WORKING PAPER – !! DRAFT, e.g., lacks most references !!

Version of December 19, 2005

Defining an Abstract Core Production Rule System

Benjamin Grosof
Massachusetts Institute of Technology,
Sloan School of Management,
50 Memorial Drive, Cambridge, MA 02142 USA,
bgrosof@mit.edu, http://ebusiness.mit.edu/bgrosof

Abstract

We define an abstract core production rule system. This is part of our overall foundational work on the Production Logic Programs knowledge representation (KR) approach of web rules that are semantically interoperable between all four of the currently most commercially important families of rule systems, including production rules, relational database management systems, event-condition-action rules, and Prolog.

Contents

1 Definition of Abstract Core Production Rule System

1.1 Introduction

Production rule systems in their full generality are fairly complicated. Indeed, they can be used as fairly general-purpose programming mechanisms. In this paper, we are concerned only with a restricted set of core capabilities and features of a production rule system, where this core is shared across the family of production rule systems. We abstract away from most of the details of a production rule system, particularly the details of the rule language syntax and the engine implementation.

Our motivation is to provide a definitional part of our overall foundational work on the Production Logic Programs knowledge representation (KR) approach of web rules that are semantically interoperable between all four of the currently most commercially important families of rule systems, including production rules, relational database management systems, event-condition-action rules, and Prolog.

This is also related to the effort in the OMG Production Rules Representation (PRR) standardization effort to develop a common abstract definition of a production rule system. In near future work, we will seek to compare the formulation in this paper to the (currently being drafted) OMG PRR formulation. Preliminary indications are that there is a close relationship but somewhat different emphases and degrees of elaboration in the two formulations.

Accordingly, we define an abstract version of a restricted production rule system, as follows.

1.2 Overall Level

An *abstract core production rule system* consists of: a *language*; a *knowledge base (KB)* expressed in that language; and an *engine*.

In production rule systems terminology, the engine is often known as the "inference" engine, since it generates conclusions drawn from the production rules. More precisely, however, it could be described as an "inferencing+action" engine since part of the behavior (in general) specified by the production rules is to perform side-effectful actions by calling attached procedures.

The knowledge base consists of: *premises* and *conclusions* expressed in the language.

An *system instance* of the abstract core production rule system consists of a particular KB (starting simply with its premises) and a particular instance of the engine process.

The (set of) premises consists of *premise facts*, *production rules*, *attached procedures (aproc's)*, and an *agenda control strategy*. The attached procedures are for *tests*

and *actions* (more details about that are given below). In production rule systems terminology, these attached procedures are often known as (a category of) "functions".

The (set of) conclusions consists of: *conclusion facts*; and *launched action calls*. Each launched action call is an invocation (i.e., call) of an aproc.

The engine consists of: a *KB store*; an *agenda*; a *pattern matcher*; a *firing process*; an *action launcher* (a.k.a. "execution engine"); and a *main engine process* . Here, and below, "process" means in the sense of business process, i.e., a method or procedure

An *episode* is a run of the main engine process. It starts with a premise KB and then generates all of its associated conclusions. This is accomplished by iteratively pattern-matching the rules against the working memory's set of facts and then firing the successfully-matched rule instances to create new conclusions. The results of calls to tests are treated as additional virtual facts for purposes of pattern-matching. New (derived) conclusion facts are added to the working memory and are available for pattern-matching in subsequent iterations. The overall iteration is performed until exhaustion — i.e., until no new rule instances can be successfully-matched. The conclusion facts consist of the derived conclusion facts together with the premise facts. In addition, the conclusions include a set of launched action calls — whose executions have side effects.

1.3 Language and KB: Details

Next, we describe the language and KB in more detail.

A *fact* is a ground atom. In production rule systems, an atom, and the language generally, is furthermore required to be **function-free**. *Function-free* means that no logical functions are permitted. In the KR literature, logical functions are sometimes known as "constructors", and function-free is sometimes known as the "Datalog" restriction. Note that here (and throughout this paper), we mean atom, ground, function, and function-free each in their usual sense from logical knowledge representation (KR), e.g., from first order logic or logic programs.

An atom can be an *ordered atom* or an *unordered atom*. An ordered atom has a sequenced argument collection, i.e., an ordered tuple of arguments. It has the form:

$$(p\ t_1\ \ldots\ t_m)$$

where p is (the name of) a predicate, each t_i is a term, and $m \geq 0$. If $m = 0$ then the parentheses may be omitted.
Each term may be a constant or a (logical) variable.
For example:

$$(loves\ mary\ john)$$

says that Mary loves John.
A variable is prefixed by "?". For example:

$$(loves\ mary\ ?x)$$

Note that in this parenthesized syntactic style for an abstract core production rule system, we follow the Lisp-like parenthesized syntactic style of CLIPS and Jess.

An unordered atom has an unsequenced argument collection, in which each argument is the value of a named *slot*. The slot names must be distinct within a particular atom.

An unordered atom is thus like a row in a relational database table. In such a table, each row has individual named data fields corresponding to the table's columns.

An unordered atom has the form:

$(p\ (n_1\ t_1)\ \ldots\ (n_m\ t_m))$

where each n_i is a *slot name*. As with ordered atoms, p is a predicate, each t_i is a term, $m \geq 0$, and each term may be a constant or a variable. For example:

$(discount\ (customer\ Fred)\ (percentage\ 30)\ (product\ basicDVD))$

says that customer Fred gets a 30% discount on the basic DVD.

In production rule systems terminology, an atom is also known as (a kind of) "*pattern*".

Premises and conclusions are specified via *statements*. A fact statement has the form of a command to assert a fact, followed by an end-of-statement delimiter ".". It has the form:

(`assert` F).

where F is a fact.

For example:

(`assert` $(loves\ mary\ john)$).

asserts the fact that Mary loves John. The assert command here is a system command — a special one.

A premise fact statement, and a conclusion fact statement, each have the form of a fact statement.

A *production rule* is a premise statement that specifies an if-then rule. The if part of the rule is also known as its "left hand side (LHS)", and the then part of the rule is also known as its "right hand side" (RHS). In the KR literature about rules in general, the if and then part of a rule are also often known as the antecedent and consequent, respectively, or as the body and head, respectively. In addition to its if and then part, a production rule has a *rule name*. This rule name is required to be unique within a given KB. A production rule has the form:

(`defrule` $lab\ \ LHS \Rightarrow RHS$).

where lab is the rule name, LHS and RHS are expressions whose form are detailed below, and "." is the end-of-statement delimiter.

The LHS and RHS are built up out of *element* expressions. The four basic kinds of elements are atoms, assertions, tests, and actions. Additional kinds of elements specify logical connectives, including "and", "not", and "or". The LHS is built up out of atoms and tests, using the logical connectives "and", "not", and "or" (which may be nested freely). The RHS is built up out of assertions and actions, using the logical connective "and" (which may be nested freely). If the LHS has a top-level "and" connective, it may be left tacit (i.e., omitted from the explicit syntax). Likewise if the RHS has a top-level "and" connective, it may be left tacit.

An atom within a rule need not be ground, i.e., logical variables may appear in it.

An *assertion* asserts an atom. It has the form:

$(\texttt{assert}\ A)$

where A is an atom. For example:

$(\texttt{assert}\ (loves\ mary\ ?x))$

A *test* is a call to an attached procedure (aproc). The test aproc is required to return a boolean value; it thus can be viewed conceptually as similar to a predicate. The test aproc is called for the sake of its return value, rather than for the sake of its side effect. A test has the form:

$(\texttt{test}\ (q\ t_1\ \ldots\ t_m))$

where q is (the name of) a test aproc, each t_i is a term, and $m \geq 0$. For example:

$(\texttt{test}\ (<\ ?x\ 50))$

where $<$ is the name of the less-than test aproc. $<$ is typically available as a system built-in. However, in general, a test aproc may be user-supplied.

An *action* is a call to an attached procedure (aproc). The action aproc is called for the sake of its side effect, rather than for the sake of its return value. (By contrast, with a test aproc, the sakes are vice versa.) An action has the form:

$(\texttt{test}\ (a\ t_1\ \ldots\ t_m))$

where a is (the name of) an action aproc, each t_i is a term, and $m \geq 0$. For example:

$(\texttt{action}\ (printout\ t\ ``Maryisinlove!''))$

which when called results in printing out "Mary is in love!". Here, $printout$ is the name of a printing action aproc. $printout$ is typically available as a system built-in. Another example is:

$(\texttt{action}\ (pay2\ ?customer\ ?seller\ 75.00\ dollars\ byMasterCard))$

which when called results in $?customer$ paying $?seller$ via a $75.00 charge on MasterCard. Here, $pay2$ is the name of a (hypothetical) user-supplied action aproc for payment.

The "*and*" connective performs logical conjunction, as usual in logical KR, e.g., in first order logic or logic programs. An "and" expression has the form:

$(\texttt{and}\ E_1\ \ldots\ E_m)$

where each E_i is an expression, and $m \geq 0$. As usual, if $m = 0$, the truth value of the "and" expression is regarded as True.

The "*not*" connective performs logical negation. Negation in production rule systems is essentially (a kind of) **negation-as-failure**, i.e., default negation, rather than classical negation. A "not" expression has the form:

$(\texttt{not}\ E)$

where E is an expression.

The "*or*" connective performs logical disjunction. An "or" expression has the form:

$(\texttt{or}\ E_1\ \ldots\ E_m)$

where each E_i is an expression, and $m \geq 0$. As usual, if $m = 0$, the truth value of the "or" expression is regarded as False. (Note that False here means essentially in the

sense of negation-as-failure, rather than classical negation).

We can summarize the syntax of a production rule as follows:

(defrule *lab* ({*nested* and|not|or} *of* {atom|test})
⇒ ({*nested* and} *of* {assert|action}))

Here, *atom*, *test*, *assert*, and *action* stand for atom, test, assertion, and action expressions. "|" stands for the choice operator. *lab* is the rule name.

We define a ***normal*** production rule to be one that has a connective structure similar to a "normal" rule in declarative logic programs. The LHS consists of an "and" of expressions that are each either an atom, a test, the "not" of an atom, or the "not" of a test. The RHS consists of single assertion or action expression.

(Note that "normal" here is terminology specific to this paper, rather than being previously in common use in production rule systems terminology. It is inspired by the logic programs KR literature, where it is in common usage.)

We can summarize the syntax of a ***normal*** production rule as follows:

(defrule *lab* ({and} *of* {atom|test|(not atom)|(not test)})
⇒ ({assert|action}))

In addition to facts and production rules, there are two other kinds of premises: aprocs (for tests and actions) and agenda control strategy.

An ***aproc declaration*** statement specifies an aproc as a premise. It has the form:

aproc A.

Here, A is an aproc name, e.g., $printout$ or $pay2$ or $<$. "." is the end-of-statement delimiter. The statement declares the aproc to be accessible, i.e., callable, by the engine process. The aproc takes an ordered tuple of input parameters (recall test and action expressions described above.)

In current commercially practical production rule systems, some aproc's (e.g., $printout$ or $<$) are built in to the system, rather than supplied by the user (e.g., $pay2$). These can be viewed as implicitly declared.

An aproc is defined as a procedure in some programming language, e.g., as a method in C/C++ or Java or C#, or as a web service in WSDL. As is normal for procedures in those languages, the aproc takes an ordered tuple of input parameters. Its interface is available (to the production rule system) indirectly via its name, in the usual manner for these programming languages. An aproc used in a `test` expression must return a boolean truth value and should have no side effect. An aproc used in a `action` expression should have a side effect.

We will describe the agenda control strategy a bit later, after first describing the agenda.

In addition to facts, there is one other kind of conclusion: launched action calls. A launched action call has the form of a ground action expression, i.e.,

(`action` $(ac_1 \ \ldots \ c_m)$)

where a is (the name of) an (action) aproc, each c_i is an individual constant, and $m \geq 0$.

1.4 Engine: Details

Next, we describe the engine in more detail.

The KB store consists of: a working memory, which is a store for premise facts and conclusion facts; a production rules store; an aprocs store, an agenda control strategy store, and a launched action-calls store.

The agenda consists of: an activation list; and an agenda ordering. The activation list is a list of "activated" ground instances of production rules. "Activated" in production rule systems terminology means that the rule instance's body is satisfied according to current state of the KB store, notably of the working memory. The agenda ordering is a partial ordering over the members of the activation list.

The pattern matcher is a component process within the engine that generates and maintains the activation list. The pattern matcher computes the rule instances whose LHS's are satisfied, i.e., that are "applicable" in production rule systems terminology. "Applicable" rule instances that have not yet fired constitute the activation list. In most current and previous commercially practical production rule systems, the pattern matcher is implemented by using Rete algorithm.

The agenda control strategy (ACS) is defined as a partial ordering ACS over the set of potential activations, i.e., over the set of ground instances of the production rules. For every pair of activations $av1$ and $av2$, the agenda control strategy indicates whether $av1$ is higher than $av2$ in the partial ordering — which can be written formally as:

$$ACS(av1, av2) \ .$$

The main engine process forms the agenda ordering based on the agenda control strategy.

Current commercially practical production rule systems often permit complicated agenda control strategy specifications. For example, in Jess, each activation includes not only the production rule and the instantiating bindings to that rule's variables, but also other information, including: the facts and test aproc calls used to satisfy the rule's LHS, and the time at which that satisfaction was detected. In Jess, the agenda control strategy is specified by a combination of an overall "conflict resolution strategy" together with salience. A few "conflict resolution strategies" are available as predefined choices, including first-in first-out (FIFO) (called "breadth" in Jess) and last-in first-out (LIFO) (called "depth" in Jess). Static salience is an optional user-defined score for each production rule statement, which takes precedence over the "conflict resolution strategy". In addition, Jess permits the salience and/or "conflict resolution strategy" to be computed dynamically using general user-defined procedures (written in Java code). However, complicated reliance upon, and extensive "hacking" of, the agenda control strategy by the user is typically discouraged in production rule systems (e.g., the Jess documentation/book cautions against it).

The *firing process* is a component process within the engine. It computes which rule instance is next to be fired, then fires that rule instance. To compute which rule instance is next to be fired, it applies the agenda control strategy to the activation

list to determine the agenda ordering, then chooses a rule instance that is maximal wrt that ordering, i.e., that is "*agenda-maximal*" (in our terminology). To fire a rule instance, the firing process executes the rule instance's RHS and then removes the rule instance from the activation list. To execute its RHS, the firing process adds its asserted facts into the working memory and launches its actions by calling the action launcher. When adding an asserted fact into the working memory, the fact is marked as derived, i.e., as a derived-conclusion.

The action launcher launches an action by adding it to the launched-action-call store and actually calling the corresponding aproc with the instantiated parameters.

The *main engine process* orchestrates an episode of generating all the conclusions from a starting premise KB. (Recall the overview in sub-section 1.2.) Figure 1 gives a pseudo-code description of it.

1.5 Additional Logical Connectives

(This subsection is described in less detail than the others in this section.)

We extend PR1 to add enhanced logical connectives and quantifiers, in roughly the manner of Lloyd-Topor, including: `or`, `and`, and `exists` in the body; nesting of all these and `not` in the body; and `and` in the head. This is defined via transformations that reduce to the case without these, in a manner that is similar, but not identical, to the usual Lloyd-Topor (treatment of) logical connectives and quantifiers in LP. One main difference from Lloyd-Topor is in regard to the `or` connective when the rule has an action head; production rules behavior (e.g., in Jess) is defined as generating a set of "*subrules*", one per OR branch. This subrule generation may lead in worst case to an exponential blowup in the size/number of rules in the corresponding PLP, when multiple `or`'s nested within `and`'s. The other main difference from Lloyd-Topor is in regard to the `exists` connective, which is a kind of twice-nested `not`.

1.6 Summary

We summarize the foregoing formally as a definition.

Definition 1 (Abstract Core Production Rule System)

An *abstract core production rule system (PR1)* is defined as above in this section.

•

The structure of the overall PR1 can be summarized as follows.

```
system := language, KB, engine.
KB := premises, conclusions.
premises := premise facts, production rules, aprocs,
             agenda control strategy.
conclusions := conclusion facts, launched action calls.
```

```
1. load the KB premises into the KB store;
     /* (first, empty out the working memory and agenda;) */
2. do loop {
  2.1. pattern match;
         /* with respect to (updated) working memory.
            I.e., find every rule instance whose LHS
            is satisfied, and that has not yet been
            fired.  This creates an (updated)
            activation list. */
  2.2. if activation list is empty, then go to step (3.) to
         finish up.
  /* begin firing process:  */
    2.3. order the agenda (using the agenda control strategy),
           and
         select an agenda-maximal activated rule instance RIN
           as next to fire;
    /* begin firing that rule instance: */
      2.4. add RIN's RHS's asserted facts as derived into the
             working memory;
             /* A derived fact may have already been present
                in the working memory. If so, just keep one
                copy of the fact, overall. */
      /* begin action launching */
        2.5. add RIN's RHS's action calls to the
               launched-action-call store.
        2.6. perform RIN's RHS's action-calls by performing
               the corresponding aproc calls;
               /* these have side effects */
      /* end of action launching */
    /* end of firing that rule instance */
    2.7. remove RIN from the activation list;
           /* since it has now been fired */
  /* end firing process */  }
3. finish up
  3.1. return the conclusions set from the conclusions store;
         /* The conclusion set includes both facts and
              launched-action-calls.
            The facts include both premise facts and
              derived facts.
            Also, the launched-action-calls when executed
              had side effects. */
  3.2. stop;
```

Figure 1: Pseudo-Code for Main Engine Process of Abstract Core Production Rule System

```
engine := KB store, agenda, pattern matcher,
              firing process, action launcher.
KB store := working memory, production rules store,
              aprocs store, agenda control strategy store,
              launched action calls store.
agenda := activation list, agenda ordering.
```

Here, "`:=`" is used to mean "consists of", as in EBNF and other definitional grammars.

1.7 Other Aspects of Production Rule Systems

There are a number of other aspects of expressiveness in production rule systems that are not captured in the above notion of an abstract core production rule system.

These aspects include: builtins; datatyping; comments; lists; "constraint expressions" (similar to logical connectives and quantifiers); retracting or updating/modifying facts in the working memory; static and dynamic "shadow facts" from the surrounding programming language environment; additional engine control mechanisms such as "modules" (multiple cooperating rulebases and engine threads); various system commands, e.g., for starting, halting, resetting, and tracing the engine process and knowledge base store, and for modifying the agenda control strategy; additional fairly general procedural-programming mechanisms, e.g., scripting; and some other features.

Some of these aspects can be, and some cannot be, reasonably expected to be treated declaratively in future. The above list is very roughly sorted from more to less declarative in prospect.

2 Acknowledgements

The author wishes to thank Said Tabet for useful discussions.

Support for this work was provided by awards from DAML, the DARPA Agent Markup Language program.