

Additional Techniques for Process Variability

In the previous sections, we looked at the overall modularization and granularity approach to achieve process variability. However, we can achieve further dynamicity and variability by working on the process implementation itself and the technical constituents of a process. All these techniques look at reducing the impact of change whether it is a change to behavior, payload, or context.

Business Rule Engines

A first category of variability improvement is provided by an externalization of the behavior from code or models into configuration, policies, or property files that can be changed at runtime. These configurations, policies, and/or files require less testing because their management cycle

comes later in the Software Development Life Cycle. These business rules express business policies as a list of conditions to meet before performing a list of actions.

The supporting middleware is usually qualified as a business rules engine. When looking at using rules externalization in business processes, we must be careful to not add additional dependency that increases the life cycle. Thus, the interactions with rules engines must be defined as services, whether it is the process that calls the rules execution through services, or the rules engine execution calling services as resulting actions.

These rules engines use models and functional domains for which there are standards initiatives somehow converging on several domains. The rules semantics are defined by the Semantics of Business Vocabulary and Business Rules (SBVR) standard from OMG looking at ontologies and vocabularies. The exchange format is the focus of the W3C Rule Interchange Format (RIF) or Web Ontology Language (OWL), and the runtime aspects and APIs are addressed by JSR 94 Java API. SBVR is a metamodel specification for capturing expressions in a controlled natural language and representing them in formal logic structures.

Even though these standards provide some convergence, rules engines differences still apply, and there will be a gradation of usage and capabilities. There is no standardized rules classification, but here is a summary of the typical set:

1. Simple rule: If condition is met, then perform some action.
2. Rule set: Perform action if all conditions of a set of rules are met with no sequence implied between the rules.
3. Rule sequence: Evaluates all rules in a linear sequence and perform action if all rules conditions are met (used for compliance, validation).
4. Decision tables: In a decision table, more than one condition decides the action. The conditional logic is represented in a table where the rows and columns intersect to determine the appropriate action. Each column represents a condition. In a decision table, several conditions may get evaluated, but only one action is acted on.
5. Rule rete²: Plexus of rules where each rule has a dependency on at least one case of multiple predecessors; otherwise, it would be a sequence (used for correlation). The action is driven by the concentration of the rules' affluent to one last rule.
6. Other inference rules: Rules that have no organized dependencies and which the system needs to combine to determine whether the resulting condition is met.
7. Rules flow: An equivalent of a process or a workflow with branches and decision points where the next rule depends on the previous rule evaluation result.
8. Complex event processing: Where the rules add time dependencies and events to the rules. For example, if a credit card is used in a New York retail shop and then in a Texas shop in less than one hour, the credit card is a false credit card.

The same modularization that applies to private processes boundaries must be applied to sets, sequences, or flows of rules to avoid mixing business ownership of rules in different

domains. Then the action of the rules can be effective or informative, meaning that the rule action consists of providing information on the evaluation of the conditions, or can call a method or service to have a direct effect on the result of the evaluation.

In the first case, the rules engine is usually called as a service just to evaluate a business condition on information passed to the engine as a service interaction and return the evaluation, and then in the calling process take some decision based on the result of the rules evaluation. In the second case, the rules themselves can drive more complex actions and can be information model changes or external service or method calls that are fired when the rules detect that the appropriate conditions are met.

The following examples (see Figure 5-11 through Figure 5-13) give you a more thorough idea of some of the rule sets discussed previously. A simple rule example is a comparison with a value that returns true or false such as a loan approval limit amount check, as shown in Figure 5-11.

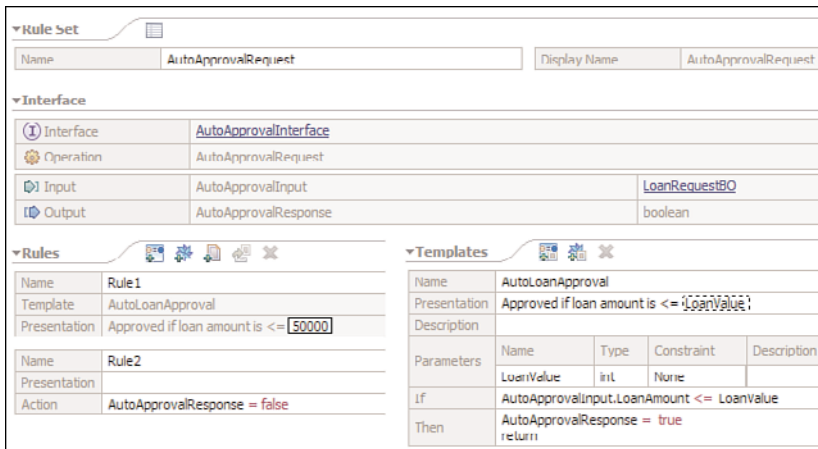


Figure 5-11 Simple rule returning Boolean

Because the limit amount may change, it has to be configurable at runtime, and a template is defined to replace the above 50000 value with whatever the LoanValue variable will be set to in the configuration user interface after the rules component has been deployed.

The example in Figure 5-12 using SBVR shows a rule that detects if the customer selects IPTV, it requires DSL broadband. It also detects which action modifies the information model accordingly. The vocabulary has previously defined elements from the information model, such as “product order” or “product order line item” and attributes such as “name.”

Figure 5-13 is a rule flow, consisting of a workflow of tasks that check the credit of a customer and select an appropriate insurance package based on the credit check rules verification result. Such rules flows should be kept in rules engines only if they consist of a cascade of rule

checks packaged as a global rule. Otherwise, it is preferable to use more services-oriented choreography engines and standards, such as WS-BPEL.

If the **product order** contains a **product order line item** with name "IPTV" and there is no product order line item with name "DSL" Then add **product order line item** "DSL" to the **product order**

Figure 5-12 Rule acting on product selection to define product dependency

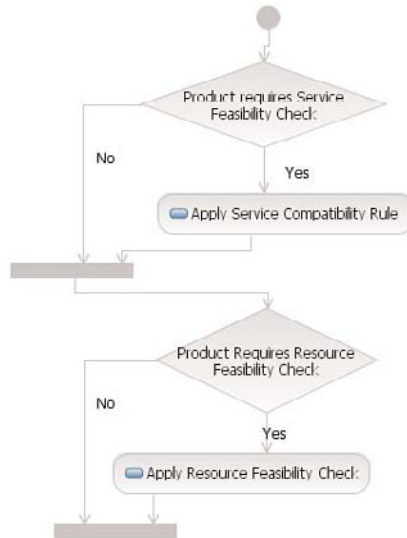


Figure 5-13 Example of rule flow in a UML activity diagram

This type of diagram is available from specific diagram builders in products such as Ilog JRules. In this type of flows rules can be conditionally applied based on the path that the rules evaluation resolves at each step. This can be used to build complex combinations of rules.

Extracting Routing Logic from Business Processes

In Figure 5-6 earlier in the chapter, we described a pattern that exposes to a process a higher level interface with the implementation of a more dynamic selection of the target service implementation.

Pursuing further simplification of the business processes to limit the cost of maintenance, the selection of target service endpoints can be removed from these processes to allow an externalized policy based selection. This alternative to maintaining binding choices is to use parametric bindings based on characteristics. Instead of a service consumer interacting directly with a service, the service is called through an endpoint assembly layer that may make an endpoint selection routing decision based on content of the service payload, context of the call such as

incoming channel information, or contract such as response time or availability of the target endpoint.

For example, in a triple play provisioning each of the specific processes for DSL, IPTV, or Voice over IP is exposed using the same interface. Adding the routing logic in the calling process makes this process dependent on the target endpoint and would require a process to change the day a “gaming” fourth play needs to be added to the possible bundles.

Since the variable payload contains the required service name it is much simpler for the calling process to loop on the set of included service orders and call the same service interface for each included service order deferring the appropriate service endpoint selection to a policy based routing layer configurable at runtime. In this case the routing will be performed based on the Service Name.

This endpoint routing layer is a semantic aware extension of an enterprise service bus, usually using ontologies to characterize the policies. The calling process logic is simplified, and additional service endpoints can be added with the appropriate policies to dynamically make decisions on the endpoints you want to invoke, based on the interface that is requested and the costs for those endpoints.

As you can see in Figure 5-14, after the endpoint selection and routing has been made there may still be the need to adapt the granularity of the exposed interface to the bottom-up reality of the applications. This adaptation needs to be performed as close as possible to the application to potentially handle units of work or manage state of data that is not exposed by the to interfaces but still needs to be preserved in the interactions. As already mentioned this topic requires specific consideration, and we address policy based endpoint matchmaking, granularity, and variability adaptation and matchmaking techniques in Chapter 6.

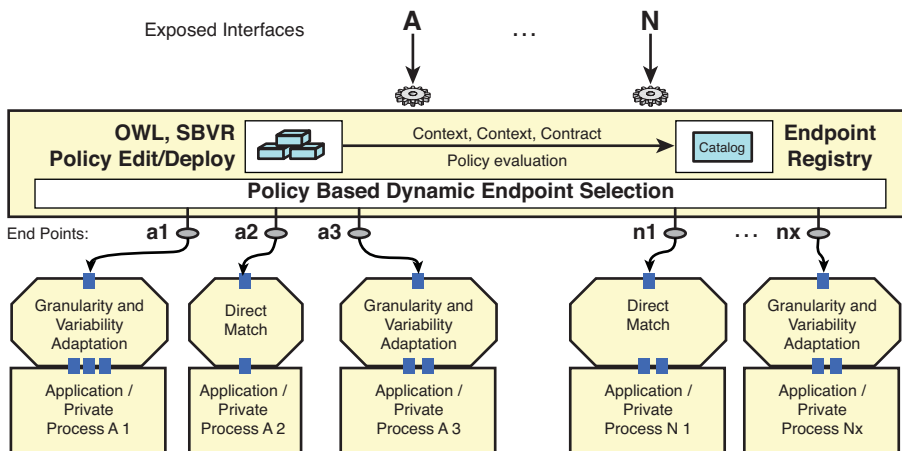


Figure 5-14 Policy based dynamic endpoint selection

Limiting Information Model Impact on Business Processes

We discussed in Chapter 3, “Implementing Dynamic Enterprise Information,” the introduction of variability in the information model, but this might still not be enough to reduce the impact of information changes on the processes.

In too many cases, I have been confronted with the change impact on processes because of indirect changes in the information structure. Even though tools make changes easy in process models, the deployment life cycle often involves a costly system test phase triggered by changes that have no direct relationship with the processes.

In consequence, processes should only carry the decisional information subset, so that changes of information that attribute structure not related to decision flows do not affect the processes. This is also where information as a service starts playing a more important role to separate the pure information persistence roles from the process flows.

For example, instead of carrying the full customer information through customer order processing, a customer key identifier and just location information can be sufficient to handle the order process. If any integrated application needs more information in its specific interfaces, it can use mediation in the bus or adaptation layer to retrieve that information using the key identifier and then call the target interface. This pushes the information dependency closer to the target integration points instead of adding a dependency on the private processes.

Figure 5-15 shows an example of separation of decisional data from the other information.

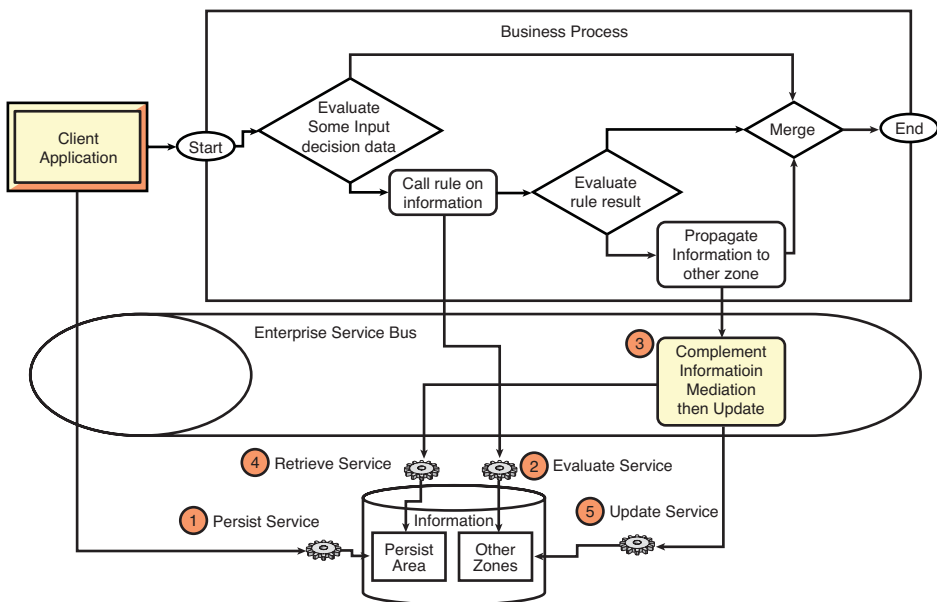


Figure 5-15 Separation of decisional data from persistence bulk

In Figure 5-15, the client application calls the Persist Service (1) first and then passes to the process only the subset that the process needs. Then in the bottom flow, some additional evaluation is required of the Evaluate Service (2), which returns the result of this data analysis. Then if that information needs to be propagated to another zone in the information space the process calls the Enterprise Services Bus, which applies a mediation (3) that uses the Retrieve Service (4) to complement the messages required by the Update Service (5). Using this approach additional data can be added without affecting the process. An additional improvement can be to use mediation between the application and the process so that the application does not perform the separation anymore but instead a bus mediation separates the data for persistence from the data for the process.

A secondary benefit of limiting the amount of data that flows through process choreography engines is that most of these engines' performance is affected by the size of the data, particularly for long-running processes requiring persistence of that information in an underlying database.

Many of the BPEL engines use JDBC to persist the process data and VARCHAR variable character strings. As soon as the VARCHAR or TABLESPACE limit is crossed (this limit depends on the database provider), some of the engines switch to a mode that may impact severely the performance. Overall, containing the process data to the smallest possible size has functional and nonfunctional benefits toward more agility and variability.

Realizing Event Driven Business Processes

A dynamic approach to the implementation of processes is to consider the processes a chain of events triggering some components with actions implementing business logic. In turn, that logic generates events but neither expects nor waits for the results of these events. The result is that the end-to-end process is realized implicitly by this chain, and a given logic in the chain does not need to know what the next step is. This gives a modular approach to processes where a new event listener can be addressed to the process chain without affecting any on the event issuers.

The true difference in an event driven approach is that pure event driven approaches are opportunistic while services approaches are deterministic. This means that if the issuer of the event message expects a specific action as a consequence of this event this is a strict service oriented architecture (SOA) interaction with a deterministic approach and not an event driven dynamic approach.

On the contrary, if the issuer of the message does not expect a specific action as the result of the event message, but if listeners examine the message and apply some rules to make out if they have to do something about it, then we are in a true event driven architecture (EDA).

After the listener has made out that it matches the condition that requires him to handle that event, it will, however, call services that can be the façade to private processes. The deterministic and nondeterministic approaches can complement themselves as business processes can be triggered by events, and asynchronous services can be considered events triggering an action. So in essence, there is not such a big difference between an event driven approach and an asynchronous

services approach. In addition, standards such as WS-Notification and WS-BPEL include all the necessary support for managing correlation of incoming events and instances of processes.

Some standards try to address that integration between event driven and services oriented architecture. An example of this continuity between the event approach and the services approach is the telecommunication industry Multi-Technology Operations System Interface (MTOSI) standard that identifies message exchange patterns and communication styles in its MTOSI XML Implementation User Guide: MSG (message) and RPC (remote procedure call) describing the interaction mode.

The message mode in Table 5-2 can be considered an event mode. As shown in Table 5-2, concrete approaches try to use both approaches where they make sense.

Table 5-2 Message Exchange Patterns Supported in MTOSI V1.0³

Communication Pattern				
Communication Style	Simple Response	Multiple Batch Response	File Bulk Response	Notification
MSG (Asynchronous)	(ARR) Asynchronous Request/Reply	(ABR) Asynchronous Batch Response	(AFB) Asynchronous (File) Bulk	(AN) Notification with async subscription
RPC (Synchronous)	(SRR) Synchronous Request/Reply	(SIT) Synchronous Iterator	(SFB) Synchronous File Bulk	(SN) Notification with sync subscription

An implication of this dynamic and modular event driven process approach is that there is no concrete process model. The end-to-end representation of event driven processes is a pure abstract representation, as no formal executable script is expected from the representation of the process. However, business monitoring requires the construction of the implicit process. This representation provides a selective view of the chain of events that happened between the initial event and the last action.

This is typically the way that we address the abstract process monitoring in Chapter 8, “Managing and Monitoring a Dynamic BPM and SOA Environment.”

Summary

In this chapter, we saw that dynamic and flexible processes require first an appropriate design for a modularity approach aligned with the enterprise business architecture. We defined rules for modularizing processes and having appropriate services granularity. I want to reinforce, at this