ILOG's position on Rule Languages for Interoperability

Colleen McClintock (cmcclintock@ilog.com), Christian de Sainte Marie (csma@ilog.fr)

ILOG (www.ilog.com)

Scope: Business Rules and Production Rules

As a major vendor of Business Rule Management Systems (BRMS), ILOG interest in rule languages is mostly focused on Business Rules and Production Rules.

Business Rules are usually defined as atomic and highly specific and structured statements that constrain some aspect of a business, process or activity, and that control or influence its behaviour. Business rules are widely used to implement regulations, procedures, business policies and other potentially volatile decisions that impact a business, a process or an activity.

In this paper, we refer more specifically to executable business rule statements for use in some rule-driven system.

The most noticeable characteristic of executable business rules is that their execution results necessarily in executing some actions: specifically, the actions that are required to enforce the underlying business policy (or non-executable business rule). From an inference point of view, the true assertion that is inferred from a business rule is thus always that some specific actions must be executed. In other words, business rules specify which actions can or have to be executed when certain conditions are met.

A car-renting business can for instance have a policy that "the driver must be 21 or older": the corresponding executable business rule – that is, the rule that makes the policy enforceable – is that "rentals must be denied to drivers below 21".

As a consequence of that characteristic, business rules are often represented as production rules when they are meant to be directly executed by an automated system: a *production rule* is an independent statement of programming logic that specifies the execution of one or more actions in the case that its conditions are satisfied; it is usually defined in the context of being executable by an inference engine. In production rule form, our example above would read: "if the driver in a car-rental request is less than 21 year old, then reject the car-rental request".

Production rules are the most convenient way to represent most business rules, as well as the most widely supported by existing inference engines, including the main commercial ones. See some examples of business rules in production rule form in appendix 1.

Requirements

A standard business rules language is a frequent request of business rule application developers. Developers view a standard rule language as a key enabling technology, allowing them to build tools and applications that can generate and manage rules, and share them between applications without having to care about the specifics of the underlying rule engine.

In our view, a standard rule language must satisfy a number of core requirements that can be classified in four categories:

- requirements with respect to the execution of the rules;
- requirements with respect to the expressiveness of the language;
- requirements with respect to the usability and the adoptability of the standards;
- and requirements with respect to its compatibility with other standards.

Execution

A standard rule language must be able to represent the rules in a way that is unambiguous with respect to execution.

A rule stating that "if there is a truck without a driver and a free driver, the driver must be assigned to the truck", applied in a context where there are three free drivers and two free trucks, can result in each truck having a different driver and one driver remaining free (which is probably the intended result), or both trucks having all three drivers assigned to them (e.g. if the rule is executed in sequential mode), or many other potential outcomes, depending on the execution semantics of the rule engine.

It is clearly not enough to have a standard rule language for exchanging business rules and then have the rules execute differently on different engines: for a standard to be useful, it must enable a representation that guarantees that two rules or sets of rules applied on the same data in two different environments will have exactly the same results.

The standard rule language must be able to represent unambiguously the operational semantics of the main business rule engines, ranging from RETE-style forward-chaining engines, to decision tables and trees to procedural engines and more.

Expressiveness

The standard rule language must be expressive enough to represent most if not all of the rules allowed by the main business rule engines, possibly starting with a useful subset determined by the use-cases (as opposed to technical considerations).

A key motivation for Business Rule Management Systems is to enable business users to author and manage business rules: it would clearly not be acceptable to have a standard rule language for exchanging business rules that would require business users to author rules using this language.

Essentially, the language must be able to represent typed first order queries, complete with variable definition and binding, and all the standard logical, arithmetical and set-theoretical operators, for the condition (left-hand part) of the rules; and a number of actions for the right-hand part, from the standard "assert", "retract" and "modify" to the execution of application specific functions.

Adoptability

The key issues here are: simplicity, ease of implementation and ease of deployment.

If one of the initial core-values of the standard is interchange, then clearly interchange must be illustrated to the stakeholders. This requires that mainstream business rule engines implement the standard.

A first requirement, closely related to but different from the language expressiveness, is that translation from and to the internal rule languages used by the relevant rule engines must be reasonably easy and reasonably efficient, where "relevant" depends on – or defines – the scope of the standard. From our viewpoint, the relevant rule engines are ILOG JRules and its main competitors.

Deployment and adoption issues also make an XML-based format the obvious choice as a basis for the standard. XML Schema, though complex, has the expressiveness required to define the required grammar.

Notice that human friendliness, and specifically, human readability is usually not a requirement: the standard rule language is not meant to be used for rule modelling or communication with the user, but for rule exchange between rule engines.

Compatibility

The major rule engine vendors, such as ILOG and Fair Isaac, are currently working within the OMG to build extensions to UML to allow business rules to be modelled [1]. Once the standardized meta-models are defined, rule engine and tools vendors can generate executable business rules (perhaps in vendor specific formats) from the models. The specification will also define an XMI XML Schema Description for production rules, based on the proposed meta-model, in order to support the exchange of production rules between modeling tools and inference engines.

A standard rule language must be compatible with that XML schema.

In the same way, JSR-94, The Java Rule Engine API, defines a lightweight-programming interface that constitutes a standard API for acquiring and using a rule engine [2]. A primary input to a rule engine is a collection of rules called a rule execution set. The rules in an execution set may be expressed in a XML-based rule language. However, the scope of the specification specifically excludes defining a standard language to describe the rules within a rule execution set.

A standard rule language must satisfy the requirements set by JSR-94 on its input rule language.

The aim is to develop a set of compatible and convergent standards for all the dimensions of the Business Rules standardisation space, including rules modelling, rules execution and rules exchange and sharing.

Discussion

There have been various attempts to define a standard rule language, and all have had limited success. For example, the Simple Rule Markup Language (SRML) was defined and published by ILOG as a basic XML representation for forward-chaining (Java) business rules [3]. Why did this initiative have such little impact on the industry?

Part of the answer is that SRML did not address the requirements regarding the execution of the rules. As a consequence, the XML Schema, while interesting, did not enable software developers and architects to solve new or existing problems using business rules technology. In fact it could be viewed as essentially disruptive: it requires an investment from the rule engine vendors to implement, and it requires an additional investment from the software developers and architects to add disambiguating information in the condition of the rules to guarantee the expected behaviour.

Other proposals, such as RuleML [4], raise similar problems of usability and adoptability, due to inadequate expressiveness or semantics (from a Business Rules point of view).

On the other hand, we are aware that a standard rule language that would attempt to accommodate all the requirements of all the use cases with respect to expressiveness, inference, and operational semantics would face implementation and adoption difficulties due to its complexity, as well as consistency and tractability problems.

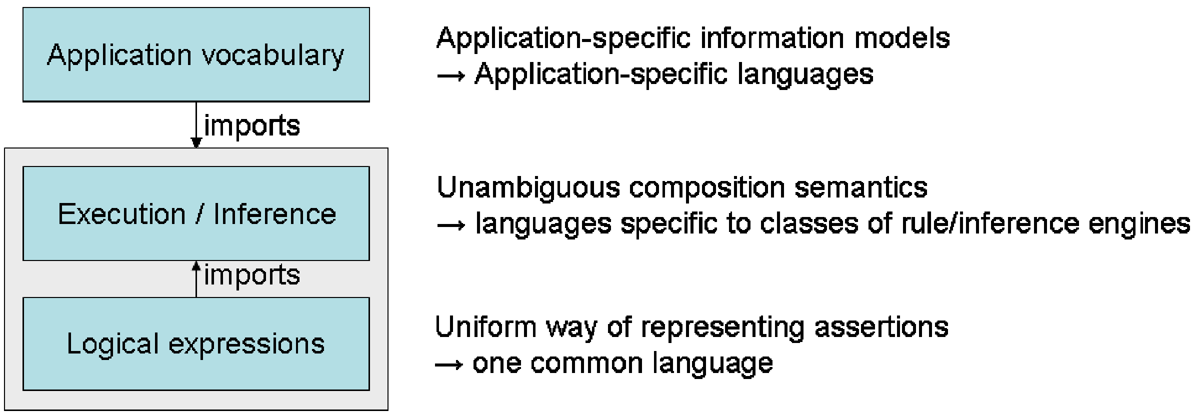
These considerations call for a modular answer to the question of a standard rules language.

One option would be to specify different standard languages for different types of rules and usages, and then, possibly, bridges between them; another option could be to specify the different components on top of a common language, in a layered architecture. ILOG would support any one of the two options. In the first case, we would contribute to the specification of a standard Production Rules exchange language. We develop our view with respect to a layered architecture in the next section.

A layered architecture

In the perspective of a standard rule language with a layered architecture, we believe that the key is to distinguish, in all the different requirements on a standard rule language, what is specific to a inference mechanism/theory from what is not: the non-specific part should be represented in a unique standard language common to all the different kinds of rules, whereas everything that is specific to a logic or inference mechanism should be represented in one of a set of independent languages specific to different (classes of) inference or rule engines. Everything that is specific to the application in a rule should, of course, be represented in a separate application specific language.

The result would be a three-tiered architecture, as illustrated below, where the top layer is composed of the languages for representing the application-specific vocabularies and functions, the middle layer provides for the unambiguous representation of the semantics of rules and rulesets, via a set of standard languages specific to classes of inference engines; and the bottom layer is a standard common language that enables an uniform representation of assertion. The two bottom layers (framed in the picture) compose the standard for exchanging rules.


Application vocabulary
imports
Execution / Inference
imports
Logical expressions
Application-specific information models
→ Application-specific languages
Unambiguous composition semantics
→ languages specific to classes of rule/inference engines
Uniform way of representing assertions
→ one common language

The representation of any specific rule requires the assertion language (bottom layer), a component from the top layer, to populate it with the application vocabulary, and a component from the middle layer, to provide the semantics of the rule.

Consider, for instance, the following business rule (a simplified version of the first rule in our example, see appendix): “If the shopping cart contains between 2 and 4 items and the purchase value is greater than $100 and the customer category is gold, then apply a 15% discount on the shopping cart value”. Let us rewrite it in a slightly more structured way:

`“Let X be a customer, let Y be X’s shopping cart, let Z be the collection of items in Y`

`If` `(the cardinal of Z is between 2 and 4) and (the sum of the price of the elements of Z is greater than 100) and (X’s category is gold)`

`Then modify` `Y’s value applying a 15% discount,` **`(engine note: only once even if the rule fires more than once for the same shopping card)`**`”`

Types, attributes or functions such as ’`customer`’, ‘`the price of`’ or ‘`apply a discount`’ , and everything that is underlined in the example, belong to the application-specific layer, that is, a not necessarily standard language for representing the business object model presented in the appendix. The structure of the rule itself, action types such as ‘`modify`’ and execution instructions (and everything that is in bold-face in the example) belong to the component of the standard that is specific to production rules (second layer). Everything else (plain courier in the example) belongs to the common core of the language (bottom layer), that is, how to declare a variable or bind it, logical connectors, arithmetical operators and constants etc..

Generally, the structure of a production rule being: “`variable declarations, conditions, actions (+ engine-specific execution instructions)`”, we suspect that the first two parts, variable declarations and conditions, could always be represented in the common core (populated with application-specific vocabulary), whereas the latter two parts will belong to the business rule engines-specific language (plus application-specific language, of course).

Since all the rule languages trace their roots back to predicate calculus, we expect the common core to consist mainly of the well-known, well-founded and non-controversial instruments of logical notations: quantifiers, logical operators, definition and binding of variables, augmented with representations for the standard arithmetical and set-theoretic types and operations, etc.

As usual, the benefit of a modular approach is scalability: the development of each component of the standard in the proposed architecture would be tractable. Even more important, consensus on the specification of each component can be achieved much more easily than it would for one single all-purpose standard rule language: although everybody uses it and relies on it, there the common core language should be largely non-controversial. On the other hand, the layer of inference-specific languages can be as fine-grained as required: the specification of each independent component will require consensus within an arbitrarily limited community of convergent interests only.

Bootstrapping adoption

By moving the inference/rule engine-specific aspects back into the hands of more limited – and thus expectedly more coherent – communities of interest, the proposed modular architecture leaves with the stakeholders the choice of their approach to the development of their component, depending on their constraints and priorities.

ILOG, for instance, would focus its effort on the production rule-specific component. With regards to that component we would propose that standardization should be approached in a very pragmatic, software engineering focused manner; starting with the requirements of a limited number of mainstream engines and working iteratively, showing and providing value to architects, engineers and vendors (the stakeholders), and slowly building a critical mass of implementations and thinking around this difficult problem. By working from the bottom-up, in a tactical way, value can be created at regular intervals without provoking complex and lengthy debate. A first version of the standard could thus be specified and start being adopted within a matter of months, provided that an initial version of the core assertion language – possibly with drastically limited expressiveness – can be standardised fast enough.

We can imagine that the approach would not fit the requirements of other communities, and that other components will be best developed in a top-down, theory-oriented way. However, the adoption bootstrapping effect can extend from one component to others and the whole standard. A benefit of the common assertion layer is, indeed, that it builds a natural bridge between different rules technology and usages: for instance, Business Rule engines such as ILOG JRules, for instance, typically use RETE-like pattern-matching techniques to check conditions; given a common assertion language, the variable declarations and condition parts could be passed to a theorem-prover when needed, instead. Adoption at one point of the rule languages space can thus provide incentives and foster adoption at other points.

Standardization therefore becomes a measurable and usable feature, with provable interoperability between a limited set of initial engines, possibly as few as two. From two initial engines a user base and developer community can bootstrap, and interoperability with the standard will assert itself as a market force (and source of requirements) for additional developers.

References

[1] Object Management Group, Production Rule Representation RFP, www.omg.org/cgi-bin/doc?br/03-09-03

[2] Java Community Process, JSR 94: JavaTM Rule Engine API, www.jcp.org/en/jsr/detail?id=94

[3] ILOG, Simple Rules Markup Language (SRML), DTD with annotation in comment declarations, http://xml.coverpages.org/SRML-simpleRules-dtd.txt

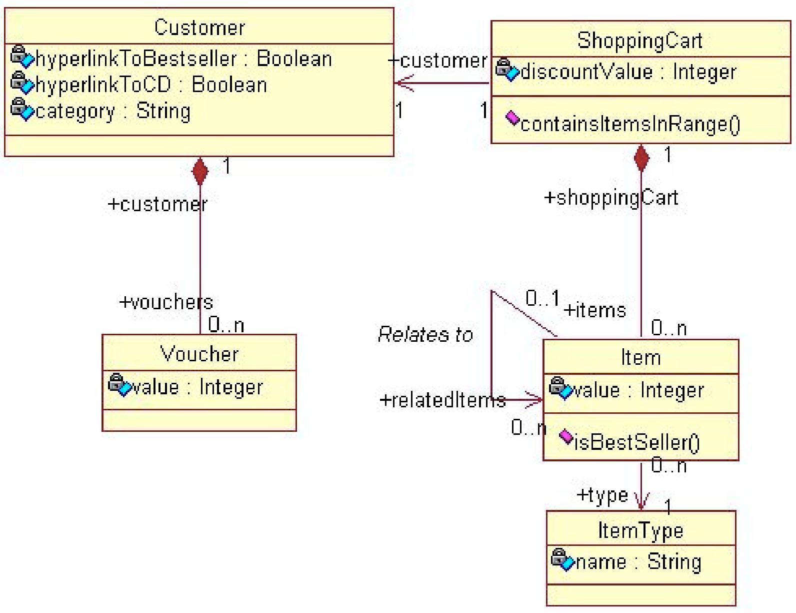
[4] RuleML: the Rule Markup Initiative, www.ruleml.org

Appendix: Production rules examples

These examples are used as an use case in our work with the team that develop the specification of a standard meta-model of production rules in answer to OMG's Production Rules Representation RFP.

Class Diagram

This section presents the UML class diagrams used to model the application rules.



Production rules with IRL translations

IRL is the internal rule language used in ILOG JRules.

Rule 1: discount

English

If the shopping cart contains between 2 and 4 items and either the purchase value is greater than $100 and the customer category is gold or the purchase value is greater than $200 and the customer category is Silver then apply a 15% discount on the shopping cart value.

IRL

```
rule discount {
when
{
    ?customer: Customer();
    ?shoppingCart: ShoppingCart(customer == ?customer);
    evaluate((?shoppingCart.containsItemsInRange(2, 4)) &&
             (((((?shoppingCart.getValue() > 100d) &&
                    (?customer.category equals "Gold")) ||
                 ((?shoppingCart.getValue() > 200d) &&
                    (?customer.category equals "Silver"))))));
}
then
{
    modify ?shoppingCart
    {
     shoppingCart.discountValue
                 = shoppingCart.discountValue + 15f);
   }
}
}
```

Rule 2: noCDItem

English

If there is no CD item in the customer shopping cart then add a hyperlink to the CD page in the customer web page.

IRL

```
rule noCDItem {
when
{
  ?customer1: Customer();
  ?shoppingCart1: ShoppingCart(customer == ?customer1);
  not Item(type == ItemType.CD ; shoppingCart == ?shoppingCart1);
}
then
{
  modify ?customer1{ hyperlinkToCD = true; }
}
}
```

Rule 3: atLeastOneBook

English

If there is at least one book item in the customer shopping cart and this book is a bestseller then add a hyperlink to the bestsellers page in the customer web page.

IRL

```
rule atLeastOneBook {
when
{
  ?customer1: Customer();
  ?shoppingCart1: ShoppingCart(customer == ?customer1);
  exists Item(shoppingCart == ?shoppingCart1 ; isBestseller());
```

```
        }
        then
        {
          modify ?customer1 { hyperlinkToBestseller = true; }
        }
        }
```

Rule 4:atLeast3Items

English

If there are at least 3 items of the same type in the customer shopping cart and each item's value is greater than $30 then give to the customer a voucher whose value is 10% of the cheapest item.

IRL

```
        rule atLeast3Items{
        when
        {
          ?customer1: Customer();
          ?shoppingCart1: ShoppingCart(customer == ?customer1);
          ?itemType1: ItemType();
          ?items: collect Item(type == ?itemType1 ; value > 30)
                                  in ?shoppingCart1.getItems()
              where (size()>3);
        }
        then
        {
          bind ?var1 = ?items.elements();
          bind ?min = 0;
          while (?var1.hasMoreElements())
          {
            bind ?elt = (Item)?var1.nextElement();
            if (?elt.value < ?min)
            {
               ?min = ?elt.value;
            }
          }
          assert Voucher
          {
            value = .1 * ?min;
            customer = ?customer1;
          }
        }
```

Rule 5: twoDifferentItems

English

If the shopping cart contains 2 items related but having different type then give to the customer a voucher of $1.

IRL

```
        rule twoDifferentItems
        when
```

```
{
  ?customer1: Customer();
  ?shoppingCart1: ShoppingCart(customer == ?customer1);
  ?item1: Item(shoppingCart == ?shoppingCart1 );
  Item(shoppingCart == ?shoppingCart1 ; type!=?item1.type)
        in getRelatedItems();
}
then
{
 assert Voucher
  {
    value = 1;
    customer = ?customer1;
  }
}
}
```

Rule 6: removeVoucher

English

It the shopping cart discount value is greater than 10% and a voucher has a value greater than $4 then remove the voucher.

IRL

```
rule removeVoucher {
when
{
  ?customer1: Customer();
  ?shoppingCart1: ShoppingCart(customer == ?customer1 ;
                              discountValue>10);
  ?voucher: Voucher(customer == ? customer1 ; value >4);
}
then
{
 retract voucher;
}
}
```