 <= 13 && x170 < 100
DR18(x171, x172, x173) -> DR18(x171, x172, 100) :|: x171 > 66 && x172 >= 23 && x172 <= 23 && x173 > 100
DR18(x174, x175, x176) -> DR18(x174, x175, 100) :|: x174 > 66 && x175 >= 23 && x175 <= 23 && x176 < 100
DR18(x177, x178, x179) -> DR18(x177, x178, 100) :|: x177 < 66 && x178 >= 23 && x178 <= 23 && x179 > 100
DR18(x180, x181, x182) -> DR18(x180, x181, 100) :|: x180 < 66 && x181 >= 23 && x181 <= 23 && x182 < 100
DR18(x183, x184, x185) -> DR18(x183, x184, 100) :|: x183 > 66 && x184 >= 24 && x184 <= 24 && x185 > 100
DR18(x186, x187, x188) -> DR18(x186, x187, 100) :|: x186 > 66 && x187 >= 24 && x187 <= 24 && x188 < 100
DR18(x189, x190, x191) -> DR18(x189, x190, 100) :|: x189 < 66 && x190 >= 24 && x190 <= 24 && x191 > 100
DR18(x192, x193, x194) -> DR18(x192, x193, 100) :|: x192 < 66 && x193 >= 24 && x193 <= 24 && x194 < 100
DR18(x195, x196, x197) -> DR18(x195, x196, 100) :|: x195 > 66 && x196 >= 26 && x196 <= 26 && x197 > 100
DR18(x198, x199, x200) -> DR18(x198, x199, 100) :|: x198 > 66 && x199 >= 26 && x199 <= 26 && x200 < 100
DR18(x201, x202, x203) -> DR18(x201, x202, 100) :|: x201 < 66 && x202 >= 26 && x202 <= 26 && x203 > 100
DR18(x204, x205, x206) -> DR18(x204, x205, 100) :|: x204 < 66 && x205 >= 26 && x205 <= 26 && x206 < 100
DR18(x207, x208, x209) -> DR18(x207, x208, 100) :|: x207 > 66 && x208 >= 27 && x208 <= 27 && x209 > 100
DR18(x210, x211, x212) -> DR18(x210, x211, 100) :|: x210 > 66 && x211 >= 27 && x211 <= 27 && x212 < 100
DR18(x213, x214, x215) -> DR18(x213, x214, 100) :|: x213 < 66 && x214 >= 27 && x214 <= 27 && x215 > 100
DR18(x216, x217, x218) -> DR18(x216, x217, 100) :|: x216 < 66 && x217 >= 27 && x217 <= 27 && x218 < 100
DR18(x219, x220, x221) -> DR18(x219, x220, 100) :|: x219 > 66 && x220 >= 33 && x220 <= 33 && x221 > 100
DR18(x222, x223, x224) -> DR18(x222, x223, 100) :|: x222 > 66 && x223 >= 33 && x223 <= 33 && x224 < 100
DR18(x225, x226, x227) -> DR18(x225, x226, 100) :|: x225 < 66 && x226 >= 33 && x226 <= 33 && x227 > 100
DR18(x228, x229, x230) -> DR18(x228, x229, 100) :|: x228 < 66 && x229 >= 33 && x229 <= 33 && x230 < 100

----------------------------------------

(1) IRSwTTerminationDigraphProof (EQUIVALENT)
Constructed termination digraph!
Nodes:
(1) DR16(id, sg, vi) -> DR16(id, sg, 100) :|: sg >= 7 && sg <= 7 && vi > 100
(2) DR16(x, x1, x2) -> DR16(x, x1, 100) :|: x1 >= 7 && x1 <= 7 && x2 < 100
(3) DR16(x3, x4, x5) -> DR16(x3, x4, 101) :|: x4 >= 12 && x4 <= 12 && x5 > 101
(4) DR16(x6, x7, x8) -> DR16(x6, x7, 101) :|: x7 >= 12 && x7 <= 12 && x8 < 101
(5) DR16(x9, x10, x11) -> DR16(x9, x10, 101) :|: x10 >= 13 && x10 <= 13 && x11 > 101
(6) DR16(x12, x13, x14) -> DR16(x12, x13, 101) :|: x13 >= 13 && x13 <= 13 && x14 < 101
(7) DR16(x15, x16, x17) -> DR16(x15, x16, 100) :|: x16 >= 23 && x16 <= 23 && x17 > 100
(8) DR16(x18, x19, x20) -> DR16(x18, x19, 100) :|: x19 >= 23 && x19 <= 23 && x20 < 100
(9) DR16(x21, x22, x23) -> DR16(x21, x22, 100) :|: x22 >= 24 && x22 <= 24 && x23 > 100
```

```
(10) DR16(x24, x25, x26) -> DR16(x24, x25, 100) :|: x25 >= 24 && x25 <= 24 && x26 < 100
(11) DR16(x27, x28, x29) -> DR16(x27, x28, 100) :|: x28 >= 26 && x28 <= 26 && x29 > 100
(12) DR16(x30, x31, x32) -> DR16(x30, x31, 100) :|: x31 >= 26 && x31 <= 26 && x32 < 100
(13) DR16(x33, x34, x35) -> DR16(x33, x34, 100) :|: x34 >= 27 && x34 <= 27 && x35 > 100
(14) DR16(x36, x37, x38) -> DR16(x36, x37, 100) :|: x37 >= 27 && x37 <= 27 && x38 < 100
(15) DR16(x39, x40, x41) -> DR16(x39, x40, 100) :|: x40 >= 32 && x40 <= 32 && x41 > 100
(16) DR16(x42, x43, x44) -> DR16(x42, x43, 100) :|: x43 >= 32 && x43 <= 32 && x44 < 100
(17) DR16(x45, x46, x47) -> DR16(x45, x46, 100) :|: x46 >= 33 && x46 <= 33 && x47 > 100
(18) DR16(x48, x49, x50) -> DR16(x48, x49, 100) :|: x49 >= 33 && x49 <= 33 && x50 < 100
(19) DR17(x51, x52, x53) -> DR17(x51, x52, 100) :|: x52 >= 7 && x52 <= 7 && x53 > 100
(20) DR17(x54, x55, x56) -> DR17(x54, x55, 100) :|: x55 >= 7 && x55 <= 7 && x56 < 100
(21) DR17(x57, x58, x59) -> DR17(x57, x58, 100) :|: x58 >= 11 && x58 <= 11 && x59 > 100
(22) DR17(x60, x61, x62) -> DR17(x60, x61, 100) :|: x61 >= 11 && x61 <= 11 && x62 < 100
(23) DR17(x63, x64, x65) -> DR17(x63, x64, 101) :|: x64 >= 12 && x64 <= 12 && x65 > 101
(24) DR17(x66, x67, x68) -> DR17(x66, x67, 101) :|: x67 >= 12 && x67 <= 12 && x68 < 101
(25) DR17(x69, x70, x71) -> DR17(x69, x70, 101) :|: x70 >= 13 && x70 <= 13 && x71 > 101
(26) DR17(x72, x73, x74) -> DR17(x72, x73, 101) :|: x73 >= 13 && x73 <= 13 && x74 < 101
(27) DR17(x75, x76, x77) -> DR17(x75, x76, 100) :|: x76 >= 21 && x76 <= 21 && x77 > 100
(28) DR17(x78, x79, x80) -> DR17(x78, x79, 100) :|: x79 >= 21 && x79 <= 21 && x80 < 100
(29) DR17(x81, x82, x83) -> DR17(x81, x82, 100) :|: x82 >= 22 && x82 <= 22 && x83 > 100
(30) DR17(x84, x85, x86) -> DR17(x84, x85, 100) :|: x85 >= 22 && x85 <= 22 && x86 < 100
(31) DR17(x87, x88, x89) -> DR17(x87, x88, 100) :|: x88 >= 23 && x88 <= 23 && x89 > 100
(32) DR17(x90, x91, x92) -> DR17(x90, x91, 100) :|: x91 >= 23 && x91 <= 23 && x92 < 100
(33) DR17(x93, x94, x95) -> DR17(x93, x94, 100) :|: x94 >= 24 && x94 <= 24 && x95 > 100
(34) DR17(x96, x97, x98) -> DR17(x96, x97, 100) :|: x97 >= 24 && x97 <= 24 && x98 < 100
(35) DR17(x99, x100, x101) -> DR17(x99, x100, 100) :|: x100 >= 27 && x100 <= 27 && x101 > 100
(36) DR17(x102, x103, x104) -> DR17(x102, x103, 100) :|: x103 >= 27 && x103 <= 27 && x104 < 100
(37) DR17(x105, x106, x107) -> DR17(x105, x106, 100) :|: x106 >= 28 && x106 <= 28 && x107 > 100
(38) DR17(x108, x109, x110) -> DR17(x108, x109, 100) :|: x109 >= 28 && x109 <= 28 && x110 < 100
(39) DR17(x111, x112, x113) -> DR17(x111, x112, 100) :|: x112 >= 30 && x112 <= 30 && x113 > 100
(40) DR17(x114, x115, x116) -> DR17(x114, x115, 100) :|: x115 >= 30 && x115 <= 30 && x116 < 100
(41) DR17(x117, x118, x119) -> DR17(x117, x118, 100) :|: x118 >= 31 && x118 <= 31 && x119 > 100
```

```
(42) DR17(x120, x121, x122) -> DR17(x120, x121, 100) :|: x121 >= 31 && x121 <= 31 &&
    x122 < 100
(43) DR17(x123, x124, x125) -> DR17(x123, x124, 100) :|: x124 >= 32 && x124 <= 32 &&
    x125 > 100
(44) DR17(x126, x127, x128) -> DR17(x126, x127, 100) :|: x127 >= 32 && x127 <= 32 &&
    x128 < 100
(45) DR17(x129, x130, x131) -> DR17(x129, x130, 100) :|: x130 >= 33 && x130 <= 33 &&
    x131 > 100
(46) DR17(x132, x133, x134) -> DR17(x132, x133, 100) :|: x133 >= 33 && x133 <= 33 &&
    x134 < 100
(47) DR18(x135, x136, x137) -> DR18(x135, x136, 100) :|: x135 > 66 && x136 >= 7 &&
    x136 <= 7 && x137 > 100
(48) DR18(x138, x139, x140) -> DR18(x138, x139, 100) :|: x138 > 66 && x139 >= 7 &&
    x139 <= 7 && x140 < 100
(49) DR18(x141, x142, x143) -> DR18(x141, x142, 100) :|: x141 < 66 && x142 >= 7 &&
    x142 <= 7 && x143 > 100
(50) DR18(x144, x145, x146) -> DR18(x144, x145, 100) :|: x144 < 66 && x145 >= 7 &&
    x145 <= 7 && x146 < 100
(51) DR18(x147, x148, x149) -> DR18(x147, x148, 100) :|: x147 > 66 && x148 >= 12 &&
    x148 <= 12 && x149 > 100
(52) DR18(x150, x151, x152) -> DR18(x150, x151, 100) :|: x150 > 66 && x151 >= 12 &&
    x151 <= 12 && x152 < 100
(53) DR18(x153, x154, x155) -> DR18(x153, x154, 100) :|: x153 < 66 && x154 >= 12 &&
    x154 <= 12 && x155 > 100
(54) DR18(x156, x157, x158) -> DR18(x156, x157, 100) :|: x156 < 66 && x157 >= 12 &&
    x157 <= 12 && x158 < 100
(55) DR18(x159, x160, x161) -> DR18(x159, x160, 100) :|: x159 > 66 && x160 >= 13 &&
    x160 <= 13 && x161 > 100
(56) DR18(x162, x163, x164) -> DR18(x162, x163, 100) :|: x162 > 66 && x163 >= 13 &&
    x163 <= 13 && x164 < 100
(57) DR18(x165, x166, x167) -> DR18(x165, x166, 100) :|: x165 < 66 && x166 >= 13 &&
    x166 <= 13 && x167 > 100
(58) DR18(x168, x169, x170) -> DR18(x168, x169, 100) :|: x168 < 66 && x169 >= 13 &&
    x169 <= 13 && x170 < 100
(59) DR18(x171, x172, x173) -> DR18(x171, x172, 100) :|: x171 > 66 && x172 >= 23 &&
    x172 <= 23 && x173 > 100
(60) DR18(x174, x175, x176) -> DR18(x174, x175, 100) :|: x174 > 66 && x175 >= 23 &&
    x175 <= 23 && x176 < 100
(61) DR18(x177, x178, x179) -> DR18(x177, x178, 100) :|: x177 < 66 && x178 >= 23 &&
    x178 <= 23 && x179 > 100
(62) DR18(x180, x181, x182) -> DR18(x180, x181, 100) :|: x180 < 66 && x181 >= 23 &&
    x181 <= 23 && x182 < 100
(63) DR18(x183, x184, x185) -> DR18(x183, x184, 100) :|: x183 > 66 && x184 >= 24 &&
    x184 <= 24 && x185 > 100
(64) DR18(x186, x187, x188) -> DR18(x186, x187, 100) :|: x186 > 66 && x187 >= 24 &&
    x187 <= 24 && x188 < 100
(65) DR18(x189, x190, x191) -> DR18(x189, x190, 100) :|: x189 < 66 && x190 >= 24 &&
    x190 <= 24 && x191 > 100
(66) DR18(x192, x193, x194) -> DR18(x192, x193, 100) :|: x192 < 66 && x193 >= 24 &&
    x193 <= 24 && x194 < 100
(67) DR18(x195, x196, x197) -> DR18(x195, x196, 100) :|: x195 > 66 && x196 >= 26 &&
    x196 <= 26 && x197 > 100
(68) DR18(x198, x199, x200) -> DR18(x198, x199, 100) :|: x198 > 66 && x199 >= 26 &&
    x199 <= 26 && x200 < 100
(69) DR18(x201, x202, x203) -> DR18(x201, x202, 100) :|: x201 < 66 && x202 >= 26 &&
    x202 <= 26 && x203 > 100
(70) DR18(x204, x205, x206) -> DR18(x204, x205, 100) :|: x204 < 66 && x205 >= 26 &&
    x205 <= 26 && x206 < 100
(71) DR18(x207, x208, x209) -> DR18(x207, x208, 100) :|: x207 > 66 && x208 >= 27 &&
    x208 <= 27 && x209 > 100
(72) DR18(x210, x211, x212) -> DR18(x210, x211, 100) :|: x210 > 66 && x211 >= 27 &&
    x211 <= 27 && x212 < 100
```

```
(73) DR18(x213, x214, x215) -> DR18(x213, x214, 100) :|: x213 < 66 && x214 >= 27 && x214 <= 27 && x215 > 100
(74) DR18(x216, x217, x218) -> DR18(x216, x217, 100) :|: x216 < 66 && x217 >= 27 && x217 <= 27 && x218 < 100
(75) DR18(x219, x220, x221) -> DR18(x219, x220, 100) :|: x219 > 66 && x220 >= 33 && x220 <= 33 && x221 > 100
(76) DR18(x222, x223, x224) -> DR18(x222, x223, 100) :|: x222 > 66 && x223 >= 33 && x223 <= 33 && x224 < 100
(77) DR18(x225, x226, x227) -> DR18(x225, x226, 100) :|: x225 < 66 && x226 >= 33 && x226 <= 33 && x227 > 100
(78) DR18(x228, x229, x230) -> DR18(x228, x229, 100) :|: x228 < 66 && x229 >= 33 && x229 <= 33 && x230 < 100

No arcs!

This digraph is fully evaluated!
----------------------------------------

(2)
TRUE
```

Listing A.8: AProVE report for a defective version of Listing A.5

```
NO
proof of crb-3.inttrs
# AProVE Commit ID: 2e6638c59cfd6c865410a35d3360fc0074b41f84 ffrohn 20140725


Termination of the given IRSwT could be disproven:

(0) IRSwT
(1) IRSwTTerminationDigraphProof [EQUIVALENT, 56.6 s]
(2) IRSwT
(3) IntTRSUnneededArgumentFilterProof [EQUIVALENT, 0 ms]
(4) IntTRS
(5) FilterProof [EQUIVALENT, 0 ms]
(6) IntTRS
(7) IntTRSPeriodicNontermProof [COMPLETE, 11 ms]
(8) NO


----------------------------------------

(0)
Obligation:
Rules:
DR16(id, sg, vi) -> DR16(id, sg, 100) :|: sg >= 7 && sg <= 7 && vi > 100
DR16(x, x1, x2) -> DR16(x, x1, 100) :|: x1 >= 7 && x1 <= 7 && x2 < 100
DR16(x3, x4, x5) -> DR16(x3, x4, 101) :|: x4 >= 12 && x4 <= 12 && x5 > 101
DR16(x6, x7, x8) -> DR16(x6, x7, 101) :|: x7 >= 12 && x7 <= 12 && x8 < 101
DR16(x9, x10, x11) -> DR16(x9, x10, 101) :|: x10 >= 13 && x10 <= 13 && x11 > 101
DR16(x12, x13, x14) -> DR16(x12, x13, 101) :|: x13 >= 13 && x13 <= 13 && x14 < 101
DR16(x15, x16, x17) -> DR16(x15, x16, 100) :|: x16 >= 23 && x16 <= 23 && x17 > 100
DR16(x18, x19, x20) -> DR16(x18, x19, 100) :|: x19 >= 23 && x19 <= 23 && x20 < 100
DR16(x21, x22, x23) -> DR16(x21, x22, 100) :|: x22 >= 24 && x22 <= 24 && x23 > 100
DR16(x24, x25, x26) -> DR16(x24, x25, 100) :|: x25 >= 24 && x25 <= 24 && x26 < 100
DR16(x27, x28, x29) -> DR16(x27, x28, 100) :|: x28 >= 26 && x28 <= 26 && x29 > 100
DR16(x30, x31, x32) -> DR16(x30, x31, 100) :|: x31 >= 26 && x31 <= 26 && x32 < 100
DR16(x33, x34, x35) -> DR16(x33, x34, 100) :|: x34 >= 27 && x34 <= 27 && x35 > 100
DR16(x36, x37, x38) -> DR16(x36, x37, 100) :|: x37 >= 27 && x37 <= 27 && x38 < 100
DR16(x39, x40, x41) -> DR16(x39, x40, 100) :|: x40 >= 32 && x40 <= 32 && x41 > 100
```

```
DR16(x42, x43, x44) -> DR16(x42, x43, 100) :|: x43 >= 32 && x43 <= 32 && x44 < 100
DR16(x45, x46, x47) -> DR16(x45, x46, 100) :|: x46 >= 33 && x46 <= 33 && x47 > 100
DR16(x48, x49, x50) -> DR16(x48, x49, 100) :|: x49 >= 33 && x49 <= 33 && x50 < 100
DR17(x51, x52, x53) -> DR17(x51, x52, 100) :|: x52 >= 7 && x52 <= 7 && x53 > 100
DR17(x54, x55, x56) -> DR17(x54, x55, 100) :|: x55 >= 7 && x55 <= 7 && x56 < 100
DR17(x57, x58, x59) -> DR17(x57, x58, 100) :|: x58 >= 12 && x58 <= 12 && x59 > 100
DR17(x60, x61, x62) -> DR17(x60, x61, 100) :|: x61 >= 12 && x61 <= 12 && x62 < 100
DR17(x63, x64, x65) -> DR17(x63, x64, 101) :|: x64 >= 12 && x64 <= 12 && x65 > 101
DR17(x66, x67, x68) -> DR17(x66, x67, 101) :|: x67 >= 12 && x67 <= 12 && x68 < 101
DR17(x69, x70, x71) -> DR17(x69, x70, 101) :|: x70 >= 13 && x70 <= 13 && x71 > 101
DR17(x72, x73, x74) -> DR17(x72, x73, 101) :|: x73 >= 13 && x73 <= 13 && x74 < 101
DR17(x75, x76, x77) -> DR17(x75, x76, 100) :|: x76 >= 21 && x76 <= 21 && x77 > 100
DR17(x78, x79, x80) -> DR17(x78, x79, 100) :|: x79 >= 21 && x79 <= 21 && x80 < 100
DR17(x81, x82, x83) -> DR17(x81, x82, 100) :|: x82 >= 22 && x82 <= 22 && x83 > 100
DR17(x84, x85, x86) -> DR17(x84, x85, 100) :|: x85 >= 22 && x85 <= 22 && x86 < 100
DR17(x87, x88, x89) -> DR17(x87, x88, 100) :|: x88 >= 23 && x88 <= 23 && x89 > 100
DR17(x90, x91, x92) -> DR17(x90, x91, 100) :|: x91 >= 23 && x91 <= 23 && x92 < 100
DR17(x93, x94, x95) -> DR17(x93, x94, 100) :|: x94 >= 24 && x94 <= 24 && x95 > 100
DR17(x96, x97, x98) -> DR17(x96, x97, 100) :|: x97 >= 24 && x97 <= 24 && x98 < 100
DR17(x99, x100, x101) -> DR17(x99, x100, 100) :|: x100 >= 27 && x100 <= 27 && x101 > 100
DR17(x102, x103, x104) -> DR17(x102, x103, 100) :|: x103 >= 27 && x103 <= 27 && x104 < 100
DR17(x105, x106, x107) -> DR17(x105, x106, 100) :|: x106 >= 28 && x106 <= 28 && x107 > 100
DR17(x108, x109, x110) -> DR17(x108, x109, 100) :|: x109 >= 28 && x109 <= 28 && x110 < 100
DR17(x111, x112, x113) -> DR17(x111, x112, 100) :|: x112 >= 30 && x112 <= 30 && x113 > 100
DR17(x114, x115, x116) -> DR17(x114, x115, 100) :|: x115 >= 30 && x115 <= 30 && x116 < 100
DR17(x117, x118, x119) -> DR17(x117, x118, 100) :|: x118 >= 31 && x118 <= 31 && x119 > 100
DR17(x120, x121, x122) -> DR17(x120, x121, 100) :|: x121 >= 31 && x121 <= 31 && x122 < 100
DR17(x123, x124, x125) -> DR17(x123, x124, 100) :|: x124 >= 32 && x124 <= 32 && x125 > 100
DR17(x126, x127, x128) -> DR17(x126, x127, 100) :|: x127 >= 32 && x127 <= 32 && x128 < 100
DR17(x129, x130, x131) -> DR17(x129, x130, 100) :|: x130 >= 33 && x130 <= 33 && x131 > 100
DR17(x132, x133, x134) -> DR17(x132, x133, 100) :|: x133 >= 33 && x133 <= 33 && x134 < 100
DR18(x135, x136, x137) -> DR18(x135, x136, 100) :|: x135 > 66 && x136 >= 7 && x136 <= 7 && x137 > 100
DR18(x138, x139, x140) -> DR18(x138, x139, 100) :|: x138 > 66 && x139 >= 7 && x139 <= 7 && x140 < 100
DR18(x141, x142, x143) -> DR18(x141, x142, 100) :|: x141 < 66 && x142 >= 7 && x142 <= 7 && x143 > 100
DR18(x144, x145, x146) -> DR18(x144, x145, 100) :|: x144 < 66 && x145 >= 7 && x145 <= 7 && x146 < 100
DR18(x147, x148, x149) -> DR18(x147, x148, 100) :|: x147 > 66 && x148 >= 12 && x148 <= 12 && x149 > 100
DR18(x150, x151, x152) -> DR18(x150, x151, 100) :|: x150 > 66 && x151 >= 12 && x151 <= 12 && x152 < 100
DR18(x153, x154, x155) -> DR18(x153, x154, 100) :|: x153 < 66 && x154 >= 12 && x154 <= 12 && x155 > 100
DR18(x156, x157, x158) -> DR18(x156, x157, 100) :|: x156 < 66 && x157 >= 12 && x157 <= 12 && x158 < 100
DR18(x159, x160, x161) -> DR18(x159, x160, 100) :|: x159 > 66 && x160 >= 13 && x160 <= 13 && x161 > 100
```

```
DR18(x162, x163, x164) -> DR18(x162, x163, 100) :|: x162 > 66 && x163 >= 13 && x163 <= 13 && x164 < 100
DR18(x165, x166, x167) -> DR18(x165, x166, 100) :|: x165 < 66 && x166 >= 13 && x166 <= 13 && x167 > 100
DR18(x168, x169, x170) -> DR18(x168, x169, 100) :|: x168 < 66 && x169 >= 13 && x169 <= 13 && x170 < 100
DR18(x171, x172, x173) -> DR18(x171, x172, 100) :|: x171 > 66 && x172 >= 23 && x172 <= 23 && x173 > 100
DR18(x174, x175, x176) -> DR18(x174, x175, 100) :|: x174 > 66 && x175 >= 23 && x175 <= 23 && x176 < 100
DR18(x177, x178, x179) -> DR18(x177, x178, 100) :|: x177 < 66 && x178 >= 23 && x178 <= 23 && x179 > 100
DR18(x180, x181, x182) -> DR18(x180, x181, 100) :|: x180 < 66 && x181 >= 23 && x181 <= 23 && x182 < 100
DR18(x183, x184, x185) -> DR18(x183, x184, 100) :|: x183 > 66 && x184 >= 24 && x184 <= 24 && x185 > 100
DR18(x186, x187, x188) -> DR18(x186, x187, 100) :|: x186 > 66 && x187 >= 24 && x187 <= 24 && x188 < 100
DR18(x189, x190, x191) -> DR18(x189, x190, 100) :|: x189 < 66 && x190 >= 24 && x190 <= 24 && x191 > 100
DR18(x192, x193, x194) -> DR18(x192, x193, 100) :|: x192 < 66 && x193 >= 24 && x193 <= 24 && x194 < 100
DR18(x195, x196, x197) -> DR18(x195, x196, 100) :|: x195 > 66 && x196 >= 26 && x196 <= 26 && x197 > 100
DR18(x198, x199, x200) -> DR18(x198, x199, 100) :|: x198 > 66 && x199 >= 26 && x199 <= 26 && x200 < 100
DR18(x201, x202, x203) -> DR18(x201, x202, 100) :|: x201 < 66 && x202 >= 26 && x202 <= 26 && x203 > 100
DR18(x204, x205, x206) -> DR18(x204, x205, 100) :|: x204 < 66 && x205 >= 26 && x205 <= 26 && x206 < 100
DR18(x207, x208, x209) -> DR18(x207, x208, 100) :|: x207 > 66 && x208 >= 27 && x208 <= 27 && x209 > 100
DR18(x210, x211, x212) -> DR18(x210, x211, 100) :|: x210 > 66 && x211 >= 27 && x211 <= 27 && x212 < 100
DR18(x213, x214, x215) -> DR18(x213, x214, 100) :|: x213 < 66 && x214 >= 27 && x214 <= 27 && x215 > 100
DR18(x216, x217, x218) -> DR18(x216, x217, 100) :|: x216 < 66 && x217 >= 27 && x217 <= 27 && x218 < 100
DR18(x219, x220, x221) -> DR18(x219, x220, 100) :|: x219 > 66 && x220 >= 33 && x220 <= 33 && x221 > 100
DR18(x222, x223, x224) -> DR18(x222, x223, 100) :|: x222 > 66 && x223 >= 33 && x223 <= 33 && x224 < 100
DR18(x225, x226, x227) -> DR18(x225, x226, 100) :|: x225 < 66 && x226 >= 33 && x226 <= 33 && x227 > 100
DR18(x228, x229, x230) -> DR18(x228, x229, 100) :|: x228 < 66 && x229 >= 33 && x229 <= 33 && x230 < 100

----------------------------------------

(1) IRSwTTerminationDigraphProof (EQUIVALENT)
Constructed termination digraph!
Nodes:
(1) DR16(id, sg, vi) -> DR16(id, sg, 100) :|: sg >= 7 && sg <= 7 && vi > 100
(2) DR16(x, x1, x2) -> DR16(x, x1, 100) :|: x1 >= 7 && x1 <= 7 && x2 < 100
(3) DR16(x3, x4, x5) -> DR16(x3, x4, 101) :|: x4 >= 12 && x4 <= 12 && x5 > 101
(4) DR16(x6, x7, x8) -> DR16(x6, x7, 101) :|: x7 >= 12 && x7 <= 12 && x8 < 101
(5) DR16(x9, x10, x11) -> DR16(x9, x10, 101) :|: x10 >= 13 && x10 <= 13 && x11 > 101
(6) DR16(x12, x13, x14) -> DR16(x12, x13, 101) :|: x13 >= 13 && x13 <= 13 && x14 < 101
(7) DR16(x15, x16, x17) -> DR16(x15, x16, 100) :|: x16 >= 23 && x16 <= 23 && x17 > 100
```

```
(8) DR16(x18, x19, x20) -> DR16(x18, x19, 100) :|: x19 >= 23 && x19 <= 23 && x20 < 100
(9) DR16(x21, x22, x23) -> DR16(x21, x22, 100) :|: x22 >= 24 && x22 <= 24 && x23 > 100
(10) DR16(x24, x25, x26) -> DR16(x24, x25, 100) :|: x25 >= 24 && x25 <= 24 && x26 < 100
(11) DR16(x27, x28, x29) -> DR16(x27, x28, 100) :|: x28 >= 26 && x28 <= 26 && x29 > 100
(12) DR16(x30, x31, x32) -> DR16(x30, x31, 100) :|: x31 >= 26 && x31 <= 26 && x32 < 100
(13) DR16(x33, x34, x35) -> DR16(x33, x34, 100) :|: x34 >= 27 && x34 <= 27 && x35 > 100
(14) DR16(x36, x37, x38) -> DR16(x36, x37, 100) :|: x37 >= 27 && x37 <= 27 && x38 < 100
(15) DR16(x39, x40, x41) -> DR16(x39, x40, 100) :|: x40 >= 32 && x40 <= 32 && x41 > 100
(16) DR16(x42, x43, x44) -> DR16(x42, x43, 100) :|: x43 >= 32 && x43 <= 32 && x44 < 100
(17) DR16(x45, x46, x47) -> DR16(x45, x46, 100) :|: x46 >= 33 && x46 <= 33 && x47 > 100
(18) DR16(x48, x49, x50) -> DR16(x48, x49, 100) :|: x49 >= 33 && x49 <= 33 && x50 < 100
(19) DR17(x51, x52, x53) -> DR17(x51, x52, 100) :|: x52 >= 7 && x52 <= 7 && x53 > 100
(20) DR17(x54, x55, x56) -> DR17(x54, x55, 100) :|: x55 >= 7 && x55 <= 7 && x56 < 100
(21) DR17(x57, x58, x59) -> DR17(x57, x58, 100) :|: x58 >= 12 && x58 <= 12 && x59 > 100
(22) DR17(x60, x61, x62) -> DR17(x60, x61, 100) :|: x61 >= 12 && x61 <= 12 && x62 < 100
(23) DR17(x63, x64, x65) -> DR17(x63, x64, 101) :|: x64 >= 12 && x64 <= 12 && x65 > 101
(24) DR17(x66, x67, x68) -> DR17(x66, x67, 101) :|: x67 >= 12 && x67 <= 12 && x68 < 101
(25) DR17(x69, x70, x71) -> DR17(x69, x70, 101) :|: x70 >= 13 && x70 <= 13 && x71 > 101
(26) DR17(x72, x73, x74) -> DR17(x72, x73, 101) :|: x73 >= 13 && x73 <= 13 && x74 < 101
(27) DR17(x75, x76, x77) -> DR17(x75, x76, 100) :|: x76 >= 21 && x76 <= 21 && x77 > 100
(28) DR17(x78, x79, x80) -> DR17(x78, x79, 100) :|: x79 >= 21 && x79 <= 21 && x80 < 100
(29) DR17(x81, x82, x83) -> DR17(x81, x82, 100) :|: x82 >= 22 && x82 <= 22 && x83 > 100
(30) DR17(x84, x85, x86) -> DR17(x84, x85, 100) :|: x85 >= 22 && x85 <= 22 && x86 < 100
(31) DR17(x87, x88, x89) -> DR17(x87, x88, 100) :|: x88 >= 23 && x88 <= 23 && x89 > 100
(32) DR17(x90, x91, x92) -> DR17(x90, x91, 100) :|: x91 >= 23 && x91 <= 23 && x92 < 100
(33) DR17(x93, x94, x95) -> DR17(x93, x94, 100) :|: x94 >= 24 && x94 <= 24 && x95 > 100
(34) DR17(x96, x97, x98) -> DR17(x96, x97, 100) :|: x97 >= 24 && x97 <= 24 && x98 < 100
(35) DR17(x99, x100, x101) -> DR17(x99, x100, 100) :|: x100 >= 27 && x100 <= 27 && x101 > 100
(36) DR17(x102, x103, x104) -> DR17(x102, x103, 100) :|: x103 >= 27 && x103 <= 27 && x104 < 100
(37) DR17(x105, x106, x107) -> DR17(x105, x106, 100) :|: x106 >= 28 && x106 <= 28 && x107 > 100
(38) DR17(x108, x109, x110) -> DR17(x108, x109, 100) :|: x109 >= 28 && x109 <= 28 && x110 < 100
(39) DR17(x111, x112, x113) -> DR17(x111, x112, 100) :|: x112 >= 30 && x112 <= 30 && x113 > 100
```

```
(40) DR17(x114, x115, x116) -> DR17(x114, x115, 100) :|: x115 >= 30 && x115 <= 30 &&
     x116 < 100
(41) DR17(x117, x118, x119) -> DR17(x117, x118, 100) :|: x118 >= 31 && x118 <= 31 &&
     x119 > 100
(42) DR17(x120, x121, x122) -> DR17(x120, x121, 100) :|: x121 >= 31 && x121 <= 31 &&
     x122 < 100
(43) DR17(x123, x124, x125) -> DR17(x123, x124, 100) :|: x124 >= 32 && x124 <= 32 &&
     x125 > 100
(44) DR17(x126, x127, x128) -> DR17(x126, x127, 100) :|: x127 >= 32 && x127 <= 32 &&
     x128 < 100
(45) DR17(x129, x130, x131) -> DR17(x129, x130, 100) :|: x130 >= 33 && x130 <= 33 &&
     x131 > 100
(46) DR17(x132, x133, x134) -> DR17(x132, x133, 100) :|: x133 >= 33 && x133 <= 33 &&
     x134 < 100
(47) DR18(x135, x136, x137) -> DR18(x135, x136, 100) :|: x135 > 66 && x136 >= 7 &&
     x136 <= 7 && x137 > 100
(48) DR18(x138, x139, x140) -> DR18(x138, x139, 100) :|: x138 > 66 && x139 >= 7 &&
     x139 <= 7 && x140 < 100
(49) DR18(x141, x142, x143) -> DR18(x141, x142, 100) :|: x141 < 66 && x142 >= 7 &&
     x142 <= 7 && x143 > 100
(50) DR18(x144, x145, x146) -> DR18(x144, x145, 100) :|: x144 < 66 && x145 >= 7 &&
     x145 <= 7 && x146 < 100
(51) DR18(x147, x148, x149) -> DR18(x147, x148, 100) :|: x147 > 66 && x148 >= 12 &&
     x148 <= 12 && x149 > 100
(52) DR18(x150, x151, x152) -> DR18(x150, x151, 100) :|: x150 > 66 && x151 >= 12 &&
     x151 <= 12 && x152 < 100
(53) DR18(x153, x154, x155) -> DR18(x153, x154, 100) :|: x153 < 66 && x154 >= 12 &&
     x154 <= 12 && x155 > 100
(54) DR18(x156, x157, x158) -> DR18(x156, x157, 100) :|: x156 < 66 && x157 >= 12 &&
     x157 <= 12 && x158 < 100
(55) DR18(x159, x160, x161) -> DR18(x159, x160, 100) :|: x159 > 66 && x160 >= 13 &&
     x160 <= 13 && x161 > 100
(56) DR18(x162, x163, x164) -> DR18(x162, x163, 100) :|: x162 > 66 && x163 >= 13 &&
     x163 <= 13 && x164 < 100
(57) DR18(x165, x166, x167) -> DR18(x165, x166, 100) :|: x165 < 66 && x166 >= 13 &&
     x166 <= 13 && x167 > 100
(58) DR18(x168, x169, x170) -> DR18(x168, x169, 100) :|: x168 < 66 && x169 >= 13 &&
     x169 <= 13 && x170 < 100
(59) DR18(x171, x172, x173) -> DR18(x171, x172, 100) :|: x171 > 66 && x172 >= 23 &&
     x172 <= 23 && x173 > 100
(60) DR18(x174, x175, x176) -> DR18(x174, x175, 100) :|: x174 > 66 && x175 >= 23 &&
     x175 <= 23 && x176 < 100
(61) DR18(x177, x178, x179) -> DR18(x177, x178, 100) :|: x177 < 66 && x178 >= 23 &&
     x178 <= 23 && x179 > 100
(62) DR18(x180, x181, x182) -> DR18(x180, x181, 100) :|: x180 < 66 && x181 >= 23 &&
     x181 <= 23 && x182 < 100
(63) DR18(x183, x184, x185) -> DR18(x183, x184, 100) :|: x183 > 66 && x184 >= 24 &&
     x184 <= 24 && x185 > 100
(64) DR18(x186, x187, x188) -> DR18(x186, x187, 100) :|: x186 > 66 && x187 >= 24 &&
     x187 <= 24 && x188 < 100
(65) DR18(x189, x190, x191) -> DR18(x189, x190, 100) :|: x189 < 66 && x190 >= 24 &&
     x190 <= 24 && x191 > 100
(66) DR18(x192, x193, x194) -> DR18(x192, x193, 100) :|: x192 < 66 && x193 >= 24 &&
     x193 <= 24 && x194 < 100
(67) DR18(x195, x196, x197) -> DR18(x195, x196, 100) :|: x195 > 66 && x196 >= 26 &&
     x196 <= 26 && x197 > 100
(68) DR18(x198, x199, x200) -> DR18(x198, x199, 100) :|: x198 > 66 && x199 >= 26 &&
     x199 <= 26 && x200 < 100
(69) DR18(x201, x202, x203) -> DR18(x201, x202, 100) :|: x201 < 66 && x202 >= 26 &&
     x202 <= 26 && x203 > 100
(70) DR18(x204, x205, x206) -> DR18(x204, x205, 100) :|: x204 < 66 && x205 >= 26 &&
     x205 <= 26 && x206 < 100
```

```
(71) DR18(x207, x208, x209) -> DR18(x207, x208, 100) :|: x207 > 66 && x208 >= 27 && x208 <= 27 && x209 > 100
(72) DR18(x210, x211, x212) -> DR18(x210, x211, 100) :|: x210 > 66 && x211 >= 27 && x211 <= 27 && x212 < 100
(73) DR18(x213, x214, x215) -> DR18(x213, x214, 100) :|: x213 < 66 && x214 >= 27 && x214 <= 27 && x215 > 100
(74) DR18(x216, x217, x218) -> DR18(x216, x217, 100) :|: x216 < 66 && x217 >= 27 && x217 <= 27 && x218 < 100
(75) DR18(x219, x220, x221) -> DR18(x219, x220, 100) :|: x219 > 66 && x220 >= 33 && x220 <= 33 && x221 > 100
(76) DR18(x222, x223, x224) -> DR18(x222, x223, 100) :|: x222 > 66 && x223 >= 33 && x223 <= 33 && x224 < 100
(77) DR18(x225, x226, x227) -> DR18(x225, x226, 100) :|: x225 < 66 && x226 >= 33 && x226 <= 33 && x227 > 100
(78) DR18(x228, x229, x230) -> DR18(x228, x229, 100) :|: x228 < 66 && x229 >= 33 && x229 <= 33 && x230 < 100

Arcs:
(21) -> (24)
(22) -> (24)
(23) -> (21)
(24) -> (21)

This digraph is fully evaluated!
----------------------------------------

(2)
Obligation:

Termination digraph:
Nodes:
(1) DR17(x57, x58, x59) -> DR17(x57, x58, 100) :|: x58 >= 12 && x58 <= 12 && x59 > 100
(2) DR17(x66, x67, x68) -> DR17(x66, x67, 101) :|: x67 >= 12 && x67 <= 12 && x68 < 101

Arcs:
(1) -> (2)
(2) -> (1)

This digraph is fully evaluated!

----------------------------------------

(3) IntTRSUnneededArgumentFilterProof (EQUIVALENT)
Some arguments are removed because they cannot influence termination. We removed arguments according to the following replacements:

   DR17(x1, x2, x3) -> DR17(x2, x3)

----------------------------------------

(4)
Obligation:
Rules:
DR17(x58, x59) -> DR17(x58, 100) :|: x58 >= 12 && x58 <= 12 && x59 > 100
DR17(x67, x68) -> DR17(x67, 101) :|: x67 >= 12 && x67 <= 12 && x68 < 101

----------------------------------------

(5) FilterProof (EQUIVALENT)
Used the following sort dictionary for filtering:
```

```
DR17(INTEGER, VARIABLE)
Replaced non-predefined constructor symbols by 0.
----------------------------------------

(6)
Obligation:
Rules:
DR17(x58, x59) -> DR17(x58, 100) :|: x58 >= 12 && x58 <= 12 && x59 > 100
DR17(x67, x68) -> DR17(x67, 101) :|: x67 >= 12 && x67 <= 12 && x68 < 101

----------------------------------------

(7) IntTRSPeriodicNontermProof (COMPLETE)
Normalized system to the following form:
f(pc, x58, x59) -> f(1, x58, 100) :|: pc = 1 && (x58 >= 12 && x58 <= 12 && x59 > 100)
f(pc, x67, x68) -> f(1, x67, 101) :|: pc = 1 && (x67 >= 12 && x67 <= 12 && x68 < 101)
Witness term starting non-terminating reduction: f(1, 12, 101)
----------------------------------------

(8)
NO
```

B. Javadoc

This appendix contains the Javadoc for the source code of our implementation. This source code is available online at `https://github.com/jss-de/drools-checker`.

B.1. Package de.jss.drools

Package Contents *Page*

Classes

Class CLI

The command line interface for the application.

Declaration

public class CLI
extends java.lang.Object

Constructor summary

CLI()

Method summary

main(String[]) The main entry point for the application.

Constructors

- **CLI**
  `public CLI()`

Methods

- **main**
  `public static void main(java.lang.String[] args)`

– **Description**

  The main entry point for the application.

– **Parameters**

  * `args` – The arguments for the command line interface.

B.2. Package de.jss.drools.analysis

Package Contents *Page*

Classes

Class INTTRSReporter

Analyzes the provided `Package` and generates an INTTRS.

Declaration

public class INTTRSReporter
extends de.jss.drools.analysis.PackageReporter (in B.2, page 105)

Constructor summary

INTTRSReporter()

Method summary

report(OutputStream, Package) Generates the INTTRS for the specified `Package` and writes it to the specified `OutputStream`.

Constructors

- **INTTRSReporter**
  `public INTTRSReporter()`

Methods

- **report**
  `public void report(java.io.OutputStream outputStream, de.jss.drools.lang.Package pkg)`

– **Description**

Generates the INTTRS for the specified `Package` and writes it to the specified `OutputStream`.

– **Parameters**

* `outputStream` – The `OutputStream` to write the INTTRS to.
* `pkg` – The `Package` to generate the INTTRS for.

Members inherited from class PackageReporter

`de.jss.drools.analysis.PackageReporter` (in B.2, page 105)

- `public abstract void` **`report`**`(java.io.OutputStream` **`outputStream`**`, de.jss.drools.lang.Package` **`pkg`**`)`

Class PackageReporter

`PackageReporter` is the abstract base class for all package reporters.

Declaration

public abstract class PackageReporter
extends java.lang.Object

All known subclasses

INTTRSReporter (in B.2, page 104)

Constructor summary

PackageReporter()

Method summary

report(OutputStream, Package) Analyzes the specified `Package` and writes a report to the specified `OutputStream`.

Constructors

- **PackageReporter**
  `public` **`PackageReporter`**`()`

Methods

- **report**
  `public abstract void` **`report`**`(java.io.OutputStream` **`outputStream`**`, de.jss.drools.lang.Package` **`pkg`**`)`
  - **Description**
    Analyzes the specified `Package` and writes a report to the specified `OutputStream`.
  - **Parameters**
    - `outputStream` – The `OutputStream` to write the report to.
    - `pkg` – The `Package` to analyze.

B.3. Package de.jss.drools.compiler

Package Contents *Page*

Class CodeGenerator

`CodeGenerator` is the abstract base class for all code generators.

Declaration

public abstract class CodeGenerator
extends java.lang.Object

All known subclasses

XMLGenerator (in B.3, page 110)

Constructor summary

CodeGenerator()

Method summary

generate(OutputStream, Package) Generates code for the specified `Package` and writes it to the specified `OutputStream`.

Constructors

- **CodeGenerator**
  `public` **`CodeGenerator`**`()`

Methods

- **generate**
  `public abstract void` **`generate`**`(java.io.OutputStream` **`outputStream`**`, de.jss.drools.lang.Package` **`pkg`**`) throws de.jss.drools.compiler.CodeGeneratorException`
  - **Description**
    Generates code for the specified `Package` and writes it to the specified `OutputStream`.
  - **Parameters**
    * `outputStream` – The `OutputStream` to write code to.
    * `pkg` – The `Package` to generate code for.
  - **Throws**
    * `de.jss.drools.compiler.CodeGeneratorException` – Indicates that an error occurred while generating code.

Class CodeParser

`CodeParser` is the abstract base class for all code parsers.

Declaration

public abstract class CodeParser
extends java.lang.Object

All known subclasses

DRLParser (in B.3, page 108)

Constructor summary

CodeParser()

Method summary

parse(InputStream) Parses the code from the specified `InputStream` into a `Package`.

Constructors

- **CodeParser**
  `public CodeParser()`

Methods

- **parse**
  `public abstract de.jss.drools.lang.Package parse( java.io.InputStream inputStream) throws de.jss.drools.compiler.CodeParserException`
  - **Description**

    Parses the code from the specified `InputStream` into a `Package`.
  - **Parameters**
    * `inputStream` – The `InputStream` to read code from.
  - **Returns** – The `Package` parsed from the code.
  - **Throws**
    * `de.jss.drools.compiler.CodeParserException` – Indicates that an error occurred while parsing code.

Class DRLParser

Parses DRL into `Package` representation.

Declaration

public class DRLParser
extends de.jss.drools.compiler.CodeParser (in B.3, page 107)

Constructor summary

DRLParser() Initializes a new instance of the `DRLParser` class.
DRLParser(ClassLoader[]) Initializes a new instance of the `DRLParser` class using the specified class loaders.
DRLParser(KnowledgeBuilderConfigurationImpl) Initializes a new instance of the `DRLParser` class using the specified configuration.

Method summary

parse(InputStream) Parses the DRL from the specified `InputStream` into a `Package`.
parse(PackageDescr) Parses the DRL from the specified `PackageDescr` into a `Package`.
parse(Reader) Parses the DRL from the specified `Reader` into a `Package`.

Constructors

- **DRLParser**
  `public DRLParser()`
  - **Description**
    Initializes a new instance of the `DRLParser` class.
- **DRLParser**
  `public DRLParser(java.lang.ClassLoader[] classLoaders)`
  - **Description**
    Initializes a new instance of the `DRLParser` class using the specified class loaders.
  - **Parameters**
    * `classLoaders` – The class loaders to use.
- **DRLParser**
  `public DRLParser(`
  `org.drools.compiler.builder.impl.KnowledgeBuilderConfigurationImpl`
  `configuration)`
  - **Description**
    Initializes a new instance of the `DRLParser` class using the specified configuration.
  - **Parameters**
    * `configuration` – The configuration to use.

Methods

- **parse**
  `public de.jss.drools.lang.Package parse(java.io.InputStream inputStream) throws de.jss.drools.compiler.CodeParserException`
  - **Description**
    Parses the DRL from the specified `InputStream` into a `Package`.
  - **Parameters**

    * `inputStream` – The `InputStream` to read DRL from.
  - **Returns** – The `Package` parsed from the DRL.
  - **Throws**
    * `de.jss.drools.compiler.CodeParserException` – Indicates that an error occurred while parsing DRL.

- **parse**
  `public de.jss.drools.lang.Package` **`parse`**`( org.drools.compiler.lang.descr.PackageDescr` **`descr`**`) throws de.jss.drools.compiler.CodeParserException`
  - **Description**
    Parses the DRL from the specified `PackageDescr` into a `Package`.
  - **Parameters**
    * `descr` – The `PackageDescr` to read DRL from.
  - **Returns** – The `Package` parsed from the DRL.
  - **Throws**
    * `de.jss.drools.compiler.CodeParserException` – Indicates that an error occurred while parsing DRL.

- **parse**
  `public de.jss.drools.lang.Package` **`parse`**`(java.io.Reader` **`reader`**`) throws de.jss.drools.compiler.CodeParserException`
  - **Description**
    Parses the DRL from the specified `Reader` into a `Package`.
  - **Parameters**
    * `reader` – The `Reader` to read DRL from.
  - **Returns** – The `Package` parsed from the DRL.
  - **Throws**
    * `de.jss.drools.compiler.CodeParserException` – Indicates that an error occurred while parsing DRL.

Members inherited from class CodeParser

`de.jss.drools.compiler.CodeParser` (in B.3, page 107)

- `public abstract Package` **`parse`**`(java.io.InputStream` **`inputStream`**`) throws CodeParserException`

Class XMLGenerator

Generates XML for `Package` representations.

Declaration

public class XMLGenerator
extends de.jss.drools.compiler.CodeGenerator (in B.3, page 106)

Constructor summary

XMLGenerator()

Method summary

generate(OutputStream, Package) Generates XML for the specified `Package` and writes it to the specified `OutputStream`.

Constructors

- **XMLGenerator**
  `public XMLGenerator()`

Methods

- **generate**
  `public void generate(java.io.OutputStream outputStream, de.jss.drools.lang.Package pkg) throws de.jss.drools.compiler.CodeGeneratorException`
  - **Description**
    Generates XML for the specified `Package` and writes it to the specified `OutputStream`.
  - **Parameters**
    * `outputStream` – The `OutputStream` to write XML to.
    * `pkg` – The `Package` to generate XML for.
  - **Throws**
    * `de.jss.drools.compiler.CodeGeneratorException` – Indicates that an error occurred while generating XML.

Members inherited from class CodeGenerator

`de.jss.drools.compiler.CodeGenerator` (in B.3, page 106)

- `public abstract void generate(java.io.OutputStream outputStream, de.jss.drools.lang.Package pkg) throws CodeGeneratorException`

Exception CodeGeneratorException

Thrown to indicate that an error occurred while generating code.

Declaration

public class CodeGeneratorException
extends java.lang.Exception

Constructor summary

CodeGeneratorException() Please refer to .
CodeGeneratorException(String) Please refer to .
CodeGeneratorException(String, Throwable) Please refer to .
CodeGeneratorException(Throwable) Please refer to .

Constructors

- **CodeGeneratorException**
  `public CodeGeneratorException()`
  - **Description**

    Please refer to .
  - **See also**
    * `java.lang.Exception()`
- **CodeGeneratorException**
  `public CodeGeneratorException(java.lang.String message)`
  - **Description**

    Please refer to .
  - **See also**
    * `java.lang.Exception(String)`
- **CodeGeneratorException**
  `public CodeGeneratorException(java.lang.String message, java.lang.Throwable cause)`
  - **Description**

    Please refer to .
  - **See also**
    * `java.lang.Exception(String,Throwable)`
- **CodeGeneratorException**
  `public CodeGeneratorException(java.lang.Throwable cause)`
  - **Description**

    Please refer to .

- **See also**
  - `java.lang.Exception(Throwable)`

Members inherited from class Throwable

`java.lang.Throwable`

- `public final synchronized void` **`addSuppressed`**`(Throwable` **`arg0`**`)`
- `public synchronized Throwable` **`fillInStackTrace`**`()`
- `public synchronized Throwable` **`getCause`**`()`
- `public String` **`getLocalizedMessage`**`()`
- `public String` **`getMessage`**`()`
- `public StackTraceElement` **`getStackTrace`**`()`
- `public final synchronized Throwable` **`getSuppressed`**`()`
- `public synchronized Throwable` **`initCause`**`(Throwable` **`arg0`**`)`
- `public void` **`printStackTrace`**`()`
- `public void` **`printStackTrace`**`(java.io.PrintStream` **`arg0`**`)`
- `public void` **`printStackTrace`**`(java.io.PrintWriter` **`arg0`**`)`
- `public void` **`setStackTrace`**`(StackTraceElement[]` **`arg0`**`)`
- `public String` **`toString`**`()`

Exception CodeParserException

Thrown to indicate that an error occurred while parsing code.

Declaration

public class CodeParserException
extends java.lang.Exception

Constructor summary

CodeParserException() Please refer to .
CodeParserException(String) Please refer to .
CodeParserException(String, Throwable) Please refer to .
CodeParserException(Throwable) Please refer to .

Constructors

- **CodeParserException**
  `public` **`CodeParserException`**`()`
  - **Description**
    Please refer to .
  - **See also**
    - `java.lang.Exception()`

- **CodeParserException**
  `public` **`CodeParserException`**`(java.lang.String` **`message`**`)`
  - **Description**

    Please refer to .
  - **See also**
    * `java.lang.Exception(String)`
- **CodeParserException**
  `public` **`CodeParserException`**`(java.lang.String` **`message`**`,`
  `java.lang.Throwable` **`cause`**`)`
  - **Description**

    Please refer to .
  - **See also**
    * `java.lang.Exception(String,Throwable)`
- **CodeParserException**
  `public` **`CodeParserException`**`(java.lang.Throwable` **`cause`**`)`
  - **Description**

    Please refer to .
  - **See also**
    * `java.lang.Exception(Throwable)`

Members inherited from class Throwable

`java.lang.Throwable`

- `public final synchronized void` **`addSuppressed`**`(Throwable` **`arg0`**`)`
- `public synchronized Throwable` **`fillInStackTrace`**`()`
- `public synchronized Throwable` **`getCause`**`()`
- `public String` **`getLocalizedMessage`**`()`
- `public String` **`getMessage`**`()`
- `public StackTraceElement` **`getStackTrace`**`()`
- `public final synchronized Throwable` **`getSuppressed`**`()`
- `public synchronized Throwable` **`initCause`**`(Throwable` **`arg0`**`)`
- `public void` **`printStackTrace`**`()`
- `public void` **`printStackTrace`**`(java.io.PrintStream` **`arg0`**`)`
- `public void` **`printStackTrace`**`(java.io.PrintWriter` **`arg0`**`)`
- `public void` **`setStackTrace`**`(StackTraceElement[]` **`arg0`**`)`
- `public String` **`toString`**`()`

B.4. Package de.jss.drools.lang

Package Contents *Page*

Interface Condition

Provides a marker interface for conditions.

Declaration

public interface Condition

All known subinterfaces

ConditionConnective (in B.4, page 126), Pattern (in B.4, page 136)

All classes known to implement interface

ConditionConnective (in B.4, page 126), Pattern (in B.4, page 136)

Method summary

clone() Creates and returns a deep copy of the condition.

Methods

- **clone**
  `Condition clone()`
  - **Description**
    Creates and returns a deep copy of the condition.
  - **Returns** – A deep copy of the condition.

Interface Consequence

Provides a marker interface for consequences.

Declaration

public interface Consequence
extends java.lang.Cloneable

All known subinterfaces

Action (in B.4, page 118), Message (in B.4, page 132)

All classes known to implement interface

Action (in B.4, page 118), Message (in B.4, page 132)

Method summary

clone() Creates and returns a deep copy of the consequence.

Methods

- **clone**
  `Consequence clone()`
  - **Description**
    Creates and returns a deep copy of the consequence.
  - **Returns** – A deep copy of the consequence.

Interface Constraint

Provides a marker interface for constraints.

Declaration

public interface Constraint
extends java.lang.Cloneable

All known subinterfaces

ConstraintConnective (in B.4, page 128), UnknownConstraint (in B.4, page 141), Attribute-Constraint (in B.4, page 123)

All classes known to implement interface

ConstraintConnective (in B.4, page 128), UnknownConstraint (in B.4, page 141), Attribute-Constraint (in B.4, page 123)

Method summary

clone() Creates and returns a deep copy of the constraint.

Methods

- **clone**
  `Constraint clone()`
  - **Description**
    Creates and returns a deep copy of the constraint.

– **Returns** – A deep copy of the constraint.

Class Action

Represents an action which changes the working memory.

Declaration

public class Action
extends java.lang.Object
implements java.lang.Cloneable, Consequence

Constructor summary

Action(String, String, ActionType) Initializes a new instance of the `Action` class using the specified data.

Method summary

clone() Creates and returns a deep copy of the action.
getAssignments() Gets the assignments of the action.
getFactTypeName() Gets the name of the fact type to which the action refers to.
getPatternName() Gets the name of the pattern to which the action refers to.
getType() Gets the type of the action.

Constructors

- **Action**
  `public` **`Action`**`(java.lang.String` **`factTypeName,`**
  `java.lang.String` **`patternName,`** `ActionType` **`type`**`)`
  - **Description**

    Initializes a new instance of the `Action` class using the specified data.
  - **Parameters**
    * `factTypeName` – The name of the fact type to which the new action refers to.
    * `patternName` – The name of the pattern to which the new action refers to.
    * `type` – The type of the new action.

Methods

- **clone**
  `public Action clone()`
  - **Description**

    Creates and returns a deep copy of the action.
  - **Returns** – A deep copy of the action.
- **getAssignments**
  `public java.util.List getAssignments()`
  - **Description**

    Gets the assignments of the action.
  - **Returns** – The assignments of the action.
- **getFactTypeName**
  `public java.lang.String getFactTypeName()`
  - **Description**

    Gets the name of the fact type to which the action refers to.
  - **Returns** – The name of the fact type to which the action refers to.
- **getPatternName**
  `public java.lang.String getPatternName()`
  - **Description**

    Gets the name of the pattern to which the action refers to.
  - **Returns** – The name of the pattern to which the action refers to.
- **getType**
  `public ActionType getType()`
  - **Description**

    Gets the type of the action.
  - **Returns** – The type of the action.

Class ActionType

Specifies the type of action in the associated (in B.4, page 118) instance.

Declaration

public final class ActionType
extends java.lang.Enum

Field summary

Insertion Inserts a new fact into the working memory.
Modification Modifies a fact in the working memory.
Retraction Retracts a fact from the working memory.

Method summary

valueOf(String)
values()

Fields

- public static final ActionType **Insertion**
  - Inserts a new fact into the working memory.
- public static final ActionType **Modification**
  - Modifies a fact in the working memory.
- public static final ActionType **Retraction**
  - Retracts a fact from the working memory.

Methods

- **valueOf**
  `public static ActionType valueOf(java.lang.String name)`
- **values**
  `public static ActionType[] values()`

Members inherited from class Enum

`java.lang.Enum`

- `protected final Object clone() throws CloneNotSupportedException`
- `public final int compareTo(Enum arg0)`
- `public final boolean equals(Object arg0)`
- `protected final void finalize()`
- `public final Class getDeclaringClass()`
- `public final int hashCode()`
- `public final String name()`
- `public final int ordinal()`
- `public String toString()`
- `public static Enum valueOf(Class arg0, String arg1)`

Class Assignment

Represents an assignment which changes the value of an attribute of a fact.

Declaration

public class Assignment
extends java.lang.Object
implements java.lang.Cloneable

Constructor summary

Assignment(String, String) Initializes a new instance of the `Assignment` class using the specified data.

Method summary

clone() Creates and returns a deep copy of the assignment.
getAttributeName() Gets the name of the attribute to which the assignment refers to.
getExpression() Gets the expression of the assignment.

Constructors

- **Assignment**
  `public` **`Assignment`**`(java.lang.String` **`attributeName`**`,`
  `java.lang.String` **`expression`**`)`
  - **Description**

    Initializes a new instance of the `Assignment` class using the specified data.
  - **Parameters**
    * `attributeName` – The name of the attribute to which the new assignment refers to.
    * `expression` – The expression of the new assignment.

Methods

- **clone**
  `public Assignment` **`clone`**`()`
  - **Description**

    Creates and returns a deep copy of the assignment.
  - **Returns** – A deep copy of the assignment.
- **getAttributeName**
  `public java.lang.String` **`getAttributeName`**`()`
  - **Description**

    Gets the name of the attribute to which the assignment refers to.

  - **Returns** – The name of the attribute to which the assignment refers to.

- **getExpression**
  `public java.lang.String getExpression()`
  - **Description**

    Gets the expression of the assignment.
  - **Returns** – The expression of the assignment.

Class Attribute

Represents the definition of an attribute of a type.

Declaration

public class Attribute
extends java.lang.Object
implements java.lang.Cloneable

Constructor summary

Attribute(String, String) Initializes a new instance of the `Attribute` class using the specified data.

Method summary

clone() Creates and returns a deep copy of the attribute.
getName() Gets the name of the attribute.
getType() Gets the type of the attribute.

Constructors

- **Attribute**
  `public Attribute(java.lang.String name, java.lang.String type)`
  - **Description**

    Initializes a new instance of the `Attribute` class using the specified data.
  - **Parameters**
    * `name` – The name of the new attribute.
    * `type` – The type of the new attribute.

Methods

- **clone**
  `public Attribute **clone**()`
  - **Description**

    Creates and returns a deep copy of the attribute.
  - **Returns** – A deep copy of the attribute.
- **getName**
  `public java.lang.String **getName**()`
  - **Description**

    Gets the name of the attribute.
  - **Returns** – The name of the attribute.
- **getType**
  `public java.lang.String **getType**()`
  - **Description**

    Gets the type of the attribute.
  - **Returns** – The type of the attribute.

Class AttributeConstraint

Represents a relation.

Declaration

public class AttributeConstraint
extends java.lang.Object
implements java.lang.Cloneable, Constraint

Constructor summary

AttributeConstraint(String, String, String) Initializes a new instance of the `Relation` class using the specified data.

Method summary

clone() Creates and returns a deep copy of the relation.
getAttributeName() Gets the attributeName of the AttributeConstraint.
getExpression() Gets the expression of the AttributeConstraint.
getRelation() Gets the type of the relation.

Constructors

- **AttributeConstraint**
  `public` **`AttributeConstraint`**`(java.lang.String` **`attributeName,`** `java.lang.String` **`relation,`** `java.lang.String` **`expression`**`)`
  - **Description**

    Initializes a new instance of the `Relation` class using the specified data.
  - **Parameters**
    * `type` – The type of the new relation.
    * `value1` – The left value of the new relation.
    * `value2` – The right value of the new relation.

Methods

- **clone**
  `public AttributeConstraint` **`clone`**`()`
  - **Description**

    Creates and returns a deep copy of the relation.
  - **Returns** – A deep copy of the relation.
- **getAttributeName**
  `public java.lang.String` **`getAttributeName`**`()`
  - **Description**

    Gets the attributeName of the AttributeConstraint.
  - **Returns** – The attributeName of the AttributeConstraint.
- **getExpression**
  `public java.lang.String` **`getExpression`**`()`
  - **Description**

    Gets the expression of the AttributeConstraint.
  - **Returns** – The expression of the AttributeConstraint.
- **getRelation**
  `public java.lang.String` **`getRelation`**`()`
  - **Description**

    Gets the type of the relation.
  - **Returns** – The type of the relation.

Class Binding

Represents the definition of a binding.

Declaration

public class Binding
extends java.lang.Object
implements java.lang.Cloneable

Constructor summary

Binding(String, String) Initializes a new instance of the `Binding` class using the specified data.

Method summary

clone() Creates and returns a deep copy of the binding.
getName() Gets the name of the binding.
getValue() Gets the value of the binding.

Constructors

- **Binding**
  `public Binding(java.lang.String name, java.lang.String value)`
  - **Description**
    Initializes a new instance of the `Binding` class using the specified data.
  - **Parameters**
    * `name` – The name of the new binding.
    * `value` – The value of the new binding.

Methods

- **clone**
  `public Binding clone()`
  - **Description**
    Creates and returns a deep copy of the binding.
  - **Returns** – A deep copy of the binding.
- **getName**
  `public java.lang.String getName()`

- **Description**

  Gets the name of the binding.
- **Returns** – The name of the binding.

- **getValue**
  `public java.lang.String getValue()`

  - **Description**

    Gets the value of the binding.
  - **Returns** – The value of the binding.

Class ConditionConnective

Represents a connective of conditions of a rule.

Declaration

public class ConditionConnective
extends java.lang.Object
implements java.lang.Cloneable, Condition

Constructor summary

ConditionConnective(ConditionConnectiveType) Initializes a new instance of the `ConditionConnective` class with the specified type.

Method summary

clone() Creates and returns a deep copy of the connective.
getConditions() Gets the connected conditions.
getType() Gets the type of the connective.

Constructors

- **ConditionConnective**
  `public ConditionConnective(ConditionConnectiveType type)`

  - **Description**

    Initializes a new instance of the `ConditionConnective` class with the specified type.
  - **Parameters**
    - `type` – The type of the new connective.

Methods

- **clone**
  `public ConditionConnective clone()`
  - **Description**
    Creates and returns a deep copy of the connective.
  - **Returns** – A deep copy of the connective.
- **getConditions**
  `public java.util.List getConditions()`
  - **Description**
    Gets the connected conditions.
  - **Returns** – The connected conditions.
- **getType**
  `public ConditionConnectiveType getType()`
  - **Description**
    Gets the type of the connective.
  - **Returns** – The type of the connective.

Class ConditionConnectiveType

Specifies the type of connective in the associated (in B.4, page 126) instance.

Declaration

public final class ConditionConnectiveType
extends java.lang.Enum

Field summary

Conjunction Connects conditions by the means of '\bigwedge'.
Disjunction Connects conditions by the means of '\bigvee'.
Negation Connects conditions by the means of '\neg \exists'.

Method summary

valueOf(String)
values()

Fields

- public static final ConditionConnectiveType **Conjunction**
  - Connects conditions by the means of '\bigwedge'.
- public static final ConditionConnectiveType **Disjunction**
  - Connects conditions by the means of '\bigvee'.
- public static final ConditionConnectiveType **Negation**
  - Connects conditions by the means of '\neg \exists'.

Methods

- **valueOf**
  `public static ConditionConnectiveType` **`valueOf`**`(`
  `java.lang.String` **`name`**`)`
- **values**
  `public static ConditionConnectiveType[]` **`values`**`()`

Members inherited from class Enum

`java.lang.Enum`

- `protected final Object` **`clone`**`() throws CloneNotSupportedException`
- `public final int` **`compareTo`**`(Enum` **`arg0`**`)`
- `public final boolean` **`equals`**`(Object` **`arg0`**`)`
- `protected final void` **`finalize`**`()`
- `public final Class` **`getDeclaringClass`**`()`
- `public final int` **`hashCode`**`()`
- `public final String` **`name`**`()`
- `public final int` **`ordinal`**`()`
- `public String` **`toString`**`()`
- `public static Enum` **`valueOf`**`(Class` **`arg0`**`, String` **`arg1`**`)`

Class ConstraintConnective

Represents a connective of constraints of a pattern.

Declaration

public class ConstraintConnective
extends java.lang.Object
implements java.lang.Cloneable, Constraint

Constructor summary

ConstraintConnective(ConstraintConnectiveType) Initializes a new instance of the `ConstraintConnective` class with the specified type.

Method summary

clone() Creates and returns a deep copy of the connective.
getConstraints() Gets the connected constraints.
getType() Gets the type of the connective.

Constructors

- **ConstraintConnective**
  `public` **`ConstraintConnective`**`(ConstraintConnectiveType` **`type`**`)`
  - **Description**

    Initializes a new instance of the `ConstraintConnective` class with the specified type.
  - **Parameters**
    * `type` – The type of the new connective.

Methods

- **clone**
  `public ConstraintConnective` **`clone`**`()`
  - **Description**

    Creates and returns a deep copy of the connective.
  - **Returns** – A deep copy of the connective.
- **getConstraints**
  `public java.util.List` **`getConstraints`**`()`
  - **Description**

    Gets the connected constraints.
  - **Returns** – The connected constraints.
- **getType**
  `public ConstraintConnectiveType` **`getType`**`()`
  - **Description**

    Gets the type of the connective.
  - **Returns** – The type of the connective.

Class ConstraintConnectiveType

Specifies the type of connective in the associated (in B.4, page 128) instance.

Declaration

public final class ConstraintConnectiveType
extends java.lang.Enum

Field summary

Conjunction Connects constraints by the means of '\bigwedge'.
Disjunction Connects constraints by the means of '\bigvee'.
Negation Connects constraints by the means of '\neg \exists'.

Method summary

valueOf(String)
values()

Fields

- public static final ConstraintConnectiveType **Conjunction**
  - Connects constraints by the means of '\bigwedge'.
- public static final ConstraintConnectiveType **Disjunction**
  - Connects constraints by the means of '\bigvee'.
- public static final ConstraintConnectiveType **Negation**
  - Connects constraints by the means of '\neg \exists'.

Methods

- **valueOf**
  `public static ConstraintConnectiveType valueOf( java.lang.String name)`
- **values**
  `public static ConstraintConnectiveType[] values()`

Members inherited from class Enum

`java.lang.Enum`

- `protected final Object clone() throws CloneNotSupportedException`
- `public final int compareTo(Enum arg0)`
- `public final boolean equals(Object arg0)`
- `protected final void finalize()`
- `public final Class getDeclaringClass()`
- `public final int hashCode()`
- `public final String name()`
- `public final int ordinal()`
- `public String toString()`
- `public static Enum valueOf(Class arg0, String arg1)`

Class Global

Represents the definition of a global.

Declaration

public class Global
extends java.lang.Object
implements java.lang.Cloneable

Constructor summary

Global(String, String) Initializes a new instance of the `Global` class using the specified data.

Method summary

clone() Creates and returns a deep copy of the global.
getName() Gets the name of the global.
getType() Gets the type of the global.

Constructors

- **Global**
  `public Global(java.lang.String name, java.lang.String type)`
  - **Description**
    Initializes a new instance of the `Global` class using the specified data.
  - **Parameters**
    * `name` – The name of the new global.
    * `type` – The type of the new global.

Methods

- **clone**
  `public Global clone()`
  - **Description**
    Creates and returns a deep copy of the global.
  - **Returns** – A deep copy of the global.
- **getName**
  `public java.lang.String getName()`

- **Description**

  Gets the name of the global.
- **Returns** – The name of the global.

- **getType**
  `public java.lang.String getType()`
  - **Description**

    Gets the type of the global.
  - **Returns** – The type of the global.

Class Message

Represents a message which does not change the working memory.

Declaration

public class Message
extends java.lang.Object
implements java.lang.Cloneable, Consequence

Constructor summary

Message(String) Initializes a new instance of the `Message` class with the specified value.

Method summary

clone() Creates and returns a deep copy of the message.
getValue() Gets the value of the message.

Constructors

- **Message**
  `public Message(java.lang.String value)`
  - **Description**

    Initializes a new instance of the `Message` class with the specified value.
  - **Parameters**
    - `value` – The value of the new message.

Methods

- **clone**
  `public Message clone()`
  - **Description**

    Creates and returns a deep copy of the message.
  - **Returns** – A deep copy of the message.
- **getValue**
  `public java.lang.String getValue()`
  - **Description**

    Gets the value of the message.
  - **Returns** – The value of the message.

Class Package

Represents a package.

Declaration

public class Package
extends java.lang.Object
implements java.lang.Cloneable

Constructor summary

Package(String) Initializes a new instance of the `Package` class with the specified name.

Method summary

clone() Creates and returns a deep copy of the package.
getFactType(String) Gets the fact type with the specified name.
getFactTypes() Gets the fact types of the package.
getGlobal(String) Gets the global with the specified name.
getGlobals() Gets the globals of the package.
getName() Gets the name of the package.
getRule(String) Gets the rule with the specified name.
getRules() Gets the rules of the package.
hasFactType(String) Checks whether the package contains a fact type with the specified name.
hasGlobal(String) Checks whether the package contains a global with the specified name.

hasRule(String) Checks whether the package contains a rule with the specified name.

Constructors

- **Package**
  `public Package(java.lang.String name)`
  - **Description**
    Initializes a new instance of the `Package` class with the specified name.
  - **Parameters**
    * `name` – The name of the new package.

Methods

- **clone**
  `public Package clone()`
  - **Description**
    Creates and returns a deep copy of the package.
  - **Returns** – A deep copy of the package.
- **getFactType**
  `public Type getFactType(java.lang.String name)`
  - **Description**
    Gets the fact type with the specified name.
  - **Parameters**
    * `name` – The name to search for.
  - **Returns** – The fact type with the specified name.
- **getFactTypes**
  `public java.util.List getFactTypes()`
  - **Description**
    Gets the fact types of the package.
  - **Returns** – The fact types of the package.
- **getGlobal**
  `public Global getGlobal(java.lang.String name)`
  - **Description**
    Gets the global with the specified name.
  - **Parameters**

    * `name` – The name to search for.
  - **Returns** – The global with the specified name.

- **getGlobals**
  `public java.util.List getGlobals()`
  - **Description**
    Gets the globals of the package.
  - **Returns** – The globals of the package.

- **getName**
  `public java.lang.String getName()`
  - **Description**
    Gets the name of the package.
  - **Returns** – The name of the package.

- **getRule**
  `public Rule getRule(java.lang.String name)`
  - **Description**
    Gets the rule with the specified name.
  - **Parameters**
    * `name` – The name to search for.
  - **Returns** – The rule with the specified name.

- **getRules**
  `public java.util.List getRules()`
  - **Description**
    Gets the rules of the package.
  - **Returns** – The rules of the package.

- **hasFactType**
  `public boolean hasFactType(java.lang.String name)`
  - **Description**
    Checks whether the package contains a fact type with the specified name.
  - **Parameters**
    * `name` – The name to search for.
  - **Returns** – true if a fact type with the specified name was found; otherwise false.

- **hasGlobal**
  `public boolean **hasGlobal**(java.lang.String **name**)`
  - **Description**
    Checks whether the package contains a global with the specified name.
  - **Parameters**
    * `name` – The name to search for.
  - **Returns** – true if a global with the specified name was found; otherwise false.
- **hasRule**
  `public boolean **hasRule**(java.lang.String **name**)`
  - **Description**
    Checks whether the package contains a rule with the specified name.
  - **Parameters**
    * `name` – The name to search for.
  - **Returns** – true if a rule with the specified name was found; otherwise false.

Class Pattern

Represents a pattern.

Declaration

public class Pattern
extends java.lang.Object
implements java.lang.Cloneable, Condition

Constructor summary

Pattern(String) Initializes a new instance of the `Pattern` class using the specified data.

Method summary

clone() Creates and returns a deep copy of the pattern.
getBindings() Gets the bindings of the pattern.
getConstraints() Gets the constraints of the pattern.
getOuterBinding() Gets the outerBinding of the Pattern.
getTypeName() Gets the name of the fact type to which the pattern refers to.
setOuterBinding(String) Sets the outerBinding of the Pattern.

Constructors

- **Pattern**
  `public **Pattern**(java.lang.String **typeName**)`
  - **Description**

    Initializes a new instance of the `Pattern` class using the specified data.
  - **Parameters**
    * `typeName` – The name of the type to which the new pattern refers to.

Methods

- **clone**
  `public Pattern **clone**()`
  - **Description**

    Creates and returns a deep copy of the pattern.
  - **Returns** – A deep copy of the pattern.
- **getBindings**
  `public java.util.List **getBindings**()`
  - **Description**

    Gets the bindings of the pattern.
  - **Returns** – The bindings of the pattern.
- **getConstraints**
  `public java.util.List **getConstraints**()`
  - **Description**

    Gets the constraints of the pattern.
  - **Returns** – The constraints of the pattern.
- **getOuterBinding**
  `public java.lang.String **getOuterBinding**()`
  - **Description**

    Gets the outerBinding of the Pattern.
  - **Returns** – The outerBinding of the Pattern.
- **getTypeName**
  `public java.lang.String **getTypeName**()`
  - **Description**

    Gets the name of the fact type to which the pattern refers to.

  - **Returns** – The name of the fact type to which the pattern refers to.

- **setOuterBinding**
  `public void setOuterBinding(java.lang.String outerBinding)`
  - **Description**
    Sets the outerBinding of the Pattern.
  - **Parameters**
    * `outerBinding` – The new outerBinding of the Pattern.

Class Rule

Represents a rule.

Declaration

public class Rule
extends java.lang.Object
implements java.lang.Cloneable

Constructor summary

Rule(String) Initializes a new instance of the `Rule` class with the specified name.

Method summary

clone() Creates and returns a deep copy of the rule.
getConditions() Gets the conditions of the rule.
getConsequences() Gets the consequences of the rule.
getName() Gets the name of the rule.

Constructors

- **Rule**
  `public Rule(java.lang.String name)`
  - **Description**
    Initializes a new instance of the `Rule` class with the specified name.
  - **Parameters**
    * `name` – The name of the new rule.

Methods

- **clone**
  `public Rule **clone**()`
  - **Description**

    Creates and returns a deep copy of the rule.
  - **Returns** – A deep copy of the rule.
- **getConditions**
  `public java.util.List **getConditions**()`
  - **Description**

    Gets the conditions of the rule.
  - **Returns** – The conditions of the rule.
- **getConsequences**
  `public java.util.List **getConsequences**()`
  - **Description**

    Gets the consequences of the rule.
  - **Returns** – The consequences of the rule.
- **getName**
  `public java.lang.String **getName**()`
  - **Description**

    Gets the name of the rule.
  - **Returns** – The name of the rule.

Class Type

Represents the definition of a fact type.

Declaration

public class Type
extends java.lang.Object
implements java.lang.Cloneable

Constructor summary

Type(String) Initializes a new instance of the `FactType` class with the specified name.

Method summary

clone() Creates and returns a deep copy of the fact type.
getAttribute(String) Gets the attribute with the specified name.
getAttributes() Gets the attributes of the fact type.
getName() Gets the name of the fact type.
hasAttribute(String) Checks whether the fact type contains an attribute with the specified name.

Constructors

- **Type**
  `public **Type**(java.lang.String **name**)`
  - **Description**

    Initializes a new instance of the `FactType` class with the specified name.
  - **Parameters**
    * `name` – The name of the new fact type.

Methods

- **clone**
  `public Type **clone**()`
  - **Description**

    Creates and returns a deep copy of the fact type.
  - **Returns** – A deep copy of the fact type.
- **getAttribute**
  `public Attribute **getAttribute**(java.lang.String **name**)`
  - **Description**

    Gets the attribute with the specified name.
  - **Parameters**
    * `name` – The name to search for.
  - **Returns** – The attribute with the specified name.
- **getAttributes**
  `public java.util.List **getAttributes**()`
  - **Description**

    Gets the attributes of the fact type.
  - **Returns** – The attributes of the fact type.

- **getName**
  `public java.lang.String getName()`
  - **Description**
    Gets the name of the fact type.
  - **Returns** – The name of the fact type.
- **hasAttribute**
  `public boolean hasAttribute(java.lang.String name)`
  - **Description**
    Checks whether the fact type contains an attribute with the specified name.
  - **Parameters**
    * `name` – The name to search for.
  - **Returns** – true if an attribute with the specified name was found; otherwise false.

Class UnknownConstraint

Represents an unknown constraint.

Declaration

public class UnknownConstraint
extends java.lang.Object
implements Constraint

Constructor summary

UnknownConstraint()

Method summary

clone() Creates and returns a deep copy of the constraint.

Constructors

- **UnknownConstraint**
  `public UnknownConstraint()`

Methods

- **clone**
  `public Constraint **clone**()`

  - **Description**

    Creates and returns a deep copy of the constraint.

  - **Returns** – A deep copy of the constraint.

Errata

The printed version of this thesis contains the following errors:

(1) On Page 30 in Rule (BindP) read $(o, \{v \mapsto o\})$ instead of $(o, \{v \mapsto o\}$.