# Oracle Business Rules: Technical Overview

*An Oracle White Paper*
*February 2007*

ORACLE

## INTRODUCTION

Changing markets, increasing competitive pressures and evolving customer needs are placing greater pressure on businesses to adjust their policies and decisions at a faster pace. Further, there is an increasing desire among business users to get into the driver seat for defining how the business is run. Moreover, regulatory constraints are increasingly demanding that businesses have transparency and consistency in their decision making, and that they are able to certify compliance. Business Rules technology has emerged as the solution for addressing these requirements.

Oracle Business Rules is a leading edge Business Rules product. It is part of the Fusion Middleware stack. It is also a core component for present and future Oracle Fusion Middleware and Fusion Applications products. Along with Oracle BPEL PM, BAM and other products, Oracle Business Rules enables its customers to become more agile.

## THE BUSINESS RULES TECHNOLOGY

Business Rules technology enables automation of business rules; it also enables extraction of business rules from procedural logic such as Java code or BPEL processes. Typically, the pillars of Business Rules technology are declarative specification and inferencing.

### Declarative Specification

Declarative means declaring the intent without encumbering it with control flow.

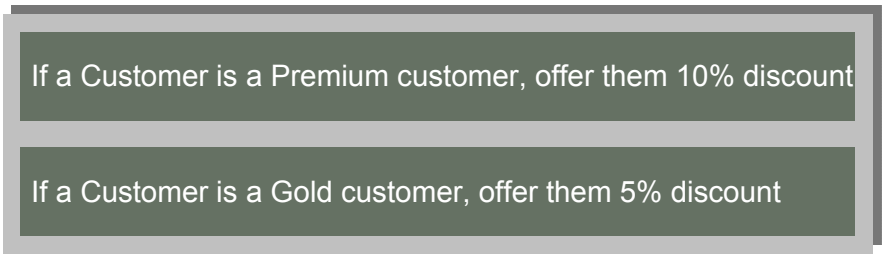


**Figure 1: Example of Declarative Specification**

The example in Figure 1 shows how a premium customer and a gold customer are defined. There is no control flow specification on the order of the rules execution. This is where a Business Rules engine comes in – it automatically figures out the correct and high performance order of execution.

#### Benefits of Declarative Specification

When business rules are implemented in procedural logic such as COBOL or Java code, it becomes a difficult exercise to understand the business rules, which requires navigating complex control flows and invocation sequences. Also, the impact of changing a business rule is difficult to understand because the surrounding control flow may or may not lead to the desired results; e.g. the control flow may not be rechecking for other rules that become applicable with this change. Finally, business users have no visibility into the business rules.

Use of declarative specification addresses the above issues and:

1. Makes business rules transparent as they are cleanly expressed in one place. This not only addresses the business need to have visibility, but also makes life easier for IT.
2. Makes changing of business rules easier. Not only is it obvious what needs to be changed due to the enhanced transparency, but also makes changes predictable, as the business rules engine will automatically do the right flow.
3. The expression of the business rules matches the business problem being solved.

### Inferencing

Some times when a rule executes because of its execution other rules may become applicable. For example in Figure 2, when the third rule determines that a customer is a premium customer, all rules for premium customers should be evaluated. Inferencing is the ability of the rule engine to automatically infer such dependencies and evaluate the needed rules.

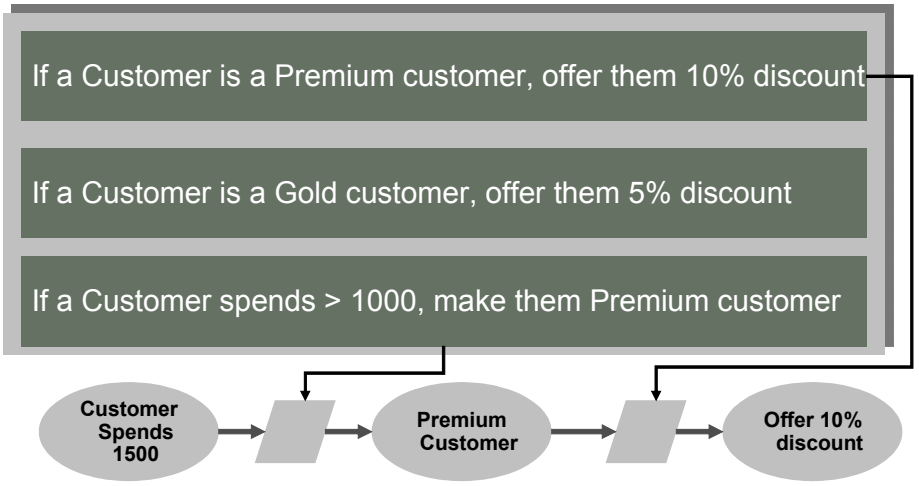


**Figure 2: Example of Inferencing**

In the example shown in Figure 2, when a customer spends 1500, the third rule fires designating the customer as a Premium customer. An inference capable rule engine then automatically evaluates the first rule, giving the customer 10% discount.

#### Benefits of Inferencing

Inferencing enables complex business rules to be declaratively specified in a modular fashion. For example, the example shown in Figure 2 separates the definition premium customers and their treatment enabling each to be changed independent of the other.

## HIGH VALUE USE CASES FOR BUSINESS RULES

While Business Rules technology may be applied to a broad spectrum of problems, following are some guidelines to help selection of problems for which Business Rules technology yield highest value:

- *Volatility* – Rules that change often are a good candidate for implementing with Business Rules technology. The ease of change provided by Business Rules technology will lead to significant cost savings over time.
- *Cost of Implementation Lags* - There may be some situations when the rules don't change that often, but when they do, the business cost of delay in implementation is too high. These high impact rules are good candidates for implementing with Business Rules technology.
- *Ownership* – Rules that business users want to own, author or edit are good candidates for implementing with Business Rules technology. Business users relate easily to the declarative and other metaphors supported by Business Rules technology.
- *Compliance* – Rules that have compliance requirements are good candidates for implementing with Business Rules technology.
- *Complexity* – Some problems naturally lend themselves to business rules as implementing them in traditional procedural logic would be too complex. Characteristics of such problems include a large number of rules and complex dependencies between them. A very good example of this class of problems is product configuration.

## ORACLE BUSINESS RULES

Oracle Business Rules provides high performance and easy to use implementation of Business Rules technology. It provides easy to use authoring environment as well as a very high performance inference capable rules engine. Oracle Business Rules is part of the fusion middleware stack and will be a core component of many Oracle products including both middleware and applications.

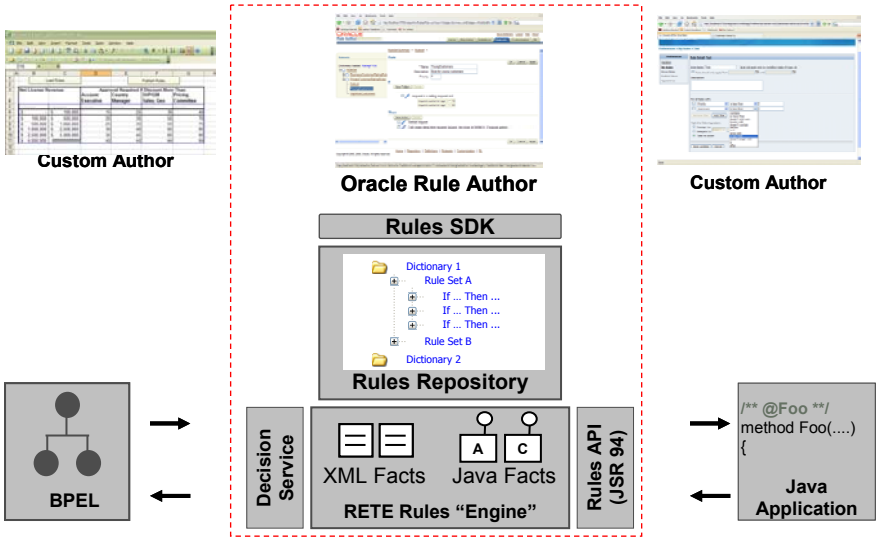


**Figure 3: Oracle Business Rules overview**

The components of the Oracle Business Rules product include:

- *Rule Author* – Rule Author is a web based graphical authoring environment that enables creation of business rules using a click and select user experience.

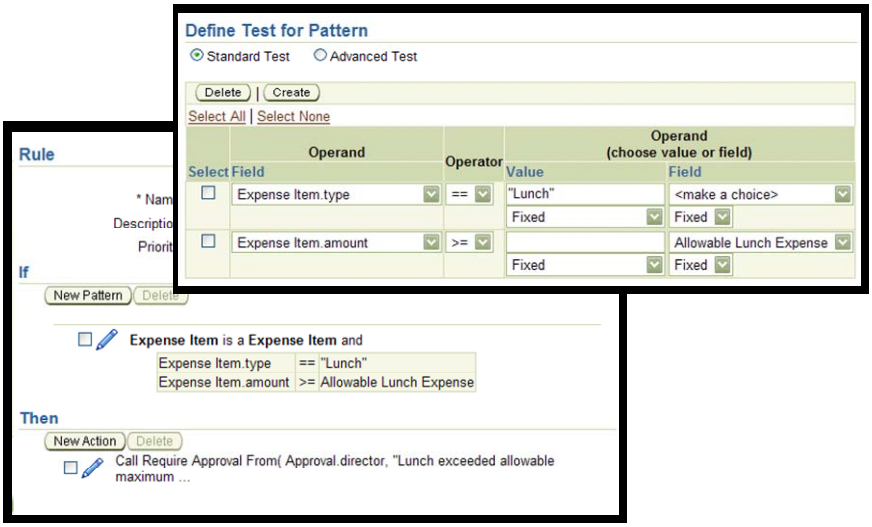


**Figure 4: Oracle Rule Author provides simple click and select authoring**

- *Rules Engine* – Rules engine is an inference capable Rete (http://en.wikipedia.org/wiki/Rete_algorithm) rules engine based on the Jess product (http://www.jessrules.com/jess/index.shtml) from Sandia labs.
- *Rules Repository* – The Rules repository enables rules to be organized in rulesets and rulesets in dictionaries. It also supports versioning of dictionaries. In current releases, the repository may be file based or webDAV based. In addition, APIs are available to plug in any desired repository. From release 11 onwards, Oracle MDS will be used as the repository consistent with all Oracle middleware and applications products.
- *Rules SDK* – The Rules SDK provides complete access to the Rule Repository and is designed to facilitate creation of rule authoring environments. The Rule Author itself uses the SDK. It is also used by the Workflow Application in Oracle BPEL PM to provide workflow specific authoring of rules. It may be used to build any custom authoring environments.
- *Rules Language (RL)* – Oracle replaced Jess' lisp like language with a Java based language called Rules Language, or commonly just RL. Typically, RL would be used directly by users only for writing supporting functions. The Rule Author abstracts away rest of RL from users.
- *Decision Service* – Decision Service enables Oracle business rules to be invoked as a web service from BPEL or other Web Service clients. The Decision Service tooling in Oracle BPEL PM provides seamless integration between BPEL and Business Rules.
- *Rules API (JSR94)* – Rules APIs including JSR 94 APIs are available to invoke Oracle Business Rules from any Java program.

## Business User Enablement with Oracle Business Rules

Oracle Business Rules provides multiple options for involving business users depending on the sophistication of the business user as well as the level of control desired. These options include:

- *Unrestricted use of rule editing in Rule Author* – Since the rule author is easy to use and provides a click and select experience, somewhat sophisticated business users desiring complete control over business rules can use it.
- *Tweaking knobs with variables* – Developers may use variables to expose simple knobs such as thresholds that the business users can control.

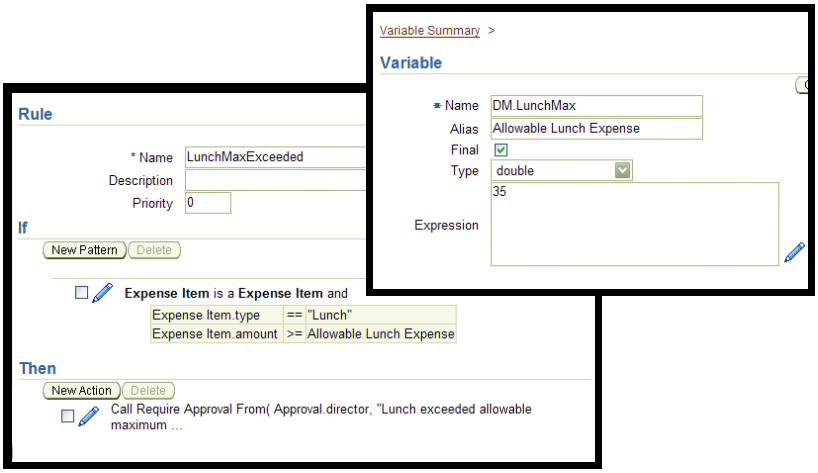


**Figure 5: Exposing knobs for business users to turn**

- *Customization* – Customization is a feature in the Rule Author that allows only those facets of a business rule to be changed that have a constraint associated with them and only within the limits of the constraint. This feature can be used to enable business users control over only certain facets and at the same time prevent them from illegal changes.

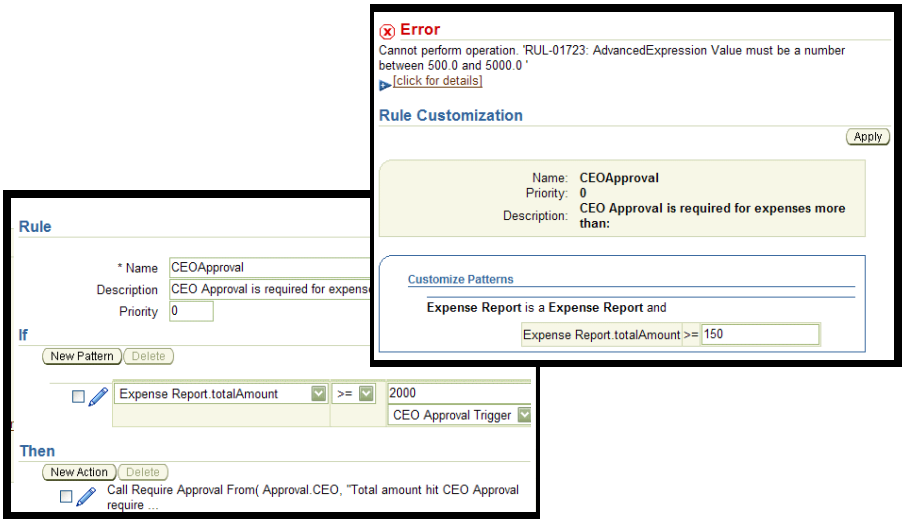


**Figure 6: Customization within constraints**

- *Custom Authoring Environments* – Some times a custom authoring environment is called for truly empowering the business users. This enables business users to control the rules from within the context of their business applications. In addition, it enables the metaphor to be completely tailored to the problem. The authoring is also simplified by limiting it to the problem on hand.

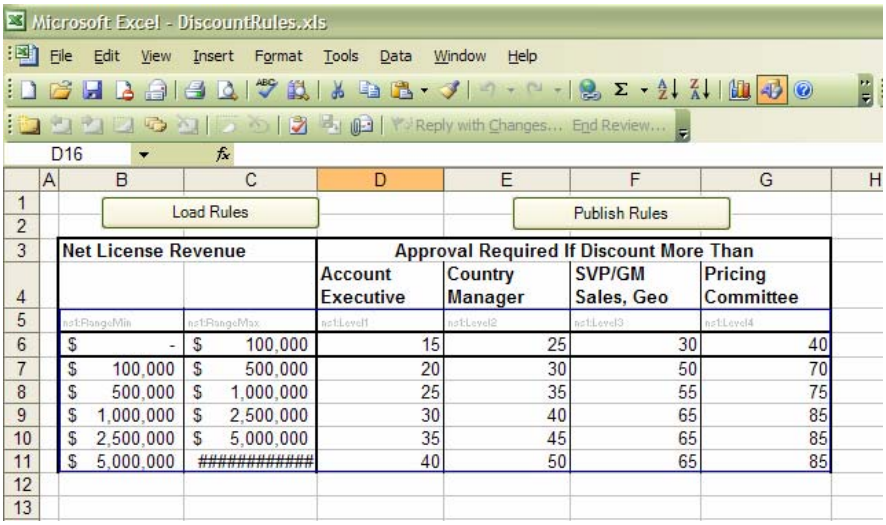


**Figure 7: Example of a custom authoring environment (Excel spreadsheet) built using SDK**

In addition to the above, RL Functions may be used to abstract complexity from business users.

In release 11, Oracle Business rules will further improve business user empowerment by introducing support for Decision Tables as well as simplifying the overall user experience.

## Run Time Architecture

Oracle Business Rules may be run in two different modes:

- *Embedded Jar file* – Typically, Java applications invoking the rules engine will embed the rules engine as a jar file (rl.jar). In this case, the rules engine scales and distributes just as rest of the application. Rule sessions may be pooled and cached for optimizing performance. These are discussed in the performance best practices section.
- *Decision Service* – Rules engine may be used as a Decision Service by BPEL and other clients desiring a service interface. The Decision Service is an application server servlet application; it scales and distributes accordingly. In this case, the Decision Service handles the session pooling and caching.

## RULE-ENABLING APPLICATIONS AND PROCESSES

As discussed above Oracle Business Rules may be used to extract business rules from applications or business processes using either the Java API interfaces or the Decision Service interface. Business Processes will use the Decision Service interface; Java applications including JSP and JSF pages may prefer to use the Java API interface. The steps for rule-enabling applications and processes include:

### Using the Decision Service Interface

1. Identify the XML Schema for the data you want to pass as input or get back as output. Import them as XML Facts
2. Follow steps for developing business rules
3. Create a Decision Service using Decision Service wizard in Oracle BPEL PM
4. Add a Decide activity in your BPEL process. Map process variables to input facts and output facts back to process variables. (For non BPEL, Decision Service clients, invoke as Web Service)

### Using the Java API

1. Identify the Java objects you want to pass as input or get back as output. Import them as Java Facts
2. Follow steps for developing business rules
3. Modify your Java application logic to invoke the Rule Engine.

### Modifying Java application logic to invoke the Rule Engine

The business analyst should determine what functionality of the application should be rule-driven. See discussion above on identifying high value use cases.

If an existing application is being rule-enabled, the programmer will need to replace procedural functionality implementing the rules with new rule-driven functionality. Note that the procedural code may need to be "mined" to extract the existing hard-coded rules.

The Java application code needs to call Rule APIs to accomplish the following:

1. Load the appropriate rule repository and dictionary
2. Create a rule session
3. Execute RL definitions for rules, functions, and variables
4. Assert some business objects as initial facts
5. Run an inference cycle
6. Retrieve results

For more details, please see http://www.oracle.com/technology/products/ias/business_rules/files/how-to-rules-java.zip.

### Developing Business Rules

1. Import the XML Schemas and Java classes you need to reason on as XML Facts and Java Facts
2. Provide a business vocabulary by editing the fact definitions to:
    a. Provide business friendly aliases (the tool provides default aliases)
    b. Hide elements to focus rule authors' experience
3. Import any needed supporting Java packages such as Date or BigInteger
4. Define *Variables*.
5. Define *Constraints*.
6. Define *RL Functions* that will simplify rule authoring. Please see Using RL Functions below. In particular, develop functions that can be used to test the rules.
7. Define the *Rulesets* and *Rules*
8. Test

Please see the section Oracle Business Rule Technical Details below for details on the steps above.

## ORACLE BUSINESS RULE TECHNICAL DETAILS

The section Oracle Business Rules above provides an overview of the Oracle Business Rules product. The discussion in this section complements it by adding technical details.

### Rule Author

As discussed in the section Oracle Business Rules above, the Rule Author is a browser-based tool for creating and customizing rules, and for defining the facts, functions, and variables on which the rules operate.

A group of related definitions and rules are stored together in a dictionary inside the repository.

**Business Rules are expressed using defined facts, constraints, variables, and functions.**

#### Facts

Fact definitions come from three places:

1. selected properties and methods of a Java class,
2. selected attributes and sub-elements of an XML element or complexType,
3. an RL class.

Fact definitions imported from Java and XML are typically used to create rules that examine the business objects of the rule-enabled application, or to return results to the application. Fact definitions based on an RL class are typically used to create intermediate facts that are used to trigger other rules in an inference chain.

**Facts cause rules to fire, and firing rules can create more facts, which in turn can fire more rules. This process is called an inference cycle.**

#### Constraints

Constraint definitions are used to mark portions of rules as customizable (Please see *Customization* above for discussion of the concept.). For example, the discount to offer to a GOLD customer could be constrained to be within the range 5..25 percent.

#### Variables

**Variables and functions make rule sets modular and maintainable.**

Variables share information among several rules and functions. For example, if the 10% GOLD discount is used in several rules, a variable "GOLD discount" should be used instead of the hard coded 10% so that if it is customized, it can be done in a single place.

Variables are also useful for exposing controls such as thresholds to business users. In addition, they may be used for holding the results of rules evaluation.

#### Functions

Function definitions can be used to share the same or similar expression among several rules, and to return results to the application . Functions can have parameters, making them more generally useful than variables.

```
function discount(String type) {
       if (type == "GOLD") { return 10; }
       else if (type == "SILVER") { return 5; }
}
```

Functions are defined using the Oracle Rule Language (RL) syntax, which is described later in this paper.

#### Rulesets and Rules

A ruleset is a collection of rules that are all meant to run at the same time. A rule consists of a condition, or *if* part, and a list of actions, or *then* part.

**A rule has the form:**

**if *conditions***
**then *actions***

***if*** - The condition part of a rule activates the rule whenever there is a combination of facts that makes the condition part true. In some respects, the rule condition is like a query over the facts in the rule engine, and for every row returned from the query, the rule is activated.

***then*** - The action part of a rule is executed when the rule fires. In order for the rule to fire, it must be activated and the RL function *run* must be called. Rules fire sequentially, not in parallel. Common rule actions include invoking functions or methods, and asserting (creating), retracting (deleting), or modifying facts. Note that rule actions often change the set of rule activations and thus change the next rule to fire.

### Rule Repository

**The Rule Repository stores Rule Author information and generates RL for execution by the Rule Engine.**

The Rule Repository is the persistence mechanism underlying the Rule Author. It is logically organized as a collection of dictionaries. The repository provides two APIs for programmatic access – the Rule SDK API and the RL Generation API.

#### Dictionary

A dictionary is a set of XML files that stores the definitions, rule sets, and customizations that are created using the Rule Author. A dictionary typically stores all the rules and definitions for a rule-enabled application. A Rule Repository can have several dictionaries, but they would typically be used for different applications or for different versions of an application.

#### Rule SDK API

The Rule SDK API is the interface between the Rule Author and the Rule Repository. This API supports both thick and thin GUI clients in the following ways:

- dictionary objects (rules, fact definitions, *etc.*) can be retrieved as nested collections of name-value pairs for easy binding to GUI controls such as navigators, property sheets, and drop-down lists,
- dictionary objects can be created with appropriate default values, and

- dictionary objects can be modified with extensive integrity checking.

Because the Rule SDK API is implemented by the Rule Repository, developers can use this API for developing custom rule authoring tools that conform to the look and feel of their rule-enabled applications.

#### RL Generation API

The RL Generation API is the interface between the Rule Engine and the Rule Repository. This API generates executable Rule Language code for the Rule Engine from the definitions and rules in a dictionary.

## Rule Engine

**The Rule Engine interprets RL. RL rules can reason on any Java or XML object. Any Java program can use the Rule Engine.**

As discussed in the section Run Time Architecture above, the Rule Engine may be used in one of two modes: Decision Service and Java Library. Some of the following discussion including the Rule Session API is not interesting to users of the Decision Service mode, as the Decision Service application abstracts them.

The Rule Engine is a Java library that a rule-enabled application invokes via the Rule Session API. The application passes facts, represented as Java or XML objects, and rules, represented in Rule Language, to the Rule Engine. The Rule Engine uses the industry-standard Rete algorithm to efficiently fire rules that match the facts.

#### Rule Session API

The Rule Session API is the interface between the application and the Rule Engine. The two most useful methods of the Rule Session API are executeRuleset and callFunction.

##### *executeRuleset*

The *executeRuleset* method executes an RL program, passed in as a String. Such a program typically contains rule, function, and variable definitions. The RL program typically comes from a dictionary in the Rule Repository.

##### *callFunction*

The *callFunction* method executes an RL function. Variants of this method may pass Java objects to RL functions, and return Java objects from RL functions. The called function may be a built-in function such as *assert*, which creates a new fact, or *run*, which enables activated rules to fire. Alternatively, the function may have been defined in a prior call to *executeRuleset*.

The *callFunction* method is typically used to

- pass business objects from the application to the Rule Engine, where they are asserted as facts,
- initiate an inference cycle using the *run* function, and

- retrieve results.

### Rule Language

The Rule Engine directly executes Oracle Rule Language (RL) RL is interpreted rather than compiled so that rules may be changed without rebuilding, redeploying, or even restarting applications. RL features Java-like syntax and type checking. RL programs can assert any Java object as a fact, and rules can reference any object property and invoke any method in its condition and actions.

Programmers can use RL as a full-featured rules programming language, or business analysts can use the Rule Author and RL can be generated from the Rule Repository behind the scenes. The rule engine has a command line interface for interactive RL development and debugging.

**Here is an example of a rule in RL:**

```
rule approvePO {
  if (fact PurchaseOrder po
     && po.amount > 1000
     && po.approvalLevel ==
null)
  {
    po.approvalLevel = "VP";
    assert(po);
  }
}
```

***Rete*** **is the Latin word for** ***network.***

### Rete

The Rule Engine uses the industry-standard Rete algorithm that was first developed by artificial intelligence researchers in the late 1970s and is at the core of rule engines from major rule vendors.

The Rete algorithm combines rule conditions for all rules into a single network of nodes. There is an input node for each fact definition. There is an output node for each rule. In between input and output nodes are test nodes and join nodes. A test occurs when a rule condition has a boolean expression. A join occurs when a rule condition ANDs two facts. Fact references flow from input to output nodes. A rule is activated when its output node contains fact references. Fact references are cached throughout the network to speed up recomputing activated rules. When a fact is added, removed, or changed, a *fact change* reference is pushed through the Rete network that updates the caches and the rule activations with only an incremental amount of work.

The Rete algorithm provides the following benefits:

- independence from rule order – rules can be added and removed without impacting other rules,
- optimization across multiple rules – rules with common conditions share nodes in the Rete network, and
- high performance inference cycles – each rule firing typically changes just a few facts and the cost of updating the Rete network is proportional to the number of changed facts, not the total number of facts or rules.

## DEVELOPMENT BEST PRACTICES

### Accessing Data

Since Oracle Business Rules has access to Java methods, it is possible to use Java (in conjunction with technologies such as Toplink) to fetch data from within Rules. However, this is not the recommended best practice. It is instead recommended that you fetch data (in general IO and Database operations) outside of Rules, possibly using BPEL and its Adapters, and then invoke Rules with all the data it needs. By doing so, you can handle things such as connection errors and timeouts in BPEL using its error handling capabilities.

### Returning Results

See section 3.12 in *Oracle Business Rules User's Guide* (http://download-east.oracle.com/docs/cd/B31017_01/web.1013/b28965/guiadvanced.htm#CIHGEFCG).

### Rule Organization

The best practices for organizing rules and rule sets parallel the practices for organizing software into classes and packages. A rule set should contain rules whose evaluation is related to producing the result(s) of that rule set. That is, they are focused on accomplishing the same goal. For example, rulesets may be organized such that one ruleset evaluates loan applications, second evaluates credit reports, and third determines the interest rate and loan amount the applicant qualifies for.

The number of rules in a rule set should be driven by the application requirements and by requirements for re-use. A smaller rule set that performs a specific evaluation with a specific result will be more manageable and more likely to be re-used. Organizing into rule sets with focused purposes also enables an application component to only load the rule sets it requires and thus optimize the resources it uses for rule evaluation. At runtime, the rules are all compiled into a single Rete network within the rule engine and whether the rules are in one or many rule sets is not a factor in resource usage.

#### Rule Set Chaining

When there are rules that need to be shared across rulesets, they should be organized in their own ruleset. For example, a Credit Processing scenario may have a rule set by each lender and a common rule set for rules applicable across all lenders. Such rulesets may be chained by pushing them on the rule stack either conditionally or unconditionally
some rules are effective only on

### Using RL Functions

Functions are a useful tool to ab

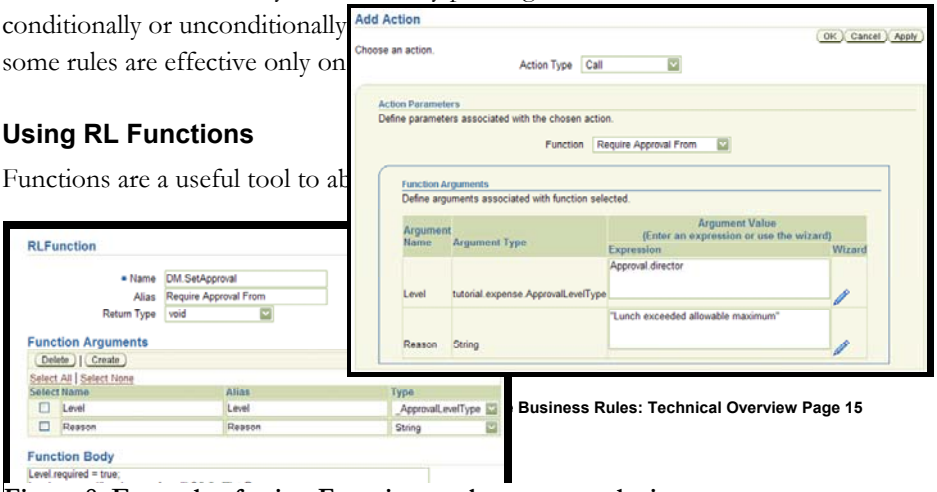




**Figure 8: Example of using Function to abstract complexity**

In the example shown in Figure 8, as part of an approval rule action, it is needed to set two values: Level.required and Level.reason; moreover, since an approval level may be required by multiple rules, the Level.reason needs to aggregate all reasons. Requiring every rule to deal with this requirement would make rule authoring difficult and error prone. Instead, encapsulating this behavior in a function enables rule authors to simply specify the Level and the Reason, as shown in the screenshot.

In the current release of Rule Author, RL function bodies must be entered using a Java-like syntax (In Release 11, easier authoring of RL Functions similar to Rule actions will be provided). Therefore, it is not advisable to use RL Functions (instead of Rulesets) to model rules that business users may change. In general, it is best to use Functions as helpers to facilitate rule editing (as Rulesets) and not for modeling rules (instead of rulesets). It may be tempting to use RL Functions for computations, especially those that are reused many times; however, if such computations are expected to change often, they should be modeled as Rulesets Rules; reuse is possible through Inferencing or ruleset chaining.

It should be noted that Java methods might be used in place of Functions. However, RL Functions have the benefit of being contained in the rule repository and being interpreted – that is they do not require deployment of Java classes etc. In addition, it is easier in RL Functions to access the rule artifacts such as variables.

### Using Inferencing

As discussed in section Inferencing above, Inferencing is a powerful tool for modular development of complex rules.

A natural scenario for inferencing is when one or more rules calculate an intermediate result in the rule action. This intermediate result would be asserted as a fact (or is a modification of an existing fact). This intermediate result is used in the rule condition of other rules.

Inferencing may also be used to split complex rules into two or more simpler rules. Rules conditions that are very complex increase the maintenance cost and reduce possible reuse. Splitting a complex rule into multiple simpler rules may improve both. Some of these simpler rules would need to assert intermediate facts (or modify existing facts) as in the previous case.

In current release, you have to be careful that inferencing does not cause a rule to fire itself, resulting in an infinite loop. In the example in Figure 10, rule 1 has an additional test that income level is not high to avoid loop.

### Enabling Business Users

Please see Business User Enablement with Oracle Business Rules above for a discussion on the various options available and their appropriateness. This section only adds some developer specific considerations to that discussion.

#### Developer Considerations using Customization

Use list-of-value constraints to make it easier for the user to choose an appropriate value from a small set. Use range constraints with numeric values to validate changes.

Be careful not to require that two or more customizable values be the same.

```
Rule1: IF p.income > 100000 && p.isHomeOwner THEN
p.offerCeditCardColor = "platinum"

Rule2: IF p.income > 100000 && p.isRenter THEN
p.offerCreditCardColor = "gold"
```

**Figure 9: Do not require two customizable values to be the same**

The example shown in Figure 9 is a problem. It can be rewritten as shown in Figure 10 using inferencing.

```
Rule0: IF p.income > 100000 && p.incomeLevel != "high" THEN
p.incomeLevel = "high"; assert(p);

Rule1: IF p.incomeLevel == "high" && p.isHomeOwner THEN
p.offerCeditCardColor = "platinum"

Rule2: IF p.incomeLevel == "high" && p.isRenter THEN
p.offerCreditCardColor = "gold"
```

**Figure 10: Rewriting Figure 9 using inference**

#### Developer Considerations using Variables

Variables can be used for many purposes. There are two kinds of variables: final and non-final. Final variables are initialized once (using their initialization expression) and may not be changed. Final variables are an alternative to customization to make rules easier to change. In the preceding example, one could define a variable to avoid repeating the literal *100000*.

```
final int HighIncomeThreshold = 100000
```

In current releases, Variable initialization expressions are not customizable so no constraint can be associated with the value.

Non-final variables can be assigned to by rule actions (but cannot be used in rule conditions), and are thus a way of returning results from rule execution.

### Using right Fact Types

The context in which the rules engine is being used often dictates the fact type to use:

- *Decision Service* - Using rules with BPEL or other Web Service clients as Decision Service dictates that XML facts are used. If the business rules need to work with existing Java fact types, please see section *18.5.3* of the *BPEL Developer's guide* (http://download-

east.oracle.com/docs/cd/B31017_01/integrate.1013/b28981/decision.htm#sthref3261).

- *Existing XML Schemas* - Data model based on an XML Schema typically dictates the use of XML fact types. However, if the application is already using JAXB, JAXB classes generated externally to rules may also be imported as regular Java facts.
- When the data model is based on a collection of Java classes, Java facts will be used.
- RL fact types are most commonly used for facts representing intermediate results.

#### Working with JAXB Numbers

JAXB maps XML *integer* type to *java.math.BigInteger* and the *decimal* type to *java.math.BigDecimal*. If a rule uses the properties of such types, the two Java classes have to be imported explicitly.

It is suggested to use XML *int* and *double* types instead in the XML Schema, in order to avoid the step to import additional Java classes. If schema change is not possible, JAXB customization can be used to map XML *integer* or *decimal* types to primitive types. Details about JAXB customization can be found in JAXB Specification (http://java.sun.com/xml/downloads/jaxb.html) section 6.

## PERFORMANCE CONSIDERATIONS

In most cases, writing of Rules should not require a focus on performance. However, as in any technology, there are tips and tricks that can be used to maximize performance when needed. Most of the considerations are around the initial set up of the data model.

### Use Java Beans

The rule engine is most efficient when the facts it is reasoning on are Java Beans (or RL classes) and the associated tests involve bean properties. The beans should expose get and set methods (if set is allowed) for each bean property. If application data is not directly available in Java Beans, it may be desirable to flatten the data to a collection of Java Beans that will be asserted as facts (and used in the rules). **Note that XML facts follow the Java Bean pattern and are ok regards this consideration**.

### Assert Child Facts instead of Multiple Dereferences

Expressions like *Account.Contact.Address* involve more than one object dereference. In a rule condition, this is not as efficient as expressions with single dereferences. It is a best practice to flatten fact types as much as possible. If the fact type has a hierarchical structure, consider using assertXPath or other means to assert object hierarchy; that is for the preceding example, assert both *Account* and *Contact* as Fact Types.

### Avoid Side Affects in Rule Conditions

Methods or functions that have side affects such as changing a value or state should not be used in a rule condition. Due to the optimizations performed when the rule engine builds the Rete network and the Rete network operations that are performed as facts are asserted, modified (and re-asserted), or retracted, the tests in a rule condition may be evaluated a greater or lesser number of times than would occur in a procedural program. Thus, if a method or function has side affects, those side affects may be performed an unexpected number of times.

### Avoid Expensive Operations in Rule Conditions

Expensive operations should be avoided in rule conditions. Expensive operations would include any operation that involves I/O (disk or network) or even intensive computations. In general, it is a best practice to avoid I/O or DBMS access from the rules engine directly. These operations should be done external to the rules engine. For other expensive operations or calculations, a best practice is to perform the computations and assert the results as a Java or RL fact. These facts are used in the rule conditions instead of the expensive operations.

### Pattern Ordering

Reordering rule patterns can improve the performance of rule evaluation in time, memory use, or both. There are two main guidelines for ordering fact clauses (patterns) within a rule condition.

- If a fact is not expected to change (or will not change frequently) during rule evaluation, place its fact clause before fact clauses that change more frequently. That is, order the fact clauses by expected rate of change from least to greatest. Ordering fact clauses in this way can improve the performance (time) of rule evaluation.
- If a fact clause (including any tests that involve only that fact) is expected to match fewer facts than other fact clauses in the rule condition, place that fact clause before the others. That is, order the fact clauses from most restrictive (matches fewest facts) to least restrictive. This can reduce the amount of memory used during rule evaluation. It may also improve the performance.

Sometimes these two guidelines conflict and it may require some experimentation to arrive at the best ordering.

### Ordering of Tests in Rule Conditions

Similar to the recommendations for fact clauses, the tests in a rule condition should be ordered such that a test that will be more restrictive is placed before a test that is less restrictive. This will reduce the amount of computation required for facts that do not satisfy the rule condition. If the degree of restrictiveness is not known or estimated to be equal for a collection of tests, then the simpler tests should be placed before more expensive tests.

### AssertXPath and Supports XPath

Most of the work done by the rules engine is done during assert, retract, or modify operations. In particular, the assertXPath method, though very convenient, may have a performance overhead. The power of this method is not only that it asserts the whole hierarchy in one call but also asserts some XLink facts for children facts to link back to parent facts. However, if these features are not needed and it is simply desired to assert a couple of levels as facts, it is better to turn off the "Supports XPath" for the relevant fact types and then use a function to do custom asserts.

```
function assertAllObjectsFromList(java.util.List objList)
{
 java.util.Iterator iter = objList.iterator();
 while (iter.hasNext())
 {
   assert(iter.next());
 }
```

```
}

function assertExpenseReport (demo.ExpenseReport expenseReport)
{
 assert(expenseReport);
 assertAllObjectsFromList(expenseReport.getExpenseLineItem());
}
```

**Figure 11: Instead of using assertXPath this example uses a function to assert ExpenseReport and ExpenseLineItems**

In 10.1.3.3, we will introduce "Enable improved assertXPath support for performance" check box in the Dictionary Properties page in Rule Author. Checking this switch will improve the performance of assertXPath. Taking advantage of this will require that

- assertXPath is only invoked with an XPath expression of "//*". Any other XPath expression will result in an RLIllegalArgumentException.
- XLink facts should not be used in rule conditions as the XLink facts will not be asserted.

### Separating Fact Types into Read-Only and Modifiable

Under the following circumstances, there may be a performance benefit in splitting a fact type into read-only and modifiable fact types:

- The fact type contains properties that are tested in rule conditions but never modified (read-only).
- The fact type contains some properties that are modified.
- The read-only properties are tested in a significant portion of the rules.
- The fact is re-asserted when a property is modified (for inferencing).

Splitting the fact type into a read-only and modifiable fact types will reduce the work performed when the fact is modified. **Note** that there is no property or flag to indicate a Fact type as read-only; this guidance only relates to the actual usage.

### Rule Sessions Pooling

A typical rule session with a hundred rules takes a couple of seconds to initialize (the rules must be compiled into a rete network), and may take just a fraction of a second to execute. Therefore, an application that needs to run the same rules against different facts should reuse rule sessions several times to amortize the initialization cost. Note that the decision service automatically pools rule sessions. It is the responsibility of the ruleset writer to ensure that each execution of rules leaves the rule session reusable (or throws an Exception). The RL builtin function "reset()" will normally prepare the rule session for reuse by retracting all facts in working memory and re-running the initialization expressions of non-final global variables. If there is additional reset work to do, a user can "register" an RL

function or method to be called at reset time by defining an initialized non-final variable as follows:

```
int resetHookVar = resetHookFcn()
```

The above will cause the RL function "resetHookFcn" to be called every time the rule session is reused. Such a hook could be used to reassert any required initial facts, or otherwise prepare for serial reuse.

Another trick that can be employed to avoid the cost of startup is to serialize the session after first load and de-serialize it for subsequent sessions. This would also require building in check to see when this cached session needs to be invalidated (reloaded).

```
RuleSession ruleSession = null;
byte[] ba = null;
ruleSession = new RuleSession();
ruleSession.executeRuleset(dmRules);
ruleSession.executeRuleset(mainRules);
ByteArrayOutputStream fos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(ruleSession);
ba = fos.toByteArray();
```

**Figure 12: Code fragment to serialize session after executing rulesets**

```
ObjectInputStream s = new ObjectInputStream(
                              new ByteArrayInputStream(ba));
ruleSession = (RuleSession)s.readObject();
```

**Figure 13: Code Fragment to initialize session from serialized cache**

## CONCLUSION

Oracle Business Rules delivers on the promise of agility. It enables its customer to change their business processes and other decisions rapidly, flexibly, and with confidence. Oracle Business Rules enables involvement of Business Users in the specification and maintenance of business rules. This document describes best practices and considerations for extracting the most value from Oracle Business Rules. Oracle Business Rules engine is a high performance engine and although this document describes many performance tips, in most cases users need not be concerned about performance implications.

**Oracle Business Rules: Technical Overview**
**February 2007**
**Author: Manoj Das**
**Contributing Authors: Gary Hallmark**

**Oracle Corporation**
**World Headquarters**
**500 Oracle Parkway**
**Redwood Shores, CA 94065**
**U.S.A.**

**Worldwide Inquiries:**
**Phone: +1.650.506.7000**
**Fax: +1.650.506.7200**
**oracle.com**