A model for Reflection in Rule-Based Languages

Van De Water, Simon; De Meuter, Wolfgang; Renaux, Thierry

A Model for Reflection in Rule-Based Languages

Simon Van de Water, Thierry Renaux, Wolfgang De Meuter
Vrije Universiteit Brussel
Pleinlaan 2
Elsene, Belgium
{svdewate, trenaux, wdmeuter}@vub.ac.be

ABSTRACT

Production systems are used to detect patterns in large sets of facts. These patterns are described by rules.

Rules frequently need to be modified to adapt to changing requirements, for instance to update the security policies encoded in the rules. The production system however needs to remain operational throughout these updates. Current systems provide no reflective language constructs to inspect rules and/or change them. Instead, updates are achieved by unloading the current ruleset and loading an updated set. This is not only a costly operation which renders the system unresponsive while the new ruleset is being loaded, it also necessitates implementing the update-logic elsewhere.

In this position-paper we introduce a meta-level to RETE; a well established pattern-matching algorithm which is used in many production systems. This meta-level embodies a reification of all the rules in the ruleset, enabling us to provide language support for reflective rules. The envisioned language constructs make it possible to write rules that

(a) leverage introspection to reason about the state of business rules, and

(b) adapt to changing requirements by changing business rules at runtime.

CCS Concepts

•**Software and its engineering** → **Software Architectures;** *Layered Systems;* Real-time systems software;

Keywords

Rule-based languages, language design, reflection

1. INTRODUCTION

Recently, we see that businesses need to deal with large continuous streams of data. Financial institutions, for example, continuously monitor all payments, withdrawals, money transfers, etc. to uncover fraud by detecting suspicious patterns. In network security, all incoming and outgoing network packets are monitored in order to detect cyber-attacks, inappropriate use of company hardware, etc.

Production systems are used to detect patterns –in these potentially enormous amounts of data– with acceptable latency. A production system is a piece of software that relies on a set of condition-action pairs (also called rules) defined by the developer to reason over the large dataset. In the heart of the production system, a matching-algorithm is at work. This algorithm decides how and in which order incoming data is matched against the rules because this can have a tremendous impact on the performance of the production system.

Although the concepts presented in this paper are applicable to most rule-based systems, we will focus on production systems that run on top of the RETE matching-algorithm [4]. The relevant parts of this algorithm will be discussed in section 2.

The kind of data that needs to be reasoned over is constantly subject to change (e.g. cyber criminals find new ways to get illegitimate access to systems). Because of this, the patterns that need to be detected also evolve continuously which means that the rules have to be modified over time. While studying how current production systems deal with updating rulesets, we identify two problems.

First, the language constructs provided in current production systems to update rules at runtime are inadequate. In Drools [5] – an enterprise-strength production system – for example, the only way to change rules at runtime is by (un-)loading entire rulesets altogether. There is no way to modify a single rule without reloading an entire ruleset. To the best of our knowledge there is no system that has language support to do this.

In this paper we introduce a meta-level for rule-based languages in which rules are first class citizens. This enables developers to monitor and modify single rules or even the different components of a single rule without having to recompile and reload the entire ruleset.

The second problem we identify is that rules become outdated because they are not maintained. Instead of updating the ruleset appropriately, rules that are no longer (entirely) relevant are often disabled rather than updated to meet new circumstances. Simply disabling rules can lead to bugs and unintended behaviour as disabled rules can have

Meta 2016 Amsterdam, Netherlands

DOI: 10.475/123_4

an impact on other rules that depend on facts generated by a disabled rule. However, as illustrated later in this paper, there are some rules that are expected to change under certain conditions. For example, let us consider a rule that raises an alarm when a fraudulent sequence of transactions is detected. When the alarm is raised, a financial expert should have a more in-depth look at the suspicious sequence in order to confirm/deny the fraud. If the financial experts denies most of the suspicious sequences, it means that the rule generates a lot of false positives and that some of its parameters should be tweaked. We are not aware of any production system that offers language constructs to encode such rule-changes that can be anticipated.

The meta-level introduced in this paper can be used to create, modify and delete rules at runtime based on information coming in from external systems. Moreover, it will also be possible to query meta-information about rules such as how many times a certain rule was successfully matched over a certain period of time. Such information can be useful to decide whether a rule should be (de-)activated and can thus be used to write meta-rules that can anticipate and trigger change on certain conditions.

The remainder of this paper is structured as follows; section 2 provides a short introduction on Rule-based Languages. Section 3 is the core of this paper in which we explain the concepts we envision in order to solve the problems presented in the introduction. In section 4 we discuss earlier work on meta-rules as well as the most important inspiration for the ideas explained in this paper. Finally, we come to a conclusion in section 5.

2. RULE-BASED LANGUAGE

In this section we will briefly introduce Rule-based languages. The terminology and assumptions we make in this section are based on rule-based languages that run on top of the RETE-algorithm [4].
Rule-based languages can be used to detect patterns in (often large) sets of data. The data-set is represented by a number of `facts` that are continuously entering the system. Every fact (similar to an object) is an instance of a fact-type (similar to a class) with a number of instantiated fields. The patterns that need to be detected are expressed by rules.

Rules, which are typically made up of a Left-Hand Side (LHS) and a Right-Hand Side (RHS), are very similar to an if-then construct. The LHS is used to define the conditions that needs to be matched whereas the RHS is used to describe the consequent. An example of a rule can be found in listing 1.

```
1 rule "ProcessPurchasePremiumMember"
2   when
3     c: Client() as: client-condition
4     PremiumMember( clientId == c.clientId ) as: premium-
        condition
5     p: Purchase(amount > 25, ! paidFor, c.clientId ==
        clientId) as: purchase-condition
6   then
7     c.chargePurchase(p, 5);
8 end
```

Listing 1: Example of a rule called `ProcessPurchasePremiumMember`. The syntax is inspired by the Drools Rule Language.

This rule, which typically would be used by a store, expresses that a `Client` who is also a `PremiumMember` gets a reduction of 5% on purchases of over $25.

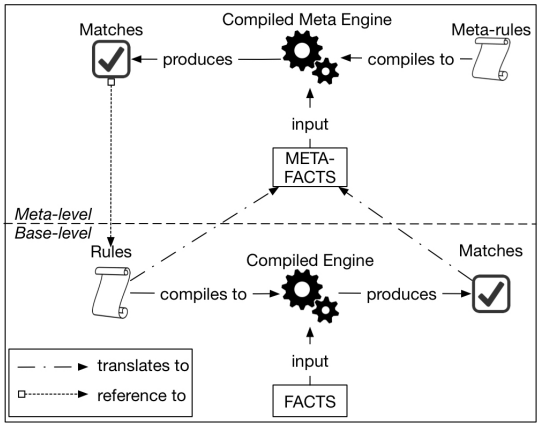

Figure 1: The architecture of the reified rule-language.

Lines 2 to 6 describe the LHS of the rule whereas line 7 describes the RHS. Line 3 considers every fact of *fact-type* `Client` and stores it in variable `c`. This variable can be used in the remainder of the rule-description. Line 4 considers every fact of the *fact-type* `PremiumMember` and tries to match its `clientId` with the `clientId` of the `Client`-fact that is stored in variable `c`. Finally, line 5 considers every fact of the *fact-type* `Purchase` and checks whether the amount of the purchase is bigger than $25 and whether it is made by a `Client` that passed the condition of line 4 (i.e. a `Client` that is also a `PremiumMember`). Every fact that is matched by this condition is stored in variable `p`. If all these conditions match, the RHS of the rule can be evaluated. In this case, the `Client c` will be charged for purchase `p` and receives a discount of 5%. Individual conditions are named by using the `as:`-construct in order to simplify reasoning about them from the meta-level.

The reason why the RETE Algorithm performs well on matching facts with patterns is because the algorithm compiles a graph based on the ruleset. Every fact that enters the system has to enter this graph and all the intermediate matches are stored in the graph. The exact description of the algorithm falls outside the scope of this paper but the interested reader is suggested to consult the original paper by Forgy [4].

3. REFLECTING OVER A RULE-BASED PROGRAM

In order to provide language constructs that allow developers to change individual rules at runtime rather than (un-)loading an entire ruleset, a reification of the ruleset is necessary. Before discussing the reification, however, it is important to discuss the general architecture of the proposed system.

3.1 Architecture

In traditional rule-based languages, it is only possible to write business rules that are concerned with the business logic. We propose to add a different kind of rules, *meta-*

rules, in order to provide language support for changing these business rules at runtime. These two types of rules live in different levels of the application.

3.1.1 Base-level

Business rules like the one discussed in listing 1 are defined in the *base-level* of a program. The rules of the *base-level* are used to describe the business logic of an application and can only be used to reason over `facts` that are associated with the business-logic. The base-level is illustrated below the dotted line in figure 1.

This figure shows that a ruleset is compiled into a running system (*Compiled Engine*) that accepts facts as input. When one of these facts matches one of the rules, a `Match` is produced and the RHS of the rule is executed.

3.1.2 Meta-level

The *meta-level* holds rules that will reason over the program running in the *base-level*. Similar to how normal rules reason over normal facts, the *meta-level* will have *meta-rules* that can reason over *meta-facts*. *Meta-facts* are facts that contain information about the runtime of the *base-level*.

The meta-level is illustrated above the dotted line in figure 1. The way the meta-level works shares a lot of similarities with the base-level. Instead of compiling business rules, meta-rules are compiled into a running system that accepts meta-facts as input. These meta-facts are the only way to share information from the base-level to the meta-level.

3.2 Meta-facts: Reifying the Base-level

Meta-rules make it possible to reason over information about the base-program, but should also make it possible to change the base-program under certain conditions. In order to do so, it is important to reify parts of the base-program.

As illustrated in figure 1 the `Rules` as well as the `Matches` are reified as *meta-facts* and fed to the *compiled meta-engine*. Listing 2 illustrates the *fact-type* of a reified business rule.

```
1  public class BusinessRule {
2    private Collection<Condition> LHS;
3    private RhsExpression RHS;
4
5    //getters and setters
6  }
7  public Class Condition {
8    private String name;
9    private LhsExpressionOneArgument condition;
10   private LhsExpressionTwoArguments condition;
11 }
12
13 public interface RhsExpression {
14   void performAction();
15 }
16
17 public interface LhsExpressionOneArgument {
18   boolean doCheck(Object arg);
19 }
20
21 public interface LhsExpressionTwoArguments {
22   boolean doCheck(Object arg1, Object arg2);
23 }
```

Listing 2: Reification of a business rule into a meta-fact. The syntax is inspired by Drools in which fact-types are defined as POJO's.

The meta-fact that represents a business rule has two main components; LHS and RHS.

`LHS` is a collection of `Conditions`. A `Condition` is made up of a name (what comes after the `as`-constructs in listing 1) and a `LhsExpression`. A `LhsExpression` is a lambda that takes one or two arguments, hence the differentiation between the interfaces `LhsExpressionOneArgument` and `LhsExpressionTwoArguments`. `RHS` is an implementation of the interface `RhsExpression` which contains the method that should be executed whenever there is a match in the meta-level. It is important to note, as illustrated in figure 1, that this lambda has access to the ruleset of the base-level. This means that `performAction` can be implemented in such a way that it changes one (or more) of the rules of the base-level which will lead to a reconfiguration of the `Compiled Engine`. Another consequence of changing a rule from the base-level is that some meta-facts might no longer be consistent with the updated situation and need to be retracted from the meta-level. The updated rule of the base-level on the other hand needs to be reified and inserted in the meta-level as a new meta-fact. How to write such meta-rules that influence the base-level is further explored in section 3.3.

Apart from reifying the business rules, it can also be observed that the matches of the base-level are reified. The *fact-type* thereof is illustrated in listing 3.

```
1 public class Match {
2   private String ruleName;
3   private Timestamp time;
4 }
5
```

Listing 3: Reification of a match into a meta-fact. The syntax is inspired by Drools in which fact-types are defined as POJO's.

The meta-fact that represents a match only has two fields. The field `ruleName` to indicate which rule was successfully matched and a field `time` of the type `Timestamp` to indicate when the rule was matched. In section 3.3 we will discuss why information about the matches is reified as well.

3.3 Meta-rules

In this section we will discuss meta-rules; the language-construct that will enable us to monitor as well as change rules at runtime. Writing meta-rules is very similar to writing regular business rules. Let us consider the example presented in listing 4.

```
1  meta-rule "giveDiscountToRegularClients"
2    when
3      r: Rule(name=="ProcessPurchasePremiumMember")
4      Number (doubleValue < ThresHold)
5        from accumulate (
6          m: Match (
7            r.name == ruleName)
8          over window:Time ( 24h ),
9          count(m))
10   then
11     r.removeCondition("premium-condition")
12 end
13
```

Listing 4: An example of a meta-rule. The syntax is inspired by the Drools Rule Language.

This meta-rule will change the business rule presented in listing 1 in such a way that a discount is applied to all `Clients` rather than only to `PremiumMembers` when the business rule was not matched enough times in the last 24 hours. As is the case with business rules, meta-rules are also made up of a name, a LHS and a RHS.

The LHS of this rule spans from line 2 to 10 whereas the RHS is defined on line 11.

In the LHS, every meta-fact of the fact-type `Rule` is considered by line 3. If the rule has the name "ProcessPurchasePremiumMember", it is stored in variable `r`. Lines 4

to 9 are responsible for counting the number of times rule `r` was matched in the last 24 hours. In order to understand this let us first focus on the `accumulate` function. `accumulate` enables one to write a function that iterates over a collection of facts. In this specific case, we apply `count` on all the meta-facts of type `Match` with the `ruleName` "ProcessPurchasePremiumMember" that were asserted during the last 24 hours. It is important to understand that `count` implements an interface that has an (a) `accumulate`-function (which increments a counter when a `Match` with `ruleName` "ProcessPurchasePremiumMember" is asserted) and a (b) `reverse`-function (which decrements the counter when a `Match` with `ruleName` "ProcessPurchasePremiumMember" is retracted from the meta-engine – because it was created longer than 24 hours ago for example). When the `accumulate`-function processed all relevant `Match`es, the result is considered as a `Number` on line 4. This `Number` has a `doubleValue`-field which is populated with the result of the `accumulate`-function. If the value of that field is smaller than the `Threshold` [1] the meta-rule is said to have matched and the RHS of the meta-rule is evaluated.

The RHS simply removes the condition of the rule "ProcessPurchasePremiumMember" that checks whether the `Client` who is making the purchase is also a `PremiumMember`. This means that from now on, regular `Client`s who are not a `PremiumMember` can also enjoy a discount of 5%. The rule is removed by invoking the method `removeCondition` on it with the name of the `Condition` that should be removed as an argument.
Apart from removing rules, it is also possible to update existing `Condition`s by invoking the method `updateCondition` on `r` while passing the name of the condition and the new `Condition` as arguments. Furthermore, adding new conditions to a rule can be achieved through invoking the method `addCondition` on `r` while passing the new `Condition` as an argument. Finally it is also possible to update the value of `meta-variables` in the RHS of a meta-rule.

3.4 Meta-variables

Apart from extending the rule-language with meta-rules to update the ruleset at runtime, we will also incorporate meta-variables into our rule-language. Meta-variables are variables that can be used and updated in the ruleset of the base-level as well as in the ruleset of the meta-level. Typically, literal values are encoded as constants in the rules. This is illustrated in listing 1 on line 5 in which the amount is encoded. We argue that by replacing the literal values with meta-variables, rules are much more customisable with little extra effort by the developer.

Let us assume that we want to give `PremiumMembers` a discount on purchases of over $50 rather than on purchases of $25. Using meta-rules, it would suffice to write a meta-rule that captures the business rule with the name `ProcessPurchasePremiumMember` and update the condition with the name "purchase-condition" accordingly.
However, rewriting the "purchase-condition" as follows (i.e. replace "25" by a meta-variable named `MinimumAmount`) would simplify the process of changing the rule even further:

[1] `Threshold` is a meta-variable which is discussed in section 3.4. For now it suffices to assume this is a numerical value.

```
1 p: Purchase(amount > MinimumAmount, ! paidFor, c.clientId
      == clientId) as: purchase-condition
```

Changing the value of meta-variables is possible in the RHS of business rules as well as in the RHS of meta-rules.
These meta-variables are furthermore particularly interesting in a RETE-implementation of the presented model as an extension was created to the algorithm that improves the performance of changing literal conditions of a rule at runtime [7].

4. RELATED WORK

To the best of our knowledge, the term *meta-rule* was first coined by Davis in the context of the MYCIN [2] expert-system and later on in the context of the TEIRESIAS [3] expert-system. In these expert systems, meta-rules are used for control; in some scenarios it might happen that there are multiple relevant business rules for a given set of data but that only one rule is allowed to fire. In order to determine which rule should be fired, meta-rules were defined.

Schild et. Herzog have introduced meta-rules in Rule Based Legal Computer Systems [6]. The authors claim that the law exhibits intrinsic vagueness and that this vagueness leaks into the rules that are written in such systems. The authors propose to use meta-rules to detect which rules are relevant for a certain set of data. Based on these relevant rules, the meta-rule can then select (or create) an appropriate business rule. However, the authors merely discuss the usefulness of meta-rules in Rule Based Legal Computer Systems and omit any technical aspects or implementation.

The ideas presented in this paper are inspired by earlier work on *Mirrors* in which there is also a clear distinction between a base-level and a meta-level [1].

5. CONCLUSION

In this paper we argue that the language support in current rule-based languages for updating the ruleset is inadequate.

We propose to reify the `Rule`s and `Match`es of the base-level program as meta-facts and feed them to a rule-engine that runs in the meta-level. This enables developers to write meta-rules which can reason over and make changes to the rule-program.

We envision language constructs to delete rules entirely, update, add and delete constraints of the LHS of a rule as well as update the RHS of rules. Finally we also incorporated meta-variables in our envisioned rule-language. These meta-variables can be modified from the RHS of meta-rules as well as business rules. This implies that the base-level program can make small changes to itself.

The concepts presented in this paper are applicable to all rule-based languages although we aim to implement this in a rule-based language that runs on top of the RETE-algorithm.

6. REFERENCES

[1] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *ACM SIGPLAN Notices*, volume 39, pages 331–344. ACM, 2004.

[2] R. Davis. Meta-rules: Reasoning about control. *Artificial intelligence*, 15(3):179–222, 1980.

[3] R. Davis and B. G. Buchanan. Meta-level knowledge. *Rulebased expert systems, The MYCIN Experiments of the Stanford Heuristic Programming Project, BG Buchanan and E. Shortliffe (Editors), Addison-Wesley, Reading, MA*, pages 507–530, 1984.

[4] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982.

[5] M. Proctor. Drools: a rule engine for complex event processing. In *Applications of Graph Transformations with Industrial Relevance*, pages 2–2. Springer, 2012.

[6] U. J. Schild and S. Herzog. The use of meta-rules in rule based legal computer systems. In *Proceedings of the 4th international conference on Artificial intelligence and law*, pages 100–109. ACM, 1993.

[7] S. Van de Water, T. Renaux, L. Hoste, and W. De Meuter. Indexing rete's working memory. 2015.