

Mapping rule-based decision services to rulesets

Choosing the best approach

Pierre Berlandier (pberland@us.ibm.com)

Senior Technical Staff Member

IBM

24 March 2010

This article addresses a recurring question for the solution architect using a Business Rule Management System (BRMS): how should I structure the rule repository and then carve out the rulesets from this repository in order to support the needed rule services? We explore different alternatives for mapping the rule repository and the rulesets to rule services, and provide the pros and cons of each approach. We then illustrate the use of one alternative for the design of a risk assessment decision service, typically occurring in financial services type of applications.

Introduction

Complex decision services implemented with business rules are usually composed of a set of elementary decision points orchestrated by a ruleflow. One such common example is the loan underwriting business process, which is commonly decomposed into a loan request *validation* phase, followed by the *scoring* of the applicants, the determination of *eligibility* and finally the *pricing* of the loan.

A rule service implementing a straight-through underwriting process would receive a loan request and respond with an *accept*, *reject* or *refer* status. If the request is accepted, details about the pricing and conditions of the loan are provided with the response. If the request is rejected or referred, the response provides the associated reasons.

Such a rule service can be efficiently implemented by a single ruleset, orchestrating the sequence of decision points using a simple rule flow. However, it often turns out that some combinations of decision points within the main process are providing a value of their own. For example, a service that implements the combination of scoring followed by eligibility could very well be shared between the underwriting process and a pre-qualification process, provided by the lender as a marketing tool.

We are then faced with two concomitant architectural decisions: how should we organize the different decision points into rule projects and rulesets and how should we implement their orchestration to provide the desired rule services?

The goal of this article is to review different approaches to the decomposition of rule services into rulesets (specifically in the context of IBM ILOG JRules V7), and present recommendations for each approach.

Possible approaches

To support the presentation of the different decomposition approaches, we'll use the example described in the [Introduction](#), assuming that:

- We have four elementary decision points implemented by business rules: validation, scoring, eligibility and pricing.
- We need to provide three different rule services:
 - A fully automated underwriting service, which chains validation, scoring, eligibility and pricing.
 - An eligibility service, which performs scoring followed by eligibility.
 - A pricing service, which performs only the pricing part.

The questions we'll address are:

- What are the possible approaches to organize the decision points into rule projects and rulesets, and to orchestrate their execution in order to support the needed rule services?
- What are the pros and cons of each approach?

The three approaches we'll discuss are:

1. A single ruleset, using an internal orchestration through a ruleflow.
2. Multiple rulesets, using some external orchestration mechanism.
3. Multiple rulesets, using an internal orchestration through a ruleflow.

Note that, unlike business process management (BPM) engines where a service naturally invokes another in order to complete its goal, invoking the execution of a ruleset from within another ruleset is not a recommended practice.

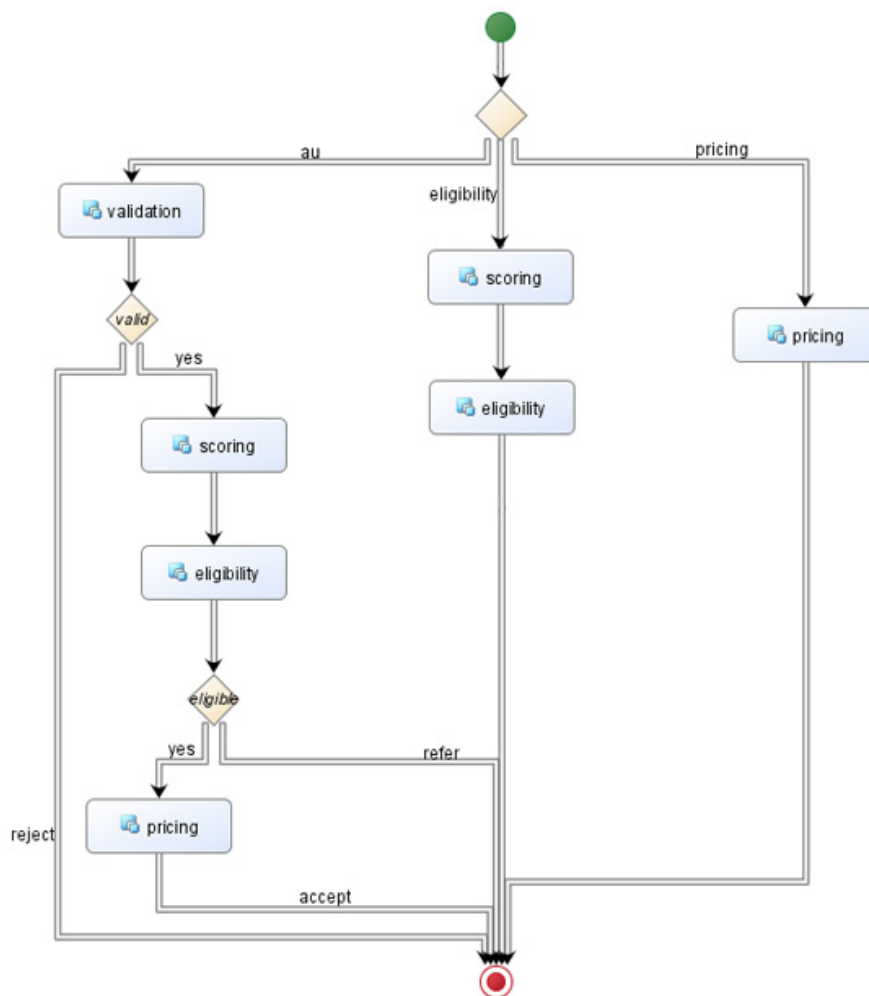
Single ruleset with internal orchestration

One approach to handle the problem is to deploy a single ruleset with one ruleflow that encapsulates the various service alternatives.

In this case, an extra parameter of the ruleset, `requested-service-name`, is used to route the request to the desired branch of the main ruleflow. All orchestration is managed internally through the ruleflow.

In our example, the resulting ruleflow is shown in [Figure 1](#). The first choice point of the ruleflow is used to check the value of the `requested-service-name` ruleset parameter, which can either be `au`, `eligibility` or `pricing`.

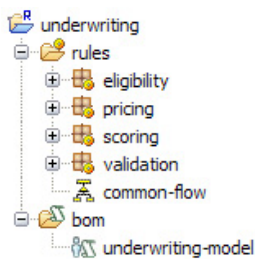
Figure 1. Using a common ruleflow for all services



An alternative to using an extra routing parameter is to address directly the desired task entry point in the ruleflow using the `setTaskName` method of the `ILrSessionRequest` class from the rule service invocation code. However, this approach is not recommended. One obvious drawback is that the name of the task gets embedded in the application code, thus adding unwanted maintenance complexity. Another is that the practice is not compatible with a simple invocation of out-of-the-box JRules Web services, such as HTDS.

From a rule project organization point of view, this approach is well supported by a single rule project, with the different decision points corresponding to different top-level rule packages, and a common flow, as shown in [Figure 2](#).

Figure 2. Sample rule project organization for a “common ruleflow for all services” solution



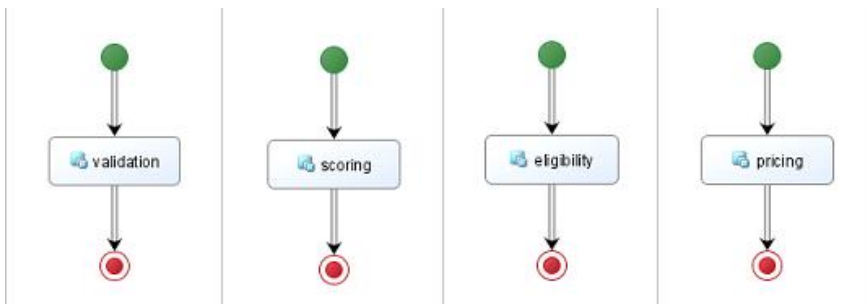
Multiple rulesets with external orchestration

This approach is the extreme opposite of the previous one. Here, we define one ruleset per individual decision point.

In this case, the orchestration of the different elementary operations is performed by the invoking application, which makes it the best fit for a BPM approach.

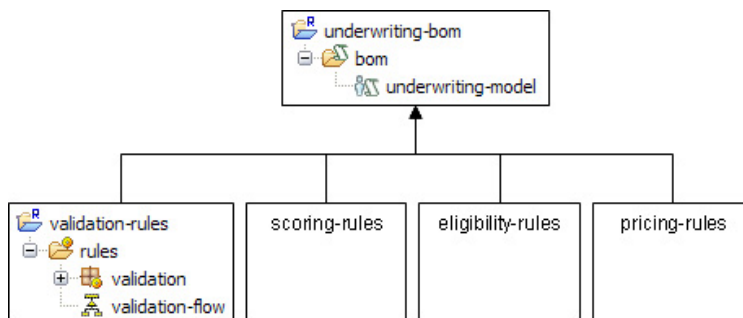
In our example, we would have four independent rulesets, whose simple ruleflows are shown in [Figure 3](#). It is the responsibility of the client application to invoke the different rulesets and manage the data flow between them.

Figure 3. Using one ruleflow per decision point



From a rule project organization perspective, this approach is best supported by having four independent rule projects. Each rule project contains the rule artifacts related to its decision point, as well as its own simple rule flow. All rule projects most likely reference a common project that encapsulates the application Business Object Model (BOM), as shown in [Figure 4](#).

Figure 4. Sample rule projects organization for a "one ruleflow per decision point" solution

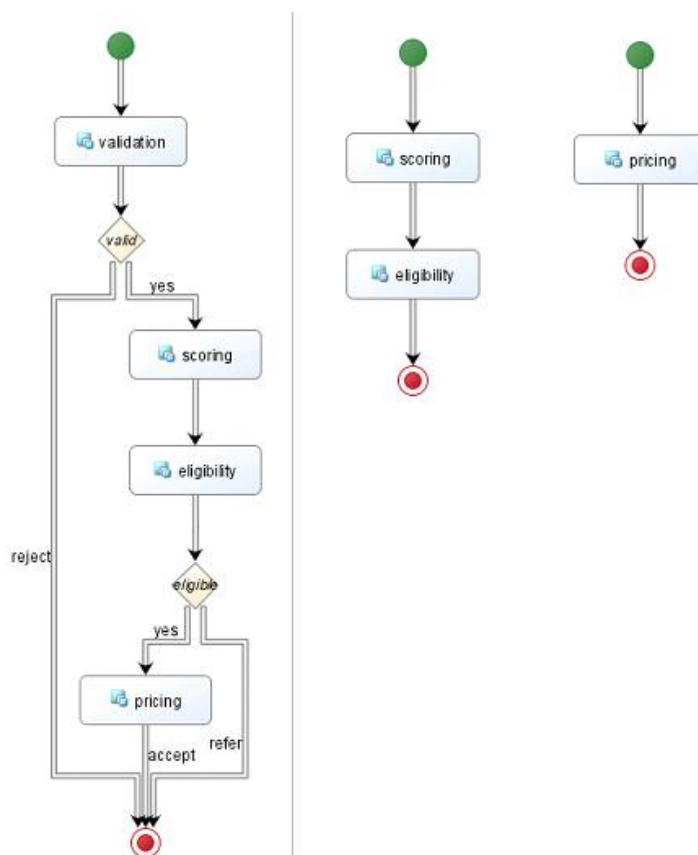


Multiple rulesets with internal orchestration

The third approach is a crossbreed between the first and the second approach. We have multiple rulesets deployed to the Rule Execution Server (RES), but each ruleset is performing a complete business service instead of an elementary operation.

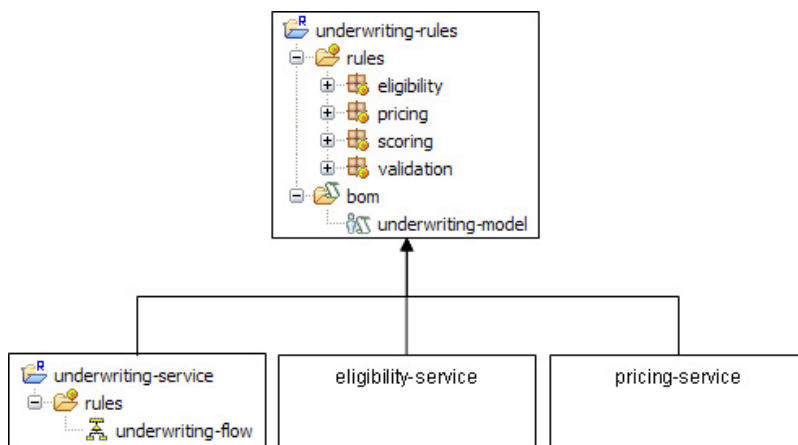
In our example, this would yield the three ruleflows shown in [Figure 5](#). The client application needs to worry only about addressing the proper ruleset, without the overhead of providing an extra routing parameter or performing any orchestration.

Figure 5. Using one ruleflow per rule service



The rule project organization for this approach also reflects its cross-breed nature: as with the first approach, we would use a common rule project which contains one top-level rule package per decision point (validation, scoring, and so on). Then we would have one rule project for each ruleset, which would reference the common rule project and define the ruleset's ruleflow, as shown in [Figure 6](#).

Figure 6. Sample rule projects organization for a "one ruleflow per rule service" solution



Selecting the right approach

Now that we have three possible approaches to choose from, let's look at the pros and cons of each, so that you can decide which one is best suited for a given situation. You should consider the following when making a decision:

- The relative homogeneity of the signatures of the different rule services.
- The number of rules involved in each decision point and the frequency of rule updates.
- The complexity of the decision point orchestration.

We'll discuss these considerations in the following sections, and present their respective impacts on the different approaches in a decision table.

Ruleset signatures

Every deployable ruleset is associated with a set of parameters (its signature) that represents the interface of the rule service implemented by the ruleset.

In JRules V7, a rule project is associated with a set of ruleset parameters, as defined in the Eclipse properties of the project. The signature of a ruleset extracted from a given rule project is determined by the transitive closure of the ruleset parameters of that rule project and all the rule projects it references.

Thus, one consideration when deciding on the rule service decomposition is whether or not the elementary decision points have the same signature. For example, if all the decision points work uniformly on a single `LoanApplication` in/out parameter, a single ruleset or ruleflow may work well.

Number of rules and rule update frequency

Using multiple instances of a rule task or a flow task within a ruleflow does not cause the duplication of the rules referenced by these tasks in the ruleset. For example, in the ruleflow illustrated in [Figure 1](#), the definition of the rules that participate in the scoring or pricing decision point appears only once in the definition of the ruleset, although the scoring and pricing flow tasks are used multiple times in the flow.

By contrast, generating multiple rulesets that use the same rule task or flow task duplicates the rules in the different rulesets.

Some of the implications of duplicating rule definitions in deployable artifacts are:

- From a runtime point of view, more CPU cycles are needed to load the rulesets when an execution unit is created and more memory is consumed by the executable units.
- From an operations point of view, when a rule that belongs to a decision point used in multiple rulesets is updated, all the rulesets that are using a copy of the rule need to be redeployed.

Orchestration

The orchestration of the different decision points is part of the business knowledge. Although usually not complex, it still belongs to business and, as much as possible, should not be buried in some Java™ servlet code, for example.

If the invocation of the rule services is orchestrated by a BPM engine, we are in an ideal situation, with the right knowledge expressed in the right place and available for update by the right people. Otherwise, we may want to keep the orchestration within the ruleflow.

Other considerations













Some other considerations when making your decision are:

- **Network latency:** With remote invocation of the rule services comes the issue of network latency. When performance is critical, it is best to limit the number of calls to the Rule Execution Server.
- **Execution units (XUs):** When using the single ruleset solution, each XU will consume significant resources, even though they may be used for small decision services such as pricing. When using one ruleset per decision point, more XUs are needed. However, they will consume fewer resources than when using the single ruleset solution. Also, the execution time of each ruleset will be shorter. The XUs will thus be released faster to the pool, improving the throughput of concurrent requests.

Decision table

The following table compares the three different approaches that we have described against the different criteria.

Table 1. Decision table

	Single ruleset Internal orchestration	Multiple rulesets External orchestration	Multiple rulesets Internal orchestration
Ruleset signature	 The ruleset signature must contain the union of the parameters needed by all the decision points. This may make some rule service calls awkward, using artificial values. This approach is thus not suited for services with different signatures.	 The ruleset signature reflects the set of parameters needed to execute the decision point.	 The ruleset signature reflects the set of parameters needed to execute the rule service.
Number of rules / update frequency	 Rules are not replicated across rulesets.	 Rules are not replicated across rulesets.	 Rules are replicated across rulesets. This approach is thus not suited for shared decision points with a large number of rules or in which rules are changing often.
Orchestration	 Orchestration is performed by the ruleflow.	 Orchestration is delegated to the calling application. This approach is thus not suited for complex or dynamic processes, unless supported by a BPM engine.	 Orchestration is performed by the ruleflow.
Network latency	 Only one call to the RES is needed to execute the desired service.	 Multiple calls to the RES are needed to execute the desired service. The approach will under-perform if the RES is a remote application and network latency is a concern.	 Only one call to the RES is needed to execute the desired service.

Special use case: Rule services with incremental need for data

A special use case where the single ruleset with internal orchestration is the best architectural choice is when the rule service, in order to complete the decision, needs additional data that must be fetched, just-in-time, by the invoking application. The execution of the decision service thus needs to be interrupted, the necessary data fetched by the application, and the decision service invoked again, starting where it left off.

A typical example of this is for the scoring activity from our example, also known as *risk assessment* decision services. Risk assessment is an essential step of loan or insurance underwriting processes, and the risk grade calculation operation usually lends itself very naturally to a business rule based implementation.

The functional requirement of a risk grading service is simple: given a set of information on both the applicants' financial history and the loan collateral or the object of the insurance policy, the service returns a recommendation (accept or reject the application, or refer it to manual underwriting) and, if the recommendation is to accept, it returns a value on a risk scaling grade.

This value reflects the risk of contracting with the applicants, and directly influences the pricing of the contract that will be established. Although the rules themselves may be very complex, the

service interface is often straightforward: the "application" object is passed as the input parameter and the output parameter mainly carries the recommendation, the risk grade, and the possible messages that have been collected during the risk grading process.

One non-functional requirement adds a little twist to the service invocation: most of the time, the background information on the applicant or on the object of the contract (car, house, and so on) has to be acquired from third party companies such as credit bureaus (such as Equifax) or other data collection companies or agencies (such as LexisNexis).

Ordering these reports is both:

- A costly part of the client risk assessment process, and ideally should be performed just-in-time and only if necessary.
- An operation that is usually carried through synchronous, possibly failing, socket communication with the bureau and therefore not appropriate as a business rule action.

For these reasons, what we need here is a combination of careful ordering of the rule tasks and a re-entrant rule flow design, so that calls to the external bureaus can be interleaved with the rule flow execution.

Ordering the rule tasks

The fail-first principle [\[1\]](#) is a time-honored heuristic to improve the performance of problem resolution that involves some amount of decision making. The principle states that "To succeed, try first where you are most likely to fail." This usually translates into trying to first verify the simplest constraints of the problems, usually the ones that involve the least number of variables.

For example, if the application can be rejected only on the ground that one of the applicants is a minor, we obviously should perform that check before ordering his or her credit report. In this context, we not only want to fail first, but also to do so with requesting the minimum amount of information from third party. So we'll start with a rule task that includes the rules that don't use any report information and are solely based on existing application data. We'll then sort the different reports based on which one brings information that is most likely to disqualify the applicant, or allow an immediate determination of their risk (usually the lowest grade).

Designing the rule flow

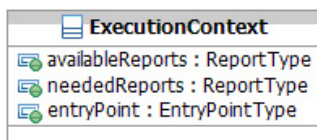
The external information needed to process risk information is most commonly acquired through synchronous socket communication with data collection agencies. Communication may (and often does) hang or fail if the agency service is busy or temporarily unavailable. Handling data acquisition, and exceptional conditions that may result from it, is not a business concern. It should be the responsibility of the component that invokes the risk grading rule service and kept outside of the rule service logic.

This means that, when it needs more information, the risk grading service should interrupt itself and return to the invoking component, which should collect the needed information, then re-enter the service where it left off, with the enriched application data.

In order to achieve this, the rule service parameters need to carry rule service execution context information, which can be grouped in an object such as the one shown in [Figure](#) . The information carried at any given point in time is the following:

- `availableReports`: The list of reports that are currently available as part of the input request parameters.
- `neededReports`: The list of reports that are needed to proceed to the next step.
- `entryPoint`: The entry point to which the execution should return after the needed report information has been acquired. This entry point variable plays the role of the `requested-service-name` parameter that described in [Single ruleset with internal orchestration](#).

Figure 7. Execution context information



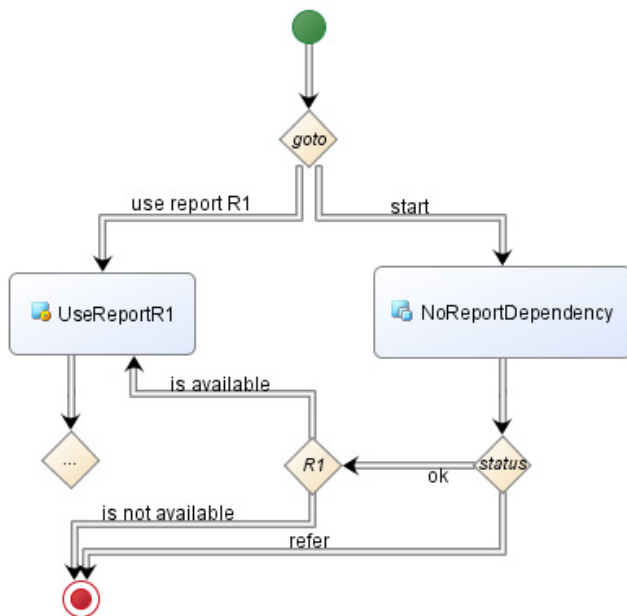
The direction of this execution context parameter should be set as `IN_OUT`, since the information it carries goes back and forth between the calling component and the rule service.

[Figure 8](#) shows a prototypical section of the rule flow that would make use of this execution context parameter. The `NoReportDependency` flow task groups the set of rules that can be executed with just the initial information passed by the service request, while `UseReportR1` is a rule task in which rules are making use of information carried by a report named `R1`.

Let's examine the successive choice nodes in this flow:

- `goto`: The first element of the rule flow should be a choice node in charge of routing the execution to the appropriate task depending on the value of the `entryPoint` attribute of the execution context object. The first time the service is invoked, the entry point is obviously initialized with a `start` value. On the subsequent service invocations, the `entryPoint` value determines where to resume the execution of the rule flow.
- `status`: This choice node tests whether the rules from the previous task have readily determined that the application should be referred (or rejected) or whether the risk determination should be pursued. If the application is referred, the entry point can be set to stop in the code of the `refer` condition branch, so as to signal to the calling component that the risk grading service has completed its execution.
- `R1`: This choice node is checking whether report `R1` is part of the available reports list. If not, `R1` is added to the `neededReports` attribute of the execution context parameter and the rule flow terminates. When the execution thread returns to the invoking application, it is the application's responsibility to read the value of the `neededReports` attribute and fetch the necessary reports accordingly.

Figure 8. Prototypical rule flow section



The choice node that follows the `UseReportR1` task and what comes after it basically replicates the flow section, starting from the status choice node: verify status, verify availability of the report, and use the report.

Summary

In this article, I've presented three different approaches for mapping rule-based decision services to deployable rulesets, and reviewed them against different flexibility and performance criteria. You've seen that each approach presents different pros and cons with respect to these criteria. However, a solution that combines multiple rulesets with external orchestration through a BPM engine is probably the one that brings the most modularity, the easiest change management, and the best performance.

Acknowledgments

Special thanks to Eduardo Rodriguez, Rajesh Rao, Zhuo Fu, Naba Gogoi, FuDong Wang and Amat Lutan for their insights and discussions on this topic.

Resources

- [Increasing tree search efficiency for constraint satisfaction problems](#), Haralick, R.M., Elliott, G.L., Artificial Intelligence 14, 1980, 263-314.
- [developerWorks BPM zone](#): Get the latest technical resources on IBM BPM solutions, including downloads, demos, articles, tutorials, events, webcasts, and more.
- [developerWorks WebSphere ILOG Business Rule Management Systems zone](#): Get the latest technical resources on IBM ILOG BRMS solutions, including downloads, demos, articles, tutorials, events, webcasts, and more.

About the author

Pierre Berlandier



Pierre Berlandier has worked for ILOG services for the past 15 years, during which time he has helped ILOG's clients leverage the successive incarnations of the rules programming paradigm from expert systems to real-time intelligent agents, and now business rules applications.

© Copyright IBM Corporation 2010

(www.ibm.com/legal/copytrade.shtml)

[Trademarks](#)

(www.ibm.com/developerworks/ibm/trademarks/)