Chapter 11
Rule Authoring in JRules

Target audience

- *Business analyst, developer, rule author*

In this chapter you will learn

- *The different rule entry languages and rule artifacts, namely, technical rules, action rules, decision tables, decision trees, and scorecards*
- *How to build your custom rule language*
- *How to orchestrate rule execution with ruleset parameters and ruleflow*
- *How to optimize rule execution by selecting the appropriate rule execution algorithm for a given rule task*

Key points

- *The Ilog Rule Language (IRL) is the foundation upon which other languages and rule artifacts are built.*
- *Action rules, decision tables, decision trees, and scorecards are translated into/executed as IRL technical rules.*
- *Be aware of the possibility to develop your own rule language (with the Business Rule Language Development Framework), but resist the temptation to.*
- *Refer to your application objects through ruleset parameters.*
- *Use ruleflows to orchestrate rule execution. They provide a high-level control mechanism and a context for rule execution.*
- *Ruleflows offer opportunities for speeding execution through run-time rule selection, and algorithm selection.*

11.1 Introduction

In Chap. 10, we explored the rule authoring *infrastructure* in JRules, where we focused on *rule projects* and *the business object model*. Rule projects and rule project dependencies enable us to modularize rule development in such a way as to facilitate the sharing and reuse of rule artifacts across different functional areas. The *business*

J. Boyer and H. Mili, *Agile Business Rule Development*,
DOI 10.1007/978-3-642-19041-4_11, © Springer-Verlag Berlin Heidelberg 2011

object model represents the business view of the application data – be it Java classes or XML data. It is defined through a powerful (*BOM to XOM*) mapping language in much the same way that relational database views are defined using the underlying database, by filtering irrelevant properties, defining computed attributes, or introducing so-called *virtual* classes that mean something to business but that are not supported in the underlying application model (Java or XSD). In particular, we saw how the architecture of the BOM and the BOM to XOM mapping is able to *absorb* a wide range of changes to the underlying application model (Java or XSD), without affecting the existing rules. If it is the rules that we want to rephrase, a *vocabulary refactoring functionality* propagates vocabulary changes to the rules that use them.

In this chapter, we first explore the different kinds of rule languages (the *ILOG Rule Language*, or IRL, and the *Business Action Language*, or BAL) and rule artifacts that use them, namely, *technical rules* (IRL), *action rules* (BAL), *decision tables* (BAL), *decision trees* (BAL), and *scorecards* (IRL). We also provide an overview of a framework for developing rule languages. In Sect. 11.3, we discuss rule execution orchestration, where the focus is on organizing the execution of rules during run-time. We will talk about ruleset variables and parameters and about *ruleflows*, which are process flows whose tasks consist of groups or rules. Ruleflows enable us, among other things, to statically select the algorithm to use to execute for a particular rule task, and to dynamically select the rules that execute within a particular task. Best practices are presented in Sect. 11.4. We conclude in Sect. 11.5.

11.2 Rule Artifacts

In this section, we give a *brief* overview of the various rule artifacts. Space limitations do not allow us to delve too deeply into any of the artifacts or languages covered; the tutorials and reference manuals included in the product documentation do a much better job at that! Our purpose is to provide the reader with a roadmap of the rule artifacts, and the relationships between them.

With this in mind, it makes sense to start with the IRL language, or *ILOG Rule Language*, which is the ancestor of all of the languages and artifacts,[1] and *technical rules* which are if-then rules written using IRL. IRL is also the only language that the rule engine understands, and every other language or artifact has to map to IRL. We then talk about the *Business Action Language*, which references the business *vocabulary* as opposed to the BOM elements directly, and *action rules* which are *if-then* rules written in BAL. We next talk about decision tables and decision trees, which are higher-level rule aggregates written using the BAL. Scorecards represent yet another high-level artifact typically used in risk assessment or credit worthiness applications. Finally, we briefly talk about the *business rule language development framework*, which is a flexible framework for developing rule languages, of which the BAL is an instance. Figure 11.1 shows the different kinds of languages, artifacts, and relations between them. Ruleflows will be discussed in Sect. 11.3 about rule execution orchestration.

[1]The first full-fledged ILOG Rule Language was based on the rule language OPS5 (see Chap. 6).

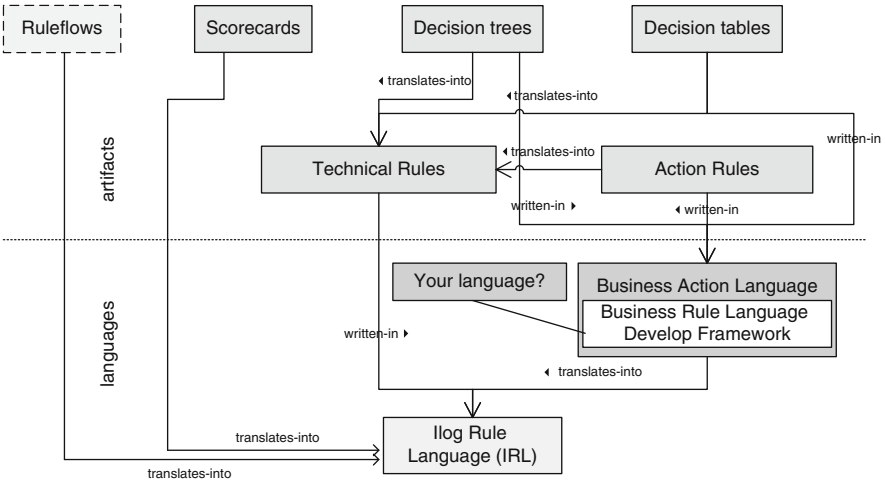


Fig. 11.1 The rule language landscape in JRules

11.2.1 IRL and Technical Rules

As shown in Fig. 11.1, the *ILOG Rule Language* is the base language for rule artifacts of JRules, including, but not limited to, if-then "technical rules". Historically, IRL was synonymous with if-then rules, and so we will start with the subset of IRL that concerns if-then rules, called *technical rules* as opposed to *action rules* which are written using the BAL. Let us first start with a simple example.

```
rule YoungDriver {
  property priority = 0;
  when {
    ?driver: Driver(age < 25);
  } then {
    System.out.println("Found a young driver:" + ?driver);
    }
}
```

A rule has a name (`YoungDriver`), some metadata/properties (**`property`** `priority = 0`), an if/when/condition part (**`when`**`{?driver: Driver (age < 25);}`), and a then/action part (**`then`** `{System.out.println("Found a young driver:" + ?driver);}`). This rule will fire for those instances of class `Driver` found in working memory, whose `age` attribute is smaller than 25. For each such driver, the rule will print the string "Found a young driver:" followed by the output of the `toString()` method on that object. The expression `Driver(age < 25)` is called a *class condition*, where the class name is `Driver`, and the condition is `age < 25`

where `age` in this case refers to a *public* data member of the class `Driver`. The expression `'"?driver : ..."` is a *variable binding*, where `?driver` is the variable name. In this case, `?driver` will be bound to an instance of `Driver` that matches the condition `age < 25`, and becomes referencable in subsequent conditions or actions of the rule. The action part of the rule shows some vanilla-flavor Java. Indeed, we can use pretty much any Java expression in the action part, with the following limitations:

1. The underlying object model ("vocabulary") is the BOM and not the XOM. This means that only the classes that are part of the BOM are "referencable."[2] For example, if the underlying Java class `Driver` has a pair of getter/setter for attribute `age`, in IRL, we set the value of `age` using the dot notation (`?driver.age = 24`), i.e., by manipulating the BOM class, as opposed to using the setter (`?driver.setAge(24)`). Further, we can refer to virtual classes and virtual class members (see Sect. 10.3.3), which do not exist in the real world (Java).
2. The definition of complex types (classes, interfaces, enums) is not supported, but who would do such a thing in the action part of a rule, anyway.
3. Some exotic expressions are not supported, e.g., the instantiation of anonymous Java classes.

Consider now this next rule:

```
rule big_claim_over_90_days_past_exp_date_policy {
  property effectiveDate = java.util.Date("1/1/2010");
  property expirationDate = java.util.Date("12/31/2010");
  property status = "development";
  when {
    ?myPolicy: Policy(?bDate: beginDate; ?eDate: endDate);
    ?claim: Claim(amount > 1000; date.before (?bDate) ||
                              date.moreThan(90,?eDate);
                              policy == ?myPolicy);
    } then {
      ?claim.status = Claim.REJECTED;
      update ?claim;
      System.out.println(?claim+ " is rejected because …" +
                                                ?myPolicy);
      }
}
```

[2]When we build a BOM entry from a Java project, we not only "import" the classes from that project, but we also "import" commonly useful classes from the Java library including basic types, collections, java.util.Date, java.util.Calendar, etc. These additional classes constitute what JRules calls the *boot bom*. Empty BOM entries actually are not empty: they contain the *boot bom*. The default boot bom can be changed.

More often than not, rules have *effectiveness periods* during which they are in force. Rules can expire if they are replaced by new rules, or if they embody a time-limited policy such as limited-duration promotional campaigns and such. The property `status` is used to assign a development status to a rule. Indeed, like other software artifacts, rules will undergo a *development lifecycle* starting with the coding of the rule (`status = "development"`) to its testing, from which a rule can either be rejected or promoted to production. A rule that is in production may return to development, for debugging or maintenance, or retirement – if it is superseded by new rules. We will talk about rule governance in Chaps. 16 and 17.

Consider now the condition part of the rule. This rule has two *class conditions* which are considered to be ANDed. The first class condition (`?myPolicy: Policy(...);`) is actually no condition: It will match *any* `Policy` object in working memory, and will bind the values of its `beginDate` and `endDate` attributes to the variables `?bDate` and `?eDate`. The second class condition on **`Claim`** consists of three test/conditions, also considered to be ANDed: (a) a condition on the amount (`amount > 1000`), (b) a condition on the `date` of the claim, which says that the `date` of the claim is *either* prior to the beginning of the policy *or* is more than 90 days past `?eDate`, i.e., more than 90 days past the `endDate` of `?myPolicy` (`date.before (?bDate) || date.moreThan(90,?eDate)`), and (c) a condition tying `?myPolicy` to `?claim`. The last condition (`policy == ?myPolicy`) ensures that, should the working memory of the engine contain several policies and several claims, the rule only matches < `?claim`,`?myPolicy` > pairs that are related. In JRules-speak, the last *two* conditions are called *join conditions* because they relate two objects together.

The action part of this rule shows three statements. The first and third statements look like regular Java. The second statement (**`update`** `?claim`) tells the engine to reevaluate the rules involving the object `?claim`. Indeed, the engine needs to be told explicitly which objects may have changed in a way that might match new rules, or invalidate existing ones.[3]

Let us take a first shot at the grammar for technical rules, using a mixture of EBNF[4] and regular expressions:

[3]Recall the discussion in Sect. 6.3.2 regarding the engine notification. The good news is that BAL rule authors do not need to worry about this, because this behavior can be configured at the BOM level. Indeed, we can set up a particular data member setter (or void function member) to automatically trigger an update when used in the action part of a rule. We will come back to this in the next section when we talk about BAL to IRL translation.

[4]EBNF stands for Extended Backus-Naur Form. People familiar with Yacc or ANTLR will recognize the syntax. Things that are supposed to appear as-is (language keywords) appear between quotes. Things that are optional appear between square brackets ([optional]). Groups of things that can appear zero or more times appear as (...)*.

- TECHNICAL_RULE ::= "**rule**" RULE_NAME "{" (RULE_PROPERTY)* "**when {**" CONDITIONS "**} then {**" ACTIONS "**}}**"
- RULE_PROPERTY ::= "**property**" PROP_NAME "**=**" PROP_VALUE "**;**"
- CONDITIONS ::= CONDITION (CONDITION)*
- CONDITION ::= CLASS_CONDITION | EXISTS_COND | NOT_COND | COLLECTION_COND | EVALUATE_COND | WAIT_COND
- CLASS_CONDITION ::= [VAR_NAME ":"] SIMPLE_CLASS_COND ";"
- SIMPLE_CLASS_COND ::= CLASS_NAME "**(**" TEST_BIND_LIST "**)**" [SCOPE_EXPRESSION]
- SCOPE_EXPRESSION ::= "**from**" SINGLE_OBJ_EXPRESSION | "**in**" COLLECTION_OBJ_EXPRESSION
- TEST_BIND_LIST ::= TEST_OR_BINDING (";" TEST_OR_BINDING)*
- TEST_OR_BINDING::= TEST | BINDING
- NOT_COND ::= "**not**" SIMPLE_CLASS_COND ";"
- EXISTS_COND ::= "**exists**" SIMPLE_CLASS_COND ";"
- EVAL_COND ::= "**evaluate(**" TEST_BIND_LIST ");"
- COLLECTION_COND ::= [VAR_NAME ":"] "**collect**" SIMPLE_CLASS_COND ["**where(**" TEST_BIND_LIST ")"] ";"
- ACTIONS ::= ACTION (ACTION)*

We will say a few words about the different types of conditions. This will help us understand the BAL to IRL translations, to be discussed in the next section.

Consider the following condition:

```
rule no_expensive_claims_in_WM {
  when {not Claim(amount > 1000);}
  then {System.out.println("No expensive claims in WM");}
}
```

The condition part is satisfied if there are no claims in working memory worth more than 1,000. Similary, the rule:

```
rule there_are_expensive_claims_in_WM {
  when {exists Claim(amount > 1000);}
  then {System.out.println("there are expensive claims in
WM");}
}
```

will fire *once* if there exist claims worth more than 1,000. In particular, if there are one or a hundred such claims, the rule will still fire only once. Contrast that with

```
rule found_an_expensive_claims_in_WM {
 when {Claim(amount > 1000);}
 then {System.out.println("found expensive claim in WM");}
}
```

which will fire for *every* claim in working memory that is worth more than 1,000.

Consider now the following rule:

```
rule policy_with_no_expensive_claims {
  when {
     ?myPolicy: Policy();
     not Claim(amount > 1000) in ?myPolicy.getClaims();
     } then {
     System.out.println("Policy with expensive claims");
     }
  }
```

In this case, we look for a policy in working memory, and check that there are *no* claims, *for that policy* (**in** `?myPolicy.getClaims()`) that are worth more than 1,000. The expression "**in** `?myPolicy.getClaims()`" corresponds to what we referred to in the grammar as `SCOPE_EXPRESSION`. We use **in** when the scope is a collection (`?myPolicy.getClaims()`) and **from** when the scope is a single object.

Let us now illustrate an example of `COLLECTION_CONDITION`. Consider the following rule:

```
rule policy_with_more_than_3_at_fault_claims {
  when {
     ?myPolicy: Policy();
     ?claims: collect Claim(amount > 1000;
      responsibility == AT_FAULT) in ?myPolicy.getClaims()
                          where (size() > 3);
  } then {
  System.out.println(?myPolicy + " had more than 3 " +
                  "at-fault claims worth more than 1000");
  }
}
```

In this case, the variable `?claims` will contain a collection[5] of the claims of `?myPolicy` that are worth more than 1,000, with `AT_FAULT` responsibility. The "**where**" clause indicates conditions on the collection, in this case `(size() > 3)`.

We conclude our overview of IRL by an illustration of the **evaluate** condition. The **evaluate** condition is a convenience construct that enables us to group variable bindings and tests outside of a class condition. For example, the rule `big_claim_over_90_days_past_exp_date_policy` shown above can be written as follows using an **evaluate**:

[5]The exact type is `ilog.rules.engine.IlrCollection`, which is a *dynamic collection* in the sense that objects will be automatically removed from the collection as soon as they no longer satisfy the conditions that got them in.

```
rule big_claim_over_90_days_past_exp_date_policy {
  property effectiveDate = java.util.Date("1/1/2010");
  property expirationDate = java.util.Date("12/31/2010");
  property status = "development";
  when {
     ?myPolicy: Policy();
     ?claim: Claim();
     evaluate(?bDate: ?myPolicy.beginDate;
              ?eDate : ?myPolicy.endDate;
              ?claim.amount > 1000;
              ?claim.date.before(?bDate)||
              ?claim.date.moreThan(90,?eDate);
              ?claim.policy == ?myPolicy);
  } then {
     …
     }
}
```

In other words, we took all of the bindings and tests out of the class conditions, and grouped them in the (external) **`evaluate`** condition. The Boolean value of an **`evaluate`** is the conjunction of the individual tests contained within. In fact, the technical rules generated from BAL rules look like this (more about the BAL to IRL translation in the next section). Note that, considering that the conditions clauses of a rule are ANDed, the **`evaluate`** enables us to write disjunctions between tests that are part of different class conditions.

In this section, we skimmed the surface of the IRL language. There are a number of rule-specific constructs that can be used within the condition and action parts of a rule that we did not talk about:

1. Constructs for event management: The IRL (and the JRules rule engine) enables us to reason about time and events. For example, we can write a rule that says "if event A occurred, wait 5 s for event B to occur, if it does, do X, if you time out, do Y."
2. Constructs for truth maintenance: There are situations where rules create objects from within their action parts when their condition part is satisfied. Consider the e xample of a monitoring application that creates an **`Alarm`** or a **`ServiceRequest`** object when some parameter of the system or device being monitored goes out of range. With normal rules, if the parameter in question returns to its normal range, the **`Alarm`** or **`ServiceRequest`** remains in working memory and will be processed. If we want the **`Alarm`** or **`ServiceRequest`** to be retracted if the conditions return to normal, we use specific constructs within the rule.[6]

[6]Called *logical conditions* and *logical assert*. The system maintains some sort of a reference-count of *justifications* for `Alarm` (or `ServiceRequest`) objects, and we need to redefine the `equals` method on the class `Alarm` (or `ServiceRequest`) accordingly. More can be found in the product documentation.

3. Constructs for working memory management within the action part of rules. Actually, we saw one: the **`update`** keyword. There is the **`insert`**, **`retract`**, **`modify`**, and **`update refresh`**. All of these have equivalent methods in the **`IlrContext`** class. More about these constructs can be found in the product documentation.
4. The **`else`** clause in rules. Indeed, IRL rules can have an **`else`** clause, but with a special meaning: the "**`else`** action part" is executed when all the conditions but the last **`evaluate`** statement yield true.[7]

Finally, as mentioned above, the IRL is not just for writing technical rules: It is also used to write *functions* and ruleflows. We will talk about ruleflows in Sect. 11.3.2.

11.2.2 BAL and Action Rules

When we deliver trainings, we lose developers at about this point. They tune out and start playing with IRL exploring the limits of the language, raising their heads only to ask questions. Alas, for all its power – we have only scratched the surface – the IRL is not appropriate for business consumption. Business cannot understand IRL rules, cannot relate them to business logic, and cannot own them, by taking over rule development and maintenance; if we code business rules in IRL, we defeat the major tenets of the business rules approach. The *Business Action Language* (BAL) enables all of the above.

The basic structure of a BAL action rule is illustrated below. A *typical* BAL rule has three parts:

1. A ***`definitions`*** part, to declare *rule variables* to be referenced in the condition part, the action part, or subsequent definitions
2. An ***`if`*** part, which consists of a Boolean expression that typically uses the variables of the rule
3. A ***`then`*** part, consisting of one or several actions that typically use the variables of the rule, ending with a semi colon (";")

[7]We would not explain it any further, especially that we strongly recommend *not* using else, because it makes rules error prone and the ruleset hard to maintain.

As mentioned earlier, when we talked about the BOM, BAL rules, much like IRL rules, refer to elements of the BOM. However, whereas IRL rules refer *directly* to the BOM elements using a Java-like object notation, BAL rules refer to BOM elements through their *verbalizations*. We show below the corresponding IRL translation.[8]

```
rule claim_date {
   property status = "new";
   when {
      current_claim: Claim();
       evaluate (current_claim.fileMoreThanXDaysAfter(90,
           current_claim.policy.startDate.toDate()));
   } then {
      current_claim.decision = "INELIGIBLE";
       update current_claim;
      current_claim.logRuleFiringWithMessage("Claim filed
too late");
   }
}
```

The BAL definition of the variable "**current claim**" yielded the simple class condition, with an object binding:

```
current_claim: com….Claim();
```

and the condition of the ***if*** part ended up in a single **evaluate** statement. This is the general translation pattern from BAL to IRL. Lest we oversimplify:

1. ***definitions*** become simple class conditions.
2. The conditions of the ***if*** part get lumped into a single **evaluate** statement.
3. The BAL ***then*** part maps to the (IRL) **then** when part.

Two points are worth noting. First, the reader may have noticed the "**update** `current_claim;`" action shown in the IRL translation. This action was inserted in the action part of the rule *after* the assignment of a new value to the `decision` attribute of the BOM class **Claim**, because that attribute had the "`Update object state`" option checked. Second, the BAL to IRL translator replaced the white space in the variable name ("**current claim**") by an underscore (`current_claim`) to make the variable name IRL/Java compliant.

In the remaining paragraphs, we will talk briefly about definitions and variables, the condition part, and the action part. Before we do that, we will present the high-level structure of the language in a similar notation to the grammar of the IRL shown in the previous section:

[8] In this and subsequent IRL translations, we omitted class package names for presentation purposes. Just be aware that the *real* IRL code shows fully qualified class names in class conditions.

- `BAL_RULE ::= [DEFINITION_PART] [CONDITION_PART] ACTION_PART [ELSE_PART]`
- `DEFINITION_PART ::= "definitions" DEFINITION ";" (DEFINITION";")*`
- `CONDITION_PART ::= "if" CONDITION ( (and | or ) CONDITION)*`
- `ACTION_PART ::= "then" ACTION";"(ACTION ";")*`
- `ELSE_PART ::= "else" ACTION";"(ACTION ";")*`

Notice that:

1. *Only* the action part is required in a BAL rule; everything else is optional. Thus, the following is a valid BAL rule.

2. A BAL rule can have an **`else`** part. As with the case of IRL, we discourage the use of the **`else`**, and for similar reasons.
3. The conditions of the condition part can be combined freely using the logical operators **`and`** and **`or`**.

We will further expand on the different components as we talk about them.

Definitions and variables. To write conditions and actions on objects, we need to refer to them through *variables*. In BAL (and IRL), there are three different kinds of variables:

1. *Rule variables*. These are variables defined in the rule itself through the `DEFINITIONS` part of the rule. Such variables have rule-scope: They are only visible within the rule, and only live while the rule is being executed.[9] In particular, the names of these variables need only be unique within the context of a single rule; several rules can use the same variable name (e.g., "**`current claim`**").
2. *Ruleset variables*. These are variables that are defined at the rule project level. They are visible within all of the rules of a project, and they live during one ruleset execution.
3. *Ruleset parameters*. These variables are also defined at the rule project level, and are visible within all of the rules of a project. They are used to pass data in and out of the engine during ruleset execution.

The difference between ruleset *variables* and rule *parameters* is like that between the *local variables* and *parameters* of a function.

[9]The lifetime issue is a bit more complex: It spans the evaluation of the condition part and the lifetime of the rule instance, if one is created. See Chap. 6.

We will talk about ruleset variables and parameters in Sect. 11.5. Here we focus on rule variables. The syntax for a rule variable definition can be described as follows:

- DEFINITION ::= **"set"** VAR_NAME **"to"**VAR_VALUE [**"where"** BOOLEAN_EXPRESSION]
- VAR_VALUE ::= CONSTANT | REFERENCED_VAL| ANON_OBJ_VALUE | ANON_OBJ_COLL
- REFERENCED_VAL ::= VAR_NAME | ATTRIBUTE REFERENCED_VAL
- ANON_OBJ_VALUE ::= BOM_TYPE [SCOPE]
- ANON_OBJ_COLL ::= **"all"** BOM_TYPE [SCOPE]

The following illustrates the first three types of definitions:

```
definitions
    set 'THRESHOLD EXPENSIVE ACT' to 1000 ;
    set 'current claim' to a claim ;
    set 'claimed service act' to a service act in the service acts of 'current claim' ;
    set 'expensive service act' to a service act in the service acts of 'current claim'
          where the cost of this service act is at least 'THRESHOLD EXPENSIVE ACT' ;
    set 'expensive service act cost' to the cost of 'expensive service act' ;
    set 'current policy' to the policy of 'current claim' ;
    set 'young policy holder' to the policy holder of 'current policy'
          where the birth date of this policy holder is after 1/1/1990 ;
then
    print "Illustrating definitions" ;
```

The first definition corresponds to setting a variable to a constant. The next three correspond to setting a variable to an anonymous object value (ANON_OBJ_VALUE), with three variants: (a) simplest, (b) with scope, and (c) with scope and test. The last three definitions correspond to REFERENCED_VAL . The first ('expensive service act cost') illustrates the case where we create a variable to hold the value of a *scalar* attribute. The last two ('current policy' and 'young policy holder') show the case of a variable that holds the value of an attribute that is a domain object, without and with a condition. The following shows the IRL translation for the first five definitions:

```
    evaluate (THRESHOLD_EXPENSIVE_ACT : 1000);
    current_claim: Claim();
    claimed_service_act: ServiceAct() in
                              current_claim.serviceActList;
    expensive_service_act: ServiceAct(?this.cost >= (float)
THRESHOLD_EXPENSIVE_ACT ) in current_claim.serviceActList;
    evaluate (expensive_service_act_cost :
                              expensive_service_act.cost);
```

The reader will notice that variables that are of scalar type are defined using a binding (VAR_NAME **":"** VAR_VALUE) embedded within an **evaluate**. The middle three are defined using simple class conditions, with or without embedded tests, and with or without scope (**in** current_claim.someAttribute).

We now illustrate the definition of collection variables:

```
definitions
    set 'some claim' to a claim ;
    set 'all claims in WM' to all claims ;
    set 'some claim service acts' to all service acts in the service acts of 'some claim' ;
    set 'expensive service acts' to all service acts in the service acts of 'some claim'
            where the cost of each service act is at least 1000 ;
then
    print "illustrating collections" ;
```

And we show below the resulting IRL:

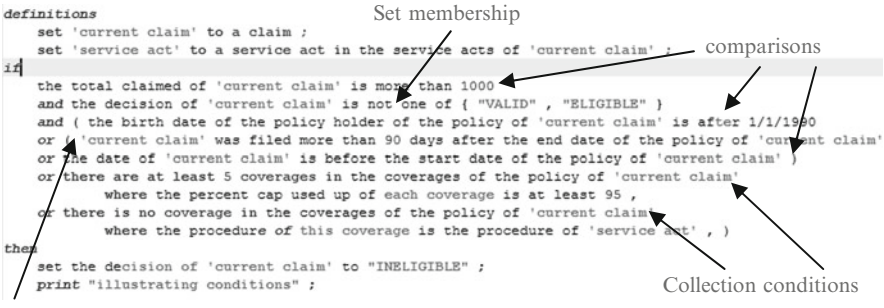
```
some_claim: Claim();
all_claims_in_WM: collect Claim();
some_claim_service_acts: collect ServiceAct() in
                                    some_claim.serviceActList;
expensive_service_acts: collect
                    ServiceAct(?this.cost >= 1000) in
                                   some_claim.serviceActList;
```

What can you do with collections? You can test the contents of a collection or its size, in the condition part, or iterate over its elements to apply a bunch of actions, in the action part. We will illustrate both uses in the subsequent discussion.

The condition part. Roughly speaking, the condition part of a rule consists of a logical combination of individual conditions using the logical operators **and** and **or**. In turn, each "top-level" condition can itself be a single Boolean term or a logical formula, with operators **and**'s, **or**'s, and parentheses. This is embodied in the next four grammar productions (rules).

- `CONDITION_PART ::= "if" CONDITION ( (and | or ) CONDITION)*`
- `CONDITION ::= BOOLEAN_TERM | BOOLEAN_TERM (and | or ) CONDITION`
- `BOOLEAN_TERM ::= BOOLEAN_TEST | "(" CONDITION ")"`
- `BOOLEAN_TEST ::= COMPARISON | PREDICATE | SET_MEMBERSHIP | COLLECTION_TEST |`

Next, we will show examples of the various Boolean tests.[10]



[10] *Please* do not write rules like this one at home ☺: This rule breaks every rule writing guideline we mentioned in Chap. 9. It is only meant to illustrate various syntactic constructs.

With the exception of collection conditions (COLLECTION_TEST), which we will address shortly, we build Boolean tests by selecting an object (a variable) or the attribute of an object, then a comparison operator (or a predicate) appropriate for that object, and then an operand of the appropriate type. In the rule above (which is utterly non-sensical), we show three comparisons. The first compares a numerical data member (`total claimed of 'current claim'`) to a constant (1,000). The second compares a date attribute (`the birth date of the policyholder of the policy of 'current claim'`) to a constant date (`1/1/1990`), while the third compares a date attribute (`the date of` ′**current claim**′) to *another* date attribute (`the start date of the policy of` ′**current claim**′). The case of a predicate is illustrated by the condition ′**current claim**′ `was filed more than 90 days after <some date>`. The SET_MEMBERSHIP case is illustrated by the condition `the decision of` ′**current claim**′ `is not one of` **{"VALID", "ELIGIBLE"}**. Set membership tests (`is not one of` and `is one of`) are available for all types (simple types, object types) and the values of the set can be enumerated *literally*, as in this example, or given as a collection variable.

Let us look now at the collection conditions. The BAL offers several types of conditions on collections, which may be described using the following grammar:

- COLLECTION_TEST ::= QUANTIFIED_COLLECTION_TEST | COLLECTION_SIZE_TEST | COLLECTION_CONTENT_TEST
- QUANTIFIED_COLLECTION_TEST ::= QUANTIFIER [NUMBER] OBJ_TYPE [scope] ["where" COLLECTION_ELEMENT_TEST]
- QUANTIFIER::= "**there are**" | "**there are at least**" | "**there are at most**" | "**there are more than**" | "**there are less than**" | "**there is no**" | "**there is at least one**" | "**there is one**" | "**there is at most one**"
- COLLECTION_SIZE_TEST ::= "**the number of**" OBJ_TYPE [scope] ["where" COLLECTION_ELEMENT_TEST] COMP_OPERATOR NUMBER
- COLLECTION_CONTENT_TEST ::= COLLECTION_NAME ("**contain**" | "**does not contain**") OBJECT

Behind the scenes (IRL), all of these conditions map to an IRL COLLECTION_CONDITION, but with different tests on the collection size (QUANTIFIED_COLLECTION_TEST and COLLECTION_SIZE_TEST) and collection contents (COLLECTION_CONTENT_TEST). The rule above shows two examples, using the "**there are at least**" and "**there is no**" forms. The following shows the IRL equivalent[11]:

[11]We simplified the underlying IRL to make it more readable, by: (a) removing class package names, (b) reducing the number of extraneous parentheses, and (c) simplifying/faking the way Date constants are handled.

```
when {
  current_claim: Claim();
  service_act: ServiceAct() in current_claim.serviceActList;
  var$_$0: collect Coverage(?this.percentCapUsedUp >= 95) in
                              current_claim.policy.coverageList;
  var$_$1: collect
      Coverage(?this.procedure.equals(service_act.procedure))
                       in current_claim.policy.coverageList;
  evaluate (
    current_claim.totalClaimed > 1000
    && current_claim.decision !in {"VALID","ELIGIBLE"}
    && (current_claim.policy.policyHolder.birth-
              Date.compareTo(new IlrDate(1990,1,1)) > 0
        || (current_claim.fileMoreThanXDaysAfter(90,
                    current_claim.policy.endDate)
            || current_claim.date.compareTo(
                    current_claim.policy.startDate) < 0
                )
            || var$_$0.size() >= 5
            || var$_$1.size() == 0
          )
          );
   }then {
         …
         }
```

The reader will notice that all of the conditions of the ***if*** part of the BAL rule ended up in the single **evaluate** statement.

There is more to BAL conditions than what we just covered. Our goal in this section is to show the "philosophy" of the BAL language. The full language reference is available in the product documentation.

BAL Actions. BAL actions are fairly straightforward. They can be of five different types:

- ACTION ::= SIMPLE_ACTION | FOREACH_COMPOUND_ACTION
- SIMPLE_ACTION ::= ATTRIBUTE_SETTER | VOID_FUNCTION_ACTION | VARIABLE_SETTER | SYSTEM_ACTION
- ATTRIBUTE_SETTER ::= "**set**" ATTRIBUTE_EXPRESSION "**to**" ATTRIBUTE_VALUE
- VARIABLE_SETTER ::= "**set**" VAR_NAME "**to**" VAR_VALUE
- FOR_EACH_COMPOUND_ACTION ::= "**for each**" OBJ_TYPE ["**called**" VAR_NAME ","] **in** COLLECTION ":" "-" SIMPLE_ACTION ("-" SIMPLE_ACTION)*

The previous rule examples illustrated ATTRIBUTE_SETTER (e.g., "`set the decision of '`**`current claim`**`' to` **`"INELIGIBLE"`**;"), SYSTEM_ACTION (e.g., "***`print`*** **`"illustrating conditions"`**;"), and VOID_FUNCTION_ACTION (e.g., "`log that this rule has fired on` **`'current claim'`** `with message` **`"Claim filed too late";`**"). The next rule illustrates the compound statement.

```
definitions
    set 'current claim' to a claim where the decision of this claim is "ELIGIBLE";
    set 'service acts' to the service acts of 'current claim' ;
then
    for each service act  called 'my service act' , in 'service acts'  :
    - set the payment of 'my service act' to the cost of 'my service act'
    - print "the procedure" + the procedure of 'my service act' + " was paid in full" ;
    set the decision of 'current claim' to "PAID";
    print "illustrating for each" ;
```

and the (simplified) IRL equivalent:

```
when {
  current_claim: Claim(?this.decision.equals( "ELIGIBLE" ));
  service_acts: collect ServiceAct() in
                          current_claim.serviceActList;}
then {
  foreach (ServiceAct my_service_act in service_acts) {
    my_service_act.payment = my_service_act.cost;
    printMessage("the procedure"+ my_service_act.procedure +
                                  "was paid in full"));
  }
  current_claim.decision = "PAID";
  update current_claim;
  printMessage("illustrating for each");
 }
```

The BAL is used in many other places, besides *action rules*. It is used to write preconditions, condition and action columns in decision tables (to be discussed next), preconditions, node conditions and actions in decision trees (Sect. 11.2.4), as well as in many places in ruleflows (function tasks, initial and final actions in all ruleflow tasks, transition guards, and rule selection, see Sect. 11.3.2).

11.2.3 Decision Tables

As mentioned in Sect. 9.2.2.2, when we have several rules whose conditions test on the same set of attributes and whose actions perform the same actions (modulo some parameter values), it pays to organize those rules in a *decision table*, both during rule analysis (see Chap. 4) and during rule authoring. JRules supports decision tables, like most BRMS. Figure 11.2 shows a decision table that sets the parameters of a coverage (deductible and yearly cap), based on the procedure covered, and on the type of policy (individual versus group policy). This table has four columns, two *conditions columns*, labeled "Covered Procedure" and "Policy Type", and two *action columns*, labeled "Deductible" and "Yearly Cap". Each line of the table corresponds to a rule. Here, we selected the line number 6, which corresponds to the case where the procedure is ECG (Electro-CardioGram), the policy type is "GROUP". In this case, the deductible is $20, and the yearly cap is $125 (per insured). As shown in the screenshot, by selecting a particular row of the

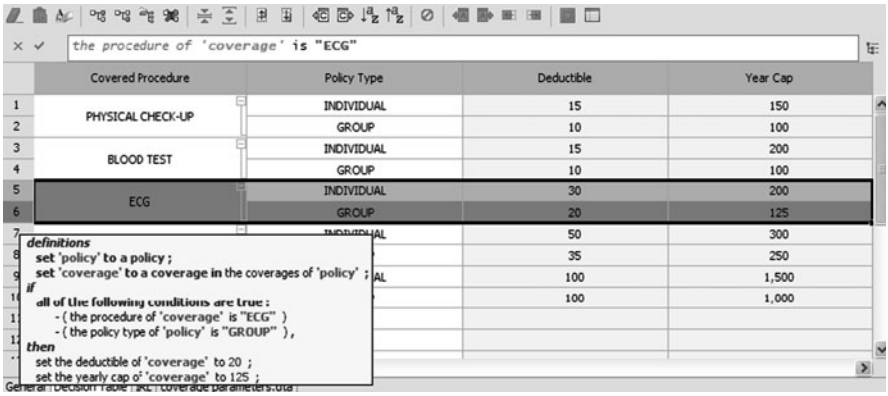


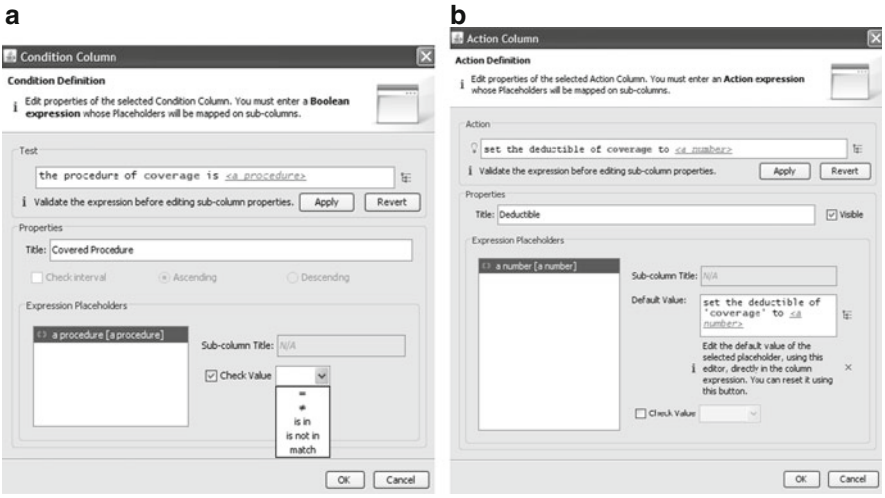
Fig. 11.2 A sample decision table

decision table, Rule Studio brings up a tool tip consisting of the BAL equivalent of the rule represented by that row.

Let us first get some vocabulary. Notice that for each value of "Covered Procedure", we have two possible values of "Policy Type". Each value of "Policy Type" is considered as a *branch* of the corresponding value of "Covered Procedure". The set of covered procedures {"PHYSICAL CHECK-UP", "BLOOD TEST", ECG, "X-RAY", and "CAT SCAN"} is said to represent a *partition* of the domain of procedures. Similarly, the set {INDIVIDUAL, GROUP} is said to represent a *partition* of the domain of the attribute "policyType" of the class Coverage. This table is called *symmetrical* because the same *partition* of "Policy Type" is used for all the values of "Covered Procedure". We now show how the table is defined.

Figure 11.3a shows the wizard for defining condition columns. A condition column enables us to enter a Boolean condition similar to the kinds of conditions we enter in a BAL action rule. The condition should be fully specified except for one (or several) value(s), which needs to be specified in the cells of the columns. In this case, the condition is "`the procedure of` **`coverage`** `is <a procedure>`", and we need only specify `<a procedure>` in the cells of the column. The column has an editable title. We can also specify conditions that column cell values must satisfy – in addition to being of the appropriate type, which is guaranteed by the table editor. Similarly, for the second condition column, the test is "`the policy type of` **`policy`** `is <a policy type>`."

Figure 11.3b shows the wizard for defining action columns. The action corresponds to any valid action we can insert in the action part of a rule (see previous section), with the exception of `FOR_EACH_COMPOUND_ACTION` (see previous section). Depending on the parameters of the action, the action column can have two or more subcolumns. In this case, the action is a setter that takes a single value. The second action column is defined in a similar way: The action is "`set the yearly cap of` **`coverage`** `to <a number>`." Notice that we can specify a *default value* for an action parameter. We can also specify additional



Wizard for defining a condition column

Wizard for defining an action column

Fig. 11.3 Wizards for defining condition and action columns. (**a**) Wizard for defining a condition column and (**b**) wizard for defining an action column

constraints on the cell values, in a way similar to values in condition columns. Finally, we can make an action column invisible.[12]

The reader may wonder where the variables that are referenced in the condition and action columns (**`coverage`** and **`policy`**) come from. The answer is: *preconditions* of the table. Generally speaking, preconditions are used to define variables and to enter conditions that hold true for all of the columns of the table. Figure 11.4 shows a screenshot of the *tab* used to define preconditions.

We will not discuss all of the wizardry of the decision table editor. However, a few features are worth mentioning:

- JRules performs different kinds of verifications and analyses on decision tables, and the results of these analyses can be presented as "Info", "Warning", or "Error":
  - *Symmetry*. A table is said to be symmetrical if for each condition column *i*, the same *partition* of values for that column is used consistently for all the values of column *i-1*. As mentioned above, the table shown in Fig. 11.2 is symmetrical because the same partition {"INDIVIDUAL", "GROUP"} for "Policy Type" is used for the values of "Covered Procedure."
  - *Overlap*. This refers to the case where the values "within a partition" overlap.[13] In the table of Fig. 11.2, we would have had an overlap if, for

[12]This is useful in those situations where (a) all the cells have the same value, or (b) the action takes no arguments – and thus no values to enter – or (c) the action represents a non-business tasks that business rule authors should not care about.

[13]This is a misnomer because mathematically speaking, the *partition* of a set *S* is a set of *non-overlapping* subsets of *S* whose union equals *S*.

Fig. 11.4 Defining preconditions for a table

example, row 5 had both "INDIVIDUAL" and "GROUP", and row 6 had "GROUP". If that were the case, when procedure="ECG" and policy type = "GROUP", we would hit both rows 5 and 6 of the table. This is not a *logical* error, but more often than not, overlaps result from data entry errors.
  - *Gaps*. Gaps are best illustrated with a numerical (range) condition column (not used here). Assume that we have a condition column based on age ranges. If our condition had age ranges [0,18], [18,30], and [65,100], then one might wonder about the age range [30,65]: What happens in such situations? Again, this is not a *definite logical* error per se, but, more often than not, indicative of a data entry error.
- JRules can enforce *locking* different aspects of the table:
  - *Preconditions*. Making sure that the preconditions part is not editable.
  - *Number of* columns. We can prevent the addition and removal of condition columns, action columns, or both.
  - *Condition column contents*. For each condition column, we can selectively lock the *tests* (preventing users from changing the column test, or column cell overrides), the *partitions*, i.e., how many different values in the partition, or the *values* themselves. In our case, if we lock the *partition* of column "Policy Type", it means that we can have *exactly* two cells/branches for every procedure, but the table author can select which values to enter in each cell. If we lock the *values* for the column, it means that values themselves are not editable, i.e., nothing about the column can be changed.
  - *Action column contents*. With action columns, we can lock the *action*, the *status*, or the *values*.
- JRules supports the graphical customization of the table. Indeed, we can change the background color, text color, text font, text style, and text size, for column headers, condition columns, and action columns (separately).

JRules supports a bunch of other features for data entry (e.g., splitting cells, merging cells, inserting the values of a *BOM domain*, etc.) that make life easier for table authors.

Looking under the hood, decision tables are actually encoded as ... a bunch of IRL rules, one per row! The following shows the beginning of the IRL file for the table of Fig. 11.2. The rule `coverage_parameters_0` represents the first row of the table, i.e., row 0. There are 10 such rules, each one corresponding to a different combination of procedure and policyType values.

```
// begin DT coverage parameters
// -- begin rule 'coverage parameters 0'
rule coverage_parameters_0 {
    property ilog.rules.dt = "coverage parameters";
    property ilog.rules.group = "coverage_parameters";
    property status = "new";
    when {
       policy: com.mywebinsurance.claimprocessing.Policy();
       coverage: com.mywebinsurance.claimprocessing.Coverage()
in policy.coverageList;
       evaluate ((((coverage.procedure.equals("PHYSICAL CHECK-
UP"))) && ((policy.policyType.equals("INDIVIDUAL")))));
    } then {
       coverage.deductible = 15;
       coverage.yearlyCap = 150;
    }
}

// -- end rule 'coverage parameters 0'
// -- begin rule 'coverage parameters 1'
rule coverage_parameters_1 {
…
```

At first glance, this may not sound like the most efficient implementation. Indeed, the table format "leads us to believe" that conditions are shared between different rows and the tests are performed only once. For example, the first and second rows of the table share the same value for procedure, i.e., "PHYSICAL CHECK-UP", but if each row is represented by a separate rule, then we lose the condition sharing. Actually, not so! Recall from Chap. 6 that the RETE algorithm ensures that if two rules start with the same condition, that condition will be shared and it will be evaluated only once for both of them. Hence, once the ruleset containing this table is parsed and the RETE network is built from it, conditions will be shared, as suggested by the table.

As mentioned in Chap. 9, JRules provides API for creating decision tables and decision trees – to be discussed next – from tabular data,[14] including csv (comma-separated values) files, Excel spreadsheets, and relational data bases. A few years back (2005), in one project for a Wall Street financial services company, we used decision tables to encode rules that figure out which kinds of financial transactions for specific types of foreign customers were subject to IRS reporting and withholding.[15] Our input from "business" was a bunch of Excel spreadsheets

[14]Check root package ilog.rules.dt, and more specifically, ilog.rules.dt.model.

[15]The Internal Revue Service expects all US entities (corporations, individuals) to file for taxes every year, and it has the necessary authority and . . . hum . . . leverage to make sure they comply. With foreign entities, because it lacks such "leverage", it requires that a percentage of their gains on *each transaction* be preemptively withheld (typically 30%, but sometimes 15% or 10%) or reported . . . unless, of course . . . (a few hundred rules and exceptions based on type of entity, country of origin, existence of treaties, type of transaction, etc.).

prepared by tax accountants. After a minor clean-up, we were actually able to load up the spreadsheets using the API, saving countless hours of data entry, but more importantly, getting rid of a major source of errors. Since JRules 7.x, there is out of the box functionality (*Rule Solutions for Office*) to *export* decision tables from Rule Team Server (RTS) as Excel 2007 spreadsheets, and to *import* them back after editing.

11.2.4 Decision Trees

The same kinds of situations that call for decision tables can also be handled by decision trees. As mentioned in Chap. 9, you may find that business people actually think in terms of decision trees, but encode the decision tree in the form of a table. With JRules decision trees, they can encode them the way they see them. However, there are situations where decision tables would be too rigid. Going back to our example decision table, it may the case that, (a) for *some* procedures, we actually do not care about the type of policy as the same deductible and yearly cap apply whether the policy is INDIVIDUAL or GROUP, and (b) for some others, the deductible and yearly cap do not only depend on the type of policy, but also depends on the number of insured. To encode such a situation with a decision table, we will need three condition columns but some columns will have empty values. Here, a *decision tree* comes in handy as different *branches* of the three can have different *tests* and different *depths*. Further, a decision tree allows different rule/action nodes to have different sets of actions; doing the same with decision tables would require some acrobatics. Figure 11.5 shows such a decision tree where we have branches of depth 1, 2, and 3. Each leaf node (box) represents the action part of a rule whose condition part consists of the path leading to that node. As was the case with decision tables, behind the scenes, decision trees are actually encoded as

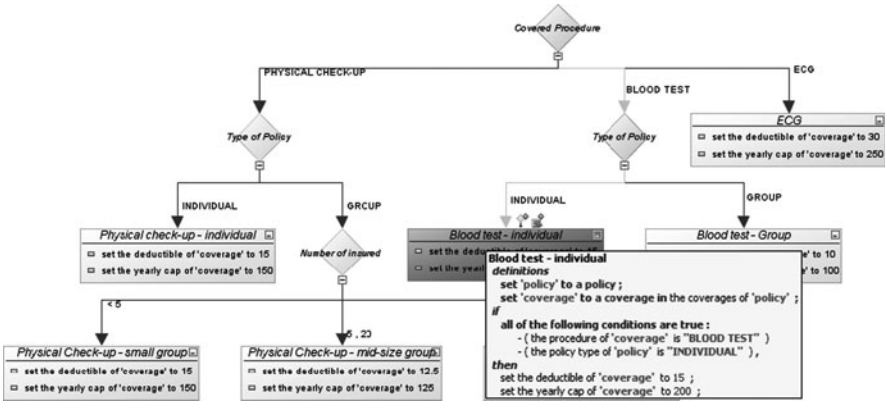


Fig. 11.5 A sample decision tree for computing coverage parameters

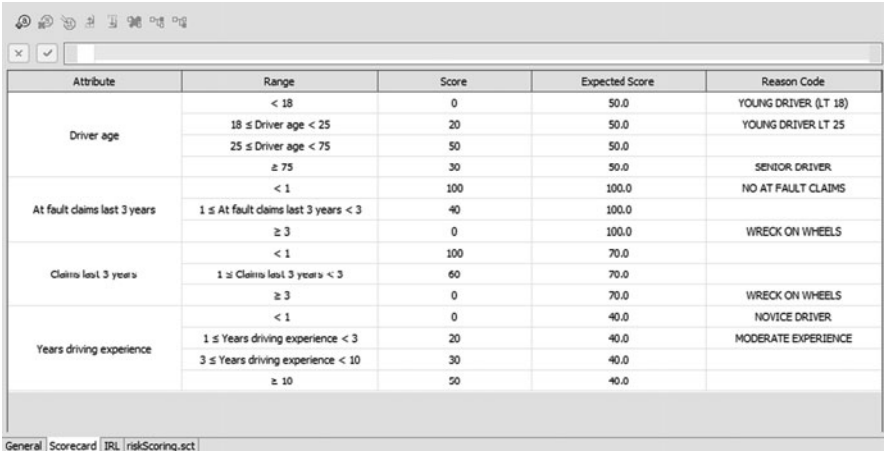
separate IRL rules. With regard to condition sharing, as explained for the case of decision tables, because of the structure of the RETE network (see Chap. 6), common conditions will indeed be shared between the different rules. In fact, the encoding of the rules of the decision tree in the RETE network mirrors the decision tree!

As is the case with decision tables, JRules enables us to check for gaps and overlaps between the different branches of the tree. In terms of GUI wizardry to create decision trees, the reader is referred to the product documentation. We will mention, however, that the decision tree editor enables us to fold rule nodes or entire tree branches, and to turn the tree sideways, with the root on the left side of the panel, and the branches going rightward. Both of these tricks enable us to somehow manage the expansive nature of decision trees: more often than not, the decision table format is much more compact than the decision tree format. However, business users love the visuals: They fit nicely in PowerPoint presentations☺.

11.2.5 Score Cards

In the insurance and financial services sector, an important rule-rich business process is *underwriting*. Simply speaking, underwriting consists of assessing the eligibility of an actual or potential customer to receive a product or service. An important aspect of underwriting is *risk scoring*. You use risk scoring when the underwriting is a multi-criteria decision – as it often is. In such a case, no single criterion is eliminatory, but the accumulation of factors, positive or negative, can tip the balance one way or the other. With risk scoring, you assign a single score to the (potential) customer based on a set of criteria. If the score falls below a certain threshold, the product or service is denied. Else, it is granted.

JRules supports *scorecards* through a product add-on called *Scorecard Modeler*. Figure 11.6 shows an example of a scorecard for *policy underwriting*. In this fictitious example, we assign a score to a (potential) policy holder based on four criteria: (a) the age, (b) the number of claims when the policy holder was at-fault in the past 3 years, (c) the number of claims of the policy holder in the past 3 years, *regardless* of responsibility, and (d) the number of years of driving experience. The higher the score, the better (i.e., lower) the risk. The first three columns assign the score per se. The last two are used to customize what is called *reasoning strategy*, to be discussed below. For the driver's age, we assign different scores to different age ranges: the highest the risk, the lower the score. Drivers between the ages of 25 and 75 are considered to possess the best mix of qualities (e.g., sobriety, reflexes). With regard to the number of claims, at-fault or in general, the smaller the number of claims, the higher the score. With regard to the driving experience, the longer the experience, the higher the score. With this scorecard, a driver who is 30 years old, with one at-fault claim and one not at-fault claim, and 11 years of driving experience,



Attribute	Range	Score	Expected Score	Reason Code
Driver age	< 18	0	50.0	YOUNG DRIVER (LT 18)
	18 ≤ Driver age < 25	20	50.0	YOUNG DRIVER LT 25
	25 ≤ Driver age < 75	50	50.0	
	≥ 75	30	50.0	SENIOR DRIVER
At fault claims last 3 years	< 1	100	100.0	NO AT FAULT CLAIMS
	1 ≤ At fault claims last 3 years < 3	40	100.0	
	≥ 3	0	100.0	WRECK ON WHEELS
Claims last 3 years	< 1	100	70.0	
	1 ≤ Claims last 3 years < 3	60	70.0	
	≥ 3	0	70.0	WRECK ON WHEELS
Years driving experience	< 1	0	40.0	NOVICE DRIVER
	1 ≤ Years driving experience < 3	20	40.0	MODERATE EXPERIENCE
	3 ≤ Years driving experience < 10	30	40.0	
	≥ 10	50	40.0	

General | Scorecard | IRL | riskScoring.sct

Fig. 11.6 A sample scorecard

would get a total score of: 50 (for age) + 40 (at-fault claims) + 60 (claims in general) + 50 (driving experience), for a total of 200. A 23-year-old driver, with no claims, and 5 years of driving experience would get: 20 (for age) + 100 (at-fault claims) + 100 (claims) + 30 (driving experience), for a total of 250.

Which driver to accept (or reject), if any? As mentioned in Chap. 9, all of the parameters of the scorecard, including which attributes to use for scoring, how many ranges to use for each attribute, what are the bounds for each range, what score to use for each range, how to compute the overall score (simple sum versus weighted sum), and *what the decision threshold should be*, are determined by statistical models.[16] For example, MyWebInsurance may have a policy to underwrite only those drivers who have less than 5% chance of making a claim worth more than $5,000 within the first year. Statistical score models will tell, among the many things mentioned above, what the threshold score should be.

With regard to the *reasoning*, Scorecards make it possible to not only return an overall score, but to also return *reason codes* to explain a particular score. Scorecard Modeler maintains lists of reason codes that can be used within a particular scorecard. In the most trivial approach, we could use one reason code *per row* of the scorecard, and ask that *all* reason codes be returned. Most business applications do not care about *that* level of precision. Instead, business may find it useful to identify those attributes that have unusually (or damningly) low scores.

Scorecard Modeler offers a number of "knobs" to tune the *reasoning strategy*: We can specify a maximum number of reason codes, and doing so, we need to specify criteria for determining which reason codes to return in case we have

[16]As is the case with statistical models, it is part science (mostly), part art. Note that the JRules Scorecard Modeler does not support those statistical analyses: They need to be done using other tools such as the SAS Enterprise Miner™.

more candidates than the maximum, and how to order them. Possible criteria for figuring out which reason codes to include: (a) reason code priority (they have one), (b) deviation relative to maximum score, (c) deviation based on expected score, or (d) custom reasoning strategy. For deviation, we can take positive deviation, or negative deviation or both. In the example of Fig. 11.6, we used *negative deviation* relative to *expected score*, meaning that we return reason codes for the attributes ranges that are farthest below the expected score. The fourth column of the scorecard shows the expected score. With regard to the ordering, we can start with reason codes corresponding to the highest deviation (i.e., worst outliers) or smallest. There are also rules for handling duplicates. And so forth.

If we look under the hood of a Scorecard, we find four IRL rules, one per attribute, that look like the rule below.[17] This rule, which is not meant for human consumption, sets the reasoning parameters in the action part, and assigned scores for the different ranges of the attribute in the action part using "**if**" statements.

```
// -- begin rule 'riskScoring_1'
rule riskScoring_1 {
  property ilog.rules.group = "riskScoring";
  property status = "new";
   when {
        scorecard: Scorecard() from riskScoring;
        PolicyHolder() from theClaim.policy.policyHolder;
        evaluate (scorecard.rejection == null);
  } then {
        scorecard.name = "riskScoring";
        scorecard.scoringStrategy = "Sum";
        scorecard.reasoningStrategy = "Deviation based on ex-
pected score";
        scorecard.reasonOrderBy = "Descending deviation";
        scorecard.reasonFilterBy = "Negative deviation";
        // -- other reasoning strategy parameters
        …
        if(theClaim.policy.policyHolder.numberAtFaultClaimsLas
tThreeYears < 1) {
         scorecard.setScore("numberAtFaultClaimsLastThree-
                                          Years", 100);
         scorecard.setReasonCode("numberAtFaultClaimsLast-
               ThreeYears","NO AT FAULT CLAIMS");
        }
        if(theClaim.policy.policyHolder.numberAtFaultClaimsLas
tThreeYears in [1, 3[) {
         scorecard.setScore("numberAtFaultClaimsLastThree-
                                          Years", 40);
       }
       …
```

[17] We greatly simplified the actual IRL to make it readable. The actual IRL has more actions, and some of the functions have more parameters.

The business logic implemented by Scorecard can easily be implemented by individual (business-friendly) BAL action rules, one per attribute, per range, such as the following:

```
definitions
    set 'my policy' to a policy ;
    set 'my policy holder' to the policy holder of 'my policy' ;
if
    the number at fault claims in the last three years of 'my policy holder' is 0
then
    add 100 to the score of 'my policy' ;
    add the reason code "NO AT FAULT CLAIMS" to 'my policy' ;
```

We could also use one decision table per attribute. The scorecard solution has the advantage of conveniently grouping the various scoring rules into one place, and presenting them in a visually intuitive/appealing fashion. It also enables us to conveniently customize the scoring calculation and manipulate reason codes.

11.2.6 The Business Rules Language Development Framework

In Chap. 4, we presented different classifications of rules. Not all classes of rules can be *conveniently* written as **if-then** rules. While we can turn every type of rule into an **if-then** rule, there are situations where a custom language can make rule authoring more familiar to the business users. Let us revisit the example we mentioned in Sect. 9.2.2.5. Assume that you are building an application for filling out tax returns. The majority of tax rules are computations. Using the rule templates described in Sect. 4.1, a computation may be stated as:

The taxable income IS-COMPUTED-AS gross income + commission–s deductions

It would be convenient to be able to enter such a rule as is within a rule editor. In this case, the rule editor would be a *formula editor* similar to the formula editor available in Excel spreadsheets. This would be more natural than entering the rule as:

if <some trivial condition or no condition> **then** taxPayer.taxableIncome = taxPayer.grossIncome + taxPayer.commission–s taxPayer.deductions

or its business-oriented language equivalent.

JRules offers a rule language development framework called the *Business Rule Language Development Framework* (BRLDF), which is a java framework for specifying the syntax of the custom rule language, and for translating rules written in this syntax to some target language. If the target language is JRules IRL, then we can reuse the entire rule development and execution infrastructure for our new language. Figure 11.7 illustrates the approach.

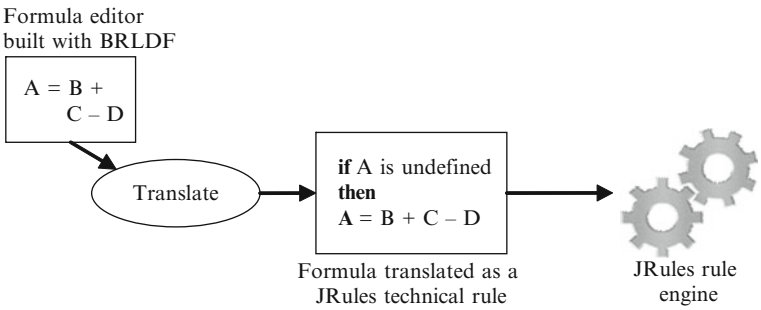


Fig. 11.7 Developing a custom rule language

In Chap. 9, we discussed situations under which it is justifiable to build a custom rule language. In this section, we provide a high-level description of how to do it with the JRules BRLDF. In the BRLDF, a rule language is defined by three components:

1. An *abstract syntax*. This syntax defines the structure of the language in a notation similar to the EBNF-like notation we used to describe the IRL and the BAL. In this case, the syntax is defined in an XML schema. We show below excerpts of the abstract syntax for our formula editor. The types prefixed with namespace "brl" are ones reused from the definition of the BAL.

```
<complexType name="T-equation">
  <sequence>
    <element name="left-hand-var" type="brl:T-local-var"/>
    <element name="right-hand-side"type="T-formula"/>
  </sequence>
</complexType>
<complexType name="T-formula">
  <choice>
    <element name="value" type="brl:T-local-var"/>
    <element name="expression" type="T-operation"/>
  </choice>
</complexType>
<complexType name="T-operation">
  <sequence>
    <element name="left-op" type="T-local-var"/>
    <element name="operator" type="brl:T-operator"/>
    <element name="right-op" type="T-expression"/>
  </sequence>
</complexType>
# etc
```

2. A *concrete syntax*. This syntax defines the *graphical properties* of the constructs of our language. This is where we specify the textual patterns (actual *tokens*), the text styles, the tool tips, prompts, whether there is a newline after a particular element, etc. We also specify the classes that process our language (see the next element). The concrete syntax is given in a *properties* file format. We illustrate the format in the excerpts below.

```
# Define the text pattern of 'if-then-else'
<T-equation>.text = <left-hand-side>=<right-hand-side>
<T-equation>.style = keyword
# Specify the parser/translator (see below) IRL
<T-equation>.translatorClass = MyTranslator
<T-equation>.codeGeneratorExtender.irl = MyCodeGenerator
# Specify graphical properties of elements
<T-equation>.<left-hand-var>.toolTip = Pick a variable
<T-equation>.<left-hand-var>.label = left-hand-side
…
```

3. *Parsers/translators*, which parse sentences of our language into an intermediate form and then translate/generate a target language. These are Java classes built using the *parsing and translation framework* that is part of the BRLDF. As mentioned above, to be able to reuse the rule deployment and execution infrastructure, it is a good idea to translate rules written using our custom language into IRL. In the above example, the class **`MyTranslator`** parses rules and generates the abstract syntax tree, whereas the class **`MyCodeGenerator`** reads such a tree and generates IRL.

Having defined the language, we now need to integrate it into the authoring environments, i.e., Rule Studio and Rule Team Server. This, in turn, involves three things:

1. Defining a new *rule class* that represents the new type of rules within these development environments. This is the class that defines which properties such rules can have. This is done through the *rule extension model*, in both RS and RTS.
2. Making sure that the language definition is available to the environment: that includes the files used to define the abstract syntax and the concrete syntax, and the Java classes that implement the parser and IRL translator. In RS, this is done through a specific plug-in.[18] In RTS, the whole thing is packaged in a jar file, and the RTS archive is repackaged to include the language jar file.
3. Customize the rule editors (Guided Editor and Intellirule) to handle the new language. Luckily, the rule editors are *parameterized* by the rule language, and thus, not much needs to be done for the customization.

Notice that the BAL language *itself* is developed using the BRLDF. In fact, the files that contain the abstract syntax and the concrete syntax are public and editable. Further, the parsers and translators for the BAL are part of the public API. This means three things:

1. If all you need is to change the ordering of BAL constructs or some of keywords, then you could simply edit the abstract syntax and concrete syntax files, with no programming involved.

[18] If you must know, we need to create an Eclipse plug-in project using the extension point `ilog.rules.studio.brl.languages`.

2. If you need to add a new kind of definition, condition, or action, then all you need to do is to define the abstract and concrete syntax for the new construct in the corresponding files, and code the corresponding parser and translator for abstract syntax tree nodes that represent the new construct.
3. If your language is too different from the BAL, you could still reuse many of the artifacts used to build the BAL, which have been conveniently modularized: (a) a component that handles bindings (variable definitions), (b) a component that handles conditions, and (c) a component that handles actions.

In our experience, developing a custom rule language is *rarely* justified, in terms of business need, development cost, and maintenance risk. Luckily, thanks to the incremental approach of the BRLDF, we can often address the most pressing BAL irritants/shortcomings using low-cost, low-impact modifications of the BAL.

11.3 Rule Execution Orchestration

In Chap. 6, we presented the principles behind rule engines and rule engine execution. Recall from Chap. 6 that an engine maintains three memory areas: (a) a *ruleset* consisting of a set of rules that embody a computation, decision, or action that the engine implements, (b) a *working memory*, containing (or referring to) the objects that the ruleset will be applied to, and (c) an *agenda* that maintains a list of so-called *rule instances*, which are candidate rules for firing. We saw earlier in this chapter the development infrastructure in JRules, namely, rule projects and the BOM, and we just covered the different rule artifacts.

What we know from Chap. 5 (prototyping) and Chap. 8 (an introduction to JRules) is that, simply speaking, a rule project – a development artifact – is mapped to a *ruleset* – a run-time concept. What we know from Chap. 6 is that the rules of a ruleset are treated as an indiscriminate "bag of rules" where all of the rules are evaluated on all of the objects in working memory with no underlying structuring or sequencing, except for the ordering on the agenda. This leaves a couple of key questions to address:

1. How to get data into the rule engine – and its working memory – in the first place, especially within the context of a *rule execution service*, as we discussed in Sect. 7.5.1. This introduces the notion of a *ruleset signature*, and more specifically, the notion of *ruleset parameters*.
2. How to structure the execution of rules within a ruleset. While each ruleset is meant to implement a single business decision, such a decision will typically be broken down into a set of sub-decisions that need to be executed in a particular sequence. In fact, the structure of these sub-decisions may be a guiding principle in the organization of rules *during* development, as illustrated in Sects. 7.4.2 and 7.4.3. This is the notion of *ruleflow* that we hinted at in many places in the previous chapters (Chaps. 5, 6, 7, and 8).

We can think of these two aspects as the *execution infrastructure* of a rule project.

In this section, we address these two aspects in detail. First, we talk about ruleset parameters: What they are, how to create them, and how to use them, both inside and outside the rule engine. Incidentally, we will also talk about *ruleset variables*. Section 11.5.2 introduces the basics of ruleflows: what they are, and how to create them. Section 11.5.3 will discuss advanced ruleflow concepts, namely, run-time selection of the contents of rule tasks, and algorithm selection.

11.3.1 Ruleset Parameters and Variables

If we think of ruleset as a *function*, ruleset parameters are parameters of that function:

1. They have a name and a type.
2. They have a direction: *in*, *out*, or *inout*.
3. They are *visible* anywhere within the "function", and can be referenced by name.
4. Their lifetime depends on the calling scope.

While this is a fairly accurate analogy, some qualifications are in order. Of course, in Java, methods have a single *out* parameter, which is the return value,[19] and all of their parameters are *inout* because Java passes variables by reference. With rulesets, the distinction between *in* and *inout* parameters makes sense within the context of a *remote invocation* of the rule engine. Let us first show how to define ruleset parameters, and then show how they can be referenced *within* the rules of a project, at rule development time, and by the rule engine calling application, at rule execution time.

To the extent that rule projects map to rulesets, ruleset parameters are defined at the *rule project* level. Figure 11.8 shows a screenshot of the wizard for defining

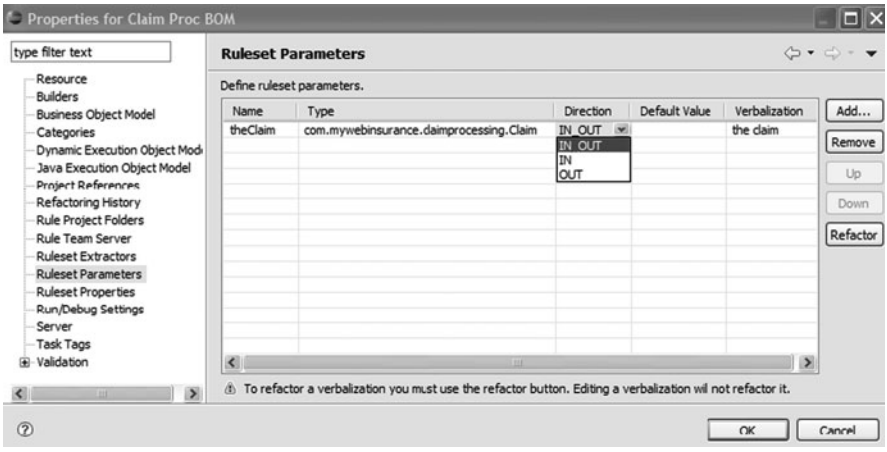


Fig. 11.8 Wizard for defining ruleset parameters

[19]Of course, we *could* have several out parameters in Java ... if we aggregate them in a single return object.

ruleset parameters. In this case, we have a single parameter, which is the Claim object, and it is inout. Incidentally, that is the only *in* object we need because it is the root of an object hierarchy that contains the various service acts, and the policy, which in turn points to the policy holder, its coverages, etc. It is the only *out* object we need because the decision and the total payment are stored in the claim object itself, and the itemized payment amounts are stored in the ServiceAct objects. Note that a ruleset parameter has a *verbalization* which enables us to refer to it in rules.

First, we look at how data is passed to the engine using ruleset parameters, as opposed to through the working memory; the full API will be discussed in Chap. 13. The following shows how data is passed through insertion into working memory, and how the result of rule execution is retrieved.

```
// initialize the rule engine (load and compile ruleset:
// see chapter 12 for details
IlrContext myEngine = …;
// get next claim object and insert into working memory
Claim myClaim = fetchNextClaim();
myEngine.insert(myClaim);
// Execute the rules. See chapter 6 for details
myEngine.execute();
// Check the outcome by examining the decision attribute
String decision = myClaim.getDecision();
if (“PAID”.equals(decision))
  System.out.println(“The Claim “+myClaim+” was paid in
the amount “ + myClaim.getTotalPaid());
```

Now with the ruleset parameter:

```
// initialize the rule engine (same as above)
IlrContext myEngine = …;
// get next claim object and pass as parameter value
Claim myClaim = fetchNextClaim();
IlrParameterMap inputs = new IlrParameterMap();
inputs.setPatameterValue(“theClaim”,myClaim);
myEngine.setParameters(inputs);
// Execute the rules, and collect the inout/out params
IlrParameterMap outputs = myEngine.execute();
Claim modClaim =(Claim)outputs.getObjectValue(“theClaim”);
// Check the outcome by examining the decision attribute
String decision = modClaim.getDecision();
if (“PAID”.equals(decision))
  System.out.println(“The Claim “+modClaim+” was paid in
the amount “ + modClaim.getTotalPaid());
```

There are a couple of subtle differences between the two “data passing” modes. In the first case, the calling application relies on the fact that the engine lives in the same JVM, and hence the variable `myClaim` stays current: Upon returning from

the call "`myEngine.execute();`" the variable `myClaim` will reflect whichever changes were made by the rules. With the parameters, the calling application does *not* rely on the fact that the engine lives in the same JVM, and will retrieve the modified value of `myClaim` into a separate variable `modClaim`. This makes the second approach more scalable in the sense of being *remotable*. In fact, the Rule Execution Server (RES) API, to be discussed in Chap. 13, relies on ruleset parameters to pass data back and forth.[20]

We now look at how ruleset parameters are referenced in rules. As mentioned above, ruleset parameters are visible within all the rule of the project, and can thus be referenced within rules. Going back to our "claim date" BAL rule from Sect. 11.2.2, we can now write the rule without a ***definitions*** part:

```
if
    'the claim' was filed more than 90 days after the start date of the policy of 'the claim'
then
    set the decision of 'the claim' to "INELIGIBLE" ;
    log that this rule has fired on 'the claim' with message "Claim filed too late" ;
```

And if we look at the IRL:

```
rule claim_date {
 property status = "new";
 when {
    com.mywebinsurance.claimprocessing.Claim() from theClaim;
    evaluate (theClaim.fileMoreThanXDaysAfter(90,
                    theClaim.policy.startDate.toDate()));
 } then {
   theClaim.decision = "INELIGIBLE";
   ?context.updateContext();
   theClaim.logRuleFiringWithMessage("Claim filed too late");
 }
}
```

If we compare this IRL to that produced for the original rule (Sect. 11.2.2), we see a couple of differences:

1. In the earlier rule (Sect. 11.2.2), we looked for the claim object in working memory, whereas, here, the claim object is *scoped* within the ruleset parameter.
2. If we look at the action part, the rule in Sect. 11.2.2 includes an update on the claim object, namely: '`update current_claim;`' whereas the above rule does an update on the rule engine itself ("`?context.updateContext()`").

[20]This is a somewhat abusive simplification, but it will do for now: (1) the API for manipulating XML data (XML XOM) is slightly different, for both working memory insertion, and parameter passing, (2) with *inout* parameters, for the case of local invocation (same JVM, as in the example above) the variable passed as inout *will* reflect the changes made by rule execution (no need to fetch the new value from outputs), and (3) the RES API does enable us to pass data that is to be inserted in working memory.

Recall that '**update** some_object;' causes the rule engine to reevaluate all of the rules relevant to some_object, which is the mechanism that underlies rule chaining. However, what if the object is *not* in working memory, as is the case with ruleset parameters? Technically, ruleset parameters are treated as data members of the rule engine object itself[21] and thus, whenever a ruleset parameter is modified, the BAL to IRL translator throws in '?context.update Context()' whose effect is to reevaluate all of the rules that concern ... the ruleset parameters!

Finally, note that passing a data object as a ruleset parameter does *not* insert it into working memory. Thus, the original form of the rule "claim date", where we defined a rule variable in the ***definitions*** part, would not work. Indeed, the class condition:

```
…
current_claim: Claim();
…
```

Would fail because there would *not* be any **Claim** object in working memory! So what do we do? We have three alternatives:

1. Rewrite the rule ... naah!
2. Find a way of inserting ruleset parameters into working memory so that "working memory-style" rules continue to work. There are several more or less elegant techniques of doing this that *do not* involve the Java code; we will see one such technique when we talk about ruleflows (Sects. 11.5.2 and 11.5.3).
3. Design the signature of the ruleset (i.e., the ruleset parameters) *before* we start writing rules, and then write rules that refer to those parameters. This is the recommended practice. We will come back to this and other practices in section on "Further Readings".

We now talk about *ruleset variables*. If *ruleset parameters* are to *rulesets* what *function parameters* are to *functions*, then *ruleset variables* are to rulesets what *function-scope local variables* are to functions: they are visible everywhere in the ruleset/rule project, and they *keep their values* during the invocation of the ruleset; we could not say "their lifetime spans a ruleset invocation", because ruleset variables actually survive a ruleset invocation, and even keep their values from one invocation t o the next... unless we clean them using "myEngine.cleanRuleset Variables();", or the more drastic "myEngine.reset();".

Like with ruleset parameters, ruleset variables can be referenced in both rule conditions and rule actions. We typically use ruleset variables to hold intermediary results of the "reasoning" of the rule engine that we wish to pass from one rule to another, with no other place to store them. A fairly common use is to implement routing logic with ruleflows, to be discussed next. For the time being, we simply show the mechanics of defining ruleset variables. Figure 11.9 shows the wizard for

[21] And in a distant past (Rules C++), they were.

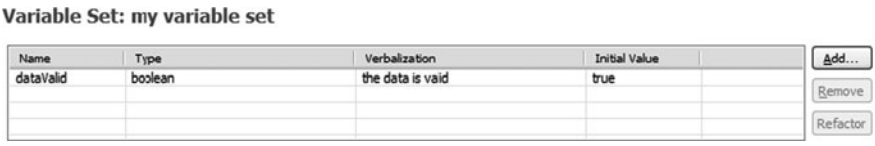


Fig. 11.9 Wizard for defining ruleset variables

defining ruleset variables. Ruleset variables are defined through *variable sets*. We can have several variables sets within the same project, but only one per package. However, the variables are accessible in all the packages of the project.

11.3.2 Ruleflows: Basics

A ruleflow is a way of organizing the execution of the rules of a rule project/ruleset in terms of groups of related rules. It is a *process* flow whose *tasks* consist – *mostly* – of the execution of groups of related rules. As mentioned in the introduction of this section, while a ruleset is meant to implement a single business decision, that decision is typically complex enough that it can be broken down into more elementary decisions. This breakdown was actually presented as one of the criteria for organizing rules during development (Sects. 7.4.2 and 7.4.3). The basic idea is that the rules of a ruleset are broken down into subsets distributed among the tasks of the ruleflow, and they will be evaluated in the sequence embodied in the ruleflow. Another way of putting it: the ruleflow becomes sort of the "main program" of the ruleset.

Figure 11.10 shows the different types of components of a ruleflow. Much like a Java function, a ruleflow has a single entry point and one or more end points. There are three types of tasks in a ruleflow:

1. *Function tasks*, which include some *imperative* IRL or BAL code to execute, i.e., the kind of code we would find in the action part of a rule (IRL or BAL) or in IRL functions. Function tasks are typically used as the starting tasks of a ruleflow to perform required initializations. For example, we could use a function task to insert ruleset parameters in working memory!
2. *(Simple) rule tasks*, which contain a bunch of rules and rule packages that will be evaluated – and fired, if applicable – when the processing reaches that particular task.
3. *Flow tasks*, which consist of the execution of a *nested* ruleflow. Indeed, some complex decisions may require two or more levels of decomposition, and some tasks of the *main* ruleflow may consist of executing *another* ruleflow.

The three types of tasks can have *initial actions* and *final actions*, which consist of imperative IRL or BAL code to be executed upon entering or exiting the task. Ruleflow tasks are linked to each other and to the start and end nodes using *transitions*. Transitions can be *guarded*, i.e., they can be crossed only when certain *conditions* are satisfied. Those conditions are *Boolean* expressions that can reference variables that have *rule project (ruleset) scope*, i.e., either global Java

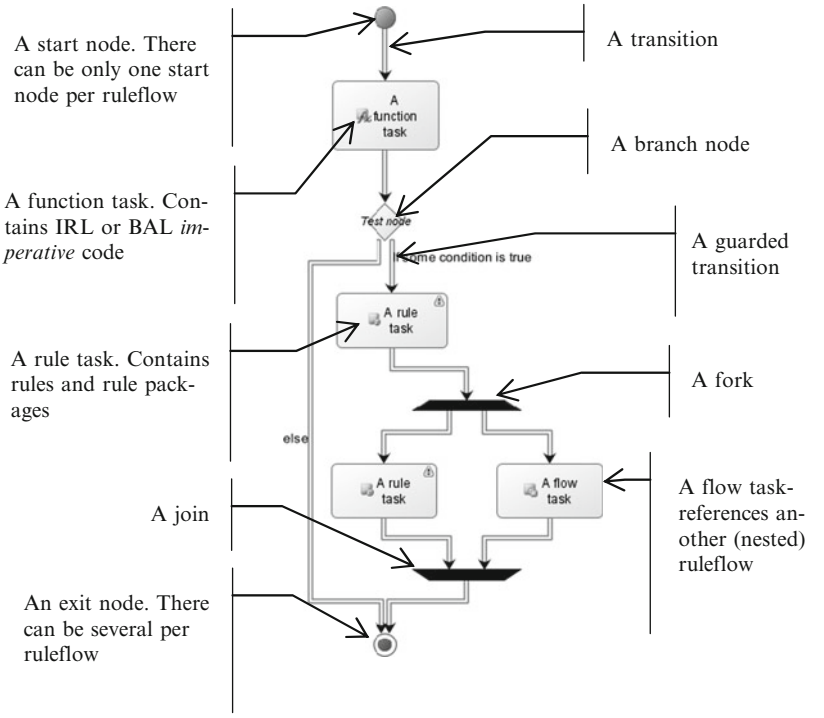


Fig. 11.10 The components of a ruleflow

variables,[22] ruleset variables or ruleset parameters. We can have several transitions coming out of the same task. If we have n transitions, $n - 1$ should be guarded, and the nth should be tagged with the *else*. When several transitions come out of a task, we can use a *branch node* for better visuals, even though it is not strictly necessary. Ruleflows can have *forks* and *joins*. Because there is no *parallel* execution of ruleflows, forks simply tell that there is no precedence between the branches of the fork. However, under the hood, the branches are serialized.

Figure 11.11 shows what a claim processing ruleflow might look like. The initialization function task does, indeed, insert the ruleset parameter theClaim into working memory so that rules that refer to a claim in working memory (i.e., non-scoped class condition) would still work. Both the data validation step and the eligibility step are complex and require ruleflows of their own. Hence, the claim processing ruleflow (Fig. 11.11a) references a data validation ruleflow (not shown) and the claim eligibility ruleflow (Fig. 11.11b).

In this ruleflow, we only check the eligibility of the claim if the data is valid, and we only adjudicate if the claim is eligible. This need not be the case. For example, we could choose to check eligibility even if some data fields are erroneous or do not

[22]For example, public static class data members.

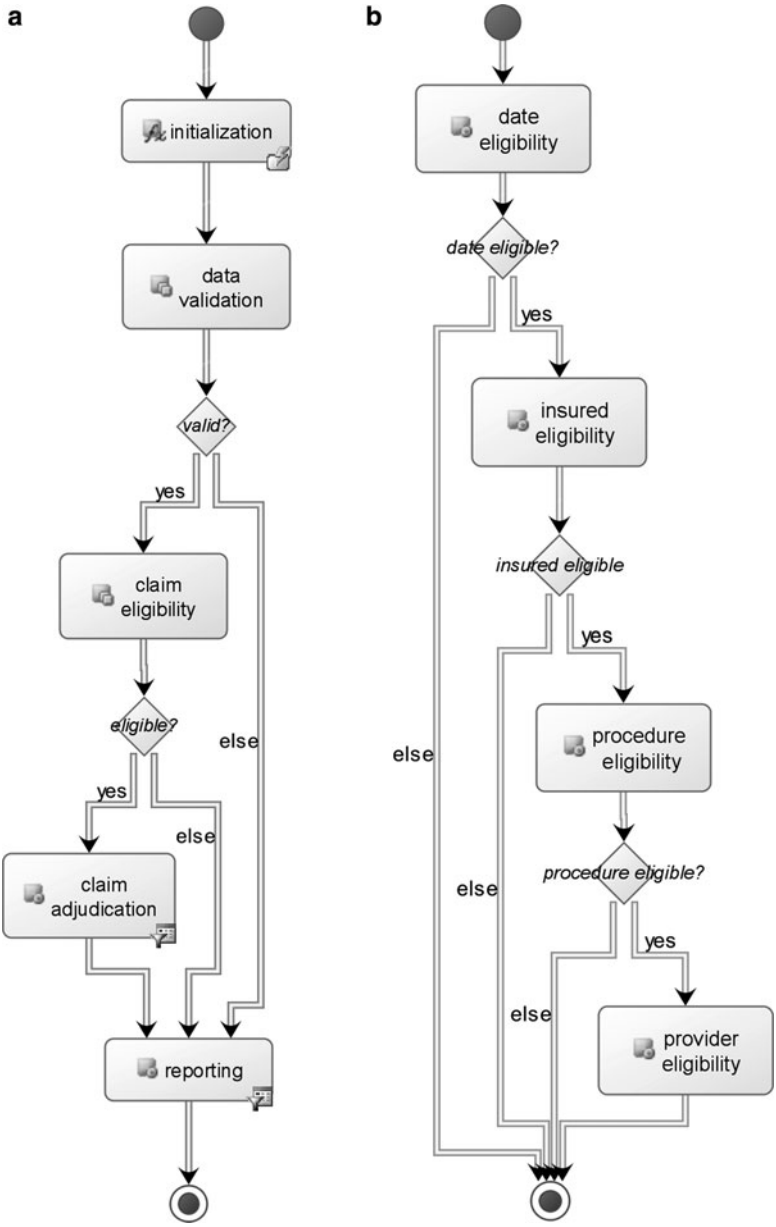


Fig. 11.11 Claim processing ruleflow. (**a**) Claim processing main flow and (**b**) claim eligibility ruleflow

make sense. Deciding which way to go is often a combination of computational constraints and business considerations. For example, in an automated system where throughput is important, we may decide to throw out a claim as soon as it

fails any of the data validation tests or any of the eligibility criteria. If there are problems past the first failure point, we will not know. However, a claims service representative may need to provide a complete diagnosis for a rejected claim for legal reasons, or for customer relationship management reasons: Tell the customer what to fix for their corrected submission, once and for all, instead of asking for yet another piece of documentation as the claim passes the various eligibility criteria.

For the sake of brevity, we will not show the Rule Studio wizards for creating and editing ruleflows. However, we discuss the corresponding IRL (Fig. 11.12).

We will comment the structure of the IRL, here displayed in multi-column format for compactness. The top pane shows the definition of the main ruleflow,

```
1  use claim_eligibility;
2  use data_validation;
3  flowtask claim_processing {
4    property mainflowtask = true;
5    body {
6      claim_processing#data_initialization;
7      claim_processing#data$_$validation;
8      if (data_validation.dataValid) {
9        claim_processing#claim$_$eligibility;
10       if ((theClaim.decision.equals("ELIGIBLE"))) {
11         claim_processing#claim_adjudication;
12         goto _node_5;
13       } else { goto _node_5; }
14     } else {
15       _node_5 : claim_processing#reporting;
16     }
17   }
18 };
```

```
19 functiontask
   claim_processing#data_initiali
   zation {
20   body {
21     insert theClaim;
22   }
23 };

24 flowtask
   claim_processing#data$_$valida
   tion {
25   body {
26     data_validation;
27   }
28 };

30 ruletask
   claim_processing#reporting {
31   algorithm = default;
32   ordering = dynamic;
33   body {
34     reporting.*
35   }
36 };
```

```
ruletask
claim_processing#claim_adjudic
ation {
  algorithm = default;
  ordering = dynamic;
  body {
    claim_adjudication.*
  }
};

flowtask
claim_processing#claim$_$eligi
bility {
  body {
    claim_eligibility;
  }
};
```

Fig. 11.12 The IRL equivalent to the "claim processing" main ruleflow

"`claim processing.`" The IRL construct for ruleflows is *flowtask* (line 3). A ruleflow/flowtask has a bunch of properties (line 4) and a body (lines 5–17). The body looks like any good old main program, even using a GOTO! Each statement in the body references a task, including the (function) task "`data initialization`" (line 6), the (flow) task "`data_validation`" (line 7), etc. Transition guards show up as simple if-then-else statements. The first transition (line 8) tests on a *ruleset variable* called "`data_valid`" defined within the rule package "`data validation.`" The second guarded transition (line 10) tests on the value of the `decision` attribute of the ruleset parameter "`theClaim.`"

Now that we have looked at the "main program", let us look at the "subroutines", i.e., the definitions of the various tasks. Lines 19–23 show the definition of the function "`data_initialization`": It consists of a single IRL statement: "`insert theClaim.`" In turn, the flowtask "`claim_processing#data$_$validation`"[23] is defined as invoking "`data_validation`" in its body, which is the name of the nested ruleflow. This externally defined entity is actually declared in line 2, with the statement "**`use`** `data_validation.`" The flowtask "`claim_processing#claim$_$eligibility`" (shown in second column) is defined in a similar fashion.

Consider now the ruletasks "`claim_processing#reporting`" (lines 30–36) and "`claim_processing#claim_adjudication`" (shown in second column). Their bodies are supposed to contain names of rules or rule packages. The notation "`reporting.*`" is similar to Java's import convention: The "`*`" means all of the contents of the package "`reporting`", including rules, and subpackages. The act of determining the body contents of a rule task is called *rule selection*, as in selecting the rules that will be evaluated and executed within the rule task. For these two cases, we talk about *static rule selection*, meaning that the *body* of a rule task is determined statically, at ruleflow development time. We can also have *run-time rule selection*, to be discussed in the next section.

Rule tasks also show two properties, "**`algorithm`**" and "**`ordering.`**" Within a ruleflow, we can *use a different rule execution algorithm for each task*. In fact, a ruleset that contains a ruleflow *behaves as* different rulesets, one per task, and the rule engine behaves as a bunch of rule engines, each with its own ruleset and agenda, but they share working memory.[24] We will discuss algorithm selection in the next section, and best practices for algorithm selection in section on "Further Reading".

[23]JRules enables us to use variable names that contain spaces. However, internally, it replaces spaces by "_" . . . and "_" by "$_$".

[24]We insist on the term *behaves as* because internally, it *is* the same ruleset object and the same rule engine, except that different subsets of the ruleset will be activated as we move from one rule task to another.

11.3.3 Ruleflows: Advanced Concepts

In this section, we talk about two features of ruleflows, *run-time rule selection*, and *algorithm selection*. *Rule selection* deals with the selection of the rule contents of a rule task. This selection is typically done at ruleflow development time where we pick a set of rules or packages to execute in the rule task. JRules enables us to compute the body of a rule task at run-time, and we will show how. As mentioned above, within a ruleflow, we can select a different rule execution algorithm for each rule task, and for each algorithm, we can specify additional parameters. We will show how to do that, and discuss situations where each algorithm is appropriate.

11.3.3.1 Run-Time Rule Selection

Figure 11.13 shows the Rule Studio wizard for selecting rules. From the property sheet of a rule task, we can select the "Rule Selection" which shows two panels. The top panel shows the list of rules and rule packages in the rule task. Initially empty, we can edit it by pressing the "Edit . . ." button which brings up the wizard shown on the left of Fig. 11.13. In the "Select Rules" wizard, we get, on the left hand side, the list of all rules and rule packages included in the current project *and in the projects that the current project depends on*. We can move rules and packages from the left to the right, and back, using the familiar $>$, $\gg$, $<$, and $\ll$ buttons. Once we press the "OK" button, the contents of the right list become the body of the rule task.

If we leave it at that, that is going to be the body of the rule task. In the example of Fig. 11.13, we are looking at the rule task "`claim adjudication`" from the "`claim processing`" ruleflow (see Fig. 11.11a). Here, we selected the rule package called "`claim adjudication.`" Note that this does not mean that *all* of the rules of the package "`claim adjudication`" will be evaluated/executed in this ruletask: Indeed, the ruleset extractor might actually filter out some rules from that package based on development status (e.g., only validated rules) or based on effective and expiration date. Thus, during run-time, this task will have all of the rules of the package "`claim adjudication`" *that were extracted by the ruleset extractor*.

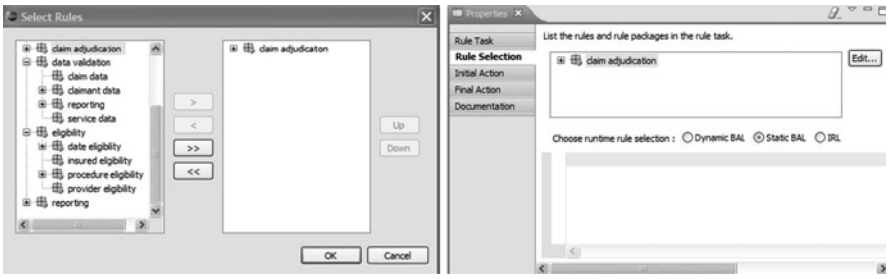


Fig. 11.13 The Rule Studio wizard for rule selection

As we mentioned above, we can also make *run-time rule selection*, which acts as an additional rule filter. Let us first consider a business scenario that requires run-time rule selection. Business policies and rules change regularly – one of the motivations for using the business rules approach! Insurance companies will update their rules regularly based on market trends, marketing studies, new actuarial studies, changing regulations, etc. When new rules come into effect, they usually have an effective date. If they are meant to replace older rules, the older rules will be made to expire on that date. However, new rules will generally apply *only* to new business. Existing contracts *will* continue to be honored according to the old rules. For example, if we decide to change the yearly cap on a particular procedure, the new cap will apply to *new* policies, or to existing policies *at renewal time* but will not apply retroactively to existing policies that are still in effect. So how do we handle that? One solution would have us use *different* rulesets, one per effectiveness period. When a claim comes in, we check the start and expiration dates of the policy, and select the ruleset to use accordingly. The yearly cap for procedure X is updated on January 20. The yearly cap for procedure Y is updated on February 18. The list of approved providers is updated on March 5 ... you get the idea: We will end up with *numerous* rulesets, and a cumbersome and error-prone ruleset dispatching mechanism. This is the business case for *run-time rule selection*: Our rule packages may contain rules with different effectiveness periods. However, for a given claim, we will select which of those rules to use, based on the effectiveness period of the policy of the claim.

Figure 11.14 shows the corresponding *run-time rule selection filter*. The BAL expression compares the "`effective date`" and "`expiration date`" of the rule to the "`start date`" and "`end date`" of the policy of the claim. Naturally, we can use any *run-time* property of a rule[25] and any property of the business data manipulated by the ruleset. Because the set of rule properties (metadata) is extensible, the possibilities are endless. For example, thanks to so-called *hierarchical properties* (properties whose values fit in a hierarchy), we can imagine filters based on the *jurisdiction of the rule*, and the place of

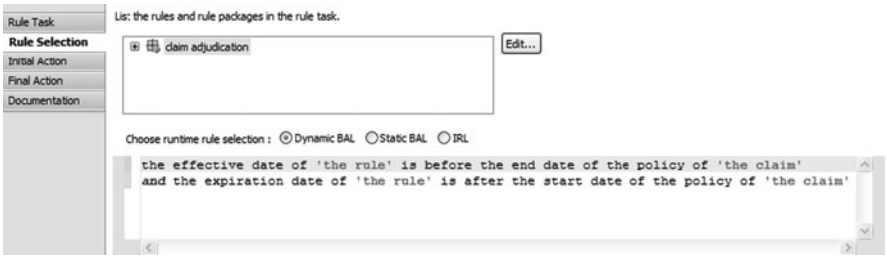


Fig. 11.14 The ***dynamic*** *run-time rule selection* filter

[25]The *properties* of a rule can be either *extractable*, in which case they are available in the *run-time* representation of rules, or *non-extractable*, in which case they are development time-only properties. The extractability of a rule property is a true/false attribute than can be set in *rule model extensions*. See Chap. 17 for more information about extending the rule *metamodel*.

```
ruletask claim_processing#claim_adjudication {
  algorithm = default;
  ordering = dynamic;
  scope {
    claim_adjudication.*,
  }
  body = dynamicselect (?rule) {
    return ((?rule.?effectiveDate.compareTo(
                theClaim.policy.endDate.toDate()) < 0
             && ?rule.?expirationDate.compareTo(
                theClaim.policy.startDate.toDate()) > 0));
  }
};
```

Fig. 11.15 IRL for a rule task with a dynamic run-time rule selection

residence of the policy holder. For example, California and US-wide rules will apply to a policy held by a San Francisco resident, whereas Michigan rules will not.

Figure 11.14 above shows a radio button labeled "Static BAL". So what is *static BAL run-time rule selection* filter? A *dynamic BAL run-time rule selection* filter is run *each time* the control flow reaches the rule task, i.e., each time the rule task is executed. In our case, that is the behavior we want, because our ruleset will be run with a *different* claim each time. By contrast, a *static BAL run-time rule selection* filter is applied *only* the *first time* the rule task is executed and the body of the task will remain constant throughout the lifetime of the ruleset object. There are not that many use cases where a *static BAL run-time rule selection* filter is appropriate.

Figure 11.15 shows the IRL for the rule task "claim adjudication". By comparing it with the IRL in Fig. 11.12, the reader may notice that the package "`claim_adjudication.*`" now represents the *scope* of the rule task, and the body consists of the filter. The filter is like a Boolean function that takes a rule as an argument, and return true if the rule should be included, and false otherwise. The *scope* determines the set of rules over which this filter will be applied?

Had we picked the *static BAL* button (Fig. 11.14) instead, the keyword `dynamicselect` would be replaced by the keyword `select`. And had we picked the button "IRL" (Fig. 11.14), we would have had to enter the body block, and would have had the leisure to pick either `dynamicselect` or `select`. We could even have used a different *signature* for the filter (static or dynamic) which takes no arguments and returns an array of rules to include in the rule task, in one shot, which makes the computation of the body more efficient. Indeed, dynamic run-time rule selection does have a performance cost, and if we are not careful, we can make it *prohibitively* costly.

11.3.3.2 Algorithm Selection

As mentioned above, we can select a different execution algorithm for each task within a ruleflow. JRules offers three execution algorithms, discussed in Chap. 6:

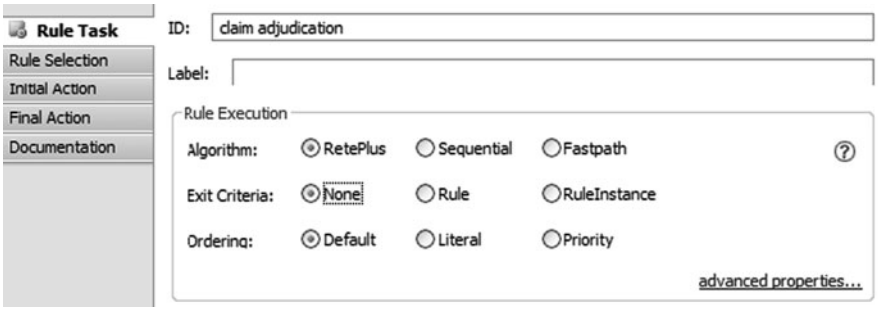


Fig. 11.16 The rule task algorithm selection wizard

1. The *RETE* algorithm, which is the default algorithm. This is the most powerful of the three, and it supports *rule* chaining (see Chap. 6).
2. The *sequential algorithm*, which applies the rules of a ruleset/task to the data of the working memory *sequentially*. Thus, for a given object tuple <object$_1$, object$_2$, ... ,object$_n$>, each rule is evaluated only once, if at all. This leads to a more efficient execution, but does not support rule chaining and has other limitations, to be discussed later.
3. The *fastpath* algorithm, which combines characteristics of the RETE algorithm and of the sequential algorithm. It does *not* support rule chaining, but it does not have many of the sequential algorithm limitations.

Figure 11.16 shows the algorithm selection wizard. In addition to the algorithm selection, we have two additional properties, with three potential values each:

1. *Exit criteria*, which defaults to "`None`" but can take the value "`Rule`" or "`RuleInstance.`" `None` means that we let the engine fire all of the rules that are satisfied. With `RuleInstance`, as soon as the engine fires *any* rule instance, we stop the execution and exit. With `Rule`, we let the engine fire all of the instances of *highest priority* rule, and then exit.
2. *Ordering*, which defaults to ... "`Default`" but which can take the values "`Literal`" or "`Priority.`" `Default` refers to the use of *dynamic priorities* to order the execution of rules. `Priority` means that rules are executed according to their static priorities, and `Literal` means that rules are executed according to their order of appearance in the task body (yuk!).

We can also enter some advanced properties in a textual format. One such property is `firinglimit` which can take any positive integer value, to mean how many rules of a particular task can fire before the task is terminated. A `firinglimit=0` means no limit, i.e., we let the algorithm run its course to the end. Generally speaking, the value `None` for "Exit Criteria" means `firinglimit=0`, and the value `RuleInstance` or `Rule` (depending on the execution algorithm) means `firinglimit=1`.

Note that not all combinations of values are legal. For example, `Default` ordering is not legal for the sequential algorithm, because the sequential algorithm

does not support dynamic priorities: we will get a ruleset parsing error.[26] Further, not all the legal combinations make sense. For example, the combinations <Algorithm=RetePlus, Ordering=Literal or Priority, Exit Criteria = anything> are legal but would yield a RETE algorithm with no agenda or rule chaining. Why bother? We will discuss below the legal combinations that do make sense. The reader can consult the product documentation for the more exotic combinations.

For the RETE algorithm. As mentioned above, only the `Default` value makes sense here. With regard to the exit criteria, if we use "`None`", we let the "while agenda not empty" loop discussed in Chap. 6 run its course until the agenda is empty. If "Exit Criteria" is `RuleInstance`, the "while agenda not empty" loop actually stops after the first rule instance is executed. This exit criterion might be needed for a rule task that contains rules that detect violations of eligibility or validation constraints: If all we are interested in is the pass/fail decision, then we can exit at the first violation, i.e., the first rule instance. With "Exit Criteria" equal to `Rule`, we take the first rule instance (i.e., the top of the agenda), and fire *it* and *all* of the other instances of the same rule that are on the agenda. Often, all of the other instances of the same rule will have the same priority as the first one, and will be "right behind" on the agenda. But there are situations where that would not be the case: For example, when that rule uses dynamic priorities – and thus, different instances will have different priorities – or when there are other rules on the agenda with the same priority – in which case other criteria such as recency (see Chap. 6) come into play.

For the sequential algorithm. As mentioned above, the `Ordering` property can be either `Literal` or `Priority` in this case. With `Literal`, for each data tuple, the rules are applied in the other in which they (or the packages that contain them) are listed in the rule task body. With `Priority`, the rules are applied according to their static priority. Regarding the `Exit Criteria` property, `None` means that all of the rules will be applied, `RuleInstance` does not make sense (because we have no agenda), and `Rule` means that as soon as a rule is fired, we drop the current data tuple, and take the next one.

For the fastpath algorithm. Because this algorithm does not rely on an agenda, the `Ordering` property can be either `Literal` or `Priority`, with the same behavior as with the sequential algorithm. Regarding the `Exit Criteria` property, `None` means that all of the rules will be applied. Using `RuleInstance` means that as soon as a rule instance is fired, we end the task. With `Rule`, we execute all of the instances of the first rule, based on the ordering property, and then we exit the task.

In this section, we discussed algorithm selection, and discussed the various parameters. We will discuss criteria for selecting one algorithm versus the other in Sect. 11.6.3.

[26] We will be able to *extract* the ruleset, but when we load it into a ruleset object, we get a ruleset parsing error.

11.4 Best Practices

In this section, we review some best practices regarding rule authoring and rule execution orchestration.

11.4.1 Best Practice 1: Design the Signature First

We saw in Sect. 11.5.1 how the use of ruleset parameters can change the way rules reference application objects, and ultimately, where rules will fire on not. Adding ruleset parameters *after* people have written rules can require some acrobatics:

1. Rewriting rules so that they now refer to the ruleset parameters
2. Add rules or ruletasks to insert ruleset parameters in working memory

It is better to start right, from the beginning. We will talk shortly about how to pick the correct signature.

The same is true with ruleflows: It is better to start designing the structure of the ruleflow *before* rule authoring starts. Indeed, we recommend that the rule architect design the high-level package structure of the rule project (see Sect. 9.4.3 for some design patterns) and the ruleflow, before rule authors start writing rules. This way, rule authors will write rules within the context of a predefined and carefully designed development structure (rule package hierarchy), and that structure is pre-mapped to the execution structure (ruleflow) through rule task rule selection. Indeed, when we write a rule, it *helps* to know the context under which the rule will be executed, i.e., the point in the process, what things are already assumed to be true, etc.

Regarding the ruleset signature, which data items should I pass back and forth between the calling application, and the rule engine? Actually, there are two aspects to this question, the *business* data contents, and the *computational* data structure. With regard to the *business* contents of the parameters . . . business knows! Policy analysts know which information they need for policy underwriting, policy renewal, or for claim processing. For each one of these processes, there is, naturally, the main document or transaction (e.g., policy application, claim), but also a bunch of ancillary or supporting data (see Fig. 11.17).

For example, for policy underwriting, we would want to know about the policy (which risks are to be covered, deductibles, restrictions), but we may also want to know about the (potential) policy holder credit file. For claim processing, in addition to the claim itself, we may want to get the policy itself, to see which coverages are included, but perhaps also some historical data about past claims, etc. Only business knows the sources of information they draw upon to make decisions.

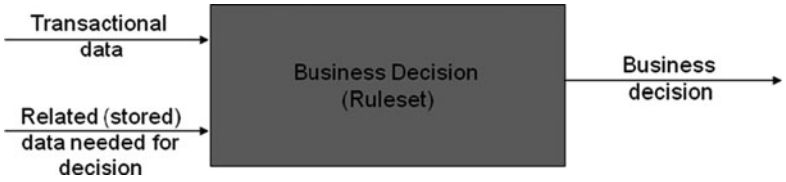


Fig. 11.17 The *business* signature of a ruleset

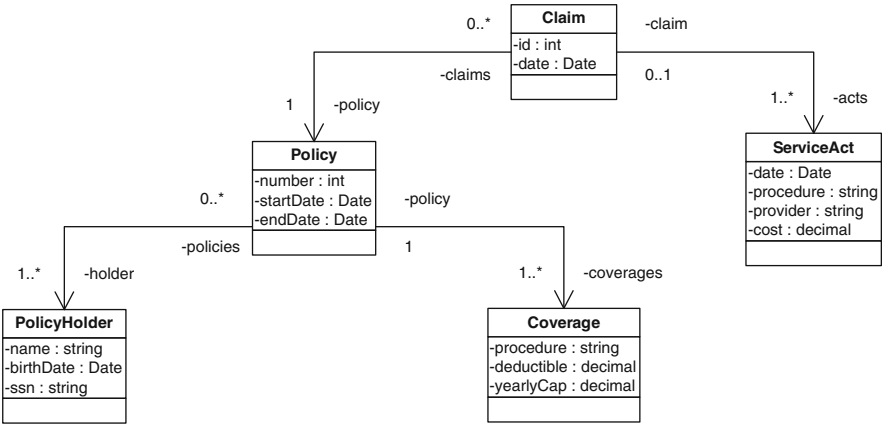


Fig. 11.18 A model of the data needed to make a decision

Having decided on the business contents of the data, the question now is how to structure it to pass it back and forth. We illustrate the issue with the model in Fig. 11.18. In this particular case, from the **`Claim`** object, we can access all of the information about the service acts, the policy, the coverages, and the policy holder. Hence, passing the **`Claim`** object is enough: The policy can be accessed as "`the policy of the claim`", the service acts can be accessed as "`a service act in the service acts of the claim`", etc. If we need the past claims, then we have several alternatives:

1. If we need the aggregated data from the past claims (e.g., total amount, number over a three year period, etc.), then those can be stored at the **`Policy`** object as attributes.
2. If we need the actual individual claims, then, either the **`Policy`** object points back to the claims made against it, in which case the main **`Claim`** object suffices, or we need to pass the set of past claims, separately, as a ruleset parameter.

Similar issues will arise regarding the decision output. Generally speaking, if the "rule team" has some control over the XOM, we can custom tailor the XOM (e.g., adding collection attributes to point from a **`Policy`** to past claims) to make the ruleset signature simpler.

11.4.2 Best Practice 2: Rulesets and Ruleflows

One of the design issues that will come up has to do with the granularity of the ruleflow. At the highest level, we have a business process that involves a number of decisions. At the lowest level, we have individual business rules. JRules provides with rulesets and ruleflows as a way of structuring rule executions. The question then becomes:

1. What should be the granularity of a ruleset?
2. Having chosen a ruleset granularity, how far down should we decompose the decision implemented by a ruleset using ruleflows?

Let us answer the ruleset question first, and then we will tackle ruleflows.

Chapters 3 and 4 argued that *decision points* within a business process are candidates for a ruleset. However, we did not talk about the granularity of the business process. Because business processes can themselves be nested, we had not answered the question entirely. Let us consider our case study. Figure 11.19 shows claim processing, at three levels of detail. At the highest level, we have the entire business process from the reception of the claim in paper format to actual payment. Starting with the process in the left, the first task consists of entering the claim data into the system, possibly scanning and archive receipts, etc. The next task consists

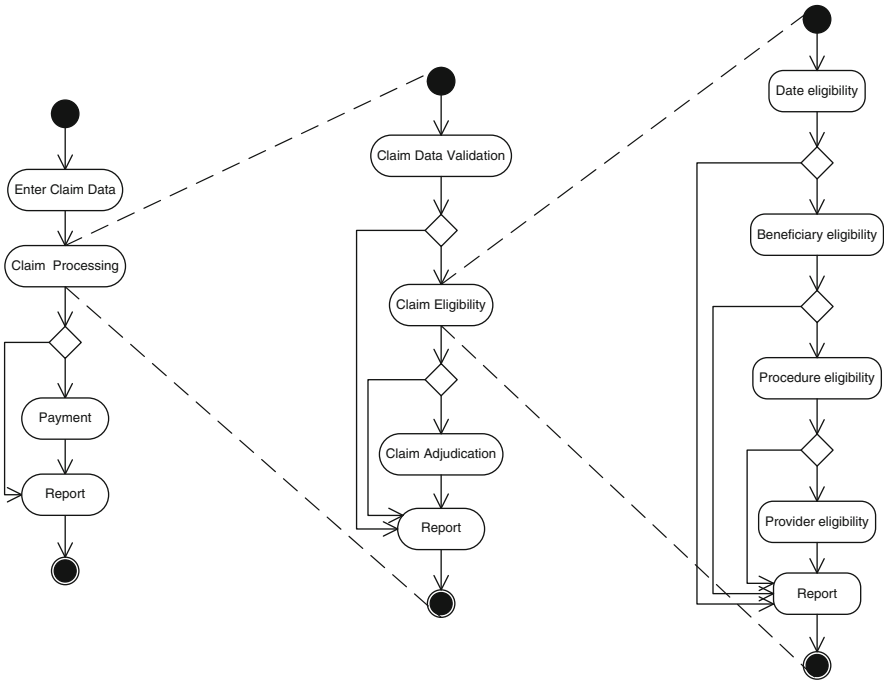


Fig. 11.19 The claim processing, at three levels of detail

of processing the claim, and if the claim is deemed payable, then we go through payment, and then report. The claim processing task itself can be broken down into data validation, eligibility, and adjudication. In turn, claim eligibility can be broken down into data eligibility, beneficiary eligibility, procedure eligibility, and provider eligibility.

So the question is which of the three processes should be a ruleset, if any? Generally speaking, a process or task should be a candidate for a ruleset if it satisfies two sets of criteria:

1. *Business criteria*:
   - The process or task should be *decision intensive*, i.e., it should involve business rules.
   - The process or task should embody a *cohesive decision*, i.e., have a single identifiable, business meaningful outcome.
2. *Computational criteria*:
   - The process or task should represent a *short-lived*, *synchronous* activity.
   - The process or task should not involve any *heavy-lifting*, e.g., accessing a legacy EIS, or making a remote connection.

The business criteria are self-explanatory. The computational criteria are justified by the fact that ruleset execution requires a single, synchronous rule engine invocation (the method `execute()`). Indeed, we would not want a rule engine invocation to last minutes, hours, or days, and lock the resources (claim object, policy object, etc.) while the engine is running. Second, we would not want to be dealing with exceptions raised by the external resources (e.g., a SQL exception, a database connection timeout, a deadlock, a remote method invocation timeout, etc.) within a ruleset execution because they are at worst, unrecoverable, and at best, leave the engine in an inconsistent state, making the entire rule engine invocation suspect.

Going back to our example (Fig. 11.19), *both* the claim processing process (middle one) and the claim eligibility process (right one) satisfy *all* the criteria, and are potential candidates for a ruleset. However, the top-level process does not:

- It is debatable whether we could call it *decision* intensive: Two tasks out of three are clerical and do not involve decisions (data entry and payment).
- It fails *both* computational criteria: It is *not* a short-lived process as it involves an external manual task (data entry, archiving), and it *does involve* accessing external resources, for both data entry and payment.[27]

Having eliminated the top process as a candidate for a ruleset, we can now worry about the next two.

[27]Data entry typically involves saving the data entered in the database, but also, pulling out the policy object from the database. Payment requires either printing checks or making automatic transfers by accessing a banking system.

If we choose to make "claim processing" a ruleset – as we have assumed in this chapter – then the internal process will be implemented using a ruleflow. One could also imagine deciding otherwise. In real life, the process labeled "claim processing" will likely require thousands of rules. Further, while data validation is generally relatively simple (e.g., checking individual property values), claim eligibility will involve lots of rules, and lots of data. If a claim fails data validation, we would have loaded all of the business data (policy, policy holder, past claims, etc.) for nothing. An architect might then choose to implement the "claim processing" process (middle of Fig. 11.19) in Java – or in BPEL or in some workflow engine – and then implement data validation, claim eligibility, and claim adjudication as separate rulesets.

Having decided on the granularity of the ruleset, now the question becomes: How fine-grained should be our ruleflows? Considering that a ruleflow is a piece of *hardcoded procedural logic*, the procedural logic needs to be *business-oriented*, so that it makes sense to the people writing rules, and it should be *stable* so that we do not have to frequently redesign the ruleflow. Indeed, the high-level package structure of a rule project and the ruleflow embody the *architecture* of the rule project and of the corresponding ruleset. We should *not* implement computational algorithms or replicate procedural code using ruleflows: We should let the engine do its job with the built-in inference mechanisms. For example, you know that you have gone too far if each rule task contains a handful of rules.

11.4.3 Best Practice 3: My Kingdom for an Algorithm

Chap. 6 explained the various rule engine execution algorithms. Section 11.5.3.2 of this chapter explained the different parameters of the various rule task execution algorithms, and how to set them. In this section, we present criteria for selecting an execution algorithm and its associated parameters for a particular task.

If you do nothing, the default execution algorithm for rule tasks is the RETE algorithm. As mentioned earlier, this is the most powerful of the three execution algorithms, and it supports all of the IRL constructs, including **`exists`**, **`not`**, truth maintenance, and event-based reasoning. This execution mode supports rule *chaining*. In the context of a ruleflow, rule chaining for a rule task means that the firing of a rule within that task can trigger the firing of another rule within the same task. Let us first refresh our memory about what rule chaining means. Consider the "procedure eligibility" task, in Fig. 11.19. A procedure is considered eligible if (a) it is covered by the policy and (b) it is *justified*. Assume that this is written using two rules as follows:[28]

[28]Note that the current BOM does not support these rules. They are used for illustration purposes.

```
  Rule 1 - coverage:
definitions
    set 'a service act' to a service act in the service acts
      of 'the claim';
if
    there exists a coverage in the coverages of the policy of
      'the claim' where the procedure of this coverage
      is the procedure of 'a service act' ,
then
    set the status of 'a service act' to "COVERED" ;

  Rule 2 - justification:
definitions
    set 'a service act' to a service act in the service acts
      of 'the claim' where the status of this service act is
      "COVERED";
if
    there exists a prescription in the documents of
      'the claim' where the procedure of this prescription
      is the procedure of 'a service act' ,
then
    set the status of ' a service act' to "JUSTIFIED" ;
```

The justification rule (Rule 2) will only be triggered for those service acts that have the status "**COVERED.**" If Rule 1 and Rule 2 are in the same task, only the RETE algorithm will ensure that if Rule 1 is executed for a particular service act, then Rule 2 will be evaluated and potentially triggered. Indeed, both the sequential algorithm and the fastpath algorithm will take a *single* pass at the rules, and if Rule 2 happens to be looked at before Rule 1 (see discussion in Chap. 6, and the rule ordering parameter in Sect. 11.5.3.2), we will never be able to establish that a service act is eligible!

Because the RETE algorithm is the least efficient of the three algorithms, we have to consider whether we need it for a particular task. Two sets of reasons would compel us to use the RETE algorithm:

1. *The decision logic*. The above example illustrated a case where rule chaining was needed for the proper execution of rules. Other cases include truth maintenance and event-based reasoning, which also require the RETE algorithm.
2. *The use of or reliance on working memory or agenda constructs in rules*. This means constructs like dynamic priorities, which are not supported in either sequential or fastpath. It also means *unscoped*[29] **exists**, **not** and collections, and their BAL equivalents, which are not supported by the sequential algorithm, and **insert**, **update** and **retract**, which will have unexpected or unpredictable behavior[30] in sequential and fastpath.

[29]That is, without the **in**/**from** constructs.

[30]For example, in RETE mode, when an object is **insert**'ed, *all* of the rules that concern it will be evaluated. In sequential and fastpath mode, the new object *may* or *may not* be considered

The two factors are not independent: business logic can also dictate the kind of IRL construct we use. For example, while we can refrain from using unscoped **exists** or **not** – by scoping them using **in**/**from** constructs!—it may be far more awkward, for a particular application, to implement the business logic without **insert** or **retract**, say.

If you have established that, *for a particular task*, the business logic does *not* require the RETE algorithm, and if the rules that do go into that task do not use the IRL constructs mentioned above, then we should aim for the more efficient alternatives, the sequential or fastpath algorithm. Which one should you use? As it turns out, this is not only a question of efficiency, but it is also a question of correctness. Indeed, if the rules within a rule task do not have a *homogeneous signature*, the sequential algorithm will not behave correctly.

Informally, the *signature* of a rule is the *tuple* of objects on which the rule applies. Formally, the *signature* of an IRL rule is the set of *simple class conditions* of the rule. At the BAL level, it is the set of *object* variables of the rule – including *object* ruleset parameters, *object* ruleset variables, and *object* local variables.[31] In the example above, the rules Rule 1 (coverage) and Rule 2 (justification) have the same signature: {**Claim**, **ServiceAct**}. By contrast, the signature of the following rule is {**Claim**, **PolicyHolder**, **ServiceAct**}.

```
  Rule 3 - different signature:
definitions
   set 'a service act' to a service act in the service acts
 of 'the claim;
   set 'a policy holder' to apolicy holder in the insureds
       of the policy of 'the claim;
if
   …
```

Recall from Chap. 6 that the default tuple generator used by the sequential algorithm (see Chap. 6) takes the *union* of the signatures of the rules within the task to generate the tuples. Thus, if Rule 3 were in the same rule task as Rule 1 and Rule 2, the tuple generator will use the signature {**Claim**, **PolicyHolder**, **ServiceAct**} as the structure for the tuples. Given the objects `claim_1`, `serviceAct_1`, `serviceAct_2`, `policyHolder_1`, `policyHolder_2`, the tuple generator will generate the tuples: T_1 = <claim_1, policyHolder_1, serviceAct_1>, T_2 = <claim_1, policyHolder_1, serviceAct_2>, T_3 =

depending on the tuple enumerator used by the sequential algorithm or the rule ordering algorithm used by fastpath.

[31] *Object* local variables are variables defined using the form "set <var name>to a <object type> [scope expression]." A variable that represents the *value* of an attribute (regardless of its type) is not mapped to an IRL class condition.

<claim_1, policyHolder_2, serviceAct_1>, and T_4 = <claim_1, policyHolder_2, serviceAct_2>. For each tuple, we will apply Rule 1, Rule 2, and Rule 3, sequentially; if a rule has a smaller signature than the tuple, we "project" the tuple on the signature of the rule, meaning that the extra objects are ignored. This means that Rule 1, for example, will be evaluated *twice* on the pair <claim_1, serviceAct_1>, first while we process T_1 and a second time while we process the tuple T_3. The same is true for the pair <claim_1, serviceAct_2>, which will be evaluated twice by Rule 1, once for T_2 and a second time for T_4. The same is true for Rule 2. Having a rule execute several times on the same tuple of objects within a single run can be anywhere from inefficient to outright wrong. Hence, if the rules within a task do not have the same signature, the sequential algorithm should not be considered.

If the rules within a task *do* have the same signature, then it becomes a matter of performance. Recall that the fastpath algorithm does build a RETE network from the rules; it is just that takes a single pass at the rules. Compiling the rules of the task into a RETE network does have a cost. The benefit is the underlying condition sharing. Thus, if the rules of the task have numerous randomly ordered conditions, the fastpath algorithm will incur the RETE network construction costs, without the benefit of condition sharing: We should use the sequential algorithm. If the rules share some conditions, then the fastpath algorithm is preferred.

We summarize our *preliminary* discussion in the decision process of Fig. 11.20. This decision process needs to be qualified. In particular, the need for the RETE algorithm and for WM or agenda constructs can, in some cases, be eliminated, or reduced in scope. This is illustrated with a couple of examples below.

Most underwriting decisions – be they for mortgage or insurance – involve two distinct phases: (a) a *risk assessment* phase, which assigns a risk score to the customer application (for a loan or an insurance policy) and (b) a decision phase, which consists of assigning a recommendation (typically, accept, reject, or send for manual referral) based on that score. The underwriting decision itself *does* require rule chaining between risk assessment rules and decision rules. *However*, if we break the underwriting decision into two tasks, then the ruleflow built-in control flow will enforce that rule chaining. This is illustrated in Fig. 11.21. With this decomposition, instead of selecting a single algorithm for the task "Policy underwriting" (left ruleflow), we can now select different algorithms for the rule tasks "Risk scoring" and "Decision". Typically, risk scoring rules compare attributes to predefined ranges and increment or decrement a cumulative score, and they do not require rule chaining. The same is true for decision rules which typically compare a single risk value, or a set of score, to predefined thresholds and assign a decision with justifications. Thus, we should be able to use the sequential or fastpath algorithm for each of the two tasks taken separately – provided that the IRL/BAL constructs that are used in the rules allow it!

While this is a useful heuristic, it should be used *sparingly*: We should resist the temptation of slicing business decisions into finely granular, sequential decisions,

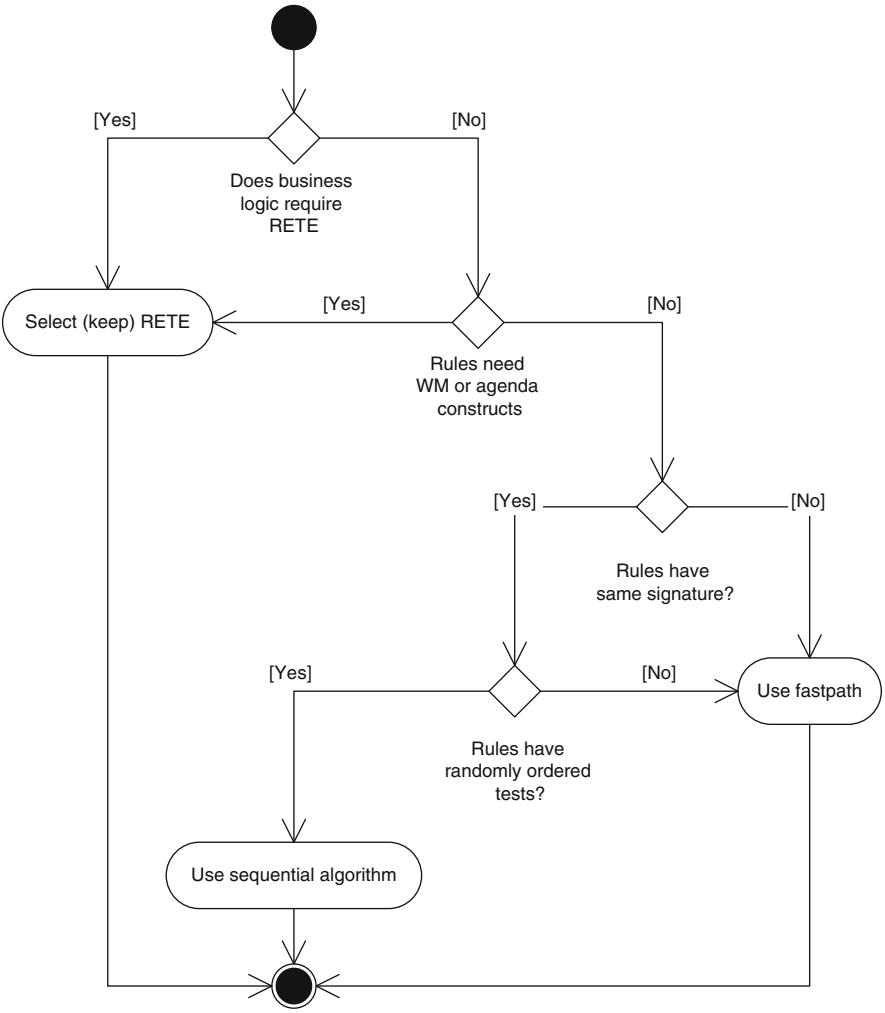


Fig. 11.20 A first-cut rule task execution algorithm selection process

just to get rid of rule chaining. Do not lose from sight the guidelines provided in Sect. 11.4.2 regarding the granularity of ruleflows.

With regard to the IRL or BAL constructs that are problematic or forbidden in sequential/fastpath, by adhering to a few stylistic guidelines, we can live without *most* of them – and never miss them again. For example, we can refrain from using unscoped **`exists`**, **`not`**, and collections in IRL, or their BAL equivalents. In particular, by using ruleset parameters to communicate business data to the engine and by refraining from inserting objects in working memory – as is the recommended practice, see Sects. 11.3.1 and 11.4.1 – we have no choice but to use the *scoped* versions of **`exists`**, **`not`**, and collections: Rules would *not* work otherwise, regardless of the execution algorithm!

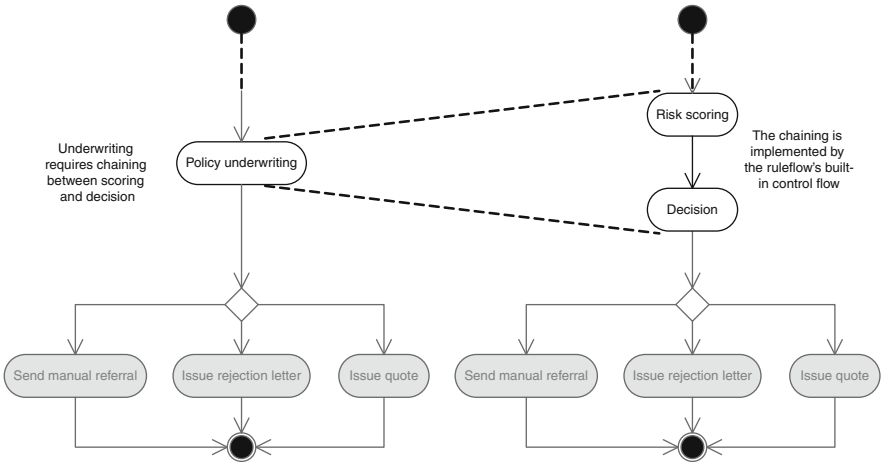


Fig. 11.21 By breaking a decision into two, we *may* obviate the need for rule chaining

This discussion raises two issues. First, *technically*, it *is always possible* to write the business rules so that they can execute in sequential or fastpath ... as it is possible to write them in Java or assembly language! The question is: ***How much of a price are we willing to pay for efficiency***. Keep in mind that business rules are supposed to become the communication language between business and IT. If that language is tweaked to the point that the business logic is no longer recognizable by a business person, be it a mortgage specialist, for a mortgage underwriting application, or a network operator, for an alarm filtering and correlation application, then we defeated the purpose of the business rules approach.

The second issue is related to the interplay between rule authoring and algorithm selection. If we design the ruleflow before we write the rules – as is the recommended practice, see Sects. 11.4.1 and 11.4.2 – then we will not know which algorithm to use for each rule task, until the rules are written. This erodes, a bit further, the separation of concerns between the *development time concerns* surrounding rule authoring and the *run-time concerns*. In particular, it raises the question of how much a rule author needs to know about the execution context of the rule that she or he is writing, for that rule to execute *correctly* and *efficiently*. This is a valid concern, but as we showed for the case of problematic IRL/BAL constructs, we can achieve quite a bit with a good preliminary design (Sects. 11.4.1 and 11.4.2), and a few stylistic guidelines, which can be enforced through the use of rule templates.

11.4.4 Best Practice 4: Do You Really Need a Custom Language?

We showed in Sect. 11.4.6 the JRules *Business Rules Language Development Framework* or *BRLDF* for short, a framework for developing custom rule authoring languages. The BRLDF, which has evolved over a dozen or so years, has a nice

modular design, and provides a nice separation between the *abstract syntax* of a language from its *concrete syntax*. It also provides a clean separation between *parsing* and *code generation*. The BRLDF also enables us to build a language *incrementally* by modifying an existing language. This provides for localized and low-cost customization of existing languages. This makes it particularly easy to customize or extend the BAL, which is built using the BRLDF.

That being the case, do you really need a separate rule language? Now? The answer is probably no, and almost certainly not now. In Chap. 9, we argued that a new rule language is justified only when the following conditions, reframed within the context of JRules, are satisfied:

- The BAL syntax represents an unnecessary burden, and an unbearably awkward syntax for the rule authors.
- The cost of developing the custom rule language, the custom rule editor, and the custom rule engine was minimal.
- You have reasonable assurance that future evolution of JRules will not invalidate your language.

With regard to the second condition, the BRLDF design ensures that the cost of developing the language is indeed minimal, if we build it by extending or reusing parts of the BAL. However, how could we have a reasonable assurance that future evolution of the product will not invalidate your custom rule language? And if so, for how long? JRules is one of the most mature – if not *most* mature – BRMSs on the market. And yet, historically, it underwent major modernizations every few years. A case in point is the change between JRules 5 and JRules 6, which came out in late 2005/early 2006. In JRules 5, BAL rules are persisted in the form of their abstract syntax trees, serialized in XML format. In JRules 6, BAL rules are persisted in BAL text format. JRules 6.x and 7.x include utilities that know how to read the old representation format (XML-based abstract syntax trees) and how to convert them to the new format. However, these utilities understand, out of the box, only the *standard* BAL syntax.[32] This means a set of painful choices:

1. Refrain from upgrading to the newest product version, thereby foregoing valuable additional functionalities, bug fixes, or architectural enhancements.
2. Manually migrate your existing rules into the new version of JRules.
3. Develop your own migration utilities.

Note that both choice two and three imply that you upgrade your implementation of the custom language into the new version of JRules/BRLDF.

Different customer circumstances have at one point or another dictated each of the three choices. We can certify that they were all painful, and we do not recall a case where it was *candidly* felt that the customization added-value was, with hindsight, worth the initial language development effort (minimal) *and* the

[32]Well. They can also handle simple extensions like specifying value editors or specializing tokens of the language, but they cannot handle different grammatical structures.

migration pain (major). So how do customers get talked – or talk themselves – into building risky custom languages? Two reasons: (a) uneducated or unreasonable user requirements, and (b) an eager development organization. Indeed, if JRules is brought into an organization to replace another niche BRMS-like product, business users may insist on (and get) keeping every single nicety – or idiosyncracy – of the niche-product it is replacing, even when there are better or cleaner way of doing it in the generalist JRules. This could mean recreating an idiosyncratic rule entry language.[33] Second, developers are often eager to please because developers ... love to develop: Any opportunity to delve into the more exotic parts of the API is a welcome relief from the often repetitive development tasks. Project managers and technical leads should know when to call off the party and say no.

As for the timing, while we believe that there is seldom a good time to develop a custom rule entry language, doing it on your first major rule project is definitely the *wrong* time. Project teams have enough to deal with on the first release of a rule-based application; they should not overburden themselves with "cosmetic" or nice-to-have features. And besides, the requirements for such a language can only be determined through practice.

11.5 Discussion

There is a lot more to what we collectively referred to as "rule authoring" than actually coding individual rules. Rule execution orchestration involves a number of complex design decisions that impact rule authoring, rule deployment, and rule execution. In this chapter, we identified these design decisions, described the design space, and discussed some best practices.

Designing rule execution orchestration falls within the purview of the *rule architect* and is of no concern to rule *writers*. As illustrated for the case of algorithm selection, the rule architect needs a deep understanding of the business logic, a deep understanding of the BAL, IRL, and an understanding of rule engine mechanics. Similarly, ruleset signature requires good business logic knowledge and software architecture knowledge.

As this chapter and last showed, the rule architect has a central role in rule authoring, management, and execution. He also needs a variety of skills straddling three different areas: business, java, and JRules. From our experience, customers often misunderstand this role and assign its tasks to individuals who lack one – and sometimes two – skill sets, or worse yet, assign different tasks to different individuals. This typically leads to suboptimal or incoherent designs.

Our experience has also been that customers underestimate the skill level required of rule writers. In most projects, we have been to where IT is responsible for authoring and maintaining rules, it was often the most junior members of the

[33]We can call it a *domain-specific language* to make it more acceptable ☺.

team that got to write rules. That is a shame because good rule authoring requires a deep understanding of the business logic, an awareness of the rule coding patterns discussed in Chap. 9, and a mastery of the business action language (BAL) and its derivatives. A junior *IT* person would typically lack at least one of the skill sets.

As with any other technology, quality is *not inevitable*. Get the wrong people, and you get the wrong results. If this is your first business rules project, get the wrong people, and not only do you get the wrong results, but you also learn the wrong lesson – and set back business rule adoption in your organization by a few years.

11.6 Further Reading

As this chapter is JRules specific, additional sources of information can be found in the product documentation and on IBM's support site for *Websphere Ilog JRules* at http:/publib.boulder.ibm.com/infocenter/brjrules/v7r1/index.jsp.

More information about the rule engine execution algorithms can be found in Chap. 6 and its references. The Web site www.agilebrdevelopment.com, which is dedicated to this book, contains complementary information.