Formalizing Both Refraction-Based and Sequential Executions of Production Rule Programs

Bruno Berstel-Da Silva

Institut für Informatik
Albert-Ludwigs-Universität Freiburg
Germany

Abstract Production systems are declarative, in that they do not explicitly specify the control flow. Yet, the concept of a production system does not include the definition of a given control strategy. The control between rules in a production rule program is, in practice, defined by each implementation of a production rule engine. Engines have traditionally been implemented using the Rete algorithm. Since the turn of the century, however, production systems have evolved into industrial products known as Business Rules Management Systems (BRMS). BRMS have introduced new compilation and execution schemes, which are often called *sequential* in contrast with the incremental behavior of Rete. This change in execution scheme came with a change in semantics for rule programs. In this paper, we propose a formal description of the execution of production rule programs. Existing descriptions either ignore the control strategy, or assume a Rete semantics. Ours isolates the handling of rule eligibility in the control strategy, which allows us to describe the sequential execution semantics of rule programs, as well as the Rete semantics, and others.

1 Introduction

Business Rules Management Systems (BRMS) [7,11,16,21,22,26] are industrial products that have gained substantial consideration as a way to lower the cost of frequent changes in business policies [18,29]. The contribution of BRMS is to externalize the business logic of an application as a rule program, and to provide business experts with tools to author and manage these rules collaboratively.

The rules in question are of the condition-action type, also known as production rules. As such, BRMS can be seen as descendants of production systems [12,19], in the tradition of OPS5 [3,13]. Indeed, the rule engines present in most BRMS compile and execute rule programs with variants of the Rete algorithm [14,15,20], which is at the heart of OPS5. However, with the transition from production systems to BRMS, rule programs have evolved from medium-sized programs implementing inference algorithms to massive programs performing simpler tasks [28]. In this context, the concern for the implementors of rule engines has shifted from providing smart control strategies in the execution of programs to designing new compilation schemes that would ensure a high throughput in the processing of data.

This new generation of compilation and execution algorithms is often referred to as *sequential* [10,17,23], both due to the way they consume input data and in contrast with the incremental behavior of Rete. Yet, the gain in performance has been achieved at the price of a change in the execution semantics of the rule programs. The main difference between the Rete and sequential semantics is that the former allows rule programs to implement complex inference schemes, whereas the latter makes assumptions (a stable working memory, a static rule set, etc.) that opens the door to faster execution. This disruption in semantics is usually acceptable for the users of BRMS, because their rule programs, although important in size, implement simple algorithms that do not require evolved inference mechanisms.

This paper is organized as follows. In Sect. 2, we expose our formal description of rule program execution. Then, in Sects. 3 and 4, we review some selection and eligibility strategies; in particular, we provide the formalization of Rete's refraction semantics and of the sequential semantics of modern BRMS. Finally, in Sect. 5 we illustrate both semantics and their formalizations on an example rule-based application, which we introduce in Sect. 1.2.

1.1 Related Work

Up to now, and to the best of our knowledge, there have been only few formalizations of the execution behavior of production rule programs with the Rete semantics, and none with the sequential semantics. Such a formalization is however useful to develop tools that help understand, verify, test, and more generally analyze, rule programs. In this paper, we present a logic-based description of rule programs and of their execution, with a focus on the distinction between the applicability and the eligibility of a rule.

In the traditional presentations of Rete, eligibility is one step of the control strategy, called *refraction*. By isolating the eligibility strategy, our formalization allows us to depict the execution of rule programs with either Rete's refraction semantics, the sequential semantics, or others. It is based on first-order logic; we also use objects with attributes to reflect the fact that rule programs in BRMS handle object-oriented data provided by an embedding application. We model states as first-order logic structures and rule execution as relations between states.

Production systems have been formalized in a number of ways, either based on first-order logic or not. In [27] and in previous publications, Schmolze and Snyder have explored the connection between production rules and term rewriting systems. However, their approach focuses on confluence and termination properties, and does not aim at describing the execution behavior of production systems.

Production systems have also been studied from the viewpoint of active databases (see [1] for a survey). Again, confluence and termination is the main focus, and the composition of rules into a rule program execution is not addressed. Safety properties are treated in [2], and mapped to constraint satisfiability problems on the transition constraints that describe the execution of the rule program. However, the construction of these constraints is not studied. In [4], propositional production systems are modeled with μ-calculus, and production systems with variables are modeled with fixed-point logic. This approach leads to using first-order logic structures as we do.

The formalizations of production systems that address their execution behavior are due to Fages and Lissajoux [9], to Cirstea *et al.* [6], and to Damásio *et al.* [8,25]. These formalizations are based on first-order logic, with [8] relying on Answer-Set Programming. All consider the execution of rule programs by the Rete algorithm. However, [9] explicitly discards the control strategy from its scope and provides a nondeterministic view of rule program execution. On the other hand, [6] includes the control strategy in its formal description of rule program execution, but does not go into any detail; furthermore, the examples given implement an explicit control flow and hence do not exhibit the role of the control strategy. Finally, [8] does include the control strategy in its formalization of rule programs semantics.

1.2 Running Example

Acme.com is an e-commerce company. It wants to introduce a rewarding program in which customers earn bonus points, and chooses to implement it with a BRMS. Customers are entered into the system with their current bonuses and their purchases, to be processed by the rule program $\mathcal{R} = \{\mathtt{P}, \mathtt{S}\}$ that implements the rewarding program.

The first rule in the program grants a customer bonus points, based on his/her purchases. Namely, if the value of the purchase p exceeds \$ 100, the customer c earns 10 % of the purchase value in bonus points.

$$\mathtt{P(c,p)}:\ \mathtt{c = p.buyer\ \&\ p.value \geq 100} \rightharpoonup \mathtt{c.bonus \coloneqq c.bonus + p.value \times .1}$$

The second rule implements a sponsorship mechanism: if a customer s sponsors another customer c, then a transfer of bonus points occurs; the transfer only occurs if the sponsor has at least 200 points.

$$\begin{aligned}\mathtt{S(s,c)}:\ \mathtt{s = c.sponsor\ \&\ s.bonus \geq 200} \rightharpoonup\ & \mathtt{s.bonus \coloneqq s.bonus - 50}\,;\\ & \mathtt{c.bonus \coloneqq c.bonus + 30}\end{aligned}$$

This short example introduces rules, with a guard that determines when the rule applies, and an action that indicates how the rule execution will evolve the system state by performing updates on object attributes. The attributes in this example program are *bonus*, *value*, *buyer*, and *sponsor*. Objects are held by rule variables: the rule variables in P are c and p, those in S are s and c. These concepts are formalized in Sect. 2.

In contrast with industrial rule languages or with RIF-PRD [25], our language does not include adding objects to, or removing objects from, the working memory. However, like industrial rule languages, it regards the update of an object attribute as a single operation.

2 Formal Description of Rule Program Execution

2.1 Expressions, States, Rules, Programs

A rule program implements part of the logic of a larger application. To this end, the rules handle the data of the application, in the format defined by the application itself. In practice, they use the data types of the embedding language, such as Booleans, numbers, enumerations, and, in an object-oriented context, the classes defined by the application. From a formal viewpoint, the application introduces a ***theory*** Θ, which contributes to the *signature* of the rule language with a set of function symbols (including constants), and provides their interpretation. Classically, the theory Θ can include Booleans with logical connectors ($\wedge$, $\vee$, $\neg$...), numbers with arithmetic operators ($+$, $-$, $\times$...), uninterpreted functions with equality, etc. In our framework, we assume that Θ includes at least Booleans, without quantifiers, and objects with attributes as defined below. The running example exposed in Sect. 1.2 also includes numbers.

Objects have a unique identity. The identities of two objects can be compared for equality; no other operation is available on the identity of objects. Object *attribute symbols* are unary function symbols.

In our rule language, objects are handled through *variables*. ***Expressions*** are built in the classical, inductive way on the signature inherited from Θ. They include arithmetic and Boolean expressions, but also *attribute references*. An attribute reference denotes the value of an object's attribute; it is written in dotted postfix notation. For example, $p.age$ refers to the attribute with symbol age of the object held by the variable p.

A ***state*** is a first-order logic structure. The common domain to all states is provided by the theory Θ; we note it $\mathcal{D}$. As mentioned previously, it includes at least the Booleans and the objects, the infinite set of which we note $\mathbb{O}$. The interpretation function $\mathcal{I}_s$ of a state s interprets function symbols as specified by the theory Θ—typically, logical connectors and arithmetic operators are interpreted in the classical way. Attribute symbols are interpreted by partial functions from $\mathbb{O}$ to $\mathcal{D}$. For an attribute symbol f, we note $\mathcal{I}_s(f)$ or f^s the function that interprets f in s. The definition domain of this partial function is noted $\mathrm{Dom}(f^s)$.

A ***rule*** $r = (\vec{o}, g, a)$ consists of the tuple $\vec{o} = (o_1, \ldots, o_m)$ of its *rule variables*, the Boolean expression g called its *guard*, and its *action* a, described further below. The *arity* of r, noted $|r|$, is m. A ***rule instance*** is a tuple $R = (r, O_1, \ldots, O_m)$ where $O_1, \ldots, O_m \in \mathbb{O}$ are objects. The objects in a rule instance provide values for the rule variables, which are used to interpret the rule guard and action, as described below.

A rule instance $R = (r, O_1, \ldots, O_m)$ is ***applicable*** in a state s if the guard g of r holds in this state and on the objects $O_1, \ldots, O_m$. That is, if the interpretation of the Boolean expression g by state s, with each variable o_j mapped onto the object O_j for $j = 1, \ldots, m$, yields true. The guard g is interpreted as false by s if one has $O_j \notin \mathrm{Dom}(f^s)$ for any attribute reference $o_j.f$ that appears in g.

When a rule instance $R = (r, O_1, \ldots, O_m)$ is ***applied***, the action a of the rule is executed on the objects in the instance. A rule action is a sequence of assignments to attributes of rule variables. An *assignment* is a statement $o_{j_k}.f_k \coloneqq e_k$ that denotes the update of the attribute f_k for the object held by o_{j_k} with the value of the expression e_k. Its semantics, when executed from a state s, is to produce a new state s', in which all attribute and function symbols are interpreted as in s, with the exception of f_k. In state s', the attribute symbol f_k is interpreted by the partial function $f_k^{s'}$ with the same domain as f_k^s, and such that

$$\forall O \in \mathrm{Dom}(f_k^s) \quad f_k^{s'}(O) = \begin{cases} \mathcal{I}_{s'}(e[O_j/o_j]_{j=1}^m) & \text{if } O = O_{j_k} \\ f_k^s(O) & \text{otherwise}\,. \end{cases}$$

To summarize, given a rule $r = (\vec{o}, g, a)$, a rule instance $R = (r, O_1, \ldots, O_m)$ is applicable in a state s if the guard g holds in s and on the objects $O_1, \ldots, O_m$. When R is applicable in s, then the application of R in s produces a new state s' that results from the execution of the action a from s on $O_1, \ldots, O_m$. We note this application $s \xrightarrow{R} s'$.

A ***rule program*** $\mathcal{R} = \{r_1, \ldots, r_n\}$ is a finite set of rules. A rule program is executed on a finite set of objects $\mathcal{M} \subset \mathbb{O}$, called the *working memory.* The set of all rule instances that can be formed out of rules in $\mathcal{R}$ and objects in $\mathcal{M}$ is noted $\mathcal{I}(\mathcal{R}, \mathcal{M}) = \{(r, \vec{O}) \mid r \in \mathcal{R}, \vec{O} \in \mathbb{O}^{|r|}\}$. Given a state s, the subset of rule instances that are applicable in s is noted A_s.

2.2 Execution of a Rule Program

To formally describe the execution of a rule program, we introduce the notion of a *configuration* of the rule engine, as a pair $\langle E, s\rangle$, where $E \subseteq \mathcal{I}(\mathcal{R}, \mathcal{M})$ is a set of rule instances and s is a state. In such a configuration, E denotes the set of *eligible* rule instances in s.

With this definition, we say that an *execution of a rule program* $\mathcal{R}$ on a working memory $\mathcal{M}$ from an initial state s_0 is defined by a potentially infinite sequence of transitions between configurations

$$\langle \mathcal{I}(\mathcal{R}, \mathcal{M}), s_0\rangle \xrightarrow{R_1} \langle E_1, s_1\rangle \xrightarrow{R_2} \langle E_2, s_2\rangle \xrightarrow{R_3} \cdots$$

in which each transition complies with the following transition rule, for $k > 0$:

$$\frac{\{R_k\} = \mathcal{S}(A_{s_{k-1}} \cap E_{k-1}) \quad s_{k-1} \xrightarrow{R_k} s_k \quad E_k = \mathcal{E}(E_{k-1}, \ldots)}{\langle E_{k-1}, s_{k-1}\rangle \xrightarrow{R_k} \langle E_k, s_k\rangle}\,. \tag{1}$$

This transition rule encodes that the rule engine can perform a transition by rule instance R_k from configuration $\langle E_{k-1}, s_{k-1}\rangle$ to configuration $\langle E_k, s_k\rangle$ if the following conditions are met: the rule instance R_k is selected among the rule instances that are both applicable and eligible in s_{k-1}, and its application in s_{k-1} produces the state s_k.

This transition rule exhibits the selection strategy $\mathcal{S}$ and the eligibility strategy $\mathcal{E}$ as two parameters of the rule engine semantics. The selection strategy $\mathcal{S}$ is a function that takes a set of rule instances and returns either the empty set or a singleton included in the set received. Sect. 3 reviews the most common selection strategies.

Which rule instances are eligible at each step of the execution is determined by the eligibility strategy $\mathcal{E}$, based on the set of previously eligible rule instances and on possibly other arguments, such as the rule instance being applied, the initial or final states of the transition, etc. At the beginning of the execution, all rule instances are eligible. Sect. 4 reviews the eligibility strategies at work in the Rete and sequential execution semantics.

As an extreme case, consider the eligibility strategy $\mathcal{E}_{\mathsf{id}}$ that always returns the previous set of eligible rule instances, and the selection strategy $\mathcal{S}_{\mathsf{nd}}$ that non-deterministically selects a rule instance. These strategies give the rule program execution semantics considered by [9].

2.3 Comparison with Traditional Presentations

The control strategy in a production rule engine is analogous to the scheduler in parallel programming or to the method call resolution in an object-oriented language. In all these control mechanisms, a variation on any criterion can change the course of program execution dramatically. However, in contrast with object-oriented languages where the method call semantics is defined with the language, rule languages do not include the definition of a control strategy. Instead, the control strategy is brought in by the algorithm used to execute the rule program.

Since the seminal implementation by OPS5, production systems have traditionally executed rule programs with the Rete algorithm or a variant. More or less formal descriptions of its control strategy can be found in the OPS5 User's Manual [13], in the RIF-PRD recommendation [25], in papers describing extensions to OPS5 [5,24], or in the documentations of BRMS [17].

The Rete control strategy applies to the set A_s of applicable rule instances in the current state s. In the context of Rete, this set is called the *conflict set* [3,19]. It is traditionally presented as the following four steps. If, after the first step, the conflict set is empty, then the program execution stops. If, as the result of any step, the conflict set contains only one rule instance, then this rule instance is selected for application.

(i) *Refraction:* Discard from the conflict set any rule instance that has already been applied, and has since remained applicable.
(ii) *Recency:* Retain the rule instances that include objects that have been inserted or modified last.
(iii) *Specificity:* Retain the rule instances that relate to the most specific rules. (We define rule specificity in Sect. 3.)
(iv) *Random:* Arbitrarily retain only one rule instance.

Implementations of production systems other than OPS5 [7,11,16,21,22,26] have adopted similar control strategies, with minor variations on the definition of refraction, and a wider range of criteria to select the rule instance to apply among those that successfully passed the refraction step.

From a broader perspective, one can identify two purposes in a control strategy for the execution of rule programs. A first goal is to discard program executions that do not make sense, for example to avoid trivial loops. In the Rete algorithm, this is the role of the refraction step. Our formalism exposed in Sect. 2.2 generalizes this filtering task with the eligibility strategy. A second goal is to choose one rule instance among the ones that have passed the filtering step. This is the role of the three last steps of Rete. We generalize this task with the selection strategy. The distinction that our framework introduces between the selection and eligibility strategies allows us to describe other execution schemes than Rete, as shown in Sect. 4.

3 Selection Strategies

When several rule instances are both applicable and eligible in a given state, the choice of which to apply in the transition to the next configuration is the role of the selection strategy. In the traditional Rete control strategy, exposed in Sect. 2.3, this is addressed by the last three steps.

In practice, selection strategies define an order on rule instances, and return the (applicable and eligible) instance that is maximal according to this order. The order is classically defined by the lexicographic combination of various orders such as the following ones [3,19].

- *Priority on rule instances.* A rule $r = (\vec{o}, g, a)$ is equipped with a numerical expression π_r in the rule variables. The order is based on the value of this expression for each rule instance in the current state.
- *Priority on rules.* This order is a simplified version of the previous one, where the expressions π_r are numerical constants. The priority of all instances of a rule r are then equal to the number π_r, independently of the state.
- *Strict ordering of the rules.* A strict order is explicitly defined on the rules, for example by setting the rule priorities to a permutation of $\{1, \ldots, n\}$.
- *Specificity of the rules.* A rule $r_1 = (\vec{o}_1, g_1, a_1)$ is said to be more specific than another rule $r_2 = (\vec{o}_2, g_2, a_2)$ when one has $g_1 \Rightarrow g_2$. This defines a partial order on rules. Some rule engines approximate this order by using empirical indications of the rule specificity, such as the number of elementary Boolean expressions in the rule guard, or the arity of the rule.
- *Recency.* This order is based on a numerical constant associated with each object in the working memory, called the object recency, with the idea that objects have been inserted into the working memory in some order. The recency of a rule instance is given by the maximal recency of the objects in the rule instance.

4 Eligibility Strategies

4.1 The Refraction Eligibility Strategy

As stated in the RIF-PRD recommendation [25]: *"The essential idea of refraction is that a given instance of a rule must not be fired more than once as long as the reasons that made it eligible for firing hold."* As a direct consequence, enforcing refraction must take the execution history into account. To this end, [25] chooses to define refraction by counting during how many execution steps each rule instance has remained applicable, and since how many steps it has been applied.

In our framework, we base the definition of refraction on the set E_k of eligible rule instances in a configuration $\langle E_k, s_k \rangle$ of the rule engine. Using the formalism introduced in Sect. 2.2, refraction can be defined as follows.

> **Refraction.** If a rule instance R has been applied in a configuration transition $\langle E_i, s_i \rangle \xrightarrow{R} \langle E_{i+1}, s_{i+1} \rangle$, it is eligible for application in a subsequent transition $\langle E_k, s_k \rangle \xrightarrow{R} \langle E_{k+1}, s_{k+1} \rangle$ only if the execution contains a configuration $\langle E_j, s_j \rangle$ such that $i + 1 \leq j \leq k$ and $R \notin A_{s_j}$.

Of course, for the transition $\langle E_k, s_k \rangle \xrightarrow{R} \langle E_{k+1}, s_{k+1} \rangle$ to occur, the rule instance R will have to be applicable in s_k, which implies $j < k$. However, this requirement does not relate to eligibility. The independence between applicability and eligibility is visible in the example discussed in Sect. 5. For instance, we shall see that in state σ_1, the rule instance $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$ is eligible but not applicable; and that in σ_3, it is applicable but not eligible.

The task of the eligibility strategy is to compute the set of eligible rule instances that results from a transition between configurations of the rule engine. As per the definition of refraction above, the refraction eligibility strategy $\mathcal{E}_{\mathsf{ref}}$ makes a rule instance ineligible when it is applied, and makes it eligible again as soon as its becomes inapplicable. That is, in a transition $\langle E, s \rangle \xrightarrow{R} \langle E', s' \rangle$:

$$\mathcal{E}_{\mathsf{ref}}(E, R, s') = E \setminus \{R\} \cup \{R' \in \mathcal{I}(\mathcal{R}, \mathcal{M}) \mid R' \text{ is not applicable in } s'\}\,.$$

In this definition, the eligibility strategy first removes the rule instance R that was just applied from the set of eligible rule instances, which corresponds to the statement *"a rule instance must not be fired more than once"*. Then, the strategy implements *"as long as the reasons that made it eligible for firing hold"* by adding the rule instances that became inapplicable due the application of R.

Note that, although the definition of $\mathcal{E}_{\mathsf{ref}}$ refers to any rule instance inapplicable in s', the rule instances added are precisely those that have been already applied in this rule program execution and that have been made inapplicable by the application of R. Indeed, the rule instances that have never been applied are in E since the beginning of the rule program execution; and the rule instances that were already inapplicable in s have been added to E in a previous transition.

Since, in the execution of the rule program, the selection is performed on $A_s \cap E$, the definition of the refraction eligibility strategy above ensures that once applied, a rule instance will not be applied a second time before it becomes first inapplicable and then applicable again, as stated by the Rete algorithm.

4.2 The Sequential Control Strategy

Definition of Sequential Execution Sequential execution schemes have appeared in BRMS about a decade ago [10,17,23], as an answer to the evolution of rule programs from the many patterns/many objects case for which Rete had been invented, to a many rules/few objects case found in business applications. In this section, we describe the sequential execution mode of IBM Websphere Operational Decision Management [16]; the corresponding execution algorithms in other BRMS [7,11,21,22,26] have similar behaviors.

The sequential execution mode considers all object tuples from the working memory in sequence, and submits each tuple to all rules, in sequence again. If the guard of a rule holds on an object tuple when they are considered together, the rule action is executed. Otherwise, the next rule is considered. There is no "second chance:" a rule instance that has already been considered and would become applicable only later, due to the application of another rule instance, will not be applied. The orders defined on object tuples and on rules are therefore crucial. Any criterion described in Sect. 3 can be chosen, although simple rule priorities are commonly used.

The semantics of the sequential execution mode can be considered poorer, since it suppresses opportunities for inference and chaining between rules. However, it is considered an acceptable trade-off by BRMS users as it allows faster execution. Furthermore, the greater control it provides through the explicit ordering of rules is found opportune with large rule programs.

The Sequential Control Strategy Unlike refraction, which is a pure eligibility strategy and hence can be combined with any selection strategy, the sequential control strategy defines both a selection and an eligibility strategy. These two strategies $\mathcal{S}_{\mathsf{seq}}$ and $\mathcal{E}_{\mathsf{seq}}$ implement the behavior just exposed. Namely, the eligibility strategy defines the set of rule instances under consideration, and the selection strategy ensures that the rule instances are picked in the proper order.

As described above, the sequential control strategy is based on a strict ordering of the rules in the rule program, and of the objects in the working memory. These two orderings define a strict order $<_{\mathsf{seq}}$ on rule instances lexicographically. The sequential control strategy is then defined as follows:

(i) The *selection strategy* $\mathcal{S}_{\mathsf{seq}}$ returns the minimal rule instance under consideration:

$$\mathcal{S}_{\mathsf{seq}}(C) = \{R \in C \mid R \text{ is minimal with respect to } <_{\mathsf{seq}}\} .$$

(ii) The *eligibility strategy* $\mathcal{E}_{\mathsf{seq}}$ retains for further consideration only the rule instances that follow (are greater than) the rule instance just applied; in a transition $\langle E, s\rangle \xrightarrow{R} \langle E', s'\rangle$:

$$\mathcal{E}_{\mathsf{seq}}(E, R) = \{R' \in E \mid R <_{\mathsf{seq}} R'\} .$$

5 Illustration

Let us consider again the example rule-based application depicted in Sect. 1.2. The rules of the program $\mathcal{R} = \{\texttt{P}, \texttt{S}\}$ are reminded below:

$$\begin{aligned}
&\texttt{P(c,p)}: \texttt{c} = \texttt{p.buyer}\ \texttt{\&}\ \texttt{p.value} \geq 100 \rightharpoonup \texttt{c.bonus} := \texttt{c.bonus} + \texttt{p.value} \times .1\\
&\texttt{S(s,c)}: \texttt{s} = \texttt{c.sponsor}\ \texttt{\&}\ \texttt{s.bonus} \geq 200 \rightharpoonup \texttt{s.bonus} := \texttt{s.bonus} - 50\,;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{c.bonus} := \texttt{c.bonus} + 30
\end{aligned}$$

As seen in Sect. 2, a rule program such as $\mathcal{R}$ is executed on a finite set of objects, called the working memory. Assume that we plan to execute $\mathcal{R}$ on a working memory $\mathcal{M} = \{\texttt{Alice}, \texttt{Bob}, \texttt{Don}, \texttt{Car}\}$ containing four objects: three customers and a purchase. An execution of $\mathcal{R}$ on $\mathcal{M}$ from an initial state s_0 consists of a sequence of transitions between configurations, starting from $\langle \mathcal{I}(\mathcal{R}, \mathcal{M}), s_0 \rangle$. Assume that the execution starts from a state s_0 in which Alice is a sponsor of Bob and Don, their respective bonuses are 230, 100, and 50 points, and Alice purchases a car for \$ 900. State s_0 can be depicted as follows:

$$s_0 \begin{cases}
\texttt{Alice}: & \textit{bonus} = 230 & \\
\texttt{Bob}\ \ \ : & \textit{bonus} = 100 & \textit{sponsor} = \texttt{Alice}\\
\texttt{Car}\ \ \ : & \textit{buyer} = \texttt{Alice} & \textit{value} = 900\\
\texttt{Don}\ \ \ : & \textit{bonus} = 50 & \textit{sponsor} = \texttt{Alice}\,.
\end{cases}$$

5.1 Sequential Executions of the Rule Program

The sequential executions of $\mathcal{R}$ will be governed by two orders: the order on object tuples, here customer pairs or customer-purchase pairs; and the order on rules, here P and S. In our example, the rules apply to distinct types of objects; as a result, their order has no impact. Let us assume that S comes before P. On the other hand, the order on object tuples determines which of earning bonus points on purchases, or of sponsoring friends, is favored.

Sponsorship over Purchases. Let us first assume that pairs of customers come before customer-purchase pairs, and that object tuples are taken in the following order:

$$(\texttt{Alice}, \texttt{Alice})\ (\texttt{Alice}, \texttt{Bob})\ \ldots\ (\texttt{Don}, \texttt{Don})\ (\texttt{Alice}, \texttt{Car})\ (\texttt{Bob}, \texttt{Car})\ (\texttt{Don}, \texttt{Car})\,.$$

By definition, all rule instances are eligible in the initial configuration. The sequential selection strategy will hence pick the first applicable rule instance that can be formed for rules P and S, with each of the object tuples in the order above.

Per the definitions of Sect. 2.1, the rule instance $(\texttt{P}, \texttt{Alice}, \texttt{Alice})$ is not applicable in s_0, because $\texttt{Alice} \notin \mathrm{Dom}(\textit{buyer}^{s_0})$. On the other hand, $(\texttt{S}, \texttt{Alice}, \texttt{Alice})$ is not applicable in s_0 either, because $\textit{sponsor}^{s_0}(\texttt{Alice}) \neq \texttt{Alice}$. Similarly, $(\texttt{P}, \texttt{Alice}, \texttt{Bob})$ is not applicable in s_0. However, $(\texttt{S}, \texttt{Alice}, \texttt{Bob})$ is applicable in s_0, and is thus selected by the sequential selection strategy.

Therefore, a transition by $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$ from $\langle \mathcal{I}(\mathcal{R}, \mathcal{M}), s_0 \rangle$ to $\langle E_1, s_1 \rangle$ occurs, where the state s_1 results from the application of $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$ in s_0, that is:

$$s_1 \begin{cases} \mathtt{Alice}: & \mathit{bonus} = \mathbf{180} \\ \mathtt{Bob} \quad : & \mathit{bonus} = \mathbf{130} \quad \mathit{sponsor} = \mathtt{Alice} \\ \mathtt{Car} \quad : & \mathit{buyer} = \mathtt{Alice} \quad \mathit{value} = 900 \\ \mathtt{Don} \quad : & \mathit{bonus} = 50 \quad \mathit{sponsor} = \mathtt{Alice} \end{cases}$$

and the set E_1 contains the rule instances that are greater than $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$, that is:

$$E_1 = \{(\mathtt{P}, \mathtt{Alice}, \mathtt{Don}), (\mathtt{S}, \mathtt{Alice}, \mathtt{Don}), (\mathtt{P}, \mathtt{Bob}, \mathtt{Alice}), \ldots, (\mathtt{S}, \mathtt{Don}, \mathtt{Car})\}\,.$$

Because Alice has less than 200 bonus points in state s_1, the first applicable rule instance in s_1 among those in E_1 is $(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})$. A transition by this rule instance therefore occurs to $\langle E_2, s_2 \rangle$, where the state s_2 results from the application of $(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})$ in s_1, that is:

$$s_2 \begin{cases} \mathtt{Alice}: & \mathit{bonus} = \mathbf{270} \\ \mathtt{Bob} \quad : & \mathit{bonus} = 130 \quad \mathit{sponsor} = \mathtt{Alice} \\ \mathtt{Car} \quad : & \mathit{buyer} = \mathtt{Alice} \quad \mathit{value} = 900 \\ \mathtt{Don} \quad : & \mathit{bonus} = 50 \quad \mathit{sponsor} = \mathtt{Alice} \end{cases}$$

and the set E_2 contains the rule instances that are greater than $(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})$, that is:

$$E_2 = \{(\mathtt{S}, \mathtt{Alice}, \mathtt{Car}), (\mathtt{P}, \mathtt{Bob}, \mathtt{Car}), (\mathtt{S}, \mathtt{Bob}, \mathtt{Car}), (\mathtt{P}, \mathtt{Don}, \mathtt{Car}), (\mathtt{S}, \mathtt{Don}, \mathtt{Car})\}\,.$$

None of these rules is applicable in s_2, that is, $A_{s_2} \cap E_2 = \emptyset$. Therefore, the transition rule (1) cannot be applied and the execution of $\mathcal{R}$ ends.

One can note that $(\mathtt{S}, \mathtt{Alice}, \mathtt{Don})$ was not applicable when it was considered in s_1, but is now applicable in s_2. However, the sequential nature of the execution, enforced by the sequential eligibility strategy, causes it to not be included in E_2, and hence not to be considered for execution in s_2.

Purchases over Sponsorship. Let us now assume that customer-purchase pairs come before customer pairs, and that object tuples are thus taken in the following order:

$$(\mathtt{Alice}, \mathtt{Car})\ (\mathtt{Bob}, \mathtt{Car})\ (\mathtt{Don}, \mathtt{Car})\ (\mathtt{Alice}, \mathtt{Alice})\ (\mathtt{Alice}, \mathtt{Bob})\ \ldots\ (\mathtt{Don}, \mathtt{Don})\,.$$

With this order on object tuples, the execution of $\mathcal{R}$ on $\mathcal{M}$ from s_0 is

$$s_0 \xrightarrow{(\mathtt{P},\mathtt{Alice},\mathtt{Car})} s_1' \xrightarrow{(\mathtt{S},\mathtt{Alice},\mathtt{Bob})} s_2' \xrightarrow{(\mathtt{S},\mathtt{Alice},\mathtt{Don})} s_3'$$

with in particular

$$s_2' \begin{cases} \mathtt{Alice}: & \mathit{bonus} = 270 \\ \mathtt{Bob} \quad : & \mathit{bonus} = 130 \quad \mathit{sponsor} = \mathtt{Alice} \\ \mathtt{Car} \quad : & \mathit{buyer} = \mathtt{Alice} \quad \mathit{value} = 900 \\ \mathtt{Don} \quad : & \mathit{bonus} = 50 \quad \mathit{sponsor} = \mathtt{Alice}\,. \end{cases}$$

In this execution, the rule instance $(\mathtt{S}, \mathtt{Alice}, \mathtt{Don})$ is considered in s'_2, where it is applicable. This contrasts with the previous execution, in which the rule instance is considered in state s_1, where it is not applicable.

5.2 Refraction-Based Executions of the Rule Program

Let us consider the same rule program $\mathcal{R}$ on a subset $\mathcal{M}' = \{\mathtt{Alice}, \mathtt{Bob}, \mathtt{Car}\}$ of the working memory $\mathcal{M}$. Let us consider an execution of $\mathcal{R}$ on $\mathcal{M}'$ from the initial state σ_0, equal to the restriction of s_0 to $\mathcal{M}'$, that is:

$$\sigma_0 \begin{cases} \mathtt{Alice}: & \textit{bonus} = 230 & \\ \mathtt{Bob} \ : & \textit{bonus} = 100 & \textit{sponsor} = \mathtt{Alice} \\ \mathtt{Car} \ : & \textit{buyer} = \mathtt{Alice} & \textit{value} = 900\,. \end{cases}$$

Let us note that Alice is a sponsor of Bob and the buyer of the car in state σ_0, and that the actions of none of the rules in $\mathcal{R}$ can change this. This implies that, per the definitions of Sect. 2.1, the only rule instances that can be applicable in any state of an execution of $\mathcal{R}$ on $\mathcal{M}'$ from σ_0 are $(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})$ and $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$. In addition, the rule instance $(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})$ is applicable in σ_0, and since the actions of none of the rules in $\mathcal{R}$ can change this, it shall remain applicable in any state of an execution of $\mathcal{R}$ on $\mathcal{M}'$ from σ_0. As a consequence, these executions would never terminate, unless at some point the control strategy did refrain from choosing $(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})$, even if it were the sole applicable rule instance. This is the role of the eligibility strategy, as illustrated below.

By definition, all rule instances are eligible in the initial configuration. The selection strategy will hence apply to the rule instances that are applicable in the initial state σ_0, namely:

$$A_{\sigma_0} = \{(\mathtt{P}, \mathtt{Alice}, \mathtt{Car}), (\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})\}\,.$$

Assume that the selection strategy chooses to apply $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$ in σ_0. This causes a transition by this rule instance from $\langle \mathcal{I}(\mathcal{R}, \mathcal{M}), \sigma_0 \rangle$ to $\langle E_1, \sigma_1 \rangle$, where the state σ_1 results from the application of $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$ in σ_0, that is:

$$\sigma_1 \begin{cases} \mathtt{Alice}: & \textit{bonus} = \mathbf{180} & \\ \mathtt{Bob} \ : & \textit{bonus} = \mathbf{130} & \textit{sponsor} = \mathtt{Alice} \\ \mathtt{Car} \ : & \textit{buyer} = \mathtt{Alice} & \textit{value} = 900 \end{cases}$$

and the set E_1 is computed by first removing $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$ from $E_0 = \mathcal{I}(\mathcal{R}, \mathcal{M})$, and then adding all the rule instances that are not applicable in σ_1. Since Alice has less than 200 bonus points in σ_1, the rule instance $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$ is not applicable in σ_1, and $E_1 = \mathcal{I}(\mathcal{R}, \mathcal{M})$.

In configuration $\langle E_1, \sigma_1 \rangle$, we have $A_{\sigma_1} \cap E_1 = \{(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})\}$. Therefore, a transition to $\langle E_2, \sigma_2 \rangle$ occurs by the application of $(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})$, which results in

$$\sigma_2 \begin{cases} \mathtt{Alice}: & \textit{bonus} = \mathbf{270} & \\ \mathtt{Bob} \ : & \textit{bonus} = 130 & \textit{sponsor} = \mathtt{Alice} \\ \mathtt{Car} \ : & \textit{buyer} = \mathtt{Alice} & \textit{value} = 900\,. \end{cases}$$

The set E_2 is computed by removing $(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})$ from E_1, and then adding all the rule instances that are not applicable in σ_2. Since $(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})$ is applicable in σ_2, it is not added back in E_2, and we have

$$\begin{aligned} E_2 &= \mathcal{I}(\mathcal{R}, \mathcal{M}) \setminus \{(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})\} \\ A_{\sigma_2} &= \{(\mathtt{P}, \mathtt{Alice}, \mathtt{Car}), (\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})\} \\ A_{\sigma_2} \cap E_2 &= \{(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})\}\,. \end{aligned}$$

Note that $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$ was not applicable in σ_1 but, in the transition to σ_2, Alice's bonus has been increased to over 200 points, hence making $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$ applicable. As this rule instance has been inapplicable since its latest application, it must not be discarded by refraction: indeed, we have $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob}) \in E_2$. The fact that $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$ is given the opportunity of being applied in σ_2 contrasts with the sequential execution, where this opportunity was denied in state s_2.

The execution of $\mathcal{R}$ continues with a second application of $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$, and results in $\langle E_3, \sigma_3 \rangle$ with

$$E_3 = E_2 \setminus \{(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})\} \cup \{R \in \mathcal{I}(\mathcal{R}, \mathcal{M}) \mid R \text{ is not applicable in } \sigma_3\}\,.$$

In this configuration, $(\mathtt{P}, \mathtt{Alice}, \mathtt{Car})$ is not included in E_3 since it is applicable in σ_3. On the other hand, $(\mathtt{S}, \mathtt{Alice}, \mathtt{Bob})$ cannot be in $A_{\sigma_3} \cap E_3$ since either it is not applicable in σ_3, or it is and is thus not eligible.

As there is no rule instance that is both applicable and eligible in $\langle E_3, \sigma_3 \rangle$, the transition rule (1) cannot be applied and the execution of $\mathcal{R}$ ends.

5.3 Discussion

As can be seen on the example discussed in this section, the set of eligible rule instances in each configuration is easier to compute in a sequential execution than in a refraction-based one. This explains why adopting a sequential execution semantics enables rule engines to execute rule programs faster.

On the other hand, this example also demonstrated that the refraction-based semantics gives more opportunity to rule instances to execute, whereas the sequential mode imposes a stricter control. This can be seen as an advantage of Rete-like execution, as this semantics seems more natural. However, some BRMS users regard this richer semantics as less predictable, and appreciate the greater control provided by the explicit ordering of rules of the sequential execution mode, especially with large rule programs as can for example result from the automatic translation of database tables into rules.

6 Conclusion

Business Rule Management Systems (BRMS) provide business applications with the ability to externalize part of their logic as rule programs. For a long time, these rule programs have been executed with the semantics linked to the Rete algorithm. More recently however, an alternative to Rete has emerged, known as sequential, with a specific semantics.

In this paper, we give a formal description of the execution of rule programs by BRMS. In this description, we isolate the handling of rule eligibility in the control strategies of rule engines. We complete our formal description with the expression of selection and eligibility strategies for the Rete algorithm and for the sequential execution mode.

Finally, we illustrate our formalism with both a sequential and a refraction-based execution of an example rule-based application.

References

1. Baralis, E., Widom, J.: An algebraic approach to rule analysis in expert database systems. In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds.) VLDB. pp. 475–486. Morgan Kaufmann (1994)
2. Berstel, B., Leconte, M.: Using constraints to verify properties of rule programs. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops. pp. 349–354. ICSTW'10, IEEE Computer Society (2010)
3. Brownston, L., Farrell, R., Kant, E., Martin, N.: Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming. Addison-Wesley, Boston, USA (1985)
4. de Bruijn, J., Rezk, M.: A logic based approach to the static analysis of production systems. In: Polleres, A., Swift, T. (eds.) RR. Lecture Notes in Computer Science, vol. 5837, pp. 254–268. Springer (2009)
5. Cheng, A.M.K., Tsai, H.Y.: A graph-based approach for timing analysis and refinement of OPS5 knowledge-based systems. IEEE Transactions on Knowledge and Data Engineering 16(2), 271–288 (Feb 2004)
6. Cirstea, H., Kirchner, C., Moossen, M., Moreau, P.É.: Production systems and Rete algorithm formalisation. Research report, LORIA, Nancy (Sep 2004), `http://hal.inria.fr/inria-00280938/PDF/rete.formalisation.pdf`
7. Corticon Business Rules Management System, `http://www.corticon.com/Products/Business-Rules-Management-System.php`
8. Damásio, C., Alferes, J., Leite, J.: Declarative semantics for the Rule Interchange Format Production Rule Dialect. In: Patel-Schneider, P., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J., Horrocks, I., Glimm, B. (eds.) The Semantic Web — ISWC 2010, Lecture Notes in Computer Science, vol. 6496, pp. 798–813. Springer, Berlin / Heidelberg (2010)
9. Fages, F., Lissajoux, R.: Sémantique opérationnelle et compilation des systèmes de production. Revue d'Intelligence Artificielle 6(4), 431–456 (1992)
10. Fair, Isaac, and Company: High-volume batch processing with Blaze Advisor. Computer World U.K. (Mar 2007), `http://www.computerworlduk.com/white-paper/business-process/5092/high-volume-batch-processing-with-blaze-advisor/`
11. FICO^{TM} Blaze Advisor ®, `http://www.fico.com/en/Products/DMTools/Pages/FICO-Blaze-Advisor-System.aspx`
12. Floyd, R.W.: A descriptive language for symbol manipulation. Journal of the ACM 8(4), 579–584 (1961)

13. Forgy, C.: OPS5 User's manual. Tech. Rep. CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh (Jul 1981)
14. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artificial Intelligence 19(1), 17–37 (1982)
15. Hanson, E., Hasan, M.S.: Gator: An optimized discrimination network for active database rule condition testing. Tech. Rep. TR93-036, University of Florida (1993)
16. IBM Websphere Operational Decision Management, `http://www.ibm.com/software/websphere/products/business-rule-management`
17. IBM: IBM Websphere Operational Decision Management v7.5 User's Manual (2011), `http://publib.boulder.ibm.com/infocenter/dmanager/v7r5/`
18. McCoy, D.W., Sinur, J.: Achieving Agility: The Agile Power of Business Rules. Gartner (2006), `http://www.gartner.com/DisplayDocument?doc_cd=138218`
19. Mettrey, W.: A comparative evaluation of expert system tools. Computer 24, 19–31 (1991)
20. Miranker, D.P.: TREAT: A better match algorithm for AI production systems. In: Proceedings of the Sixth National Conference on Artificial Intelligence – Volume 1. pp. 42–47. AAAI'87, AAAI Press (1987)
21. Oracle Business Rules, `http://www.oracle.com/technetwork/middleware/business-rules/overview`
22. Red Hat: JBoss Enterprise BRMS, `http://www.jboss.com/products/platforms/brms`
23. Red Hat: Drools Expert User Guide (2010), `http://downloads.jboss.com/drools/docs/5.1.1.34858.FINAL/drools-expert/html_single/index.html#d0e2086`
24. Rosenthal, D.: Adding meta rules to OPS5: A proposed extension. ACM SIGPLAN Notices 20(10), 79–86 (Oct 1985)
25. de Sainte Marie, C., Hallmark, G., Paschke, A.: Rule Interchange Format, Production Rule Dialect. Recommendation, W3C (2010), `http://www.w3.org/TR/rif-prd/`
26. SAP NetWeaver Business Rules Management, `http://www.sap.com/platform/netweaver/components/brm/index.epx`
27. Schmolze, J.G., Snyder, W.: Detecting redundancy among production rules using term rewrite semantics. Knowledge-Based Systems 12(1–2), 3–11 (1999)
28. Soloway, E., Bachant, J., Jensen, K.: Assessing the maintainability of XCON-in-RIME: Coping with the problems of a VERY large rule-base. In: AAAI. pp. 824–829 (1987)
29. Taylor, J., Raden, N.: Smart (enough) systems: How to deliver competitive advantage by automating the decisions hidden in your business. Prentice Hall, Upper Saddle River, NJ, USA (2007)