Progress® | Corticon®

PROGRESS® CORTICON®

Corticon Studio:
Rule Modeling Guide

PROGRESS software

Notices

For details, see the following topics:

- Copyright

Copyright

See Table of Contents for location of Third party acknowledgements within this documentation.

Table of Contents

Preface

For details, see the following topics:

- Progress Corticon documentation
- Overview of Progress Corticon

Progress Corticon documentation

The following documentation, as well as a *What's New in Corticon* document, is included with this Progress Corticon release:

Corticon Tutorials	
Corticon Studio Tutorial: Basic Rule Modeling	Introduces modeling, analyzing, and testing rules and decisions in Corticon Studio. Recommended for evaluators and users getting started. *See also the PowerPoint-as-PDF version of this document that is accessed from the Studio for Analysts' **Help** menu.*
Corticon Studio Tutorial: Advanced Rule Modeling	Provides a deeper look into Corticon Studio's capabilities by defining and testing vocabularies, scope, collections, messages, filters, conditions, transient data, and calculations in multiple rulesheets that are assembled into a Ruleflow. *See also the PowerPoint-as-PDF version of this document that is accessed from the Studio for Analysts' **Help** menu.*
Corticon Tutorial: Using Enterprise Data Connector (EDC)	Introduces Corticon's direct database access with a detailed walkthrough from development in Studio to deployment on Server. Uses Microsoft SQL Server to demonstrate database read-only and read-update functions.

Corticon Studio Documentation: Defining and Modeling Business Rules	
Corticon Studio: Installation Guide	Step-by-step procedures for installing Corticon Studio and Corticon Studio for Analysts on computers running Microsoft Windows. Also shows how use other supported Eclipse installations for integrated development. Shows how to enable internationalization on Windows.
Corticon Studio: Rule Modeling Guide	Presents the concepts and purposes the Corticon Vocabulary, then shows how to work with it in Rulesheets by using scope, filters, conditions, collections, and calculations. Discusses chaining, looping, dependencies, filters and preconditions in rules. Presents the Enterprise Data Connector from a rules viewpoint, and then shows how database queries work. Provides information on versioning, natural language, reporting, and localizing. Provides troubleshooting and many *Test Yourself* exercises.
Corticon Studio: Quick Reference Guide	Reference guide to the Corticon Studio user interface and its mechanics, including descriptions of all menu options, buttons, and actions.
Corticon Studio: Rule Language Guide	Reference information for all operators available in the Corticon Studio Vocabulary. A Rulesheet example is provided for many of the operators. Includes special syntax issues, handling arithmetic and character precedence issues.
Corticon Studio: Extensions Guide	Detailed technical information about the Corticon extension framework for extended operators and service call-outs. Describes several types of operator extensions, and how to create a custom extension plug-in.
Corticon Server Documentation: Deploying Rules as Decision Services	
Corticon Server: Deploying Web Services with Java	Details installing the Corticon Server as a Web Services Server, and then and deploying and exposing Decision Services as Web Services on Tomcat and other Java-based servers. Presents the features and functions of the browser-based Server Console.
Corticon Server: Deploying Web Services with .NET	Details installing the Corticon Server as a Web Services Server and deploying and exposing decisions as Web Services with .NET. Provides installation and configuration information for the .NET Framework and Internet Information Services (IIS) on various supported Windows platforms.
Corticon Server: Integration & Deployment Guide	An in-depth, technical description of Corticon Server deployment methods, including preparation and deployment of Decision Services and Service Contracts through the Deployment Console tool. Discusses relational database concepts and implementation of the Enterprise Data Connector. Goes deep into the server to discuss state, persistence, and invocations by version or effective date. Includes samples, server monitoring techniques, and recommendations for performance tuning.

Overview of Progress Corticon

Progress® Corticon® is the Business Rules Management System with the patented "no-coding" rules engine that automates sophisticated decision processes.

Progress Corticon products

Progress Corticon distinguishes its development toolsets from its server deployment environments.

- **Corticon Studios** are the Windows-based development environment for creating and testing business rules:
  - **Corticon Studio for Analysts.** is a standalone application, a lightweight installation that focuses exclusively on Corticon.
  - **Corticon Studio** is the *Corticon Designer* perspective in the **Progress Developer Studio** (PDS), an industry-standard Eclipse 3.7.1 and Java 7 development environment. The PDS enables integrated applications with other products such as Progress OpenEdge and Progress Apama.

  The functionality of the two Studios is virtually identical, and the documentation is appropriate to either product. Documentation of features that are only in the *Corticon Designer* (such as on integrated application development and Java compilation) will note that requirement. Refer to the *Corticon Studio: Installation Guide* to access, prepare, and install each of the Corticon Studio packages.

  Studio Licensing - Corticon embeds a time-delimited evaluation license that enables development of both rule modeling and Enterprise Data Connector (EDC) projects, as well as testing of the projects in an embedded Axis test server. You must obtain studio development licenses from your Progress representative.

- **Corticon Servers** implement web services for business rules defined in Corticon Studios:
  - **Corticon Server for deploying web services with Java** is supported on various application servers, and client web browsers. After installation on a supported Windows platform, that server installation's deployment artifacts can be redeployed on various UNIX and Linux web service platforms as Corticon Decision Services. The guide *Corticon Server: Deploying web services with Java* provides details on the full set of platforms and web service software that it supports, as well as installation instructions in a tutorial format for typical usage.
  - **Corticon Server for deploying web services with .NET** facilitates deployment of Corticon Decision Services on Windows .NET Framework 4.0 and Microsoft Internet Information Services (IIS). The guide *Corticon Server: Deploying web services with .NET* provides details on the platforms and web service software that it supports, as well as installation instructions in a tutorial format for typical usage.

  Server Licensing - Corticon embeds a time-delimited evaluation license that enables evaluation and testing of rule modeling projects on supported platform configurations. You must obtain server deployment licenses and server licenses that enable the Enterprise Data Connector (EDC) from your Progress representative.

1

Building the Vocabulary

This section describes the concepts and purposes of a Corticon Vocabulary. You see how to build a Vocabulary from general business concepts and relationships.

For details, see the following topics:

What is a Vocabulary?

Depending on your point of view, a Vocabulary represents different things and serves different purposes. For the rule modeler, the Vocabulary provides the basic elements of the rule language – the building blocks with which business rules are implemented in Corticon. For a systems analyst or programmer, a vocabulary is an abstracted version of a data model that contains the objects used in those business rules implemented in Corticon.

A vocabulary serves the following purposes:

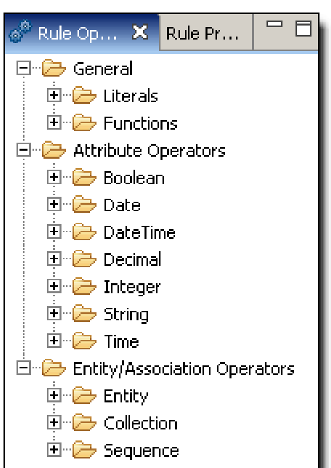
- Provides terms that represent business "things". Throughout the product documentation, we will refer to these things as *entities,* and properties or characteristics of these things as *attributes*. Entities and their attributes in underlying data sources (such as tables in a relational database or fields in a user interface) can be represented in the Vocabulary.
- Provides terms that are used to hold temporary or *transient* values within Corticon (such as the outcome of intermediate derivations). These entities and attributes usually have a business meaning or context, but do not need to be saved (which we will also refer to as *persisting*) in a database, or communicated to other applications external to Corticon. An example of this might be the following two simple computational rules:

  1. itemSubTotal is equal to the product of itemCount and itemPrice
  2. orderTotal is equal to the sum of all itemSubTotals

  In these two rules, `itemSubTotal` is the intermediate or transient term. We may never use `itemSubTotal` by itself; instead, we may only create it for purposes of subsequent derivations, as in the calculation of `orderTotal` in rule #2. Since a transient attribute may be the result of a very complicated rule, it may be convenient to create a Vocabulary term for it and use it whenever rewriting the complex rule would be awkward or unclear. Also see Extended Transients, in the Vocabulary.

- Provides a federated data model that consolidates entities and attributes from various enterprise data resources. This is important because a company's data may be stored in many different databases in many different physical locations. *`Corticon`* believes that rule modelers need not be concerned with where data is, only how it is used in the context of building and evaluating business rules. The decision management system should ensure that proper links are maintained between the Vocabulary and the underlying data. We often refer to this concept as *abstraction* – the complexities of an enterprise's data storage and retrieval systems have been hidden so that only the aspects relevant to rule writing are presented to the rule modeler.
- Provides a built-in library of *literal* terms and operators that can be applied to entities or attributes in the Vocabulary. This part of the Vocabulary, the "lower half" of the **Vocabulary** window shown in the following figure, is called the "Operator Vocabulary" because it provides many of the verbs (the "operators") needed in our business rules. Many standard operators such as the mathematical functions { +, -, *, /} and comparator functions {<, >, =} as well as more specialized functions are contained within this portion of the Vocabulary. See the *Rule Language Guide* for descriptions and examples of all operators available, as well as detailed instructions for extending the library.

  Figure 1: Operator Vocabulary



- Defines a schema that supplies the contract for sending data to and from a `Corticon` Decision Service (`Rulesheet`s deployed in production). Since XML messaging is used to carry data to and from the rules for evaluation, data must be organized in a pre-defined structure that can be understood and processed by the rules. An XML schema that accomplishes this purpose can be automatically generated directly from the Vocabulary. This schema is called a Vocabulary-Level service contract and details can be found in the `Server` *Integration & Deployment Guide*.

Scope

An important point about a Vocabulary: there does not need to be a one-to-one correlation between terms in the Vocabulary and terms in the enterprise data model. In other words, there may be terms in the data model that are not included in or referenced by rules – such terms need not be included in the Vocabulary. Conversely, the Vocabulary may include terms (such as extended transient attributes) that are used only in rules – these terms need not be present in the data model. Two guiding principles:

- If the rule modeler wants to use a particular term in a business rule, then that term must be part of the Vocabulary. This can include terms that exist only within the Vocabulary – these are the transient attributes introduced above.
- If a rule produces a value that must be retained, persisted, or otherwise saved in a database (or other means external to the rules), then that Vocabulary term must also be present in the enterprise data model. There are many methods for linking or mapping these Vocabulary terms with corresponding terms in the data model, but a discussion of these methods is technical in nature and is not included in this manual.

There are two basic starting points for creating a Vocabulary: starting from an existing data model or starting from scratch. We will start by examining the latter since it is typically more challenging.

Starting from scratch

Investigation

The first step in creating a Vocabulary from scratch is to collect information about the specifics of the business problem you are trying to solve. This usually includes research into the more general business context in which the problem exists. Various resources may be available to you to help in this process, including:

- **Interviews** – the business users and subject matter experts themselves are often the best source of information about how business is conducted today. They may not know how the process is *supposed* to work, or how it *could* work, but in general, no one knows better how a business process or task is performed today than those who are actually performing it.
- **Company policies and procedures** – when they exist, written policies and procedures can be an excellent source of information about how a process is *supposed* to work and the rules that govern the process. Understanding the gaps between what is supposed to happen and what is actually happening can provide valuable insight into the root problems.
- **Existing systems & databases** – systems are usually created to address specific business needs, but needs often change faster than systems can keep up. Understanding what the systems were designed to do versus how they are actually being used often provides clues about the core problems. Also, business logic contained in these legacy systems often captures business policies and procedures (i.e., the business rules) that are not recorded anywhere else.
- **Forms and reports** – even in heavily automated businesses, forms and reports are often used extensively. These documents can be very useful for understanding the details of a business

process. Reports also illustrate the expected output from a system, and highlight the information users require.

Analyze the chosen scenario and/or existing business rules in order to identify the relevant terms and the relationships between these terms. We refer to statements expressing the relevant terms and relationships as "facts" and recommend developing a "Fact Model" to more clearly illustrate how they fit together. We will use a simple example to show the creation of a Fact Model and its subsequent development into a Vocabulary for use in Corticon Studio.

Designing the Vocabulary

Example

An air cargo company has a manual process for generating flight plans. These flight plans assign cargo shipments to specific aircraft. Each flight plan is assigned a flight number. The cargo company owns a small fleet of three airplanes, 2 Boeing 747s and 1 McDonnell-Douglas DC-10 freighter. Each airplane type has a maximum cargo weight and volume that cannot be exceeded. Each aircraft also has a tail number which serves to identify it. A cargo shipment has characteristics like weight, volume and a manifest number to identify it.

Now let's assume the company wants to build a system that automatically checks flight plans to ensure no scheduling rules or guidelines are violated. One of the many business rules that need to be checked by this system is:

1. An aircraft must not carry a cargo shipment that exceeds its maximum cargo weight.

Step 1 – Identify the Terms

We identify the terms (entities and attributes) for our Vocabulary by circling or highlighting those nouns that are used in the business rules we seek to automate. The previous example is reproduced below:

An air cargo company has a manual process for generating flight plans. These flight plans assign cargo shipments to specific aircraft. Each flight plan is assigned a flight number. The cargo company owns a small fleet of three airplanes, 2 Boeing 747s and 1 McDonnell-Douglas DC-10 freighter. Each airplane type has a maximum cargo weight and volume that cannot be exceeded. Each aircraft also has a tail number which serves to identify it. A cargo shipment has characteristics like weight, volume, packaging method, and a manifest number to identify it.

Step 2 – Separating the Generic Terms from the Specific

Why did we only circle the "aircraft" term above and not the names of the aircraft in the fleet? It is because 747 and DC-10 are *specific* types of the *generic* term aircraft. The *type* of aircraft can be said to be an attribute of the generic aircraft entity. Along these same lines, we also know from the example that several cargo shipments and flight plans can exist. Like the specific aircraft, these are *instances* of their respective generic terms. For the Vocabulary, we are only interested in identifying the generic (and therefore reusable) terms. But ultimately, we also will need a way to identify specific cargo shipments and flight plans from within the set of all cargo shipments and flight plans – assigning *values* to attributes of a generic entity will accomplish this goal, as we will see later.

Step 3 – Assembling and Relating the Terms

None of the terms we have circled exists in isolation – they all relate to each other in one or more ways. Understanding these relationships is the next step in Vocabulary construction. We begin by simply stating facts observed or inferred from the example:

- An aircraft *carries* a cargo shipment.
- A flight plan *schedules* cargo for shipment *on* an aircraft.
- A cargo shipment *has* a weight.
- A cargo shipment *has* a manifest number.
- An aircraft *has* a tail number.
- An aircraft *has* a maximum cargo weight.
- A 747 *is* a type of aircraft.

And so on…

Notice that some of these facts describe how one term relates to another term; for example, an aircraft *carries* a cargo shipment. This usually provides a clue that the terms in question, aircraft and cargo shipment, are entities and are two of the primary terms we are interested in identifying.

Also notice that some facts describe what Business Rule Solutions, LLC (BRS) calls "has a" relationships; for example, an aircraft "has a" tail number, or a cargo "has a" weight. This type of relationship usually identifies the subject (aircraft) as an entity and the object (tail number) as an attribute of that entity. By continuing the analysis, we discover that the problem reduces to a Vocabulary containing 3 main entities, each with its own set of attributes:

Entity: aircraft

Attributes: aircraft type, max cargo weight, max cargo volume, tail number

Entity: cargo shipment

Attributes: weight, volume, manifest number, packaging

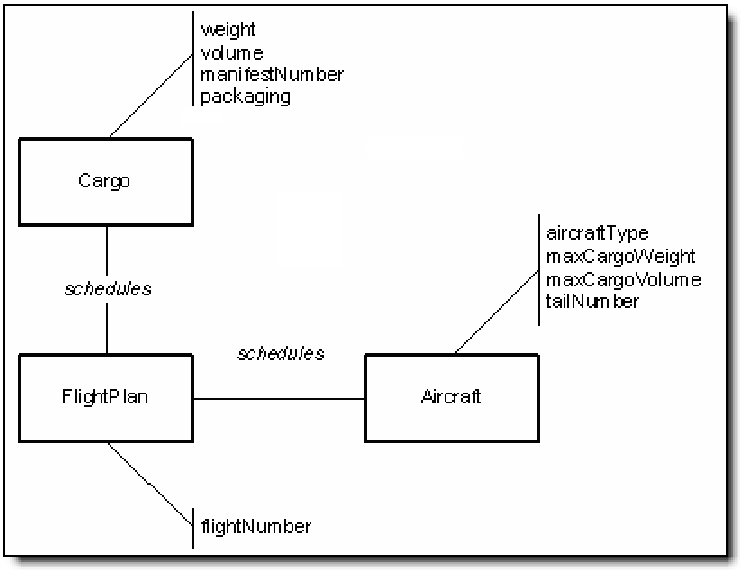
Entity: flight plan

Attributes: flight number

Step 4 – Diagramming the Vocabulary

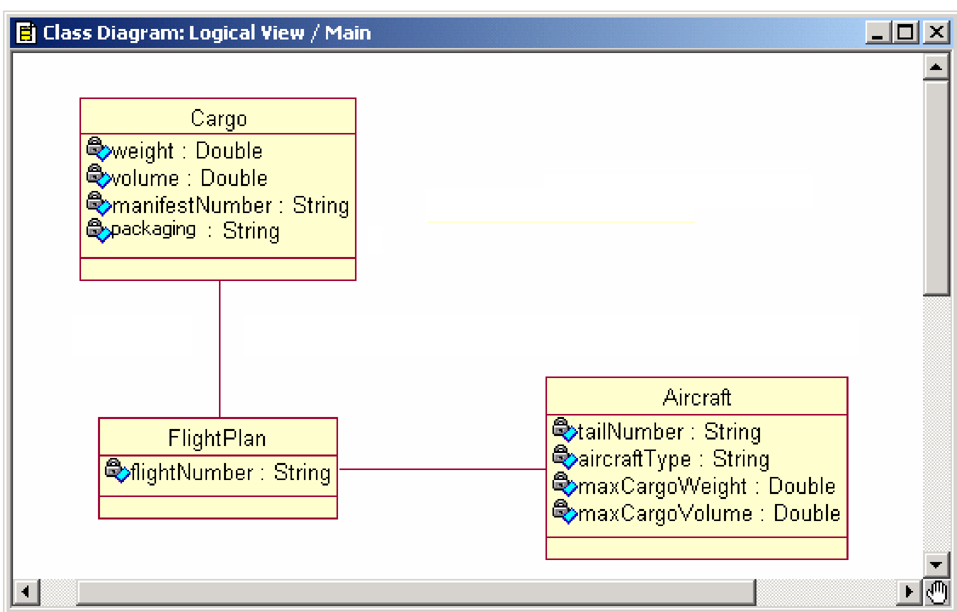
Using this breakdown, we can sketch a simple Fact Model that illustrates the entities and their relationships, or *associations*. In our Fact Model, we will represent entities as rectangular boxes, associations between entities as straight lines connecting the entity boxes, and entity-to-attribute relationships as a diagonal line from the associated entity. The resulting Fact Model appears below in the following model:

Figure 2: Fact Model



The UML Class diagram contains the same type of information, and may be more familiar to you:

Figure 3: UML Class Diagram



It is not a requirement to construct diagrams or models of the Vocabulary before building it in Corticon Studio. But it can be very helpful in organizing and conceptualizing the structures and relationships, especially for very large and complex Vocabularies. The BRS Fact Model and UML Class Diagram are appropriate because they remain sufficiently abstracted from lower-level data models which contain information not typically required in a Vocabulary.

Modeling the Vocabulary in Corticon Studio

Our next step is to transform the diagram into our actual Vocabulary. This can be done directly in Corticon Studio using the built-in **Vocabulary Editor** feature.

Refer to the *Quick Reference Guide's* "Vocabulary" chapter for complete details on building a Vocabulary inside Studio.

The following considerations apply to this transformation process:

- The same naming conventions for entities and attributes used in the Fact Model will also be used in the Vocabulary.
- All attributes in our Vocabulary must have a data type specified. These may be any of the following common data types: **String**, **Boolean**, **DateTime**, **Date**, **Time**, **Integer** or **Decimal.**
- Attributes may be classified according to the method by which their values are assigned. They may be either:
  - **Base** (in other words, obtained directly from input data or request message) or
  - **Extended Transient** (created, derived, or assigned by rules in Studio.

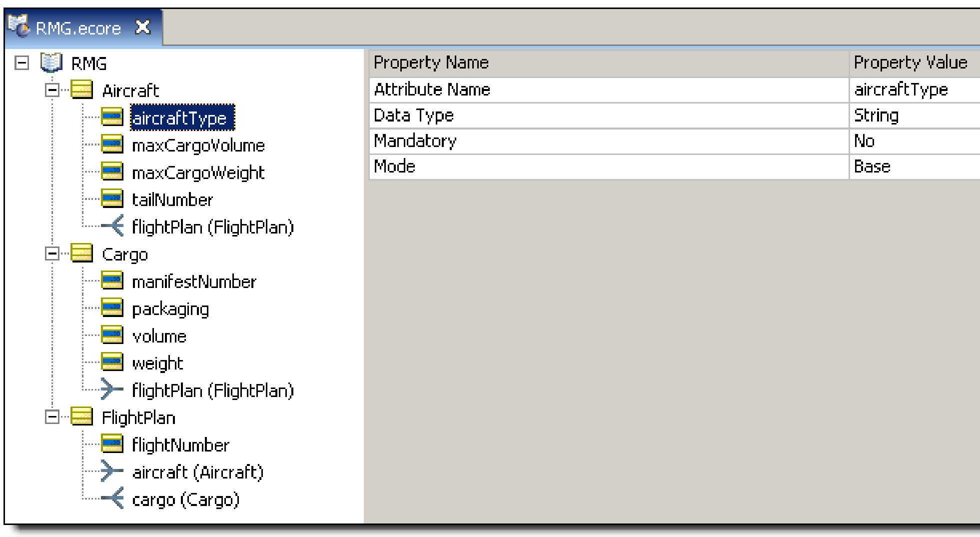
    It may be convenient to use a naming convention to distinguish transient attributes from base; for instance, we might add a lowercase letter `d` to the start of `totalWeight` to indicate it is a derived attribute. However, we caution against modifying the names of terms too much, since the intent is to express them in a language accessible to business users, not just developers.

- Associations between entities have role names that are assigned when building the associations in the UML class diagram or Vocabulary Editor. Default role names simply duplicate the entity name with the first letter in lowercase. For example, the association between the `Cargo` and `FlightPlan` entities would have a role name of "flightPlan" as seen by the `Cargo` entity, and "cargo" as seen by the `FlightPlan` entity. Roles are useful in clarifying context in a rule – a topic covered in more detail within the Scope chapter.
- Associations between entities can be directional (one-way) or bi-directional (two-way). If the association between `FlightPlan` and `Aircraft` were directional (with `FlightPlan` as the "source" entity and `Aircraft` as "target"), we would only be able to write rules that traverse *from* `FlightPlan` *to* `Aircraft`, but not the other way. This means that a rule <u>may use</u> the Vocabulary term `flightPlan.aircraft.tailNumber` but <u>may not use</u> `aircraft.flightPlan.flightNumber`. Bi-directional associations allow us to traverse the association in either direction, which clearly allows us more flexibility in writing rules. Therefore, it is strongly recommended that all associations be bi-directional whenever possible. New associations are bi-directional by default.
- Associations also have cardinality, which indicates how many instances of a given entity may be associated with another entity. For example, in our air cargo scenario, each instance of `FlightPlan` will be associated with only one instance of `Aircraft`, so we can say that there is a "one-to-one" relationship between `FlightPlan` and `Aircraft`. The practice of specifying

cardinality in the Vocabulary deviates from the UML Class modeling technique because the act of assigning cardinality can be viewed as defining a constraint-type rule. For example, "a `flightPlan` schedules exactly one `aircraft` and one `cargo` shipment" is a constraint-type business rule that can be implemented in a Corticon Studio as well as "embedded" in the associations within a Vocabulary. In practice, however, it may often be more convenient to embed these constraints in the Vocabulary, especially if they are unlikely to change in the future.

- Another consideration when creating a Vocabulary is whether derived attributes must be saved (or persisted) external to Corticon Studio, for example, in a database. It is important to note that while the structure of your Vocabulary may closely match your data model (often persisted in a relational database), the Vocabulary is *not required* to include all of the database entities/tables or attributes/columns, especially if they will not be used for writing rules. Conversely, our Vocabulary may contain attributes that are used only as transient variables in rules and that do not correspond to fields in an external database.
- Finally, the Vocabulary must contain all of the entities and attributes needed to build rules in Corticon Studio that reproduce the decision points of the business process being automated. This will most likely be an iterative process, with multiple Vocabulary changes being made as the rules are built, refined, and tested. It is very common to discover, while building rules, that the Vocabulary does not contain necessary terms. But the flexibility of Corticon Studio permits the rule developer to update or modify the Vocabulary immediately, without programming.

Figure 4: Vocabulary Window in Corticon Studio



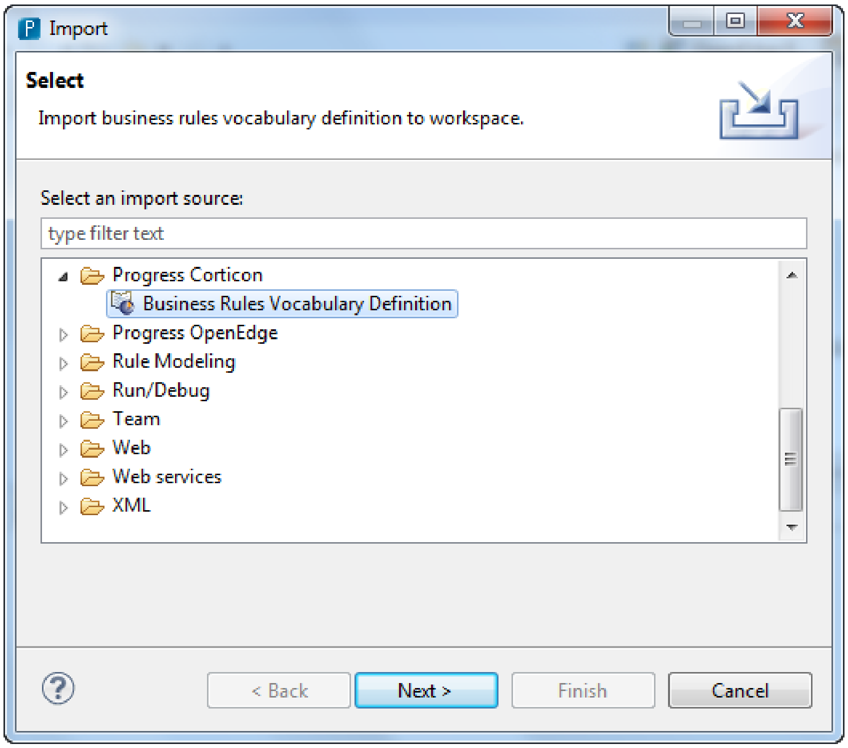
Note: In this figure, Corticon Studio is shown in **Rule Modeling** mode. If in **Integration Deployment** mode, the **Property Name** column will contain additional rows. For more information on Integration Deployment mode, see the *Corticon Server: Integration & Deployment Guide*.

Importing an OpenEdge Business Rules Vocabulary Definition (BRVD) file

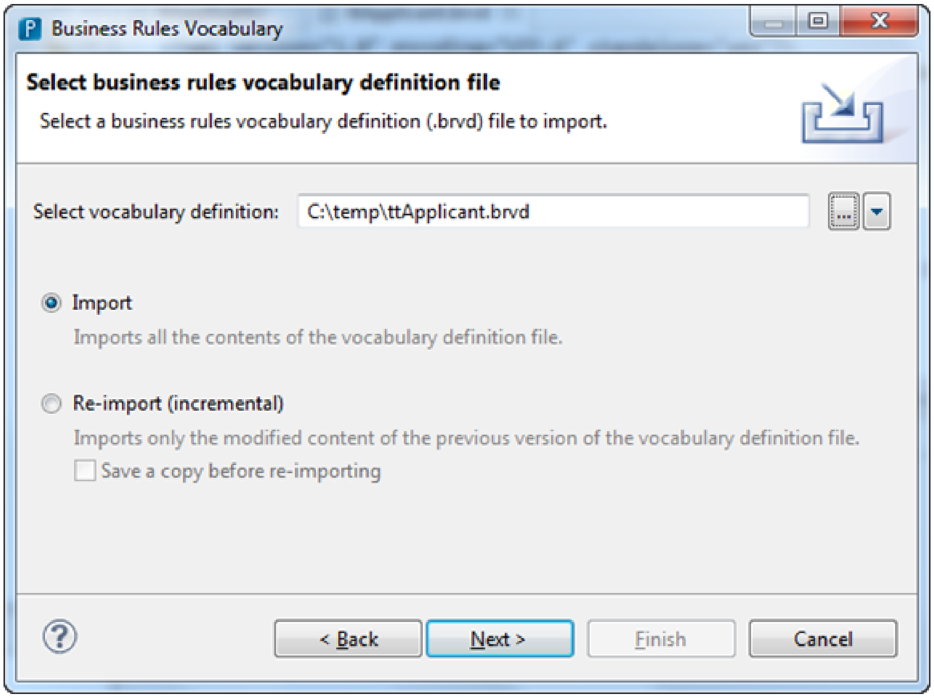
A schema exported from Progress OpenEdge can be imported and used as the basis for Vocabulary entities and attributes in Corticon Studio.

To import a Vocabulary definition created in OpenEdge into the Corticon perspective:

1. In the integrated OpenEdge/Corticon Eclipse development environment, choose the menu command **Window > Open Perspective > Corticon Designer**.
2. Choose the menu command **File > New > Rule Project**, name the project -- in this example, `CorticonProject` -- and then click **Finish**.
3. Choose the menu command **File > New > Progress Corticon > Rule Project**.
4. In the **New Corticon Project** wizard, specify the name and location of the project.
5. Optionally, add the project to working sets and selected project references.
6. Click **Finish**. The new project is created and displayed in the **Project Explorer** view.
7. Click on the project name, and then choose the menu command **File > Import > Import**
8. In the **Import** dialog, expand **Progress Corticon**, and then click on **Business Rules Vocabulary Definition**, as shown:

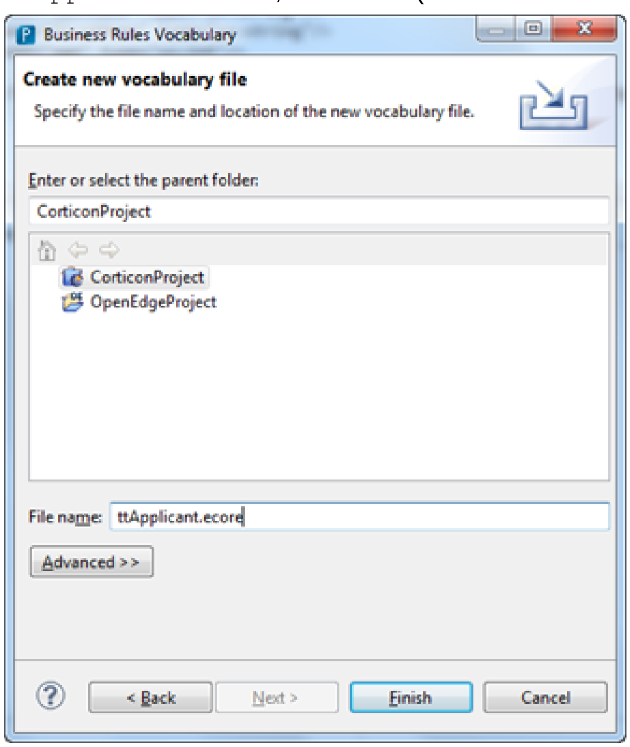


9. In the **Business Rules Vocabulary** dialog, locate the `.brvd` file -- in this example, `ttApplicant.brvd`, that was staged in the `c:\temp` folder -- that was created in OpenEdge.
10. Select **Import**, as shown:



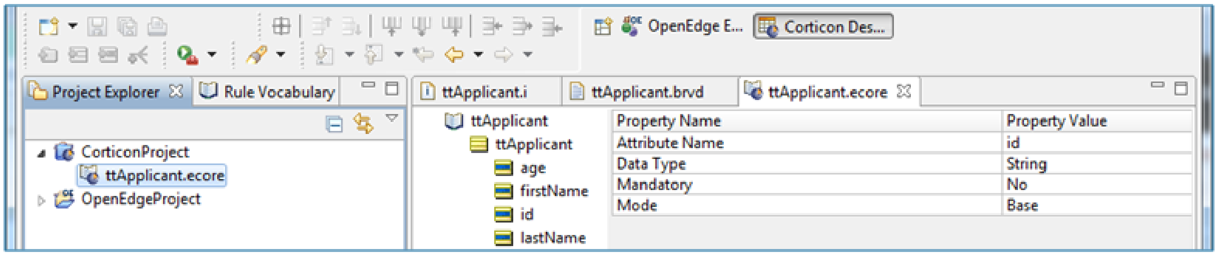
Note: Once you have completed an import, subsequent updates to the Vocabulary should use the **Re-import** function. It is a good practice to choose the **Save a copy before re-importing** to backup the existing Vocabulary before applying changes to it.

11. Click **Next**.
12. Select the **CorticonProject**, and then enter the Vocabulary **File name** -- in this example, `ttApplicant.ecore`, as shown. (The name must have the `.ecore` extension.)



13. Click **Finish**.

14. Double-click on the `.ecore` file name in the Project Explorer to open it in the Corticon Vocabulary editor. The example looks like this:



15. Review the Vocabulary to ensure that it represents the exported data correctly.

The import processing of the OpenEdge BRVD file into a Corticon Vocabulary is complete.

You can now create and test Rulesheets and Ruleflows, and then publish a Decision Service to Corticon Server (such as the one that runs in an OpenEdge Web Server).

See the Progress OpenEdge documentation for details about a complete end-to-end workflow involved in an integrated OpenEdge Business Rules environment.

Constraints on OpenEdge BRVD Vocabularies

OpenEdge developers can use Corticon for their business rules, using Progress Developer Studio to integrate their ABL projects with Corticon Decision Services. During import of a Business Rules Vocabulary Definition (BRVD) created in Open Edge, a Corticon mechanism flags vocabulary entities and attribute as read-only. This prevents you from accidentally changing an entity or attribute derived from OpenEdge, thereby invalidating calls from OpenEdge. You are alerted when you attempt to perform an action that was prevented from writing to the OpenEdge database.

Custom Data Types

Corticon uses seven basic data types: Boolean, Decimal, Integer, String, DateTime, Date, and Time. An attribute must use one of these types. Yet you also have the option of creating custom data types that "extend" any one of these basic seven.

You define and maintain Custom Data Types in a Vocabulary by selecting the Vocabulary name in the tree view.

Data Type Name

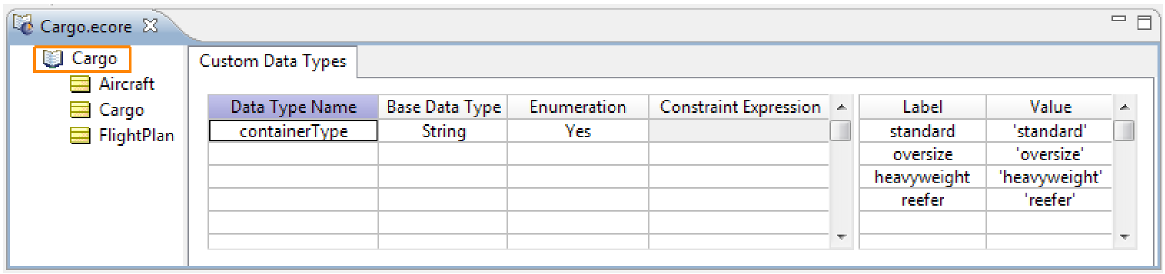
When defining a custom data type, you must give it a name. The name must comply with standard entity naming conventions (see the *Quick Reference Guide* for details) and must not overlap (match) any of the base data types, any other custom data type names, or the names of any Vocabulary entities.

Base Data Type

The selection in this field determines which base data type the custom data type extends.

We already used this feature in the custom data type `containerType`, a `String`, in the *Basic Tutorial*. It lists its labels and values.

Figure 5: Vocabulary Editor Showing the Custom Data Type containerType



Enumeration Or Constraint Expression?

Enumeration - When the **Enumeration** for a Custom Data Type is set to `Yes`, as shown above, the **Constraint Expression** field is disabled, and the **Label** and **Value** columns are enabled.

Constraint Expression - When the **Enumeration** for a Custom Data Type is set to `No`, the **Constraint Expression** field is enabled and the **Label** and **Value** columns are disabled.

The following sections explore each of these features.

Constraint Expressions

When you want to prompt Rulesheet and Ruletest designers to use a specific range values for an attribute, a constraint expression will validate entries when the associated Ruletest runs.

Constraint expressions are optional for non-enumerated Custom Data Types, but if none are used then the Custom Data Type probably isn't necessary because it reduces to a base attribute with a custom name.

All **Constraint Expressions** must be `Boolean` expressions, in other words they must return or resolve to a Boolean value of **`true`** or `false`. The supported syntax is the same as Filter expressions with the following rules and exceptions:

- Use the `value` to represent the Custom Data Type value.
- Logical connectors such as `and` and `or` are supported
- Parentheses may be used to form more complex expressions
- The expression may include references to Base and Extended Operators which are compatible with the Base Data Type chosen.
- No Collection operators may be referenced in the expression.
- There should be NO references to `null`. This is because `null` represents a lack of value and is not a real value. The Constraint Expression is intended to constrain the value space of the data type and expressions such as attribute `expression <> null` do not belong in it. An attribute that must not have a null value can be so designated by selecting `Yes` in its **Mandatory** property value.

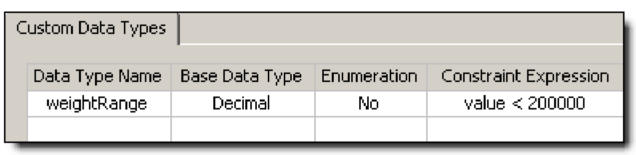
The following are typical Constraint Expressions:

Constraint Expression	Meaning
value > 5	Integer values greater than 5
value >= 10.2	Decimal values greater than or equal to 10.2
value in (1.1..9.9]	Decimal values between 1.1 (exclusive) and 9.9 (inclusive)
value in [‘1/1/2000 12:30:00 PM’..’1/2/2000 11:00:00 AM’)	DateTime values between ‘1/1/2000 12:30:00 PM’ (inclusive) and ‘1/2/2000 11:00:00 AM’ (exclusive)
value in [‘1:00:00 PM’..’2:00:00 PM’]	Time values between ‘1:00:00 PM’ (inclusive) and ‘2:00:00 PM’ (inclusive)
value.size >= 6 and (value.indexOf(1) > 0 or value.indexOf(2) > 0)	String values of minimum 6 characters in length that contain at least a 1 or 2

Using non-enumerated Custom Data Types in Rulesheets and Ruletests

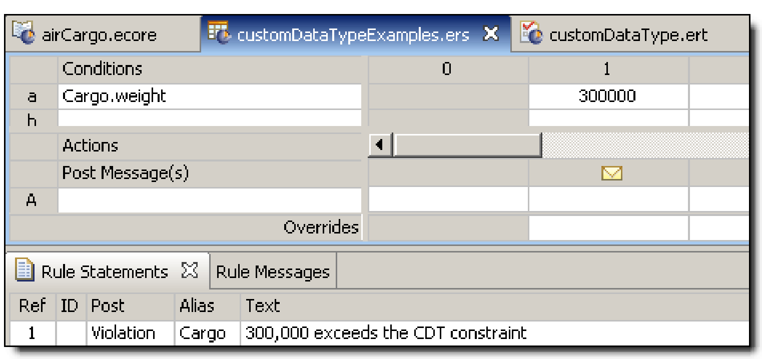
Non-enumerated custom data types use **Constraint Expressions** and do not cause Rulesheet drop-downs to become populated with custom sets. Also, manually entering a cell value that violates the custom data type's **Constraint Expression** is not prohibited in the *`Rulesheet`*. For example, in the example below, `weightLimit` is defined as a non-enumerated custom data type with **Base Data Type** of `Integer`.

Figure 6: Non-enumerated Custom Data Types



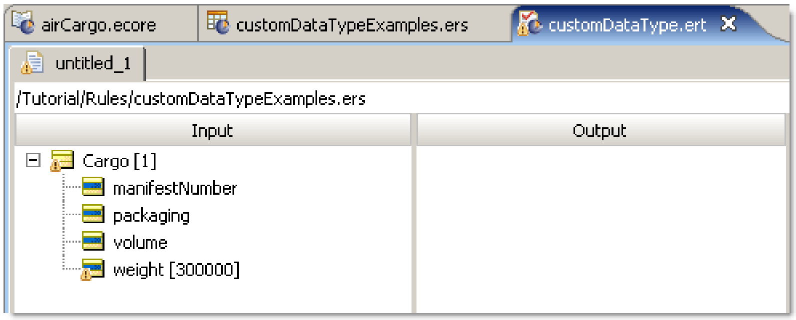
Then, after assigning it to the Vocabulary attribute `Cargo.weight`, it is used in a *`Rulesheet`* Condition row as shown below:

Figure 7: Using Custom Data Types in a Rulesheet



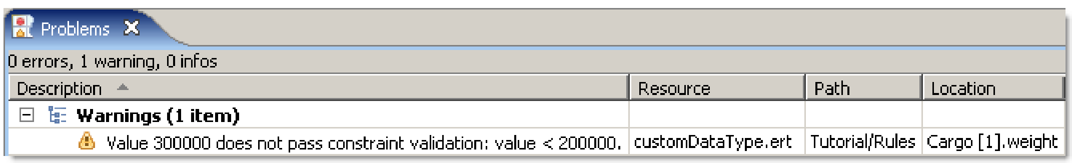
Notice in `Using Custom Data Types in a Rulesheet` that the `55000` entry violates the **Constraint Expression** of the custom data type assigned to `Cargo.weight`, but *does not turn red or otherwise indicate a problem*. The indication comes when data is entered for the attribute in a `Ruletest`, as shown below:

Figure 8: Violating a Custom Data Type's Constraint Expression



Notice that the small yellow warning icon indicates a problem in the attribute, entity, and both `Ruletest` tabs. Such an error is hard to miss! Also, a Warning message will appear in the **Problems** tab (if open and visible) as shown below. If the Problems tab is closed, you can display it by selecting **Window** > **Show View** > **Problems** from the ***Studio*** menubar.

Figure 9: Violating the Constraint Expression of a Custom Data Type



A Warning will not prevent you from running the `Ruletest`. However, an Error, indicated by a small red icon, will prevent the Ruletest execution. You must fix any errors before testing.

Enumerations

When you want to prompt Rulesheet and Ruletest designers to use a specific list of values, you can specify an explicit list, either maintained directly in the Vocabulary, or retrieved and updated from a database.

Enumerations defined in the Vocabulary

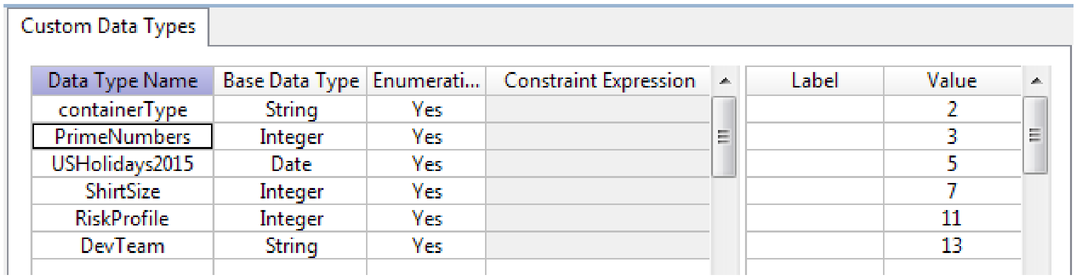
If your custom data type is a local enumeration, then you need to enter the enumerated values in the **Label** and **Value** columns.

The **Label** column is optional: you enter **Labels** only when you want to provide an easier-to-use or more intuitive set of names for your enumerated values.

The **Value** column is mandatory: you need to enter the enumerations in as many rows of the **Value** column as necessary to complete the enumerated set. Be sure to use normal syntax, so custom data types that extend String, DateTime, Date, or Time base data types must be enclosed in single quote characters.

Here are some examples of enumerated custom data types:

Figure 10: Custom Data Type, example 1

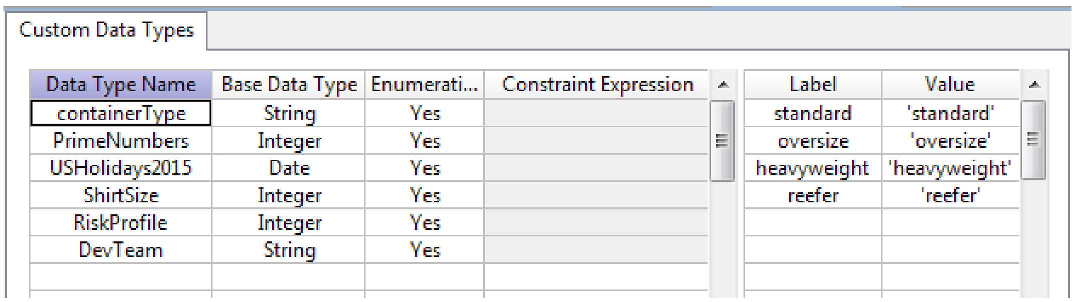

Custom Data Types

Data Type Name	Base Data Type	Enumerati...	Constraint Expression
containerType	String	Yes	
PrimeNumbers	Integer	Yes	
USHolidays2015	Date	Yes	
ShirtSize	Integer	Yes	
RiskProfile	Integer	Yes	
DevTeam	String	Yes	

Label	Value
	2
	3
	5
	7
	11
	13

`PrimeNumbers` is an Integer-based, enumerated custom data type with Value-only set members.

Figure 11: Custom Data Type, example 2

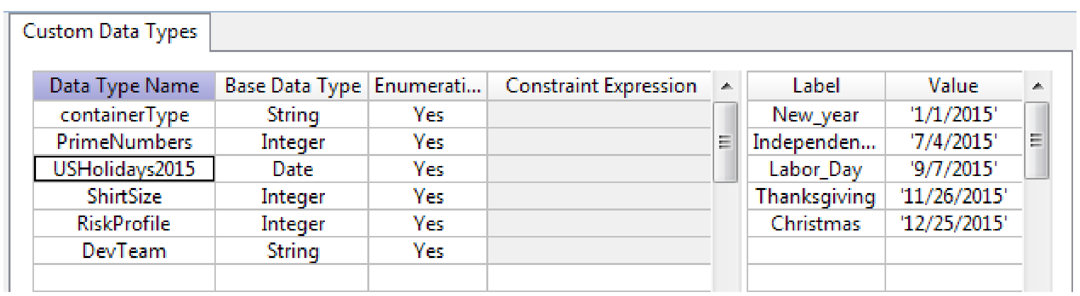

Custom Data Types

Data Type Name	Base Data Type	Enumerati...	Constraint Expression
containerType	String	Yes	
PrimeNumbers	Integer	Yes	
USHolidays2015	Date	Yes	
ShirtSize	Integer	Yes	
RiskProfile	Integer	Yes	
DevTeam	String	Yes	

Label	Value
standard	'standard'
oversize	'oversize'
heavyweight	'heavyweight'
reefer	'reefer'

`packingType` is a String-based, enumerated custom data type with Label/Value pairs.

Figure 12: Custom Data Type, example 3

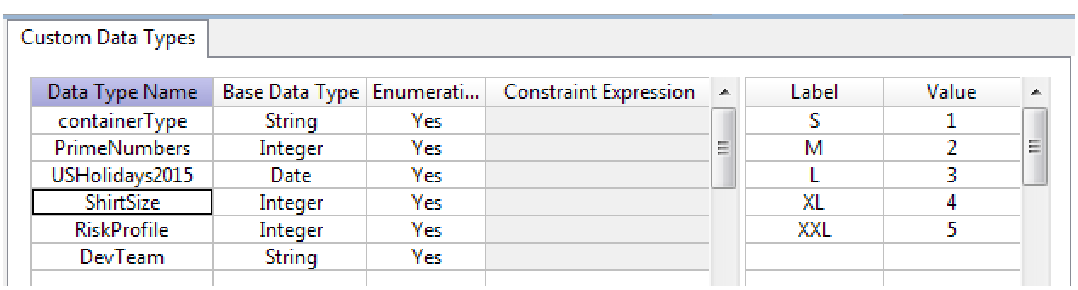

Custom Data Types

Data Type Name	Base Data Type	Enumerati...	Constraint Expression
containerType	String	Yes	
PrimeNumbers	Integer	Yes	
USHolidays2015	Date	Yes	
ShirtSize	Integer	Yes	
RiskProfile	Integer	Yes	
DevTeam	String	Yes	

Label	Value
New_year	'1/1/2015'
Independen...	'7/4/2015'
Labor_Day	'9/7/2015'
Thanksgiving	'11/26/2015'
Christmas	'12/25/2015'

`USHolidays2015` is a Date-based, enumerated custom data type with Label/Value pairs.

Figure 13: Custom Data Type, example 4

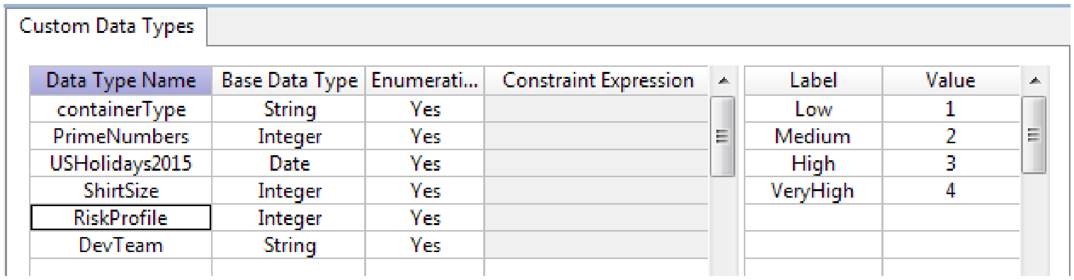

Custom Data Types

Data Type Name	Base Data Type	Enumerati...	Constraint Expression
containerType	String	Yes	
PrimeNumbers	Integer	Yes	
USHolidays2015	Date	Yes	
ShirtSize	Integer	Yes	
RiskProfile	Integer	Yes	
DevTeam	String	Yes	

Label	Value
S	1
M	2
L	3
XL	4
XXL	5

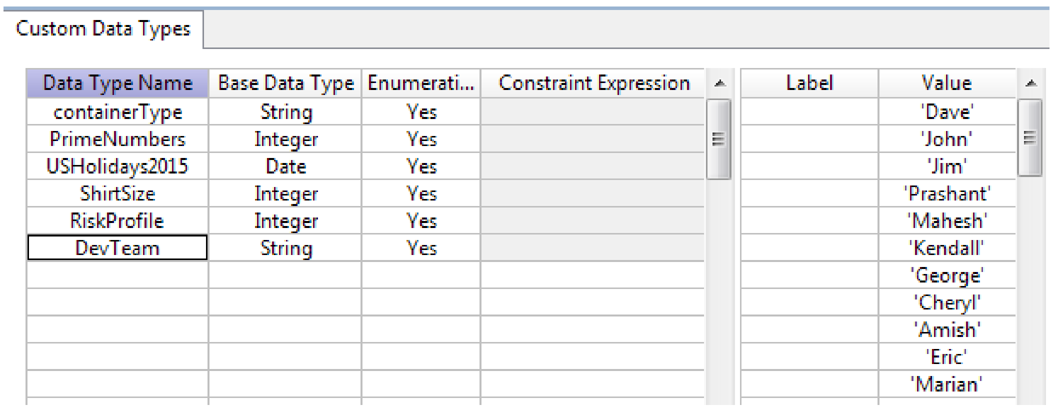
`ShirtSize` is an Integer-based, enumerated custom data type with Label/Value pairs.

Figure 14: Custom Data Type, example 5



`RiskProfile` is an Integer-based, enumerated custom data type with Label/Value pairs

Figure 15: Custom Data Type, example 6



`DevTeam` is a String-based, enumerated custom data type with Value-only set members.

Use the **Move Up** or **Move Down** toolbar icons to change the order of Label/Value rows in the list.

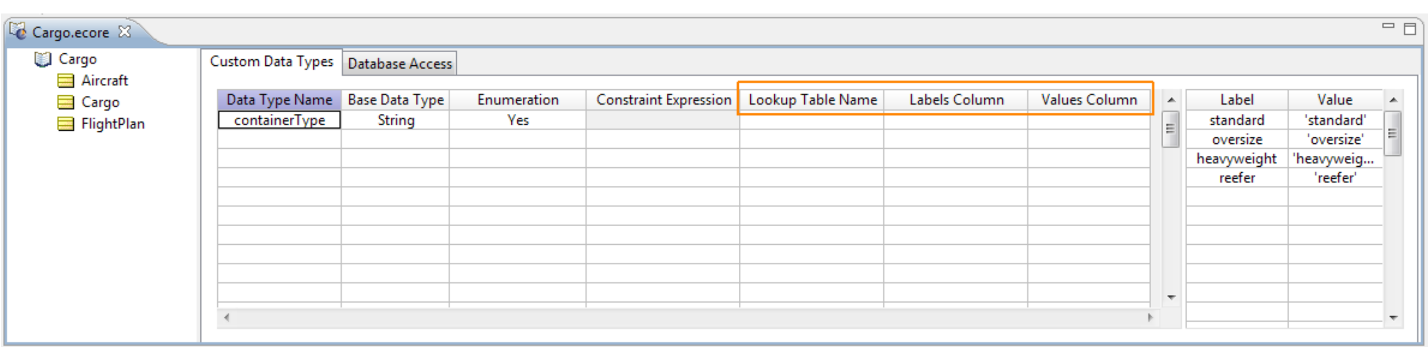
Enumerations retrieved from a database

If you want your custom data type to gets its enumerated labels and values from a database, then you need to define the database table and columns that will be accessed.

This topic covers the significant points of this feature in the context of the Vocabulary.

When you have EDC enabled -- you are in Integration and Deployment mode, and you have a verified connection to a supported database -- the **Custom Data Types** tab presents three additional columns, as shown:

Figure 16: Custom Data Type columns for defining database retrieval

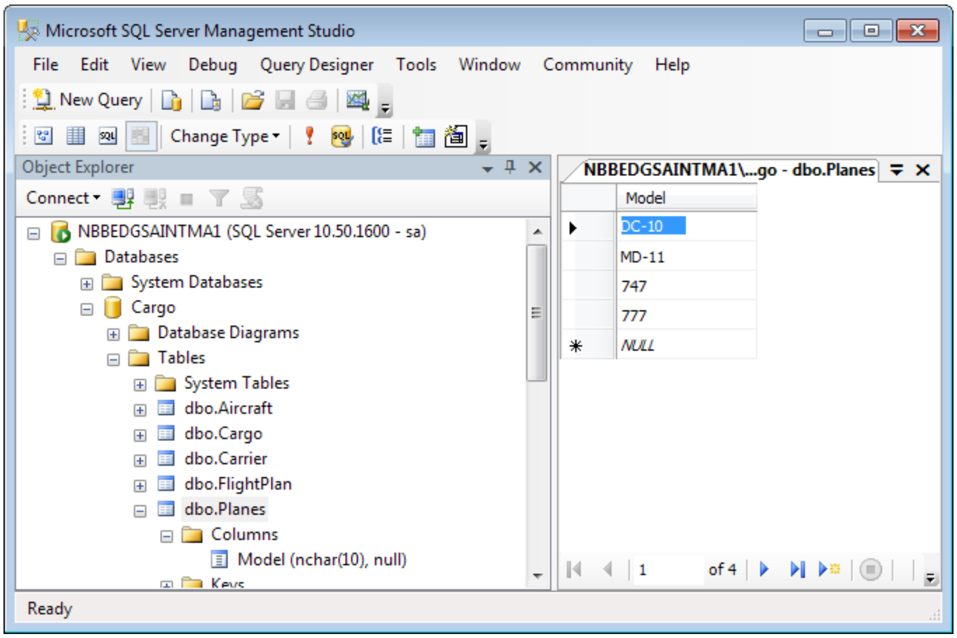


These columns are how you specify:

- **Lookup Table Name** - The database syntax that specifies the table that has the enumerations.
- **Labels Column** - The column in the lookup table that holds the label. This is optional as you can elect to use only values.
- **Values Column** - The column in the lookup table that holds the value associated with the label, or the solitary value. This is required.

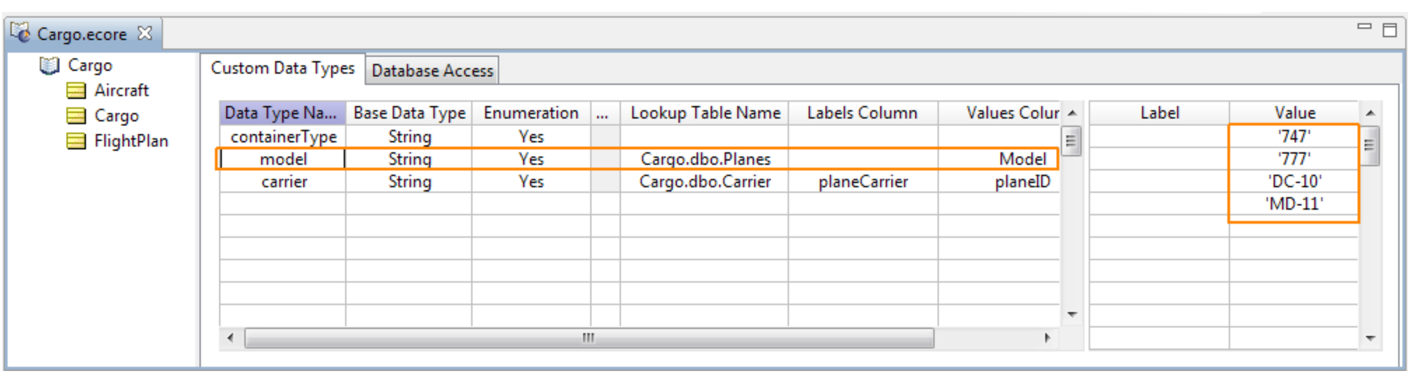
The following examples show two options:

Figure 17: SQL Server table with values to use in the Vocabulary



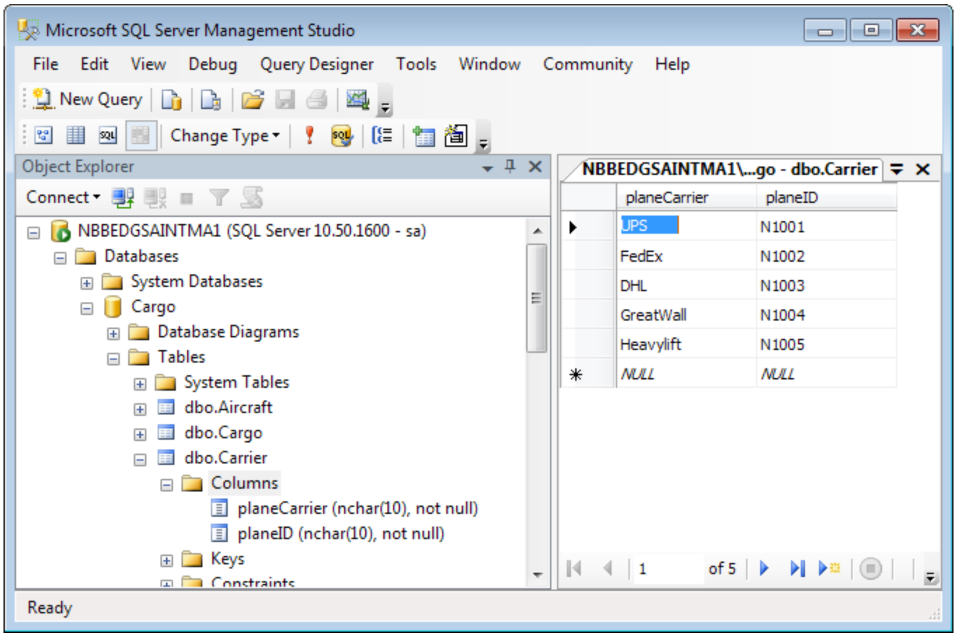
The value data is retrieved into the Vocabulary as highlighted:

Figure 18: Definition and retrieved values in the Corticon Studio



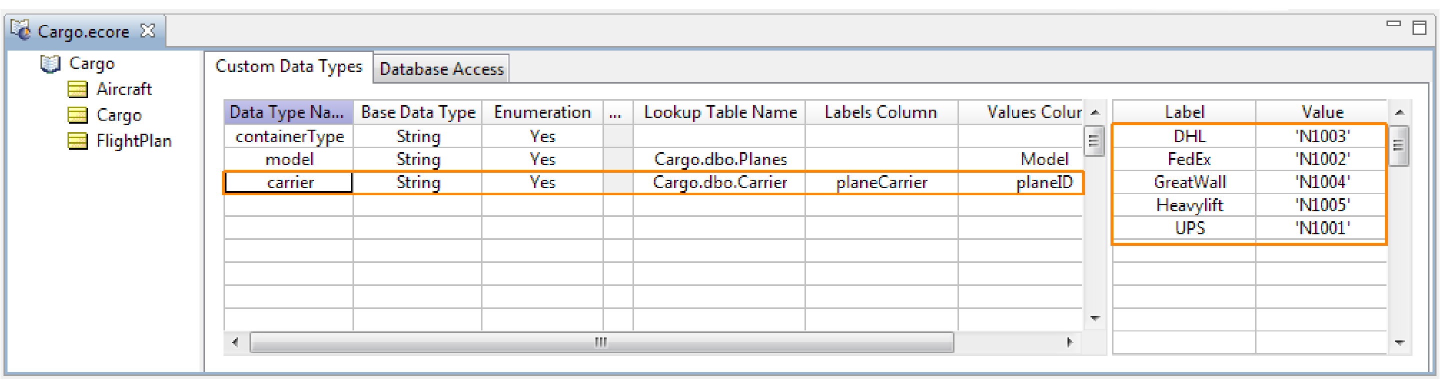
Another example retrieves name-value pairs.

Figure 19: SQL Server table with labels and values to use in the Vocabulary



The label-value data is retrieved into the Vocabulary as highlighted:

Figure 20: Definition and retrieved label-values in the Corticon Studio

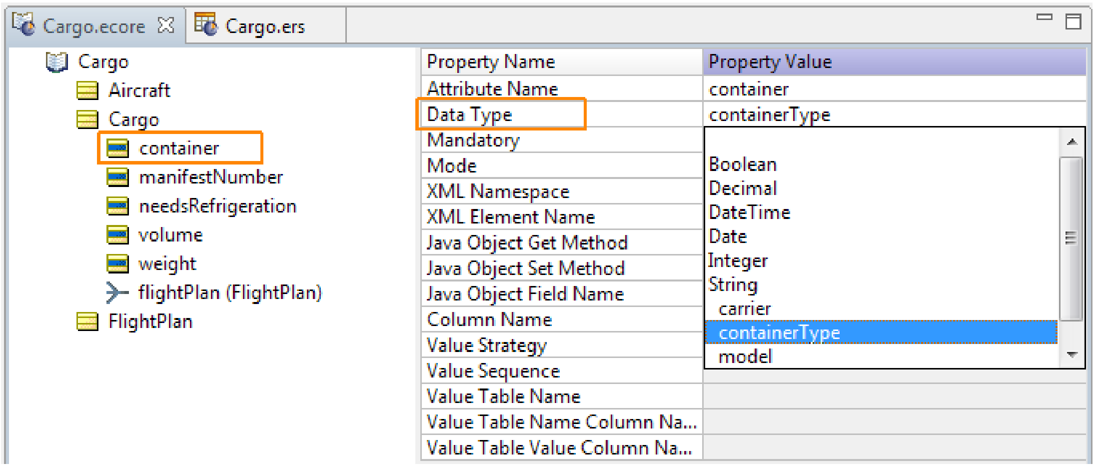


Note: This functionality uses Corticon's Enterprise Data Connector. A section of the EDC tutorial covers this topic in detail, *"Importing an attribute's possible values from database tables" in the Using EDC Guide*

Using Custom Data Types in a Vocabulary

Once a Custom Data Type has been defined as shown above, it may be used and reused throughout the Vocabulary's attribute definitions.

Figure 21: Using Custom Data Types in the Vocabulary

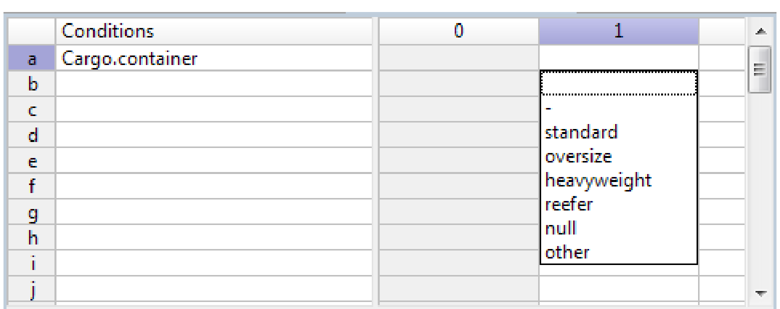


Notice in this figure that multiple attributes can use the same custom data type; the.custom data type `containerType` is shown in the drop-down as a sub-category of the String-based data type. The other custom data types will be grouped with their base data types as well.

Using enumerated Custom Data Types in Rulesheets

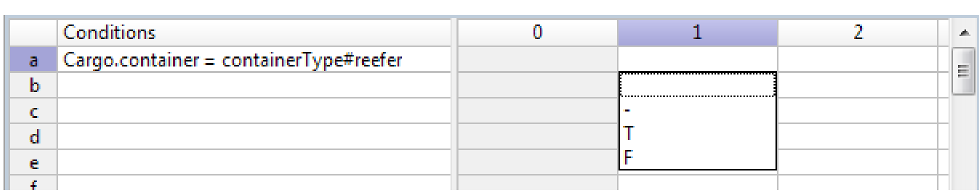
Once an enumerated, custom data type has been defined and assigned to an attribute, its labels are displayed in selection drop-downs in both Conditions and Actions expressions, as shown below. If **Labels** are not available (since Labels are optional in an enumerated custom data type's definition), then **Values** are shown. The `null` option in the drop-down is only available if the attribute's **Mandatory** property value is set to `No`.

Figure 22: Using Custom Data Types in the Rulesheet



You can test a condition bound to an attribute by evaluating the attribute against a custom data type value using the # tag, as shown:

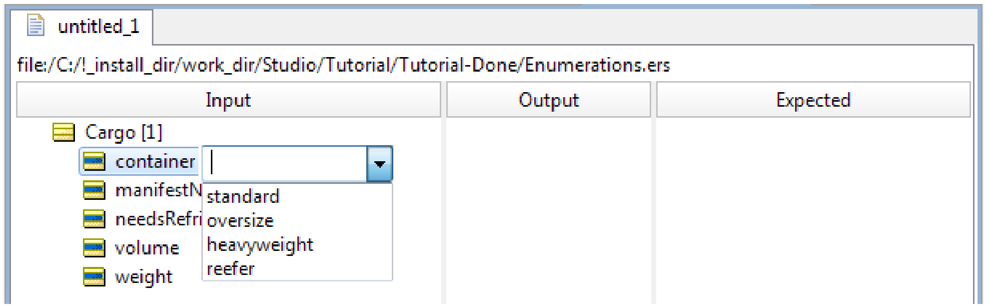
Figure 23: Using # tag to test a custom data type



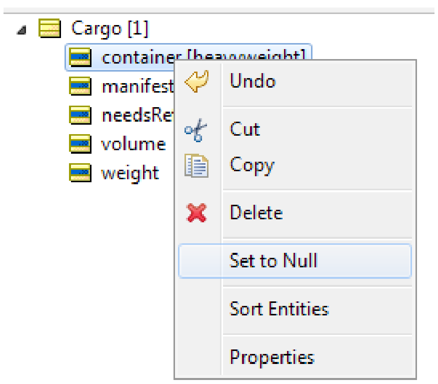
Using enumerated Custom Data Types in Ruletests

Once an enumerated Custom Data Type has been defined and assigned to an attribute, its enumerated Labels or Values become available as selectable inputs in a Ruletest, as shown:

Figure 24: Ruletest selecting container's `containerType` list



If you want the attribute value to be null, right-click on the attribute and then select **Set to Null**, as shown:



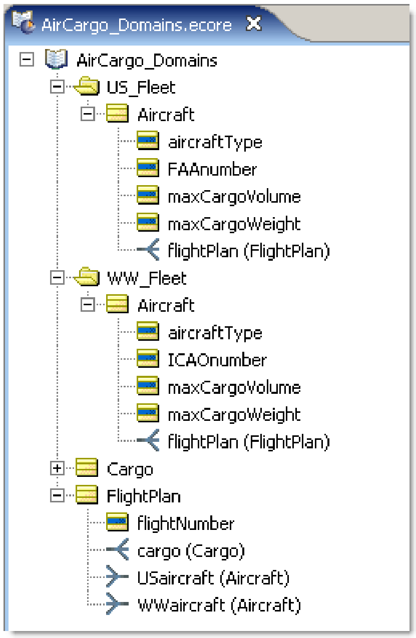
Domains

Occasionally, it may be necessary to include more than one entity of the same name in a Vocabulary. This can be accomplished using *Domains* (similar to Java *packages* and XML *namespaces*.) Domains allow us to "bundle" one or more entities in a *subset* within the Vocabulary, allowing us to reuse entity names so long as the entity names are unique within each Domain. Additional Domains, referred to as *sub-Domains*, can be defined within other Domains.

Select **Vocabulary** > **Add Domain** from the ***`Studio`*** menubar or click from the ***`Studio`*** toolbar, as shown in *`Creating Domains in the Vocabulary`*.

A new folder is listed in the Vocabulary tree. Assign it a name. The example in the following figure shows a Vocabulary with 2 Domains, `US_Fleet` and `WW_Fleet`:

Figure 25: Using domains in the Vocabulary>

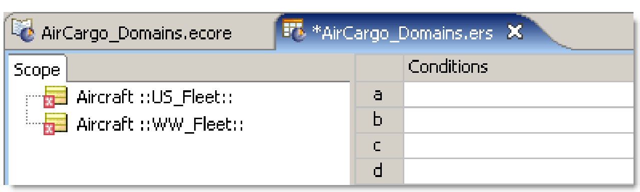


Notice that the entity `Aircraft` appears in each Domain, using the same spelling and containing slightly different attributes (`FAAnumber` vs. `ICAOnumber`). Notice too that the association role names from `FlightPlan` to `Aircraft` have been named manually to ensure uniqueness: one is now `USaircraft` and the other is `WWaircraft`.

Domains in a Rulesheet

When using entities from domains in a *`Rulesheet`*, it is important to ensure uniqueness, which means aliases must be used to distinguish one entity from another.

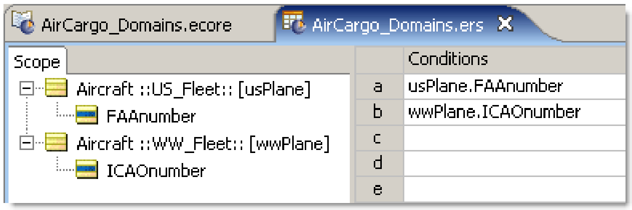
Figure 26: Non-unique Entity names prior to defining Aliases



In *`Non-unique Entity names prior to defining Aliases`*, both `Aircraft` entities have been dropped into the **Scope** section of the *`Rulesheet`*. But because their names are not unique, an error icon appears. Also, the "fully qualified" domain name has been added after each to distinguish them. By fully qualified, we mean the `::US_Fleet::` designator that follows the first Aircraft and `::WW_Fleet::` that follows the second.

But it would be inconvenient (and ugly) to use these fully qualified names in `Rulesheet` expressions. So we require that you define a unique alias for each. The aliases will be used in the Rulesheet expressions, as shown in *Non-unique Entity names after defining Aliases*.

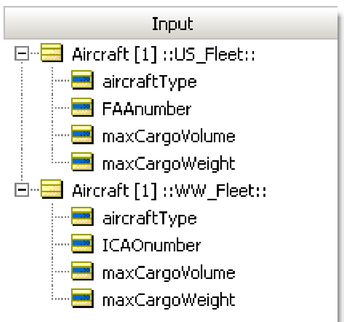
Figure 27: Non-unique Entity names after defining Aliases



Domains in a Ruletest

When using Vocabulary terms in a *Ruletest*, just drag and drop them as usual. You will notice that they are automatically labeled with the fully qualified name, as shown in **Domains in a Ruletest**.

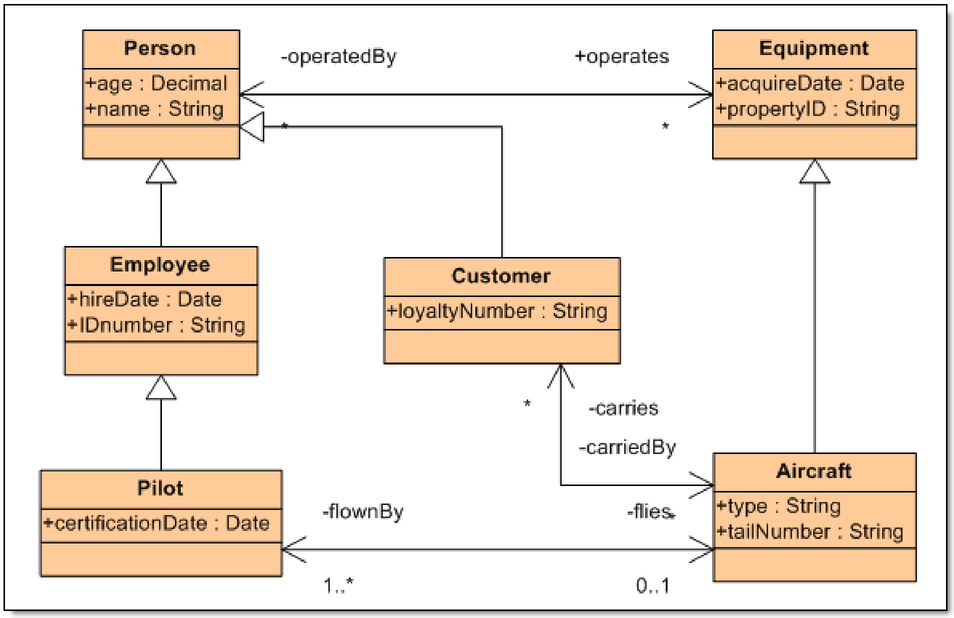
Figure 28: Domains in a *Ruletest*



Support for inheritance

UML Class diagrams frequently include a modeling/programming concept called inheritance, whereby a class may "inherit" attributes and/or associations from another class. For example:

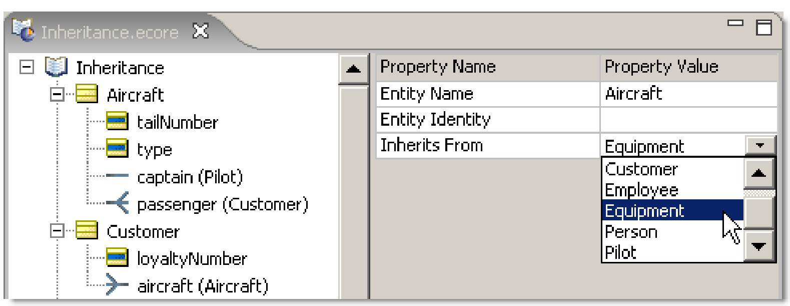
Figure 29: Rose UML Model Showing Inheritance



In this diagram, we see a UML model that includes inheritance. The solid-headed arrow symbol indicates that the `Employee` class is a descendant of the `Person` class, and therefore inherits some of its properties. Specifically, the `Employee` class inherits the `age` and `name` attributes from `Person`. In other words, `Employee` has all the same attributes of a `Person` plus two of its own, `hireDate and IDnumber`. Likewise, `Aircraft` inherits all of `Equipment's` attributes (`acquireDate` and `propertyID`)plus has attributes of its own, `type` and `tailNumber`.

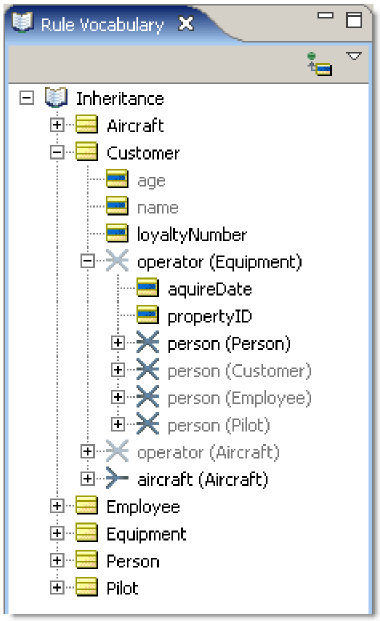
Modeling this UML Class Diagram as a *Studio* Vocabulary is straightforward. All Entities, Attributes and Associations are created as per normal practice. To incorporate the elements of inheritance, we only need add one additional setting for each of the descendant entities, as shown:

Figure 30: Selecting Ancestor Entity for Descendant



Once all descendant entities have been configured to inherit from their proper ancestor entities, we can save the Vocabulary and view it in the **Rule Vocabulary** window:

Figure 31: Vocabulary with Inheritance



Notice that many of the term names and icons are varying shades of gray - these color codes help us to understand the inherited relationships that exist in the Vocabulary.

Inherited Attributes

Attributes with names displayed in **solid black type**, such as `Customer.loyaltyNumber` in *`Vocabulary with Inheritance`*, are "native" attributes of that entity.

Attributes with names displayed in **dark gray type**, such as `Customer.age`, are inherited attributes from the ancestor entity (in the case of `Customer`, `Person`).

Inherited Associations

Inherited Associations are a bit more complicated. An entity may be directly associated with another entity or that entity's descendants. An entity may also inherit an association from its ancestor.

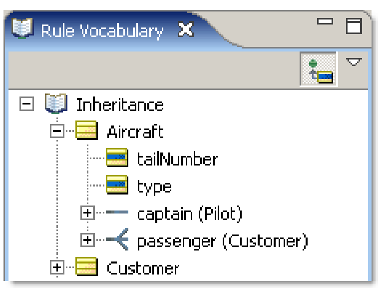
Using the example shown in *`Selecting Ancestor Entity for Descendant`* and *`Vocabulary with Inheritance`* above, let's dissect each of these combinations.

- `Customer.aircraft` is a direct association between `Customer` and `Aircraft` entities. No inheritance is involved, so the association icon is **black** and the rolename is **black**
- `Customer.operator` (Equipment) is an association "inherited" from `Customer's` ancestor entity `Person`, which has a direct association with `Equipment` and the rolename `operator` in our Vocabulary (the UML Class Diagram in *`Selecting Ancestor Entity for Descendant`* shows the rolename as `operates` because it is more conventional in UML to use verbs as rolenames, whereas nouns usually make better rolenames in a Corticon Vocabulary). Because the association is inherited from the ancestor's direct association, the icon is **dark gray** and the rolename is **black**.
- `Equipment` (which we can see equally well in the expanded `operator` rolename) has several associations with `Person`. One of these is a direct association with the `Person` entity. In this case, both association icon and rolename are **black**. But `Equipment` also has associations with descendants of the `Person` entity, specifically `Employee`, `Customer`, and `Pilot`. We call these "filtered" associations, and display their rolenames as **dark gray.**
- Finally, `Customer` has another association with `operator` (Aircraft) because `Aircraft` is a descendant of `Equipment`. So we combine the "inherited" **dark gray** icon and the "filtered" **dark gray** rolename to display this association.

Controlling the tree view

In cases where a Vocabulary contains inheritance (and includes the various icons and color schemes described above) but the modelers who use it do not intend to use inheritance in their rules, the inherited associations and filtered rolenames can be hidden from view by clicking the  icon in the upper right corner of the Rule Vocabulary window, as shown in *`Vocabulary with Inheritance Properties Hidden`*:

Figure 32: Vocabulary with Inheritance Properties Hidden



`Person` and `Equipment` are associated (using named roles), but what relationship does `Employee` have with `Equipment` or `Aircraft`, if any? This version of Corticon Studio supports inherited associations.

Note: In versions 4.1 and earlier, Corticon Studio supported inherited attributes only.

Using aliases with inheritance

Any Entity, Attribute, or Association can be dragged into the Scope section for use in Rulesheets. But if two or more terms share the same name, they must be assigned unique alias names before they can be used in rules.

For example, in *Vocabulary with Inheritance*, we see that there are four `Customer.operator.person` terms in the Vocabulary due to the various forms of inheritance used by the entities and associations. If two or more of these nodes are dragged into the Scope window (as shown in *Non-Unique Nodes used in the Scope Window*), they will be assigned error icons to indicate that their names are not unique. Without unique names, Corticon Studio does not know which one is intended in any rule that uses one of the nodes. To ensure uniqueness, aliases must be assigned and used in rules, as shown in *Uniqueness Established using an Alias*.

Figure 33: Non-Unique Nodes used in the Scope Window

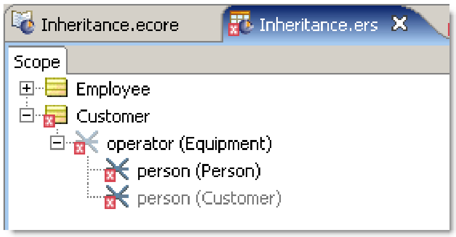
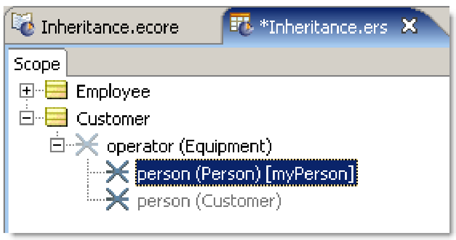


Figure 34: Uniqueness Established using an Alias



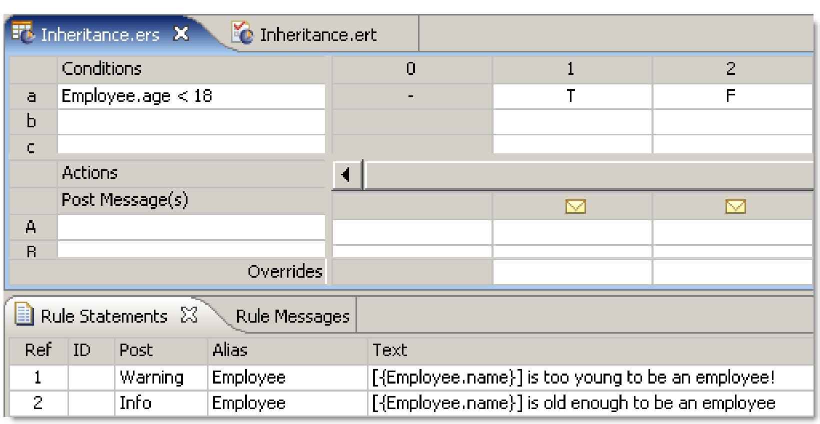
Inheritance's effects on rule execution

The point of inheritance is not to complicate the Vocabulary. The point is to be able to write rules on ancestor entities and have those rules affect descendant entities automatically. Here are simple examples:

Inherited Conditions and Actions

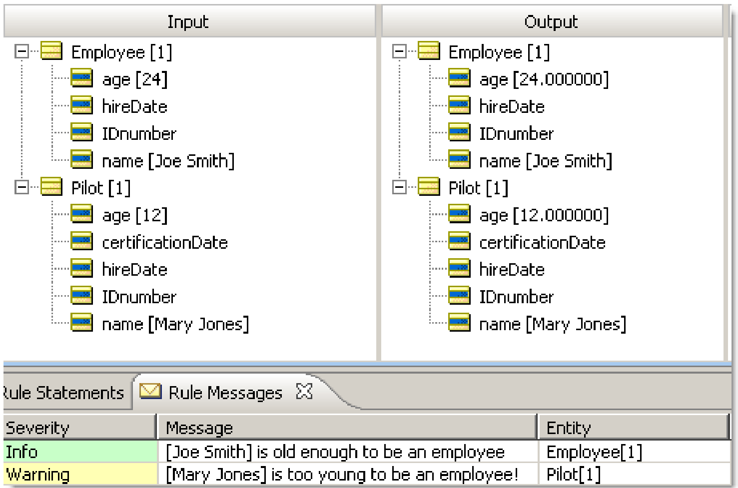
A very simple Rulesheet, shown in *Rules written on Employee*, contains 2 rules that test the `age` value of the `Employee` entity. There are no explicit Actions taken by these rules, only the posting of messages.

Figure 35: Rules written on Employee



A *Ruletest* provides an instance of `Employee` and an instance of `Pilot`. Recall from the Vocabulary that `Pilot` is a descendant of `Employee`, which means it inherits its attributes and associations. But more importantly from a rule execution perspective, a `Pilot` will also be affected by any rules that affect an `Employee`. This is shown in the following figure:

Figure 36: Inheritance in action

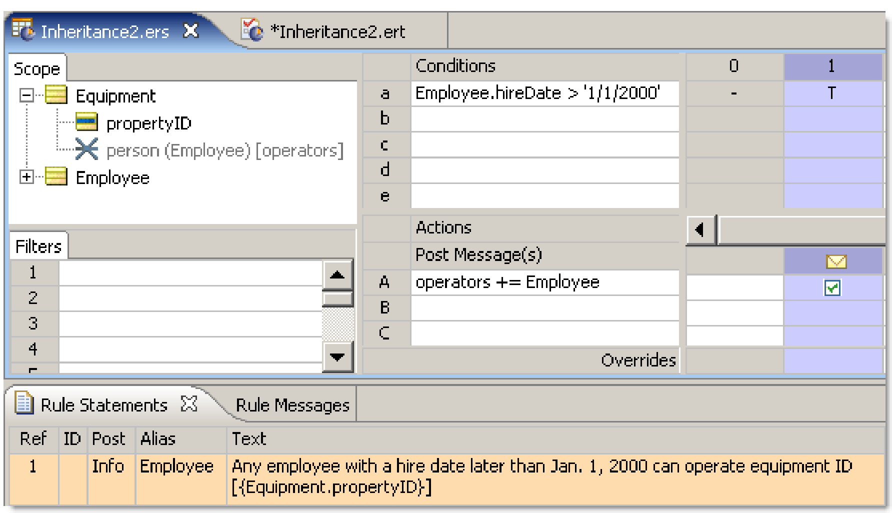


Using inheritance can be an efficient and powerful way to write rules on many different types of employees (such as pilots, gate agents, baggage handlers, and mechanics) without needing to write individual rules for each.

Inherited Association

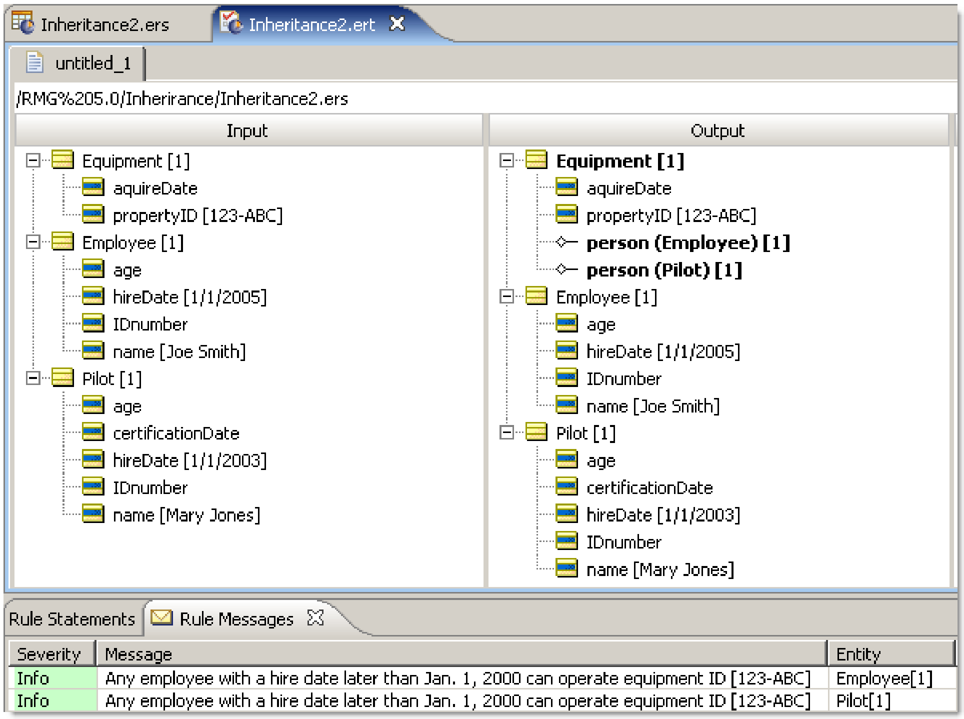
A similar test demonstrates how associations are inherited during rule execution. In this case, we test `Employee.hireDate` to see who's "qualified" to operate a piece of Equipment. The `+=` syntax used by the Action row is explained in more detail in the *Rule Language Guide*.

Figure 37: Rulesheet populating the `operators` collection



Now in, we provide a sample `Equipment` entity, one `Employee`, and one `Pilot`. Both `hireDates` satisfy the rule's Condition (the `Pilot` inheriting `hireDate` from its `Employee` ancestor as before). When the `Employee` is added to the `operators` collection alias, an instance of the association between `Equipment` and `Employee` is created. But what may be surprising is that the same occurs for `Pilot`, which also has an association to `Equipment` that it inherited from `Employee`!

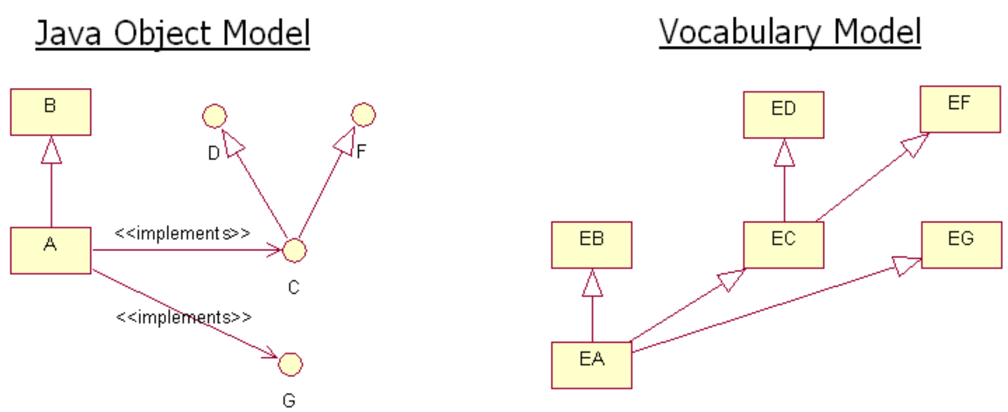
Figure 38: Inheriting an Association



Inheritance and Java object messaging

Each Entity in a Vocabulary can be mapped to a Java Class or Java Interface. Java Classes may have one ancestor. Java Interfaces may have multiple ancestors. A Java Class may implement one or more Interfaces. Say a Java Class A inherits from Java Class B and implements Java Interfaces C & G. Say Java Interface C has as its ancestors Java Interfaces D & F. Say these Classes and Interfaces are mapped to Entities EA, EB, EC, ED, EF & EG in the Vocabulary. The relationships amongst the Java Classes shall be reflected in the Vocabulary using multiple inheritance. Entity EA shall have as its ancestors Entities EB, EC & EG. Entity EC shall have as its ancestors entities ED & EF as shown below:

Figure 39: How the Vocabulary Incorporates Inheritance from a Java Object Model



When a collection of Java objects are passed into the engine through the JOM API, the Java translator determines how to map them to the internal Entities using the following algorithm:

Naming conventions used in the graphic above:

- DS = Decision Service
- JO = Java Object in input collection
- JC = Java Class for the JO and any of its direct or indirect ancestors
- JI = Java Interfaces implemented directly or indirectly by JO
- E = A Vocabulary Entity with no descendents found in DS context
- AE = An Ancestor Entity (one with descendents) found in DS context
- CDO = In memory Java Data Object created by Corticon for use in rule execution

For each E:

- If there is a JO whose JC or JI is mapped to E then
  - Instantiate a CDO for E and link to JO
  - Put CDO in E bucket
- Traverse E's inheritance hierarchy one level up
  - For each AE discovered in current level:
    - Put CDO in AE bucket
- If E has another level of inheritance hierarchy, repeat last step

This design effectively attempts to instantiate the minimum number of CDOs possible and morphs them into playing multiple Entity roles. Ideally, no duplicate copies of input data exists in the engine's working memory thus avoid data synchronization issues.

Test yourself - questions – building the vocabulary

Note: Try this test, and then go to **Answers:** Building the vocabulary on page 273 to correct yourself.

1. Give 3 functions of the Vocabulary.
2. True or False: All Vocabulary terms must also exist in the object or data model?
3. True or False: All terms in the object or data model must also exist in the Vocabulary?
4. True or False: In order to create the Vocabulary, an object or data model must already exist.
5. The Vocabulary is an __________ model, meaning many of the real complexities of an underlying data model are hidden so that the rule author can focus on only those terms relevant to the rules.
6. The UML model that contains the same types of information as a Vocabulary is called a ______________
7. What are the three components (or nodes) of the Vocabulary?
8. Which of the following are acceptable attribute names?

Hair_color	hairColor	HairColor	hair color

9. Which color is used in the Entity icon?
10. Which of the three Vocabulary components can hold an actual value?
11. What are the five main data types used by Vocabulary attributes?
12. Which colors are used in the Base attribute icon?
13. Which colors are used in the Extended Transient attribute icon?
14. What is the purpose of an Extended Transient Vocabulary term?
15. Associations are ________________ by default.

16. Association icons indicate:

optionality	singularity	cardinality	musicality

17. Which of the following icons represents a one-to-many association?

18. Which of the following icons represents a one-to-one association?

19. If an association is one-directional *from* the Source entity *to* the Target entity, then which term is not available in the Vocabulary?

Target.attribute	Target.source.attribute	Source.target.attribute	Source.attribute

20. The default role name of an association *from* the Source entity to the Target entity is:

role1	source	target	theTarget

21. Sketch a model for the following scenario:

> A Purchase Order has a customer name, order date, total amount and an unlimited number of Line Items. Each Line Item has a part number, quantity, price-per-unit and total price.

22. Create a Corticon Studio Vocabulary for the model sketched in 22.
23. List the (4) steps in generating a Vocabulary from scratch.
24. Cardinality of an association determines:
    a. The number of possible associated entities.
    b. The number of attributes for each entity.
    c. The number of associations possible within an entity.
    d. The number of attributes for each association.
25. The Vocabulary terms are the nouns of `Corticon` rules. What are the verbs?
26. What `Corticon` document contains the complete list of all Vocabulary Operators, descriptions of their usage, and actual examples of use in Rulesheets?
27. True or False. In addition to the supported vocabulary data types, you can create *any* type of custom data type you want?
28. You must name your custom data type. Which of the following are *not* custom data type naming convention requirements?
    a. Cannot match any other vocabulary entity names
    b. May match other Custom Data Type Names

c. Base Data Type names may not be re-used.

d. The name must comply with the standard entity naming rules.

29. True or False. The Enumeration section of the Custom Data Types exposses the Label/Value columns and allows you to create a list of acceptable value rows.

30. Selecting `no` in the *Enumeration* section of the Custom Data Types enables the Contraint Expression. Give an example of a Constraint Expression:

____

31. True or False. Constraint Expressions must be equivalent to a Boolean expression to be vaild.

32. In an Enumeration, are both the **Label** and **Value** columns required?

33. When you create Enumerated Custom data Types, which of the following are acceptable entries for the Value column:

12/12/2011	"12/12/2011"	Airbus	‘Airbus'

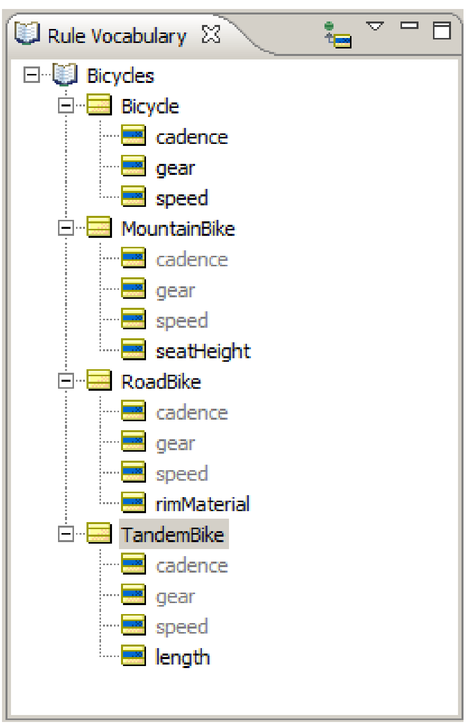
34. Name an advantage to using Enumerated Custom Data Types when it comes to testing your rules in a Ruletest.

35. Explain what Domains do in the Vocabulary?

36. True or False. If you use a Domain, then you will be required to create an alias for each unique Entity/Domain pair?

37. True or False. Inheritance can be modeled in a Vocabulary.

38. In the following vocabulary, which Entities have "native" attributes and which Entities has "inherited" attributes?



39. Give two examples of inherited attributes from the vocabulary above:

_______________ _______________

40. True or False. Using Inheritance can be a way to write efficient and powerful rules. For example, one rule could be used to modify the cadence attribute for all the entities in the Vocabulary example above.

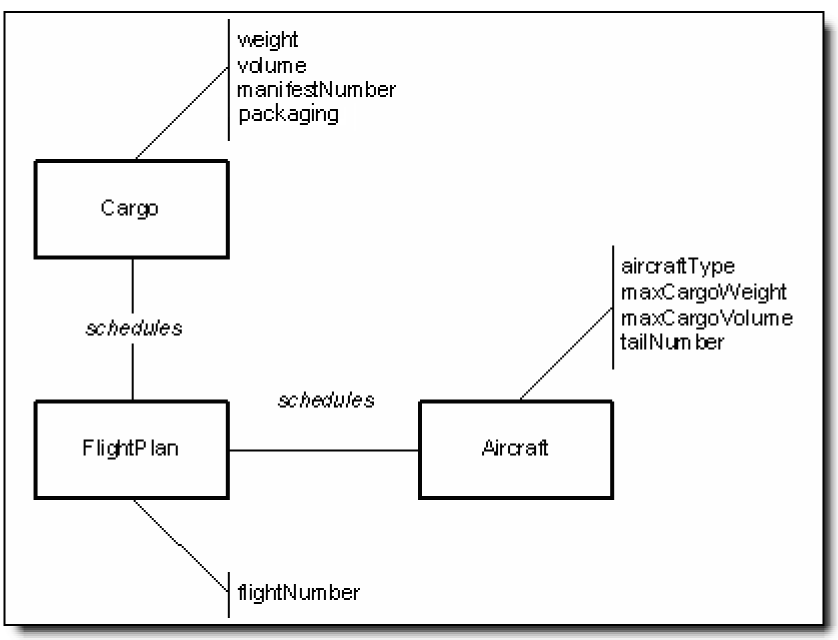
2

Rule scope & context

The air cargo example that we started in the Vocabulary chapter is continued here to illustrate the important concepts of *scope* and *context* in rule design.

A quick recap of the fact model:

Figure 40: Fact Model



According to this Vocabulary, an Aircraft is related to a Cargo shipment through a FlightPlan. In other words, it is the FlightPlan that connects or relates an Aircraft to its Cargo shipment. The Aircraft, by itself, has *no direct relationship* to a Cargo shipment unless it is scheduled by a FlightPlan; or, no Aircraft may carry a Cargo shipment without a FlightPlan. Similarly, no Cargo shipment may be transported by an Aircraft without a FlightPlan. These facts constitute business rules in and of themselves and constrain creation of other rules because they define the Vocabulary we will use to build all subsequent rules in this scenario.

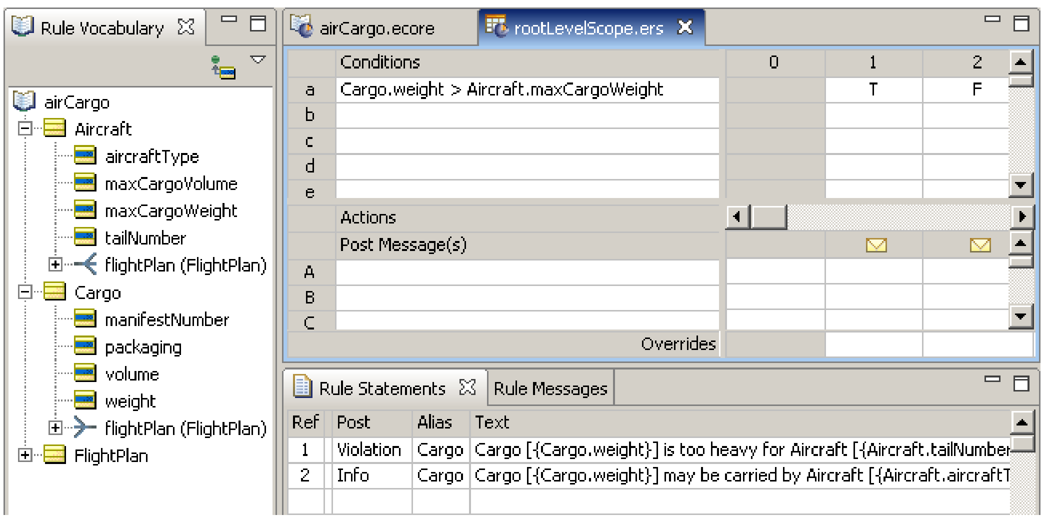
Also recall that the company wants to build a system that automatically checks flight plans to ensure no scheduling rules or guidelines are violated. One of the many business rules that need to be checked by this system is:

1. An Aircraft must not carry a Cargo shipment that exceeds its maximum Cargo weight

With our Vocabulary created, we can build this rule in the Studio. As with many tasks in Studio, there is often more than one way to do something. We will explore two possible ways to build this rule – one correct and one incorrect.

To begin with, we will write our rule using the "root-level" terms in the Vocabulary. In the following figure, column #1 (the **true** Condition) is the rule we are most interested in – we've added the **false** Condition in column #2 simply to show a logically complete `Rulesheet`.

Figure 41: Expressing the Rule Using "Root-Level" Vocabulary Terms

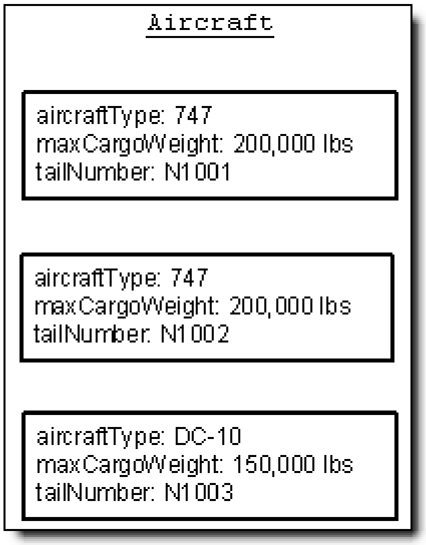


Refer to the *Embedding Attributes in Posted Rule Statements* section of the *Rule Language Guide* for additional details regarding the syntax introduced in the **Rule Statements** portion of the following figure, example 5, in the Custom data types topic.

We can build a `Ruletest` to test the rule using the Cargo company's actual data, as follows:

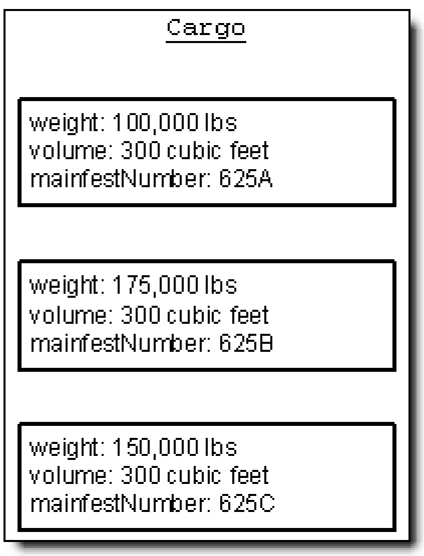
The company owns 3 Aircraft, 2 747s and a DC-10, each with different tail numbers. The 3 Aircraft are shown in the following figure, example 6 in the Custom data types topic. Each box represents a real-life example (or *instance*) of the `Aircraft` term from our Vocabulary.

Figure 42: The Cargo Company's 3 `Aircraft`



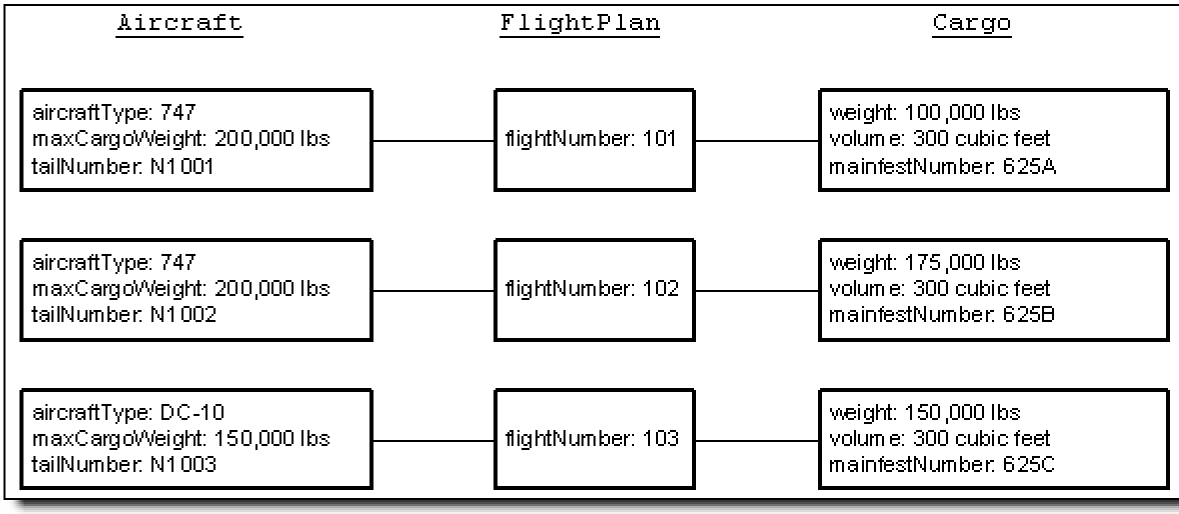
These `Aircraft` give the company the ability to schedule 3 `Cargo` shipments each night {there is another business rule implied here – "an Aircraft must not be scheduled for more than one flight per night", but we won't address this now because it is not relevant to the discussion}. On a given night, the `Cargo` shipments look like those shown below. Again, like the `Aircraft`, these `Cargo` shipments represent specific *instances* of the generic `Cargo` term from the Vocabulary.

Figure 43: The 3 `Cargo` Shipments for the Night of June 25th



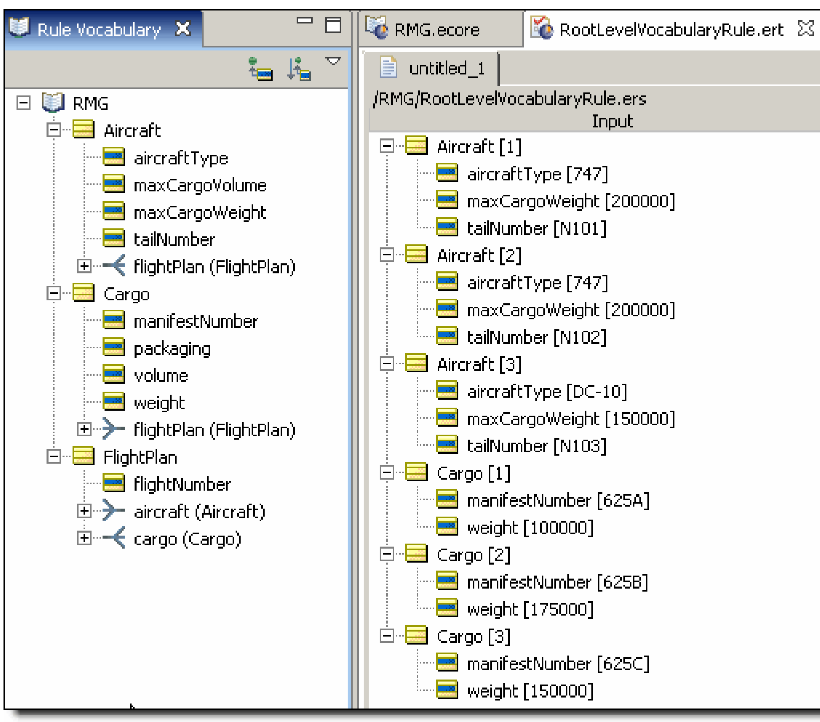
Finally, our sample business process manually matches specific aircraft and cargo shipments together as three flightplans, shown below. This organization of data is consistent with the structure and constraints implicit in our Vocabulary.

Figure 44: The 3 `FlightPlans` with their related `Aircraft` and `Cargo` instances



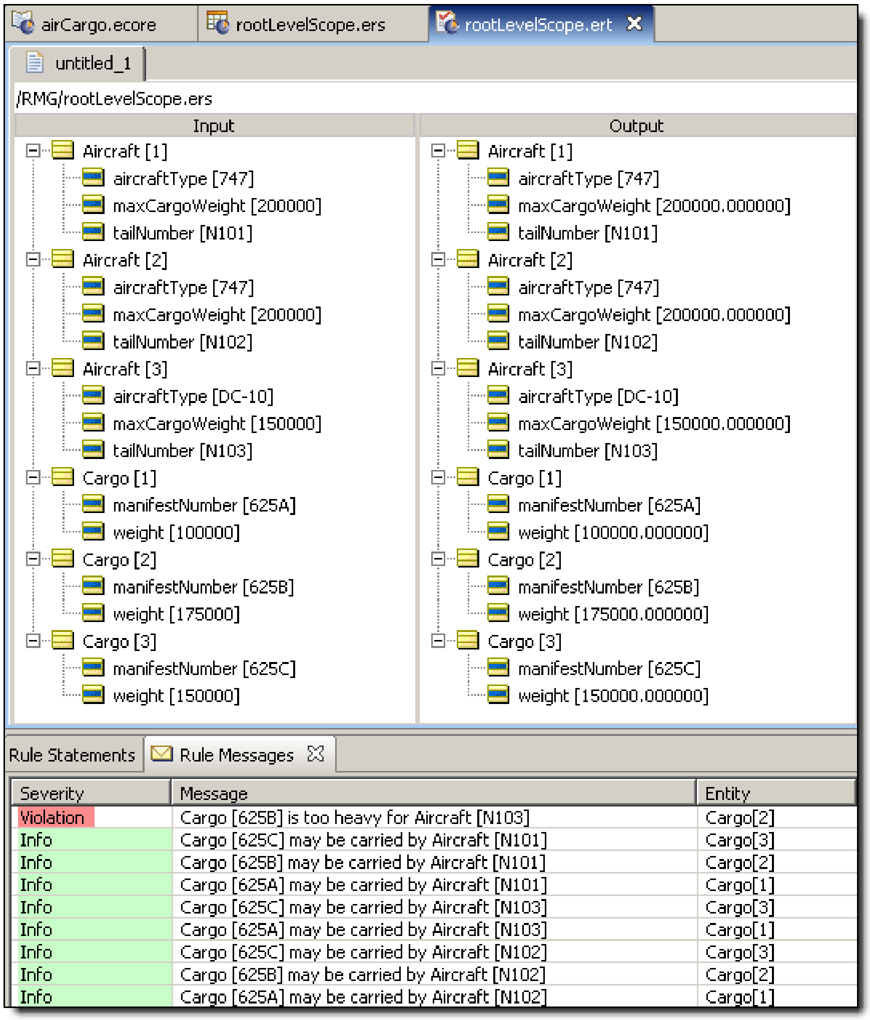
We can construct a *`Ruletest`* (in the following figure)so that the company's actual data will be evaluated by the rule. Since the rule used "root-level" Vocabulary terms in its construction, we will use "root-level" terms in the *`Ruletest`* as well:

Figure 45: Test the Rule Using "Root-Level" Vocabulary Terms



Running the *Ruletest*:

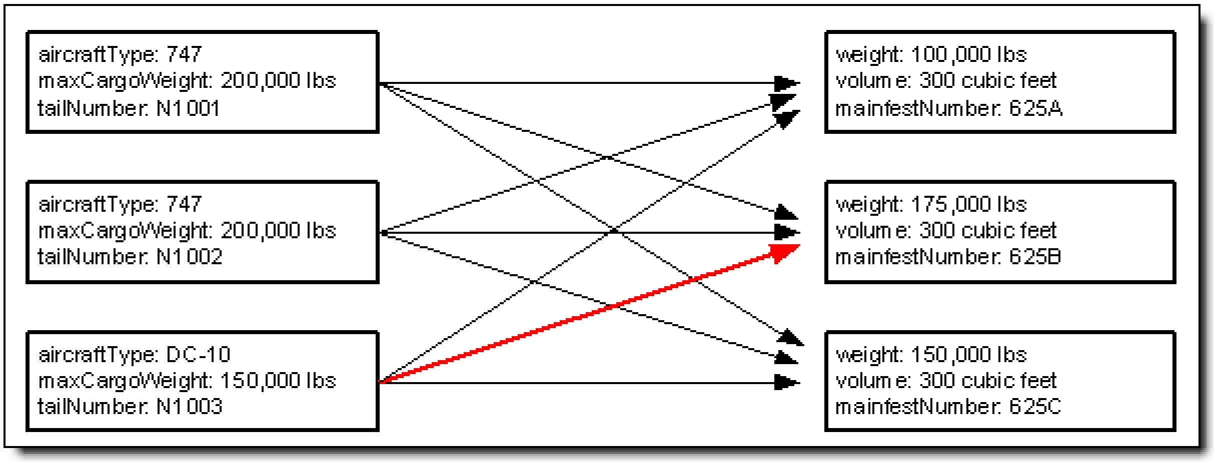
Figure 46: Results of the *Ruletest*



Note the messages returned by the *Ruletest*. Recall that the intent of the rule is to verify whether a given `Flightplan` is in violation by scheduling a `Cargo` shipment that is too heavy for the assigned `Aircraft`. But we already know there are only three `Flightplans`. And we also know, from examination of **The 3 FlightPlans with their related Aircraft and Cargo instances**, that the combination of aircraft `N1003` and cargo `625C` does not appear in any of our three `Flightplans`. So why was this combination, one that does not actually exist, evaluated? For that matter, why has the rule fired *nine* times when only *three* sets of Aircraft and Cargo were present? The answer lies in the way we defined our rule, and in the way the *Server* evaluated it.

We gave the `Ruletest` three instances of both `Aircraft` and `Cargo`. Studio treats Aircraft as a "collection" or "set" of these three specific instances. When Studio encounters the term Aircraft in a rule, it applies all instances of `Aircraft` found in the `Ruletest` (all three instances in this example) to the rule. Since both `Aircraft` and `Cargo` have three instances, there are a total of nine *possible combinations* of the two terms. In the following figure, the set of these nine possible combinations is called a "cross product", "Cartesian product", or "tuple set" in different disciplines. We tend to use cross-product when describing this outcome.

Figure 47: All Possible Combinations of `Aircraft` and `Cargo`

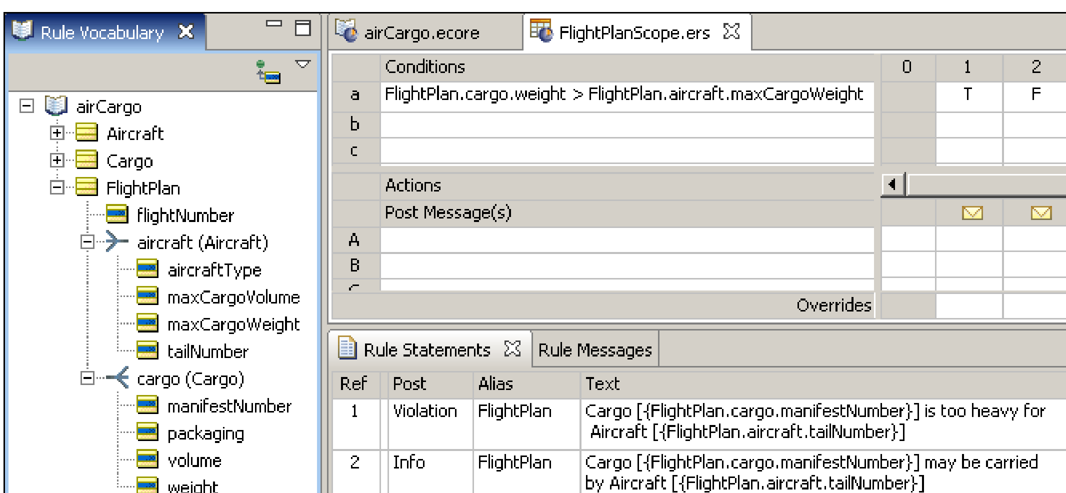


One pair, the combination of manifest `625B` and plane `N1003` (shown as the red arrow in the figure above), is indeed illegal, since the plane, a `DC-10`, can only carry `150,000` lbs, while the cargo weighs `175,000` lbs. But this pairing does not correspond to any of the three `FlightPlans` created. Many of the other combinations evaluated (five others, to be exact) are not represented by real flight plans either. So why did Studio bother to perform three times the necessary evaluations? It is because our rule, as implemented in Figure 41 on page 50, does not capture the essential elements of **scope** and **context**.

We want our rule to express the fact that we are only interested in evaluating the `Cargo—Aircraft` pair for *each* `FlightPlan`, not for *all* possible combinations. How do we express this intention in our rule? We use the associations included in the Vocabulary.

Refer to the following figure:

Figure 48: Rule Expressed Using `FlightPlan` as the Rule Scope



Here, we've rewritten the rule using the `aircraft` and `cargo` terms from *inside* the `FlightPlan` term.

Note: By "inside" we mean the aircraft and cargo terms that appear when the FlightPlan term is opened in the Vocabulary tree, as shown by the orange circles in **Rule Expressed Using `FlightPlan` as the Rule Scope**.

This is significant. It means we want the rule to evaluate the `Cargo` and `Aircraft` terms *only in the context of* a `FlightPlan`. For example, on a different night, the Cargo company might have eight Cargo shipments assembled, but only the same three planes on which to carry them. In this scenario, three flight plans would still be created. Should the rule evaluate all eight Cargo shipments, or only those three associated with actual flight plans? From the original business rule, it is clear we are only interested in evaluating those Cargo shipments *in the context of* actual flight plans. To put it differently, the rule's application is limited to only those `Cargo` shipments assigned to a specific `Aircraft` via a specific `FlightPlan`. We express these relationships in the *`Rulesheet`* by including the `FlightPlan` term in the rule, so that `cargo.weight` is properly expressed as `FlightPlan.cargo.weight`, and `Aircraft.maxCargoWeight` is properly expressed as `FlightPlan.aircraft.maxCargoWeight`. By attaching `FlightPlan` to the terms `aircraft.maxCargoWeight` and `cargo.weight`, we have indicated mandatory *traversals* of the associations between `FlightPlan` and the other two terms, `Aircraft` and `Cargo`.This instructs *`Corticon Server`* to evaluate the rule using the intended context. In writing rules, it is extremely important to understand the context of a rule and the scope of the data to which it will be applied.

For details, see the following topics:

- Rule scope
- Aliases
- Scope and perspectives in the vocabulary tree
- Test yourself: Rule scope and context

Rule scope

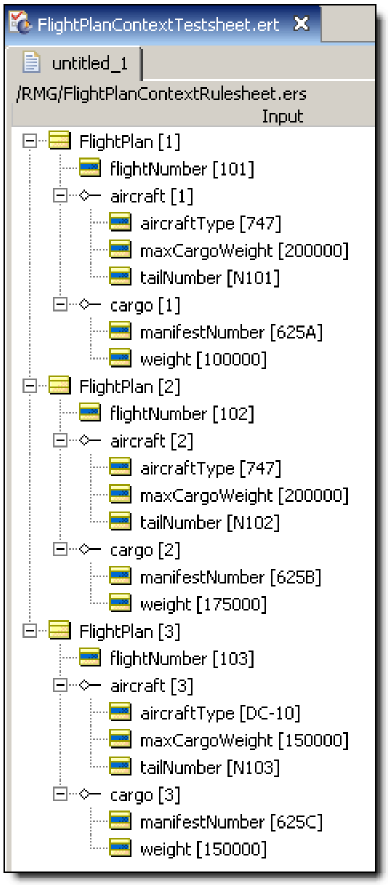
Because the rule is evaluating both `Cargo` and `Aircraft` in the context of a `FlightPlan`, we say that the rule has *scope*, which means that *the rule evaluates only' that data which matches the rule's scope*. This has an interesting effect on the way the rule is evaluated. When the rule is executed, its scope ensures that the *`Server`* evaluates only those pairings that *matches the same* `FlightPlan`. This means that a `cargo.weight` will **only** be compared to an `aircraft.maxCargoWeight` **if** both the `cargo` and the `aircraft` share the same `FlightPlan`. This simplifies rule expression greatly, because it eliminates the need for us to specify *which* `FlightPlan` we are talking about for each `Aircraft-Cargo` combination. When a rule has context, the system takes care of this matching automatically by sending *only* those Aircraft - Cargo pairs that *share the same* `FlightPlan` to be evaluated by the rule. And, since Corticon Studio automatically handles multiple instances as *collections*, it sends *all* pairs to the rule for evaluation.

Note: See the following topic, Collections, for a more detailed discussion of this subject.

To test this new rule, we need to structure our `Ruletest` differently to correspond to the new structure of our rule and reflect the rule's scope. For more information on the mechanics of creating associations in `Ruletest`s, also refer to the "Set Up the `Ruletest` Scenario" section in the *Corticon Studio Tutorial: Basic Rule Modeling* and the "Creating Associations" chapter in the *Quick Reference Guide*.

Finally, one `FlightPlan` is created for each `Aircraft-Cargo` pair. This means a total of three `FlightPlans` are generated each night. Using the terms in our Vocabulary *and the relationships between them*, we have the possibilities shown in `The 3 FlightPlans with their related Aircraft and Cargo instances`. The rule will evaluate these combinations and identify any violations.

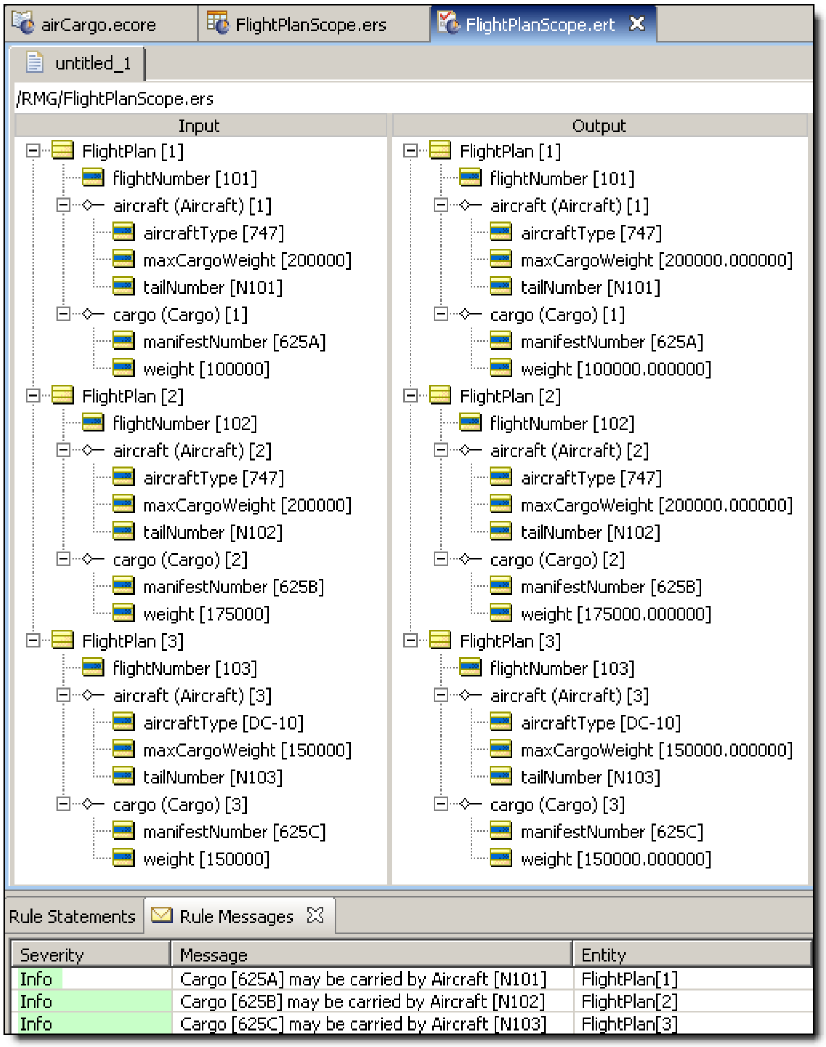
Figure 49: New Ruletest Using `FlightPlan` as the Rule Scope



What is the expected result from this `Ruletest`? If the results follow the same pattern as in the first `Ruletest`, we might expect the rule to fire nine times (three `Aircraft` evaluated for each of three `Cargo` shipments).

But refer to `Ruletest Results Using Scope – Note no Violations` and you will see that the rule, in fact, fired only 3 times – and only for those `Aircraft-Cargo` pairs that are related by common `FlightPlans`. This is the result we want. The `Ruletest` shows that there are no `FlightPlans` in violation of our rule.

Figure 50: Ruletest Results Using Scope – Note no Violations



One final point about scope: it is critical that the context you choose for your rule supports the intent of the business decision you are modeling. At the very beginning of our example, we stated that the purpose of the application is to check flightplans *that have already been created*. Therefore, the context of our rule was chosen so that the rule's design was consistent with this goal – no aircraft-cargo combinations should be evaluated unless they are already matched up via a common flightplan.

But what if our business purpose had been different? What if the problem we are trying to solve was modified to: "Of all possible combinations of aircraft and cargo, determine which pairings must **not** be included in the same FlightPlan." The difference here is subtle but important. Before, we were identifying invalid combinations of pre-existing FlightPlans. Now, we are trying to identify invalid combinations from all possible cargo-aircraft pairings. This other rule might be the first step in a screening or filtering process designed to discard all the invalid combinations. In this case, the original rule we built, root-level context, would be the appropriate way to implement our intentions, because now we are looking at all possible combinations *prior to creating new FlightPlans*.

Aliases

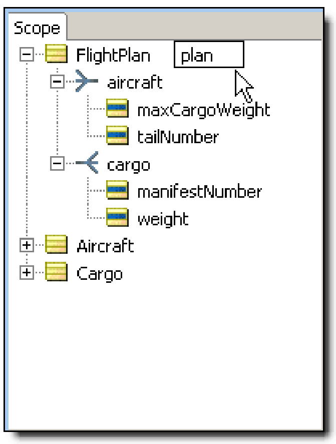
To clean up and simplify rule expression, Corticon Studio allows you to declare *aliases* in a Rulesheet Using an alias to express scope results in a less cluttered Rulesheet .

To define an alias, you need to open the **Scope** tab on the Rulesheet. Either click the toolbar button  to open the advanced view, or choose the Rulesheet menu toggle **Advanced View**.

If rules have already been modeled in the Rulesheet, then the **Scope** window already contains those Vocabulary terms used in the rules so far. If rules have not yet been modeled, then the Scope window is empty.

To define an alias, double-click to the term, and then type a unique name in the entry box, as shown:

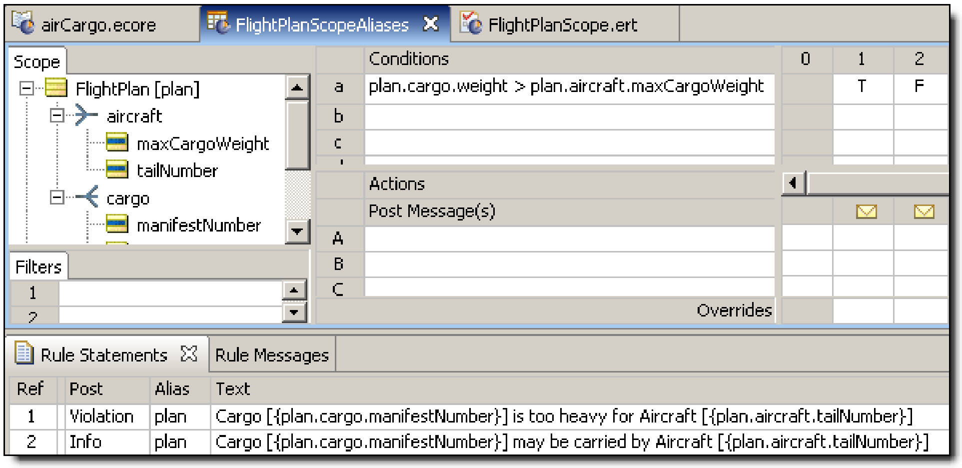
Figure 51: Defining an Alias in the Scope window



Once an alias is defined, any subsequent rule modeling in the `Rulesheet` automatically substitutes the alias for the Vocabulary term it represents.

In *`Rulesheet with FlightPlan Alias Declared in the Scope Section`*, notice that the terms in the Condition rows of the Rulesheet do not show the `FlightPlan` term. That's because the alias `plan` substitutes for `FlightPlan`. The small "c" in `cargo` and "a" in `aircraft` provide other clues that these terms exist *within the context* of the `FlightPlan` term defined in the **Scope** window.

Figure 52: *`Rulesheet`* with `FlightPlan` Alias Declared in the Scope Section



Once an alias is defined, any new Vocabulary term dropped onto the Rulesheet is adjusted accordingly. For example, dragging and dropping `FlightPlan.cargo.weight` onto the Rulesheet displays as `plan.cargo.weight`.

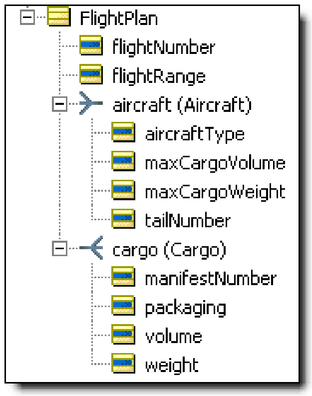
Aliases work in all sections of the *`Rulesheet`*, including the **Rule Statement** section. Modifying an alias name defined in the Scope section causes the name to update everywhere it is used in the *`Rulesheet`*.

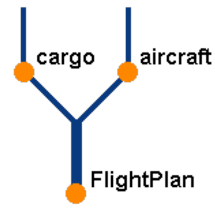
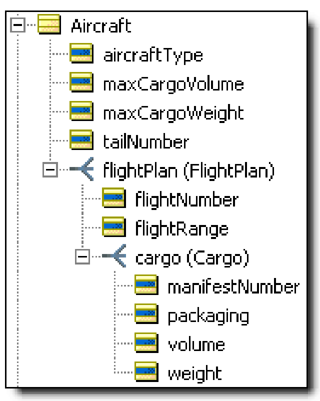
Note: Rules modeled without aliases do not update automatically if aliases are defined later. So if you intend to use aliases, define them as you start your rule modeling - that way they apply automatically when you drag and drop from the Vocabulary or Scope windows.

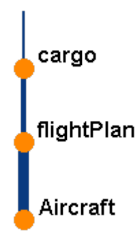
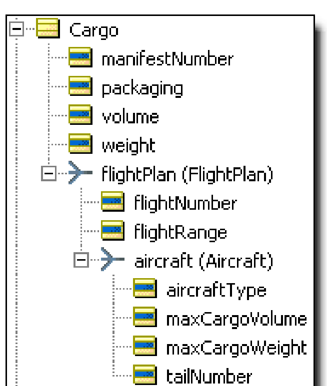
Scope and perspectives in the vocabulary tree

Because our Vocabulary is organized as a "tree" view in Corticon Studio, it may be helpful to extend the tree analogy to better understand what aliases do. The tree view permits us to use the business terms from a number of different "perspectives", each perspective corresponding to one of the root-level terms and an optional set of one or more branches.

Table 1: Table: Vocabulary Tree Views and Corresponding Branch Diagrams

<table>
<tr><th>Vocabulary Tree</th><th>Description</th><th>Branch Diagram</th></tr>
<tr><td>
</td><td>This portion of the Vocabulary tree can be visualized as the branch diagram shown to the right. Because this piece of the Vocabulary begins with the <code>FlightPlan</code> "root", the branches also originate with the FlightPlan root or trunk. The FlightPlan's associated <code>cargo</code> and <code>aircraft</code> terms are branches from the trunk.

Any rule expression that uses <code>FlightPlan</code>, <code>FlightPlan.cargo</code>, or <code>FlightPlan.aircraft</code> is using scope from this perspective of the Vocabulary tree.</td><td>
</td></tr>
<tr><td>
</td><td>This portion of the Vocabulary tree begins with <code>Aircraft</code> as the root, with its associated <code>flightPlan</code> branching from the root. A <code>cargo</code>, in turn, branches from its associated <code>flightPlan</code>.

Any rule expression that uses <code>Aircraft</code>, <code>Aircraft.flightPlan</code>, or <code>Aircraft.flightPlan.cargo</code> is using scope from this perspective of the Vocabulary tree.</td><td>
</td></tr>
<tr><td>
</td><td>This portion of the Vocabulary tree begins with <code>Cargo</code> as the root, with its associated <code>flightPlan</code> branching from the root. An <code>aircraft</code>, in turn, branches from its associated <code>flightPlan</code>.

Any rule expression that uses <code>Cargo</code>, <code>Cargo.flightPlan</code>, or <code>Cargo.flightPlan.aircraft</code> is using scope from this perspective of the Vocabulary tree.</td><td>
</td></tr>
</table>

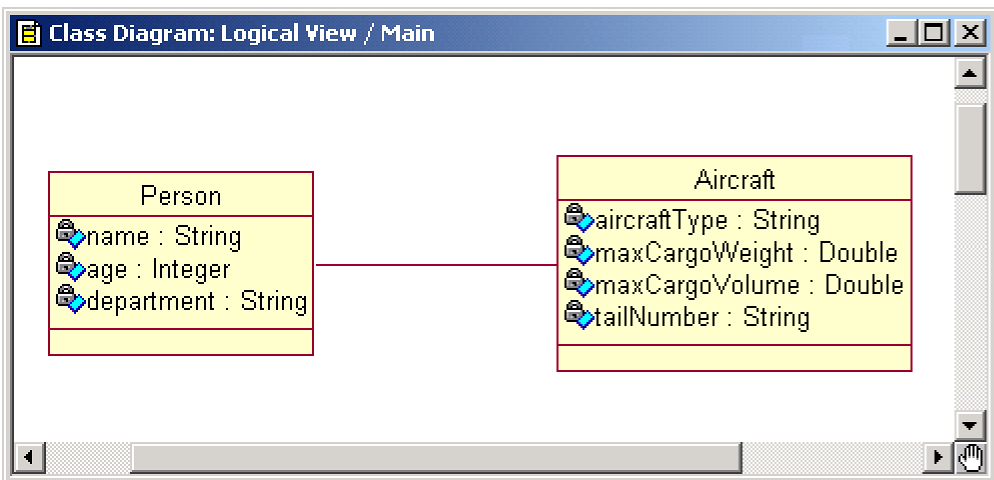
Scope can also be thought of as hierarchical, meaning that a rule written with scope of `Aircraft` applies to all root-level `Aircraft` data. And other rules using some piece (or branch) of the tree beginning with root term `Aircraft`, including `Aircraft.flightPlan` and `Aircraft.flightPlan.cargo`, also apply to this data and its associated collections. Likewise, a rule written with scope of `Cargo.flightPlan` does not apply to root-level `FlightPlan` data.

This provides an alternative explanation for the different behaviors between the *`Rulesheet`*s in Expressing the Rule Using Root-Level Vocabulary Terms and Rule Expressed Using FlightPlan as the Rule Scope. The rules in Expressing the Rule Using Root-Level Vocabulary Terms are written using different root terms and therefore different scopes, whereas the rules in Rule Expressed Using FlightPlan as the Rule Scope use the same `FlightPlan` root and therefore share common scope.

Roles

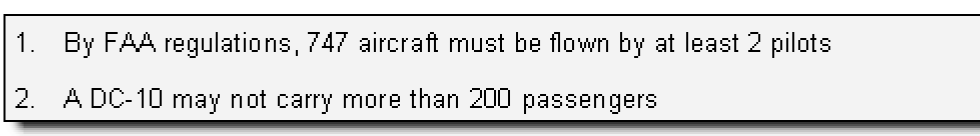
Using roles in the Vocabulary can often help to clarify rule context. To illustrate this point, we will use a slightly different example. The UML class diagram for a new (but related) sample Vocabulary is as shown:

Figure 53: UML Class Diagram without Roles



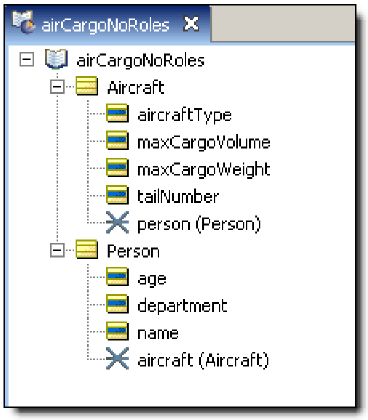
As shown in this class diagram, the entities `Person` and `Aircraft` are joined by an association. However, can this single association sufficiently represent multiple relationships between these entities? For example, a prior Fact Model might state that "a pilot flies an aircraft" and "a passenger rides in an aircraft" – both pilot and passenger are descendants of the entity `Person`. Furthermore, we can see that, in practice, some instances of `Person` may be pilots and some may be passengers. This is important because it suggests that some business rules may use `Person` in its pilot context, and others may use it in its passenger context. How do we represent this in the Vocabulary and rules we build in Corticon Studio?

Let's examine this problem in more detail. Assume we want to implement two new rules:



We call these rules "cross-entity" because they include more than one entity (both Aircraft and Person) in their expression. Unfortunately, with our Vocabulary as it is, we have no way to distinguish between pilots and passengers, so there is no way to unambiguously implement these 2 rules. This class diagram, when imported into Corticon Studio, looks like this:

Figure 54: Vocabulary without Roles



However, there are several ways to modify this Vocabulary to allow us to implement these rules. We will discuss these methods and examine the advantages and disadvantages of each.

Use Inheritance

Use two separate entities for Pilot and Passenger instead of a single Person entity. This may often be the best way to distinguish between pilots and passengers, especially if the two types of Person reside in different databases or different database tables (an aspect of deployment that rule modelers may not be aware of). Also, if the two types of Person have some shared and some different attributes (Pilot may have attributes like licenseRenewalDate and typeRating while Passenger may have attributes like farePaid and seatSelection) then it may make sense to set up entities as descendants of a common ancestor entity (such as Employee).

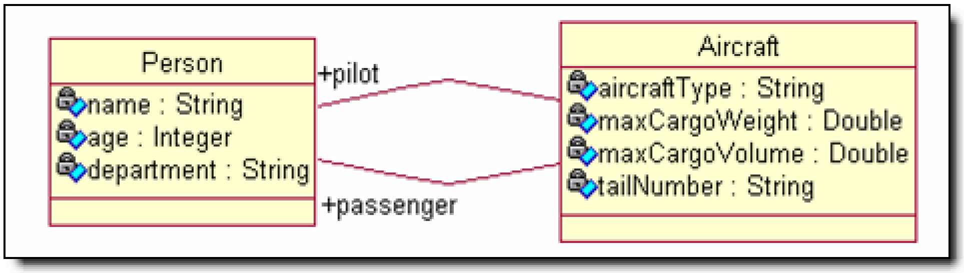
Add an Attribute to Person

If the two types of person differ only in their type, then we may decide to simply add a personType (or similar) attribute to the entity. In some cases, personType will have the value of pilot, and sometimes it will have the value of passenger. The advantage of this method is that it is flexible: in the future, persons of type manager or bag handler or air marshal can easily be added. Also, this construction may be most consistent with the actual structure of the employee database or database table and maintains a normalized model. The disadvantage comes when the rule modeler needs to refer to a specific type of Person in a rule. While this can be accomplished using any of the filtering methods discussed in Rule Writing Techniques, they are sometimes less convenient and clear than the final method, discussed next.

Use Roles

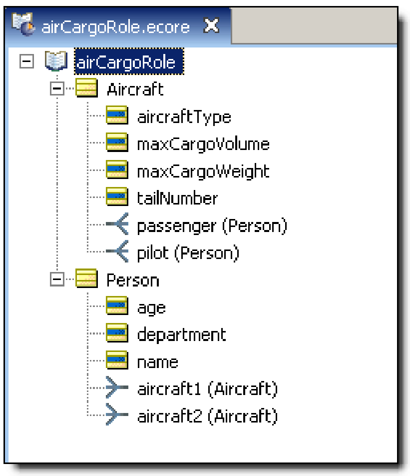
A role is a noun that labels one end of an association between two entities. For example, in our `Person-Aircraft` Vocabulary, the `Person` may have more than one role, or more than one kind of relationship, with `Aircraft`. An instance of `Person` may be a `pilot` or a `passenger`; each is a different role. To illustrate this in our UML class diagram, we add labels to the associations as follows:

Figure 55: UML Class Diagram with Roles



When the class diagram is imported into Corticon Studio, it appears as the Vocabulary below:

Figure 56: Vocabulary with Roles

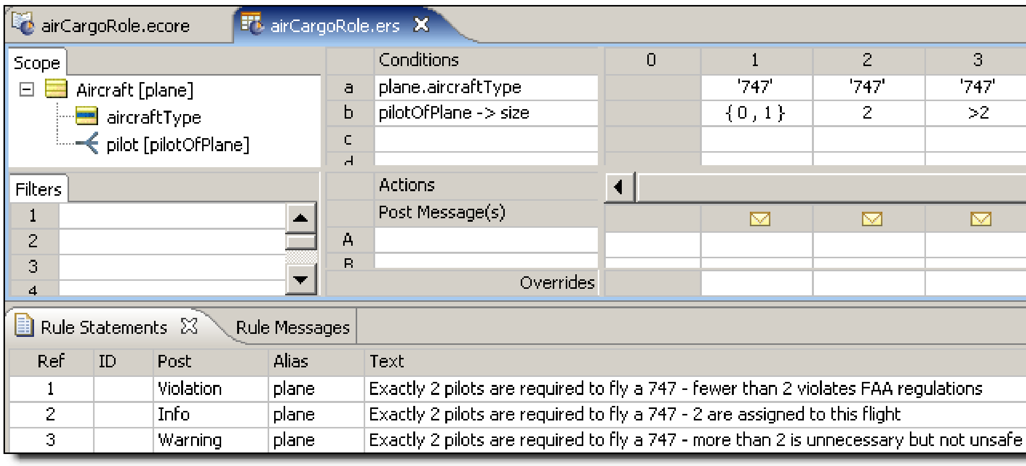


Notice the differences between **Vocabulary with Roles** and **Vocabulary without Roles** – in **Vocabulary with Roles**, `Aircraft` contains 2 associations, one labeled `passenger` and the other `pilot`, even though both associations relate to the same `Person` entity. Also notice that we have updated the cardinalities of both `Aircraft—Person` associations to "one-to-many".

Written using roles, the first rule appears below. There are a few aspects of the implementation to note:

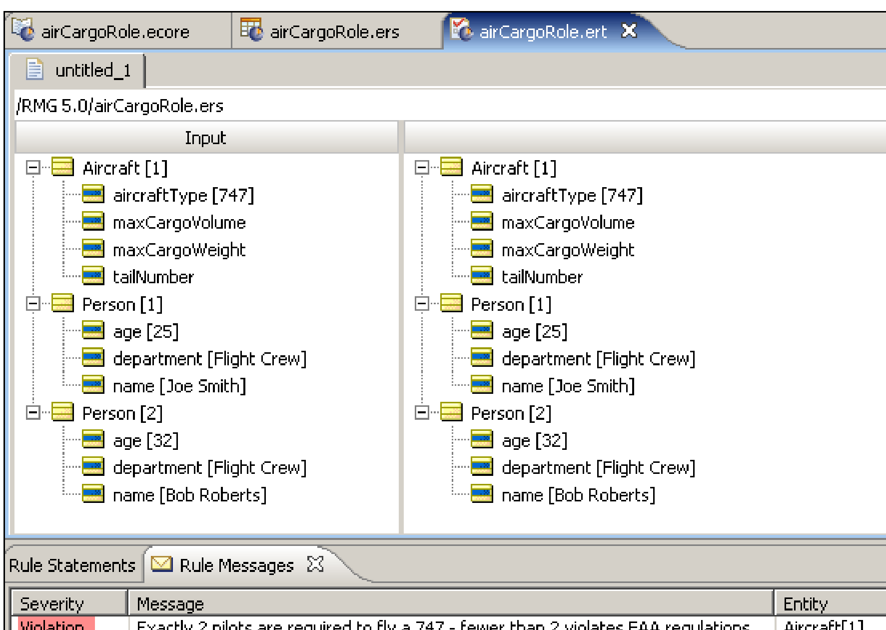
- Use of aliases for `Aircraft` and `Aircraft.pilot` (`plane` and `pilotOfPlane`, respectively). Aliases are just as useful for clarifying rule expressions as they are for shortening them.
- The rule Conditions evaluate data within the context of the `plane` and `pilotOfPlane` aliases, while the Action posts a message to the `plane` alias. This enables us to act on the `aircraft` entity based upon the attributes of its associated pilots. Note that Condition row b uses a special operator (`->size`) that "counts" the number of pilots associated with a plane. This is called a collection operator and is explained in more detail in the following chapters.

Figure 57: Rule #1 Implemented using Roles



To demonstrate how Corticon Studio differentiates between entities based on rule scope, we will construct a new *Ruletest* that includes a single instance of `Aircraft` and 2 `Person` entities, neither of which has the role of pilot.

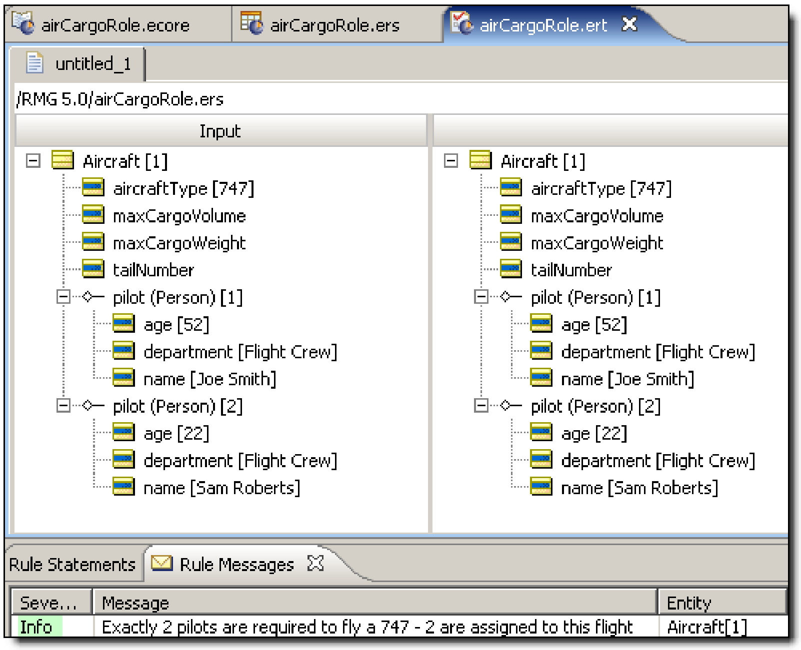
Figure 58: Ruletest with no `Person` entities in `Pilot` role



Despite the fact that there are two `Person` entities, both of whom are members of the `Flight Crew` department, the system recognizes that neither of them have the role of pilot (in relation to the `Aircraft` entity), and therefore generates the violation message shown.

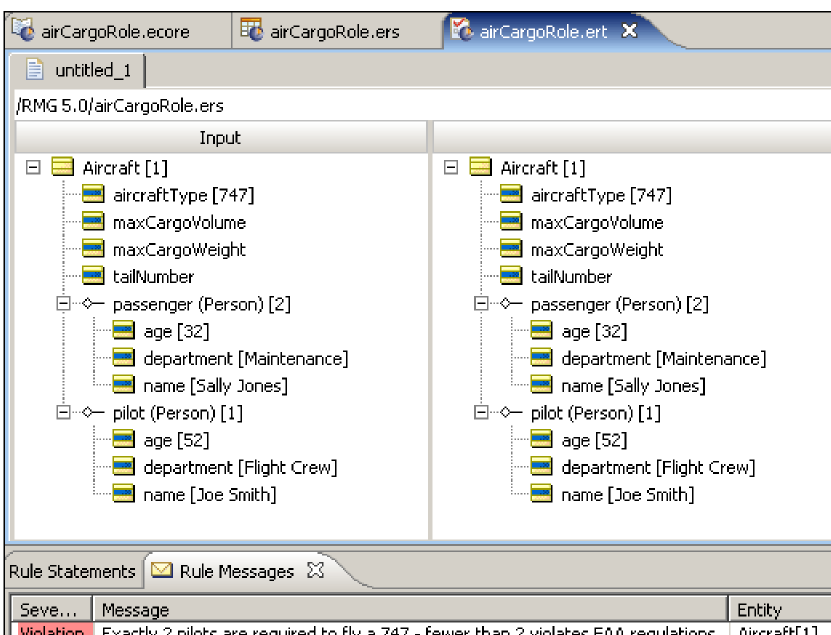
If we create a new Input *`Ruletest`*, this time with both persons in the role of pilot, we see a different result, as shown:

Figure 59: Ruletest with both `Person` entities in role of `Pilot`



Finally, the rules are tested with one pilot and one passenger:

Figure 60: Ruletest with one `Person` entity in each of `Pilot` and `Passenger` roles



We see that despite the presence of two `Person` elements in the collection of test data, only one satisfies the rules' scope – `pilot` associated with `aircraft`. As a result, the rules determine that one pilot is insufficient to fly a 747, and the violation message is displayed.

These same concepts apply to the DC-10/Passenger business rule, which will not be implemented here.

Technical aside

Understanding Rule Associations and Scope as Relationships Between Tables in a Relational Database

Although it is not necessary for the rule modeler or developer to understand database theory, a business or systems analyst who is familiar with it may have already recognized that the preceding discussion of rule scope and context is an abstraction of basic relational concepts. Actual relational tables that contain the data for our Cargo example might look like the following:

Figure 61: Tables in a Relational Database

Aircraft		
tailNumber*	aircraftType	maxCargoWeight
N1001	*747*	*200,000*
N1002	*747*	*200,000*
N1003	*DC-10*	*150,000*

Cargo		
manifestNumber*	volume	weight
625A	*300*	*100,000*
625B	*300*	*175,000*
625C	*300*	*150,000*

FlightPlan		
flightNumber*	tailNumber	manifestNumber
101	*N1001*	*625A*
102	*N1002*	*625B*
103	*N1003*	*625C*

Each one of these tables has a column that serves as a unique identifier for each row (or *record*). In the case of the `Aircraft` table, the `tailNumber` is the unique identifier for each Aircraft record – this means that no two Aircraft can have the same `tailNumber.ManifestNumber` is the unique identifier for each `Cargo` record. These unique identifiers are known as "primary keys". Given the primary key, a particular record can always be found and retrieved. A common notation uses an asterisk character (*) to indicate those table columns that serve as primary keys. If a Vocabulary has been connected to an external database using Enterprise Data Connector features, then you may notice asterisks next to attributes, indicating their designation as primary keys. See the *`Server`* *Integration and Deployment Guide*, *Direct Database Access* chapter for complete details.

Notice that the `FlightPlan` table contains columns that did not appear in our Vocabulary. Specifically, *tailNumber* and *manifestNumber* exist in the *Aircraft* and *Cargo* entities, respectively, but we did not include them in the `FlightPlan` Vocabulary entity. Does this mean that our original Vocabulary was wrong or incomplete? No - the extra columns in the `FlightPlan` table are really duplicate columns from the other two tables – `tailNumber` came from the `Aircraft` table and `manifestNumber` came from the `Cargo` table. These extra columns in the `FlightPlan` table are called *foreign keys* because they are the primary keys *from other tables*. They are the mechanism for creating relations in a relational database.

For example, we can see from the `FlightPlan` table that `flightNumber` 101 (the first row or record in the table) includes `Aircraft` of `tailNumber` N1001 and `Cargo` of `manifestNumber` 625A. The foreign keys in `FlightPlan` serve to link or connect a specific Aircraft with a specific `Cargo`. If the database is queried (using a query language like SQL, for example), a user could determine the weight of `Cargo` planned for `Aircraft` N1001 – by "traversing" the relationships from the `Aircraft` table to the `FlightPlan` table, we discover that `Aircraft` N1001 is scheduled to carry `Cargo` 625A. By traversing the `FlightPlan` table to the `Cargo` table, we discover that `Cargo` 625A weighs 100,000 lbs. Matching the foreign key in the `FlightPlan` table with the primary key in the `Cargo` table makes this traversal possible.

The *`Corticon`* Vocabulary captures this essential feature of relational databases, but abstracts it in a way that is friendlier to non-programmers. Rather than deal with concepts like foreign keys in our Vocabulary, we talk about "associations" between entities. Traversing an association in the Vocabulary is exactly equivalent to traversing a relationship between database tables. When we use a term like `Aircraft.tailNumber` in a rule, Studio creates a collection of `tailNumbers` from all records in the `Aircraft` table. This collection of data is then "fed" to the rule for evaluation. If however, the rule uses `FlightPlan.aircraft.tailNumber`, then Studio will create a collection of only those `tailNumbers` from the `Aircraft` table that have `FlightPlans` related to them – it identifies these aircraft instances by matching the `tailNumber` in the `Aircraft` table with the `tailNumber` (foreign key) in the `FlightPlan` table. If the `Aircraft` table contains 7 instances of aircraft (i.e., 7 unique rows in the table), but the `FlightPlan` table contains only 3 unique instances of flight plans, the term `FlightPlan.aircraft.tailNumber` will create a collection of only 3 tail numbers – those instances from the `Aircraft` table which have flight plans listed in the `FlightPlan` table. In database terminology, the scope of the rule determines how the tables are "joined".

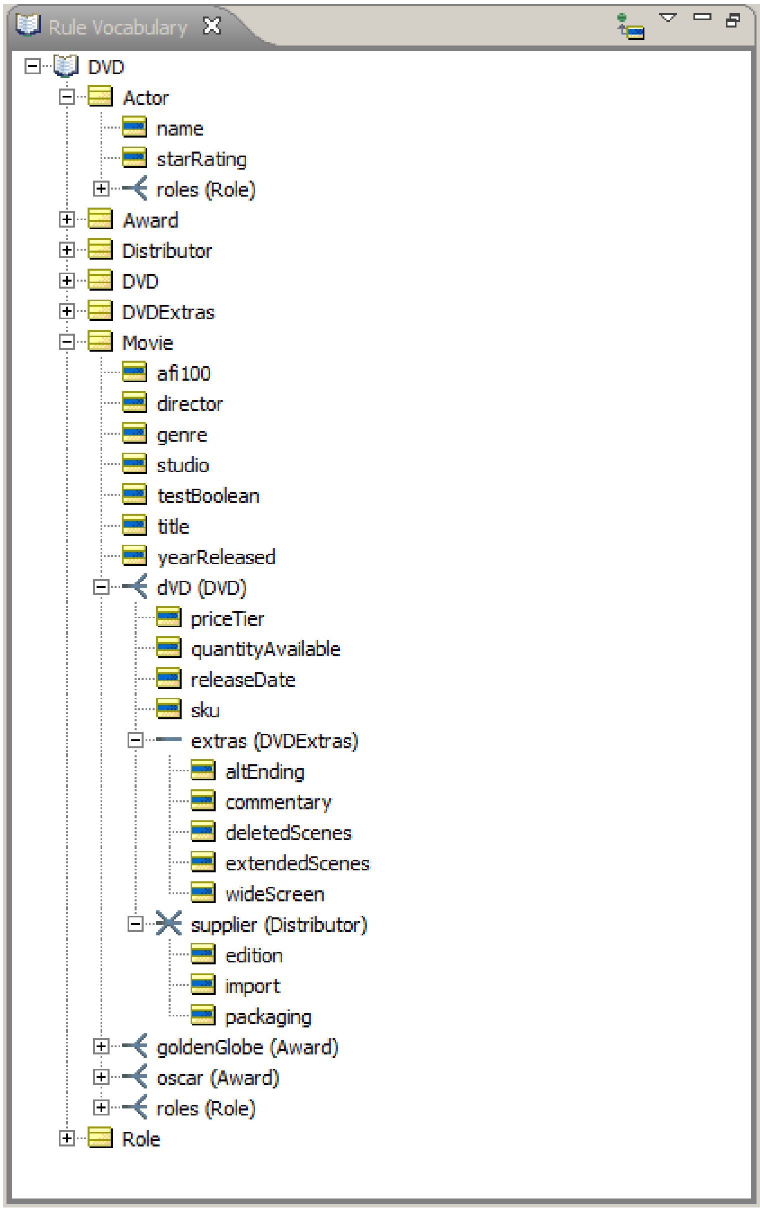
When `FlightPlan` is used as the scope for our rule, Corticon Studio automatically ensures that the collection of data contains matching foreign keys. That's why, when we rewrote the rule using proper scope, the rule only fired 3 times – there are only 3 examples of `Aircraft-Cargo` combinations where the keys match. This also explains why, prior to using scope, the rule produced 6 spurious and irrelevant outcomes – 6 combinations of `Aircraft` and `Cargo` that were processed by the rule do not, in fact, exist in the `FlightPlan` table.

While the differences in processing requirements are not extreme in our simple example, for a large company like Federal Express, with a fleet of hundreds of aircraft and several thousand unique cargo shipments every day, the system performance differences could be enormous.

Test yourself: Rule scope and context

Note: Try this test, and then go to **Answers:** Rule scope & context on page 275 to correct yourself.

Use the following Vocabulary to answer the next questions.


Rule Vocabulary
DVD
Actor
name
starRating
roles (Role)
Award
Distributor
DVD
DVDExtras
Movie
afi100
director
genre
studio
testBoolean
title
yearReleased
dVD (DVD)
priceTier
quantityAvailable
releaseDate
sku
extras (DVDExtras)
altEnding
commentary
deletedScenes
extendedScenes
wideScreen
supplier (Distributor)
edition
import
packaging
goldenGlobe (Award)
oscar (Award)
roles (Role)
Role

1. How many root-level entities are present in the Vocabulary?
2. Which of the following terms are allowed by the Vocabulary?

Movie.roles	Actor.roles	DVD.actor	Award.movie

3. Which of the following terms are **not** allowed by the Vocabulary?

Movie.oscar	Movie.supplier	Movie.roles.actor	Movie.dVD.extras

4. Which Vocabulary term represents the following phrases?
   - A movie's Oscars ______________________
   - A movie's roles ______________________
   - An actor's roles ______________________
   - A DVD's distributor ______________________
   - A movie's DVD extras ______________________
   - An actor's Oscars ______________________
5. Which of the following terms represents the phrase "an actor in a role of a movie"

Movie.roles.dVD	Actor.roles.movie	DVD.actor.movie	Actor.movie.roles

6. Since the association between Actor and Role is bidirectional, we can use both Actor.roles and ______________ in our rules.
7. Which two entities are associated with each other by more than one role?
8. What are the role names?
9. Besides roles, how else could these two relationships be represented in the Vocabulary to convey the same business meaning?
10. What is the advantage of using roles in this way?
11. When more than role is used to associate two entities, each role name must be:

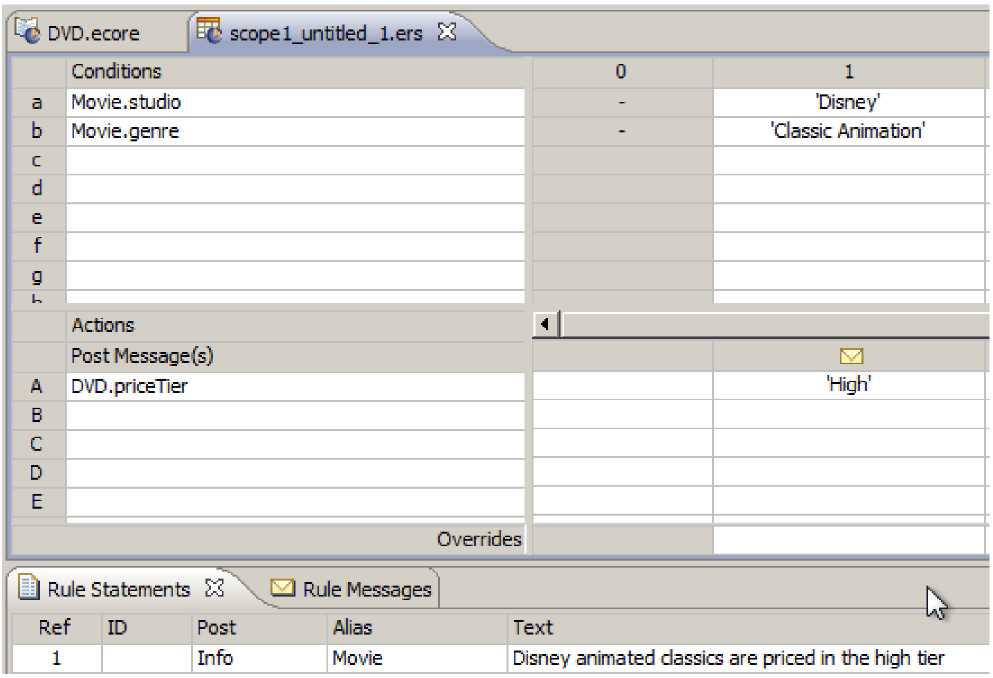
friendly	unique	colorful	melifluous

12. True or False. Rules evaluate only data that shares the same scope
13. Write a conditional expression in a `Rulesheet` for each of the following phrases:
    - If a movie's DVD has deleted scenes…
    - If an actor played a role in a movie winning an Oscar…
    - If the DVD is an import…
    - If the Movie was released more than 50 years before the DVD…
    - If the actor ever played a leading role…
    - If the movie was nominated for a Golden Globe…

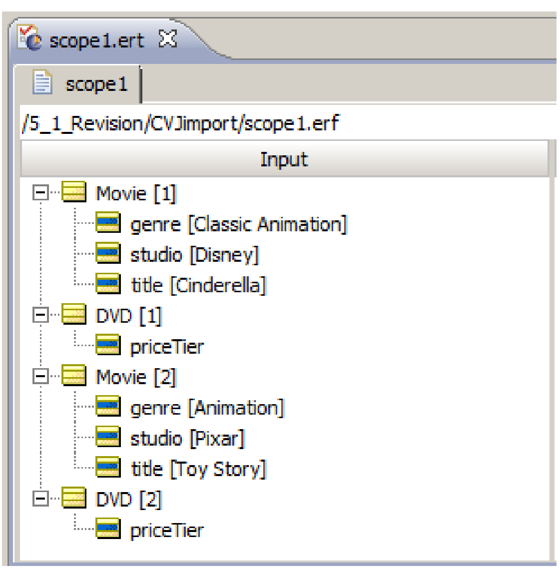
- If the Distributor offers any drama DVDs…

Given the rule "Disney animated classics are priced in the high tier", answer the following questions:

14. Which term should be used to represent Movie?
15. Which term should be used to represent DVD?
16. True or False. The following `Rulesheet` correctly relates the Movie and DVD entities?

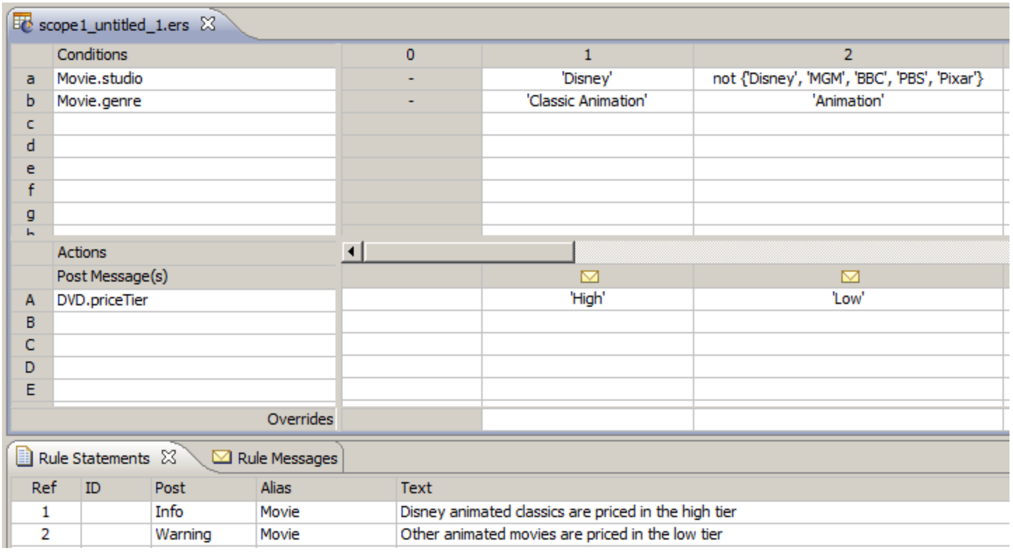


17. Given our business intent, how many times do we want the rule to fire given the Input Testsheet below?



18. Given the Ruletest Input above, how many times does the rule actually fire?

19. Assume we update the `Rulesheet` to include another rule, as shown below. Answer the following questions:



- Assuming the same Ruletest Input as question 57, what result do we *want* for Cinderella?
- What result do we *want* for Toy Story?
- What results do we *get* when the Test is executed?
- How many times does *each* rule fire?
- How many *total* rule firings occurred?
- This set of combinations is called a ________________
- Does our result make business sense?
- What changes should be made to the `Rulesheet` so that it functions as we intend?

20. True or False. Whenever our rules contain scope, we must define aliases in the Scope section of the `Rulesheet`.

21. Scope is another way of defining a specific _______________ in the Vocabulary

22. If you change the spelling of an alias in the Scope section, then everywhere that alias is used in the `Rulesheet` will:

turn red	be deleted	be updated	be ignored

23. True or False. The spelling of an alias may be the same as the Vocabulary entity it represents?

3

Rule writing techniques – logical equivalents

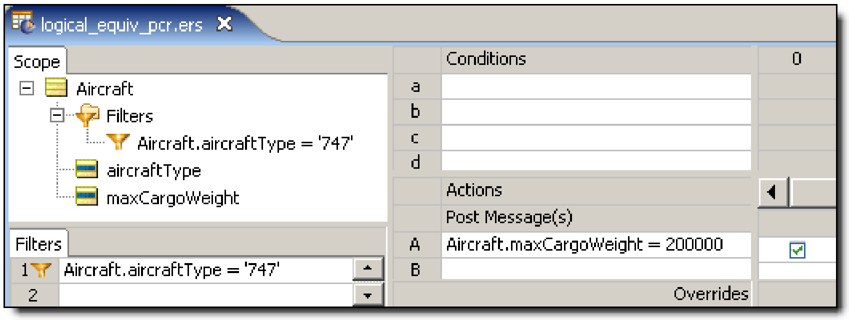
The Corticon Studio *Rulesheet* is a very flexible device for writing and organizing rules. It is often possible to express the same business rule multiple ways in a *Rulesheet*, with all forms producing the same logical results. Some common examples, as well as their advantages and disadvantages, are discussed in this chapter.

For details, see the following topics:

Filters vs. conditions

The Filters section of a *Rulesheet* can contain one or more "master" conditional expressions for that *Rulesheet*. In other words, other business rules will fire if and only if data a) survives the Filter, and b) shares the same scope as the rules. Using our air cargo example from the previous chapter, we model the following rule:

Figure 62: *`Rulesheet`* Using a Filter and Nonconditional Rule

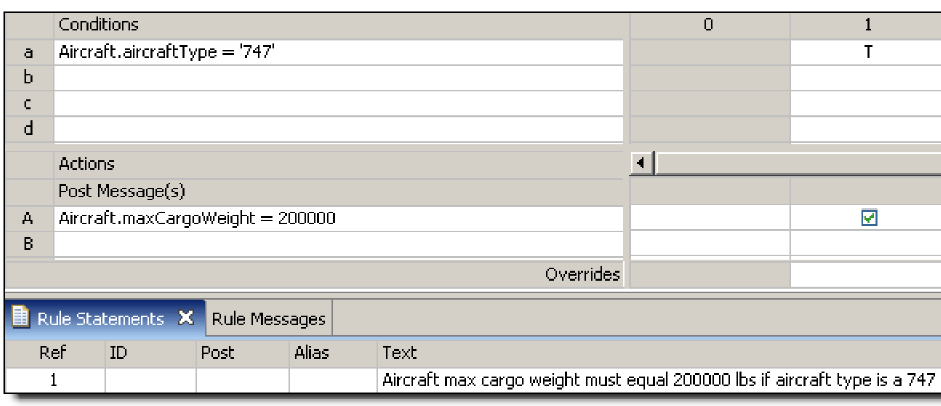


Here, the value of an Aircraft's `maxCargoWeight` attribute is assigned by column 0 in the Conditions/Actions pane (what we sometimes call a "Nonconditional" rule because it has no Conditons). The Filter acts as a master conditional expression because only Aircraft that satisfy the Filter - in other words, only those aircraft of `aircraftType = '747'`, successfully "pass through" to be evaluated by rule column 0, and are assigned a `maxCargoWeight` of `200,000`. This effectively "filters out" all non-747 aircraft from evaluation by rule column 0.

If this Filter were not present, *all* `Aircraft`, regardless of `aircraftType`, would be assigned a `maxCargoWeight` of `200,000` lbs. Using this method, additional *`Rulesheet`*s may be used to assign different `maxCargoWeight` values for each `aircraftType`. The Filter section may be thought of as a convenient way to quickly add the same conditional expression or constraint to all other rules in the same *`Rulesheet`*.

We can also achieve the same results without using Filters. The following figure shows how we use a Condition/Action rule to duplicate the results of the previous *`Rulesheet`*. The rule is restated as an "if-then" type of statement: "if the `aircraftType` is `747`, then its `maxCargoWeight` equals `200,000` lbs".

Figure 63: *`Rulesheet`* Using a Conditional Rule



Regardless of how you choose to express logically equivalent rules in a *`Rulesheet`*, the results will also be equivalent

---

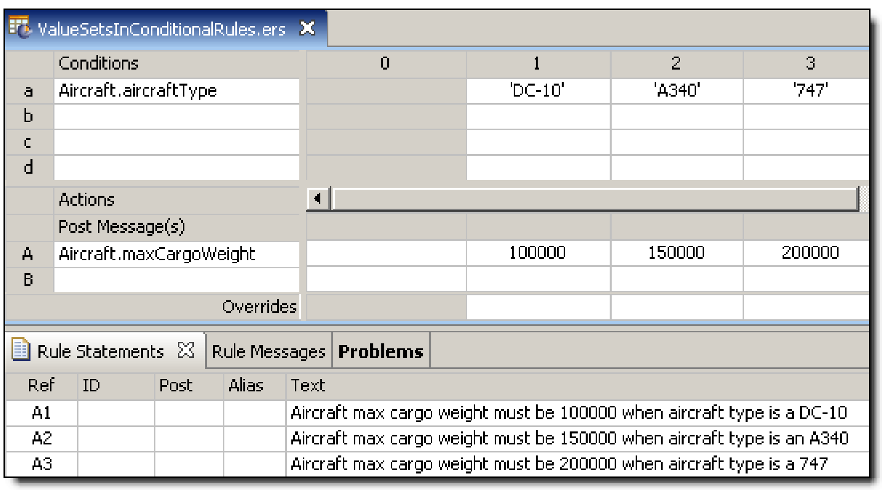
Note: While the logical result may be identical, the time required to produce those results may not be. See Optimizing Rulesheets in the Logical Validation chapter of this Guide for details.

That said, there may be times when it is advantageous to choose one way of expressing a rule over another, at least in terms of the visual layout, organization and maintenance of the business rules and *Rulesheet*s. The example discussed in the preceding paragraphs was very simple because only one Action was taken as a result of the Filter or Condition. In cases where there are multiple Actions that depend on the evaluation of one or more Conditions, it may make the most sense to use the Filters section. Conversely, there may be times when using a Condition makes the most sense, such as the case where there are numerous values for the Condition that each require a different Action or set of Actions as a result. In our example above, there are different types of Aircraft in the company's fleet, and each has a different `maxCargoWeight` value assigned to it by rules. This could easily be expressed on one *Rulesheet* by using a single row in the Conditions section. It would require many *Rulesheet* s to express these same rules using the Filters section. This leads us to the next topic of discussion.

Boolean condition Vs. values set

Rulesheet Using a Conditional Rule illustrates a simple Boolean Condition that evaluates to either `True` or `False`. The Action related to this Condition is either selected or not, on or off, meaning the value of `maxCargoWeight` is either assigned the value of `200,000` or it is not (Action statements are "activated" by selecting the check box that automatically appears when the cell is clicked). However, there is another way to express both Conditions and Actions using Values sets.

Figure 64: Rulesheet Illustrating use of Multiple values in the same Condition Row



By using different values in the column cells of Condition and Action rows in **Rulesheet Illustrating use of Multiple values in the same Condition Row**, we can write multiple rules (represented as different columns in the table) for different Condition-Action combinations. Expressing these same rules using Boolean expressions would require many more Condition and Action rows, and would fail to take advantage of the semantic pattern these three rules share.

Exclusionary syntax

The following examples are also logically equivalent:

Figure 65: Exclusionary Logic Using Boolean Condition, Pt. 1

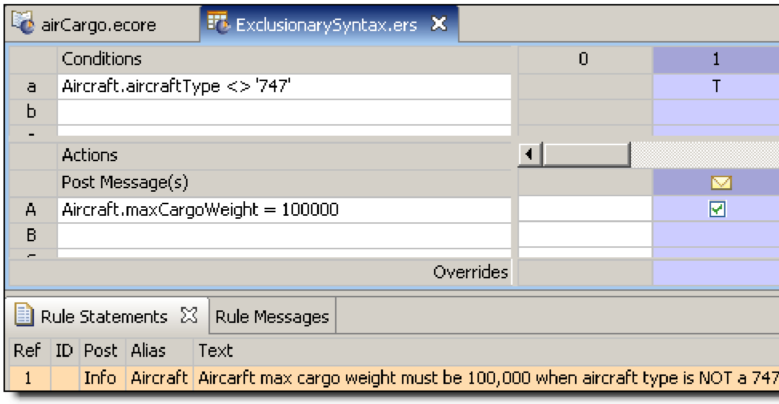


Figure 66: Exclusionary Logic Using Boolean Condition, Pt. 2

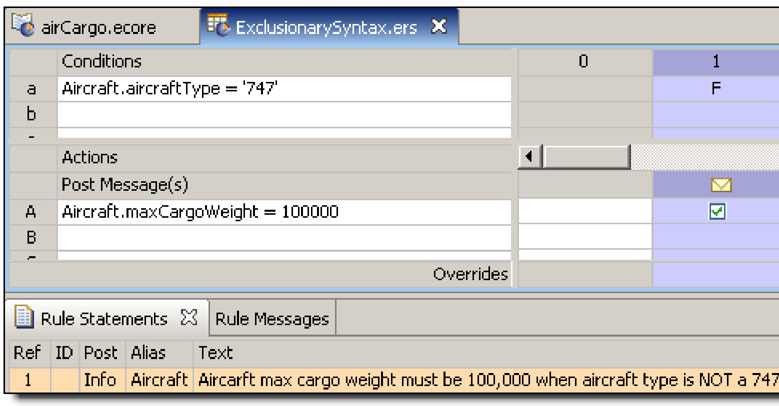
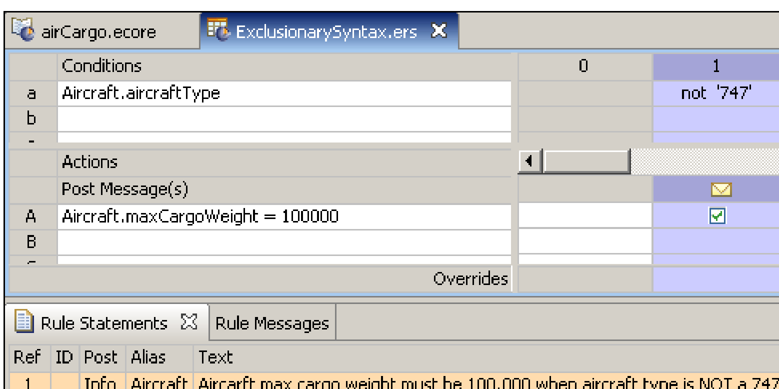


Figure 67: Exclusionary Logic Using Negated Value



Notice that that the last example uses the unary function `not`, described in more detail in the *Rule Language Guide*, to negate the value `747` selected from the Values set.

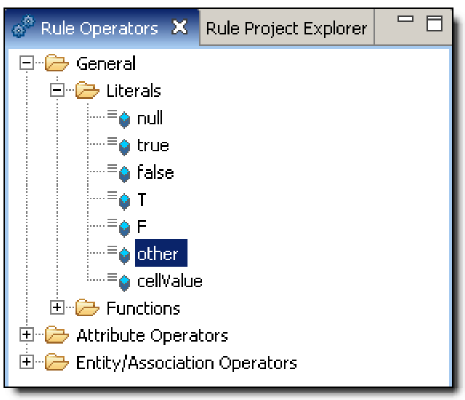
Once again we see that the same rule can be expressed in different ways on the `Rulesheet`, with identical results. It is left to the rule modeler to decide which way of expressing the rule is preferable in a given situation. We recommend, however, avoiding double negatives. Most people find it easier to understand `attribute=T` instead of `attribute<>F`, even though logically the two expressions are equivalent

Note: This assumes bi-value logic. If tri-value logic is assumed (such as, for a non-mandatory attribute), meaning the null value is available in addition to true and false, then these two expressions are not equivalent. If attribute = null, then the truth value of attribute<>F is true while that of attribute=T is false.

Using "other" in condition cells

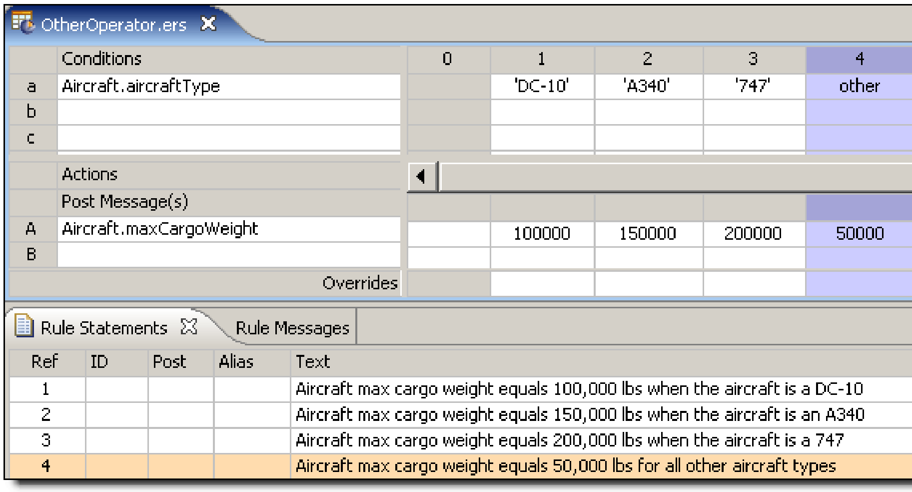
Sometimes it is easier to define values we don't want matched than it is to define those we do. In the example shown above in **Exclusionary Logic Using Negated Value**, we specify a `maxCargoWeight` to assign when `aircraftType` is not a `747`. But what would we write in the Conditions Cell if we wanted to specify any `aircraftType` *other than* those specified in *any of the other* Conditions Cells? For this, we use a special term in the Operator Vocabulary named `other`, shown in the following figure:

Figure 68: Literal Term `other` in the Operator Vocabulary



The term `other` provides a simple way of specifying any value *other than* any of those specified in other Cells of the same Conditions row. The following figure illustrates how we can use `other` in our example.

Figure 69: Rulesheet Using `other` in a Condition Cell

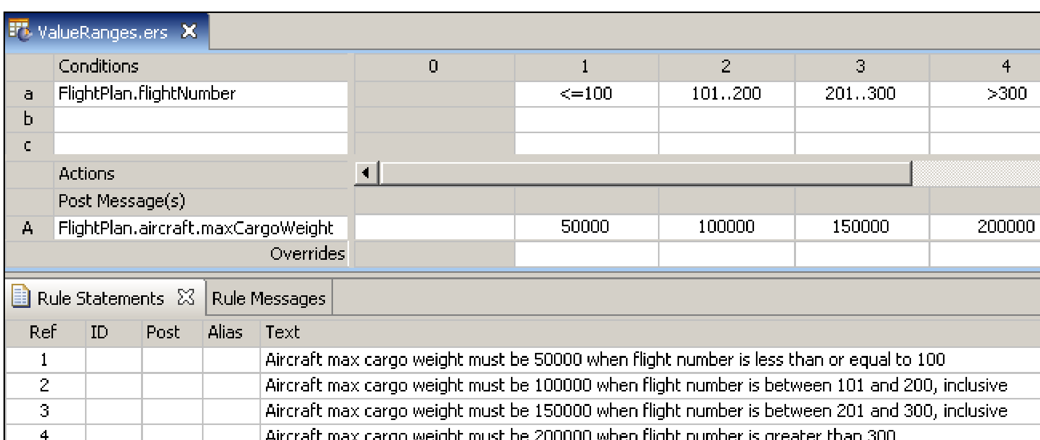


Here, we added a new rule (column 4) that assigns a `maxCargoWeight` of `50000` to any `aircraftType` *other than* the specific values identified in the cells in Condition row a (for example, a `727`). Our *`Rulesheet`* is now complete because all possible Condition-Action combinations are explicitly defined by columns in the decision table.

Use of ranges in condition cells

When using values in Condition Cells for attributes of any data type except Boolean, the values do not need to be discreet – they may be in the form of a range. A value range is typically expressed in the following format: `x..y`, where `x` and `y` are the starting and ending values for the range *inclusive* of the endpoints if there is no other notation to indicate otherwise. This is illustrated in the following figure:

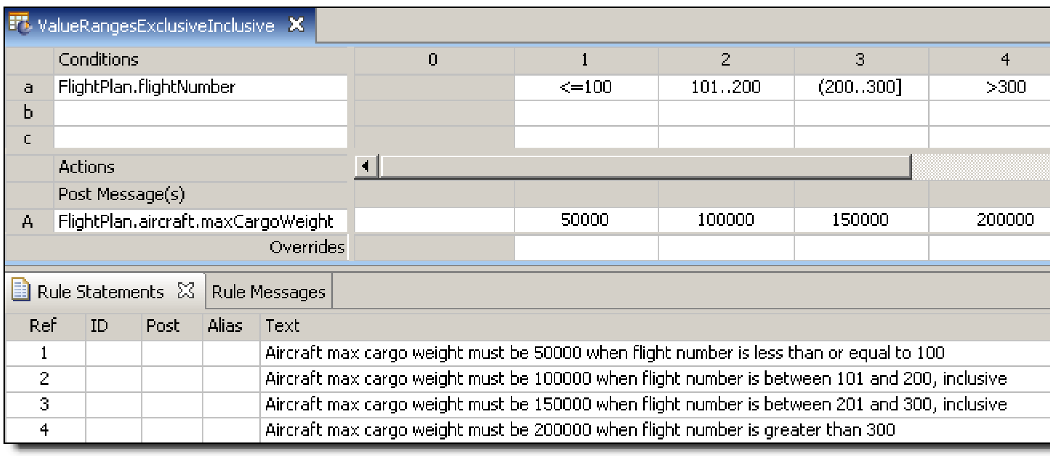
Figure 70: Rulesheet Using Value Ranges in the Column Cells of a Condition Row



In this example, we are assigning a `maxCargoWeight` value to each `Aircraft` depending on the `flightNumber` value from the `FlightPlan` that the `Aircraft` is associated with. The value range `101..200` represents all values (Integers in this case) between `101` and `200`, including the range "endpoints" `101` and `200`. This is an inclusive range in that the starting and ending values are included in the range.

Corticon Studio also gives you the option of defining value ranges where one or both of the endpoints are "exclusive", meaning that they are **not** included in the range of values – this is the same idea as the difference between "greater than" and "greater than or equal to". The following figure, **Rulesheet Using Open-Ended Value Ranges in Condition Cells**, shows the same *`Rulesheet`* shown in the previous figure, but with one difference: we have changed the value range `201..300` to `(200..300]`. The starting parenthesis `(` indicates that the starting value for the range, `200`, is exclusive – it is **not** included in the range. The ending bracket `]` indicates that the ending value is inclusive. Since `flightNumber` is an Integer value and there are therefore no fractional values allowed, `201..300` and `(200..300]` are equivalent.

Figure 71: Rulesheet Using Open-Ended Value Ranges in Condition Cells



Listed below are all of the possible combinations of parenthesis and bracket notation for value ranges and their meanings:

Figure 72: Rulesheet Using Open-Ended Value Ranges in Condition Cells

`(x..y)` - is the range between `x` & `y`, excluding both `x` & `y`

`(x..y]` - is the range between `x` & `y`, excluding `x` and including `y`

`[x..y)` - is the range between `x` & `y`, including `x` and excluding `y`

`[x..y]` - is the range between `x` & `y`, including both `x` & `y`

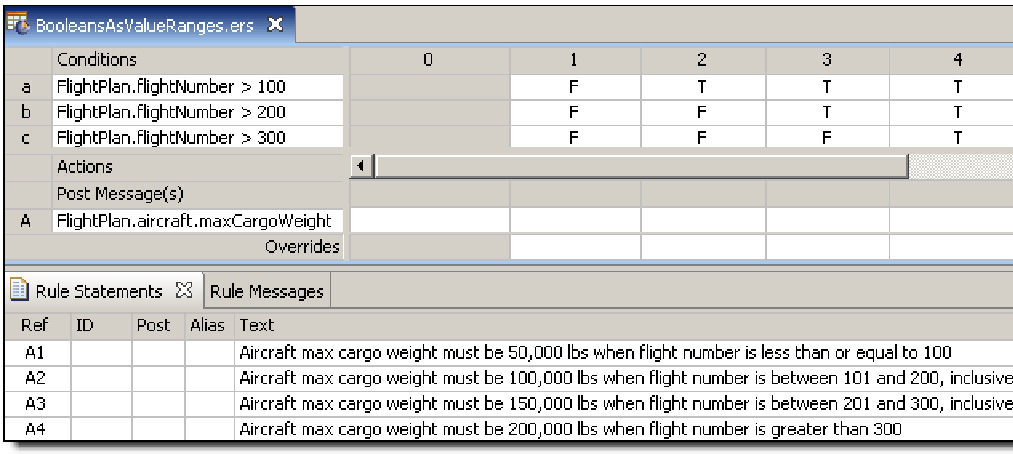
As illustrated in columns 2-3 of **Rulesheet Using Value Ranges in the Column Cells of a Condition Row** and column 2 of **Rulesheet Using Open-Ended Value Ranges in Condition Cells**, if a value range has no enclosing parentheses or brackets, it is assumed to be inclusive. It is therefore not necessary to use the `[..]` notation for a closed range in Corticon Studio; in fact, if you try to create a value range with `[..]` in Corticon Studio, the square brackets will be automatically removed. However, should either end of a value range have a parenthesis or a bracket, then the other end must also have a parenthesis or a bracket. For example, `x..y)` is not allowed, and is properly expressed as `[x..y)`.

Value ranges can also be used in the Filters section of the *`Rulesheet`*. See the *Rule Language Guide's Special Syntax* chapter for full details on usage.

Alternatives to value ranges

As you might expect, there is another way to express a rule which contains a range of values. One alternative is to use a series of Boolean Conditions that cover the ranges of concern. This is illustrated in the following figure:

Figure 73: Rulesheet Using Boolean Conditions to Express Value Ranges



The rules here are identical to the rules in **Rulesheet Using Value Ranges in the Column Cells of a Condition Row** and **Rulesheet Using Open-Ended Value Ranges in Condition Cells**, but are expressed using a series of three Boolean Conditions. Recall that in a decision table, values aligned vertically in the same column represent **AND** 'ed Conditions in the rule. So rule 1, as expressed in column 1, reads:

if `flightNumber` is not greater than `100` and `flightNumber` is not greater than `200` and `flightNumber` is not greater than `300`, then its `maxCargoWeight` must equal `50,000` lbs.

Expressing this rule in friendlier, more natural English, we might say:

An Aircraft's max cargo weight must be 50,000 lbs when flight number is less than or equal to 100.

This is how the rule is expressed in the **Rule Statements** section in **Rulesheet Using Boolean Conditions to Express Value Ranges**. The same rules may also be expressed using a series of *`Rulesheet`*s with the applicable range of `flightNumber` values constrained by Filters. Corticon Studio gives you the flexibility to express and organize your rules any number of possible ways – as long as the rules are logically equivalent, they will produce identical results when executed.

In the case of rules involving numeric value ranges as opposed to discrete numeric values, the value range option allows you to express your rules in a very simple and elegant way. It is especially useful when dealing with Decimal type values.

Standard boolean constructions

A decision table is a graphical method of organizing and formalizing logic. But if you have a background in computer science or formal logic, you may have seen alternative methods. One such method is called a truth table.

Test yourself - questions – rule writing techniques – logical equivalents

Note: Try this test, and then go to **Answers:** Rule writing techniques – logical equivalents on page 277 to correct yourself.

1. Filters act as master rules for all other rules in the same `Rulesheet` that share the same _________.
2. An expression that evaluates to a True or False value is called a _________ expression.
3. True or False. Condition row values sets must be complete.
4. True or False. Action row values sets must be complete.
5. The special term __________ can be used to complete any Condition row values set.
6. What operator is used to negate a Boolean expression?
7. If a Boolean expression is written in a Condition row, what values are automatically entered in the Values set when **Enter** is pressed?
8. A Filter expression written as `Entity.boolean1=T` is equivalent to (circle all that apply)

`Entity.boolean1`	`Entity.boolean1<>F`	`Entity.boolean1=F`	`not (Entity.boolean1=F)`

9. Of all alternatives listed in Question 71, which is the best choice? Why?
10. Describe the error (if any) in each of the following value ranges. Assume all are used in Conditions values sets.
    a. {1…10, other}
    b. {1..a, other}
    c. {'a'..other}
    d. {1..10, 5..20, other}
    e. {1..10, [10..20), other}
    f. {'red', 'green', 'blue'}
    g. {<0, 0..15, >3}
11. True or False. The special term `other` may be used in Action row values sets.
12. Using best practices discussed in this chapter, model the following rules on a single `Rulesheet`
    - If the part is in stock and it has a blue tag, then the part's discount is 10%
    - If the part is in stock and it has a red tag, then the part's discount is 15%
    - If the part is in stock and it has a yellow tag, then the part's discount is 20%
    - If the part is in stock and it has a green tag, then the part's discount is 25%
    - If the part is in stock and it has any other color tag, then the part's discount is 5%

13. True or False. A Nonconditional rule is equivalent to an Action expression with no Condition.
14. True or False. A Nonconditional rule is governed by any Preconditions on the same *`Rulesheet`*.

4

Collections

Collections enable operations to be performed on a set items specified by an alias.

For details, see the following topics:

Understanding how Corticon Studio handles collections

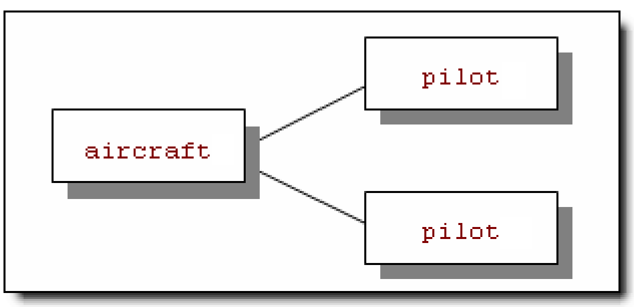
Support for using collections is extensive in Corticon Studio – in fact, the integration of collection support in the Rules Language is so seamless and complete, the rule modeler will often discover that rules are performing multiple evaluations on collections of data beyond what he/she anticipated! This is partly the point of a declarative environment – the rule modeler need only be concerned with *what* the rules do, rather than *how* they do it. How the system actually iterates or cycles through all the available data during rule execution should not be of concern.

As we saw in previous examples, a rule with term `FlightPlan.aircraft` was evaluated for every instance of `FlightPlan.aircraft` data delivered to the rule, either by an XML message or by a *`Ruletest`* (which are really the same thing, as the *`Ruletest`* simply serves as a quick and convenient way to create XML payloads and send them to the rules). A rule is expressed in Corticon Studio the same way regardless of how many instances of data are to be evaluated by it – contrast this to more traditional *procedural* programming techniques, where "for-do" or "while-next" type looping syntax is often required to ensure all relevant data is evaluated by the logic.

Visualizing collections

Collections of data may be visualized as discrete portions, subsets, or "branches" of the Vocabulary tree – a "parent" entity associated with a set of "child" entities, which we call *elements* of the collection. Looking back at the role example from a previous chapter, the collection of pilots can be illustrated as:

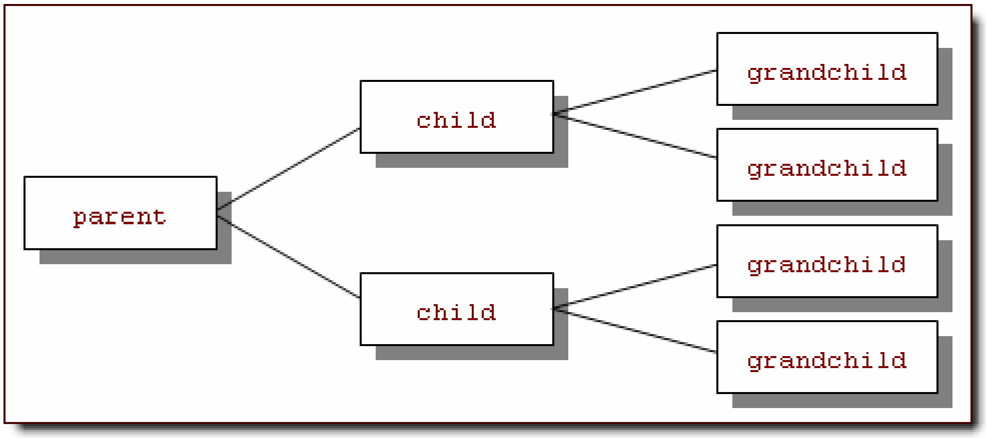
Figure 74: Visualizing a Collection of Pilots



In this figure, the `aircraft` entity is the parent of the collection, while each `pilot` is a child element of the collection. As we saw in the role example, this collection is expressed as `aircraft.pilot` in CRL. It is important to reiterate that this collection contains scope – we are seeing the collection of pilots as they relate to this aircraft. Or, put more simply, we are seeing a plane and its 2 pilots, arranged in a way that is consistent with our Vocabulary. Whenever a rule exists that contains or uses this same scope, it will also *automatically* evaluate this collection of data. And if there are multiple collections with the same scope (for example, several aircraft, each with its own collection of pilots), then the rule will automatically evaluate all those collections, as well. In our lexicon, "evaluate" has a different meaning than "fire". *Evaluate* means that a rule's scope and Conditions will be compared to the data to see if they are satisfied. If they are satisfied, then the rule *fires* and its Actions are executed.

Collections can be much more complex than this simple pilot example. For instance, a collection can comprise more than one type or "level" of association:

Figure 75: 3-Level Collection



This collection is expressed as parent.child.grandchild in CRL. Now let's look at a simple collection operator and understand how it works given the collection in **Visualizing a Collection of Pilots**.

Note: The "parent" and "child" nomenclature is a bit arbitrary. Assuming bidirectional associations, a child from one perspective could also be a parent in another.

A basic collection operator

As an example, let's use the `->size` operator (see the *Rule Language Guide* for more about this operator). This operator returns the number of elements in the collection that it follows in a rule expression. Using the collection from Visualizing a Collection of Pilots:

```
aircraft.pilot -> size
```

returns the value of 2. In the expression:

```
aircraft.crewSize = aircraft.pilot -> size
```

`crewSize` (assumed to be an attribute of Aircraft) is assigned the value of 2.

Corticon Studio requires that all rules containing collection operators use unique aliases to represent the collections. Using aliases to represent collections is described in greater detail in this chapter. A more accurate expression of the rule above becomes:

```
plane.pilot -> size
```

or

```
plane.crewsize = plane.pilot -> size
```

where `plane` is an alias for the collection of `pilots` on `aircraft`.

Filtering collections

The process of screening specific elements from a collection is known as "filtering", and the Corticon Studio supports filtering by a special use of Filter expressions. See the Filters & Preconditions chapter of this manual for more details.

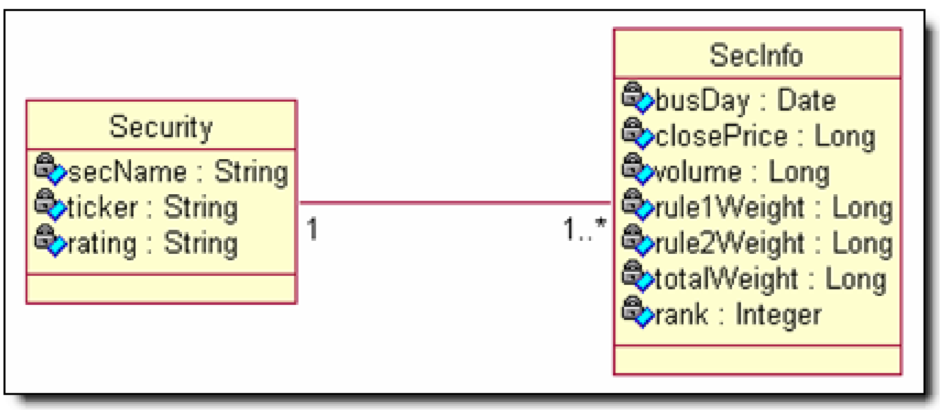
Using aliases to represent collections

Aliases provide a means of using scope to specify elements of a collection; more specifically, we are using aliases (expressed or declared in the Scope section of the `Rulesheet`) to represent *copies* of collections. This concept is important because aliases give you the ability to operate on and compare multiple collections, or even copies of the same collection. There are situations where such operations and comparisons are required by business rules. Such rules are not easy (and sometimes not possible) to implement without using aliases.

Note: To ensure that the system knows precisely which collection (or copy) you are referring to in your rules, it is necessary to use a **unique alias** to refer to each collection.

For the purposes of illustration, we will introduce a new scenario and business Vocabulary. This new scenario involves a financial services company that compares and ranks stocks based on the values of attributes such as closing price and volume. A model for doing this kind of ranking can get very complex in real life; however, we will keep our example simple. Our new Vocabulary is illustrated in a UML class diagram in the following figure:

Figure 76: Security Vocabulary UML Class Diagram



This Vocabulary consists of only two entities:

`Security` – represents a security (stock) with attributes like security name (`secName`), `ticker` symbol, and `rating`.

`SecInfo` – is designed to record information for each stock for each business day (`busDay`); attributes include values recorded for each stock (`closePrice` and `volume`) and values determined by rules (`totalWeight` and `rank`) each business day.

The association between `Security` and `SecInfo` is `1..*` (one-to-many) since there are multiple instances of `SecInfo` data (multiple days of historical data) for each `Security`.

In our scenario, we will use three rules to determine a security's rank:

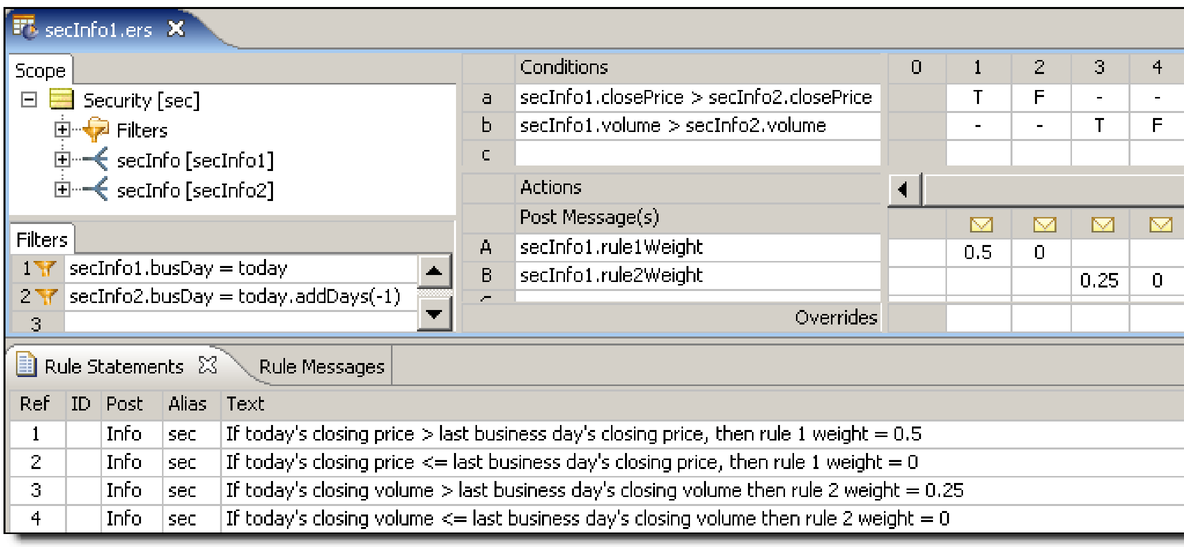
1. A security whose closing price today is higher than its closing price on the previous business c must have a value of 0.5 assigned to its rule 1 weight; otherwise, a value of 0 must be assigne rule 1 weight.

2. A security whose trading volume today is greater than its trading volume on the previous busin day must have a value of 0.25 assigned to its rule 2 weight; otherwise, a value of 0 must be as to its rule 2 weight.

3. A security's total weight is equal to the sum of its rule 1 weight and its rule 2 weight.

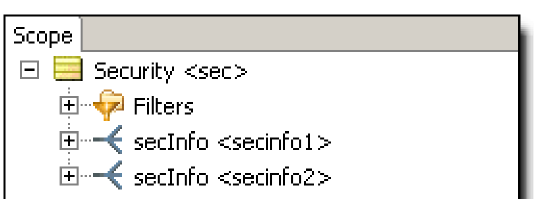
Finally, rules will be used to assign a rank based on the total weight. It is interesting to note that although the rules refer to a security's closing price, volume, and rule weights, these attributes are actually properties of the `SecInfo` entity. *`Rulesheet`*s that accomplish these tasks are shown in the next two figures.

Figure 77: Rulesheet with Ranking Model Rules 1 and 2



In the figure above, we see two business rules expressed in a total of four rule models (one for each possible outcome of the two business rules). The rules themselves are straightforward, but the shortcuts (alias values) used in these rules are different than any we have seen before. In the Scope section, we see the following:

Figure 78: Close-up of the Scope Section from Rulesheet with Ranking Model Rules 1 and 2



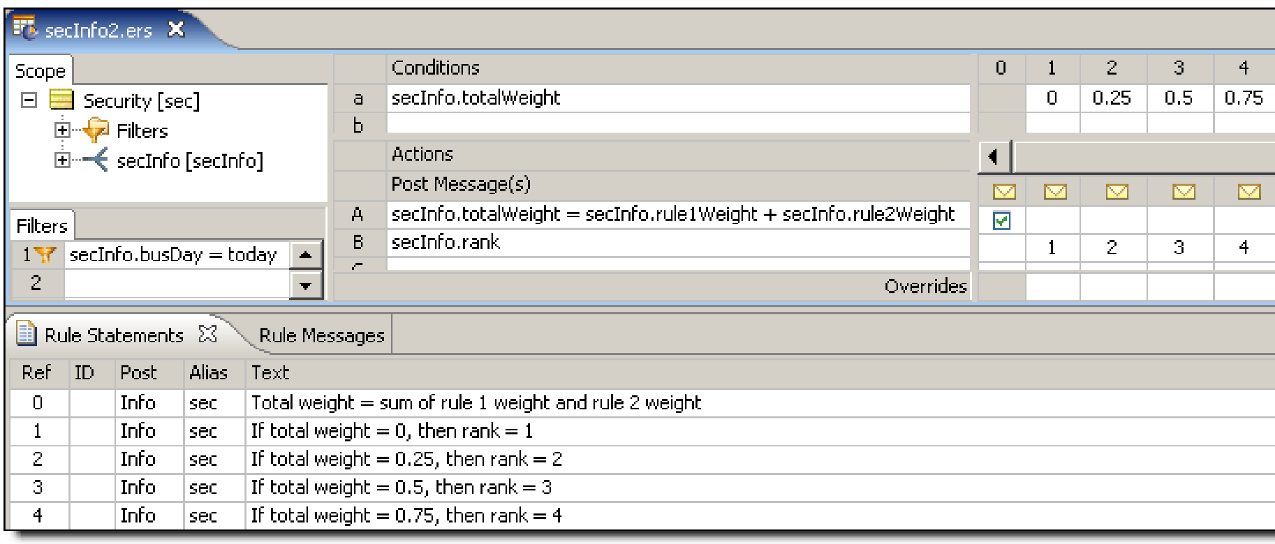
Security is the scope for our *Rulesheet*, which is not a new concept. But then we see that there are two aliases for the SecInfo entities associated with Security: secinfo1 and secinfo2. Each of these aliases represents a separate but identical collection of the SecInfo entities associated with Security. In this *Rulesheet*, we constrain each alias by using Filters; in a later example, we will see how more loosely constrained aliases can represent many different elements in a collection when the *Corticon Server* evaluates rules. In this specific example, though, one instance of SecInfo represents the current business day (today) and the other instance represents the previous business day (today.addDays(-1).)

Note: For details on the .addDays operator, refer to that topic in the *Rule Language Guide*.

Once the aliases are created and constrained, we can use them in our rules where needed. In the figure **Rulesheet with Ranking Model Rules 1 and 2**, we see that the use of aliases in the Conditions section allows comparison of closePrice and volume values from one specific SecInfo element (the one with today's date) of the collection with another (the one with yesterday's date).

The following figure shows a second *Rulesheet* which uses a Nonconditional rule to calculate the sum of the partial weights from our model rules as determined in the first *Rulesheet*, and Conditional rules to assign a rank value between 1 and 4 to each security based on the sum of the partial weights. Since we are only dealing with data from the current day in this *Rulesheet* (as specified in the Filters), only one instance of SecInfo per Security applies and we do not need to use aliases.

Figure 79: Rulesheet with Total Weight Calculation and Rank Determination

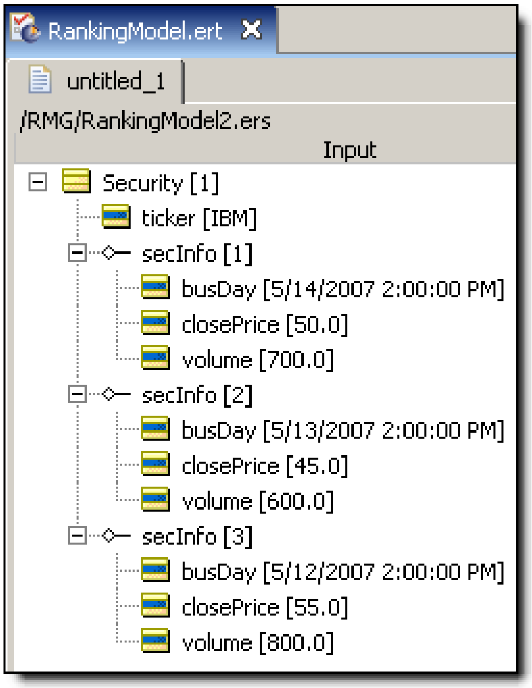


We can test our new rules using a *Ruleflow* to combine the two *Rulesheet*s. In a *Ruletest* which executes the *Ruleflow*, we would expect to see the following results:

1. The Security.secInfo collection that contains data for the current business day (we expect that this collection will reduce to just a single secinfo element, since only one secinfo element exists for each day) should be assigned to alias secinfo1 for evaluating the model rules.
2. The SecInfo instance that contains data for the previous business day (again, the collection filters to a single secinfo element for each Security) should be assigned to alias secinfo2 for evaluating the model rules.
3. The partial weights for each rule, sum of partial weights, and resulting rank value should be assigned to the appropriate attributes in the current business day's SecInfo element.

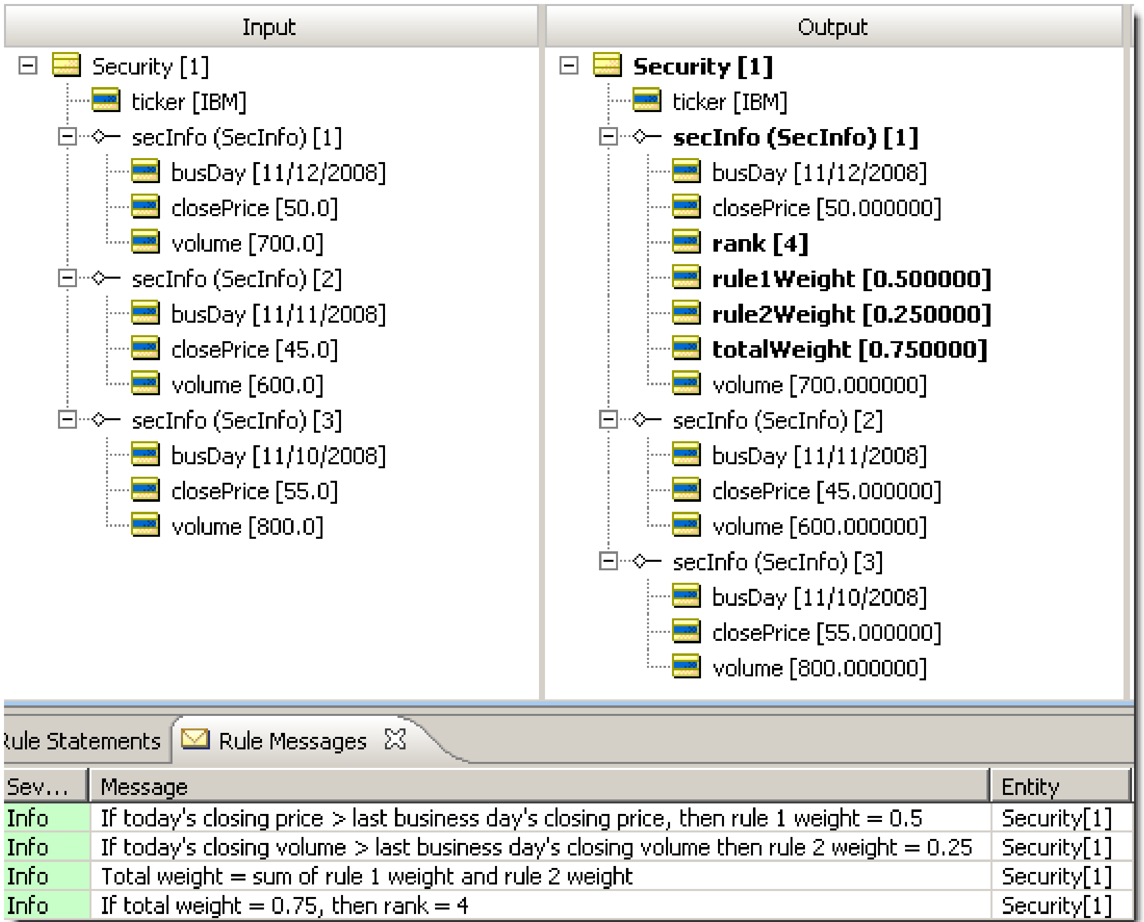
A *Ruleflow* constructed for testing the ranking model rules is as shown:

Figure 80: Ruletest for Testing Security Ranking Model Rules



In this figure, we have added one `Security` object and three associated `SecInfo` objects from the Vocabulary. The current day at the time of the *Ruletest* is `5/14/2003`, so the three `SecInfo` objects represent the current business day and two previous business days. The third business day is included in this *Ruletest* to verify that the rules are using only the current and previous business days – none of the data from the third business day should be used if the rules are executing correctly. Based on the values of `closePrice` and `volume` in our two `SecInfo` objects being tested, we expect to see the highest rank of 4 assigned to our security in the current business day's `SecInfo` object.

Figure 81: Ruletest for Security Ranking Model Rules

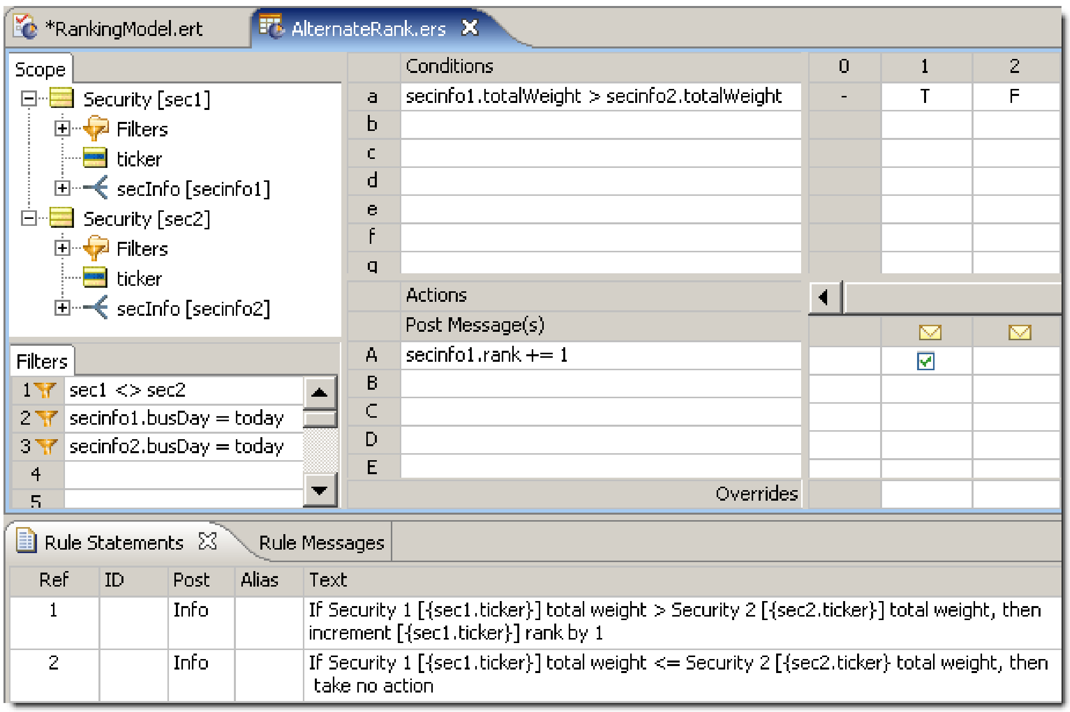


We see the expected results produced above. Both `closePrice` and `volume` for 11/12/2008 were higher than the values for those same attributes on 11/11/2008; therefore, both `rule1Weight` and `rule2Weight` attributes were assigned their "high" values by the rules. Accordingly, the `totalWeight` value calculated from the sum of the partial weights was the highest possible value and a `rank` of `4` was assigned to this security for the current day.

As previously mentioned, the example above was tightly constrained in that the aliases were assigned to two very specific elements of the referenced collections. What about the case where there are multiple instances of an entity that you would like to evaluate with your rules? We will discuss just such an example next.

Our second example is also based on our security ranking scenario but, in this example, the rank assignment that was accomplished will be done in a different way. Instead, we would like to rank a number of securities based on their relative performance to one another, rather than against a preset ranking scheme like the one in Figure 77 on page 87. In the rules for our new example, we will compare the `totalWeight` value that is determined for each security for the current business day against the `totalWeight` of every other security, and determine a `rank` based on this comparison of `totalWeight` values. A `Rulesheet` for this alternate method of ranking securities is shown in the next figure.

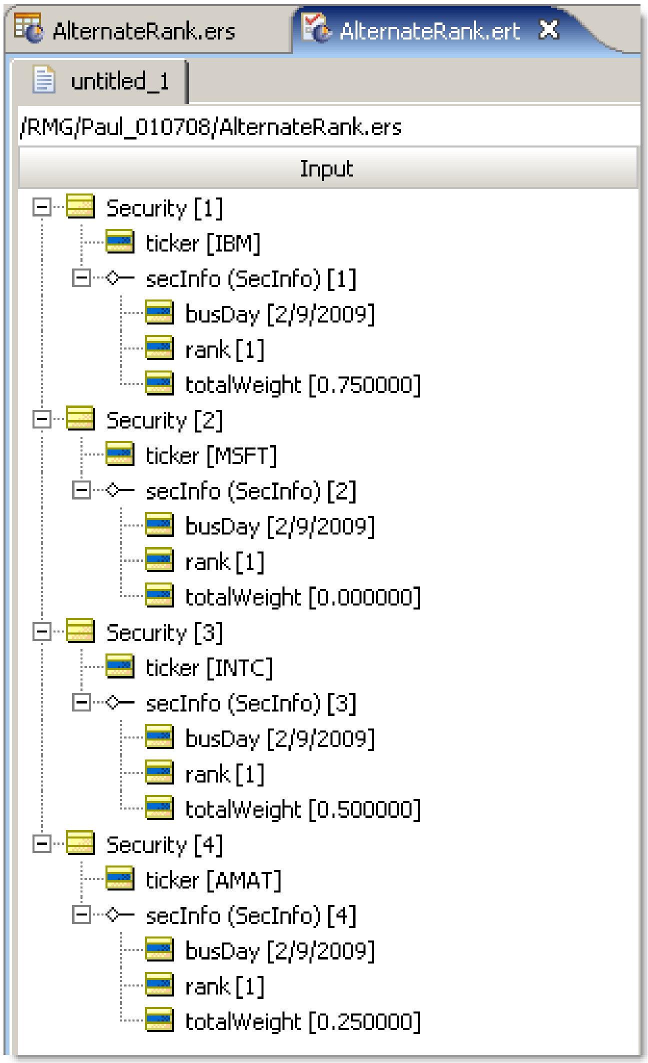
Figure 82: Rulesheet with Alternate Rank Determination Rules



In these new ranking rules, we have created aliases to represent specific instances of `Security` and their associated collections of `SecInfo`. As in the previous example, Filters constrain the aliases, most notably in the case of the `SecInfo` instances, where we filter `secInfo1` and `secInfo2` for a specific value of `busDay` (today's date). However, we have only loosely constrained our `Security` instances – we merely have a Filter that prevents the same element of `Security` from being compared to itself (when `sec1` = `sec2`). No other constraints are placed on the `Security` aliases.

Note that we are not assigning specific, single elements of `Security` to our aliases. Instead, we are instructing the *`Server`* to evaluate all *allowable* combinations (i.e., all those combinations that satisfy the 1st Filter) of `Security` elements in our collection in each of the aliases (`sec1` and `sec2`). For each allowable combination of `Security` elements, we will compare the `totalWeight` values from the associated `SecInfo` element for `busDay = today`, and increment the rank value for the first `SecInfo` element (`secinfo1`) by 1 if its `totalWeight` is greater than that of the second `SecInfo` object (`secinfo2`). The end result should be the relative performance ranking of each security that we want.

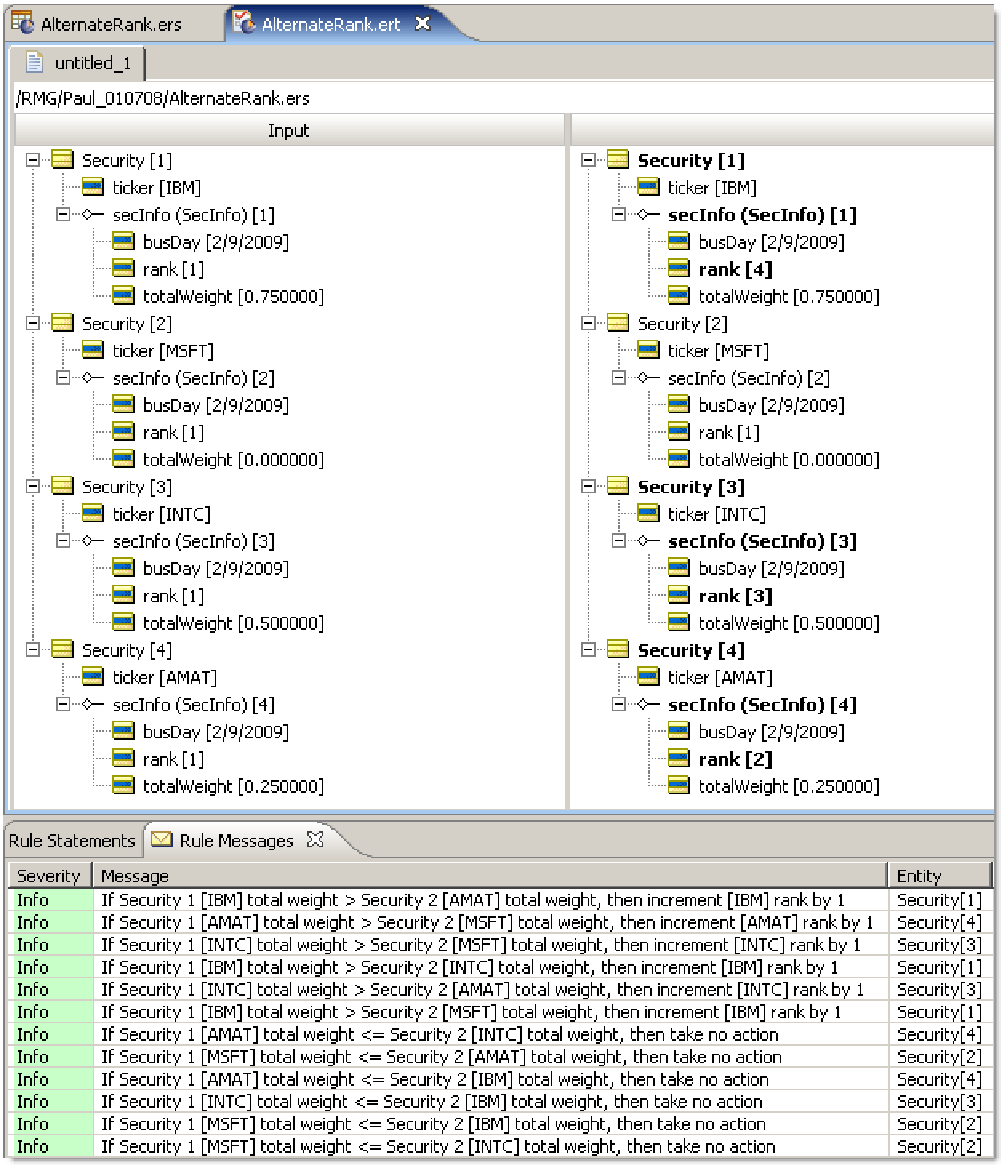
Figure 83: Rulesheet for Testing Alternate Security Ranking Model Rules



This figure shows a *`Ruletest`* constructed to test these ranking rules. In our data, we have added four `Security` elements and an associated `secInfo` element for each (note that each alias will represent ALL 4 `security` elements AND their associated `secInfo` elements). The current day at the time of the *`Ruletest`* is 2/9/2009, so each `Security.secInfo.busDay` attribute is given the value of 2/9/2009 (if we had included additional `secinfo` elements in each collection, they'd have earlier dates, and therefore would be filtered out by the Preconditions on each alias). We have initially set (or "initialized") each `Security.secInfo.rank` equal to 1, so that the lowest ranked security will still have a value of 1. The lowest ranked security will be the one that "loses" all comparisons with the other securities - in other words, its weight is less than the weights of all other securities. If a security's weight is less than all the other security weights, its rank will never be incremented by the rule, so its rank will remain 1. The values of `totalWeight` for the `SecInfo` objects are all different; therefore, we expect to see each security ranked between 1 and 4 with no identical `rank` values.

Note: If there were multiple `Security.secInfo` elements (multiple securities) with the same `totalWeight` value for the same day, then we would expect the final `rank` assigned to these objects to be the same as well. Further, if there were multiple `Security.secInfo` entities sharing the highest relative `totalWeight` value in a given *`Ruletest`*, then the highest `rank` value possible for that *`Ruletest`* would be lower than the number of securities being ranked, assuming we initialize all `rank` values at 1.

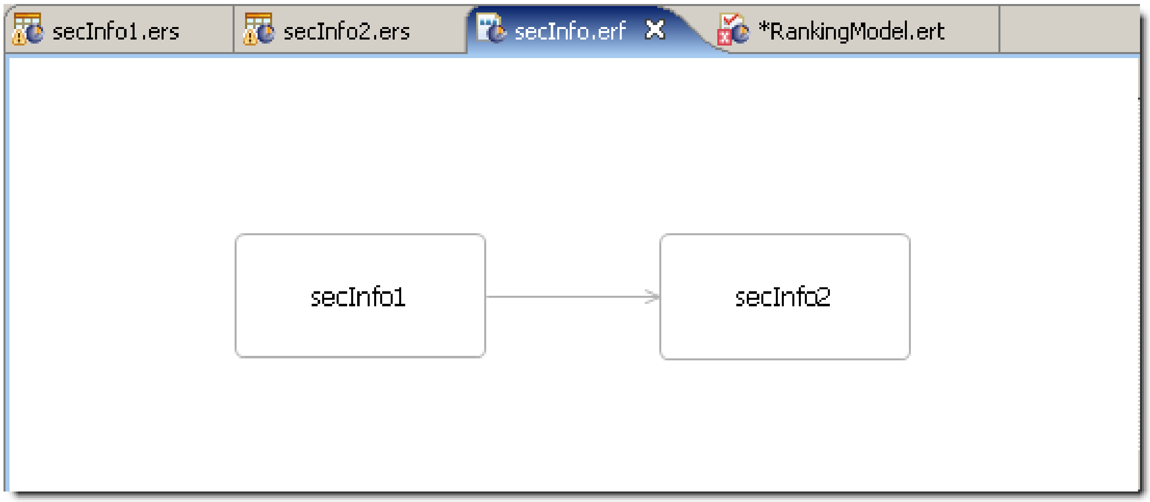
Figure 84: Rulesheet for Alternate Security Ranking Model Rules



In this figure, our *Ruletest* results are as expected. The security with the highest relative `totalWeight` value ends the *Ruletest* with the highest `rank` value after all rule evaluation is complete. The other securities are also assigned `rank` values based on the relative ranking of their `totalWeight` values. The individual rule firings that resulted in these outcomes are highlighted in the message section at the bottom of the results sheet.

It is interesting to note that nowhere in the rules is it stated how many security entities will be evaluated. This is another example of the ability of the declarative approach to produce the intended outcome without requiring explicit, procedural instructions.

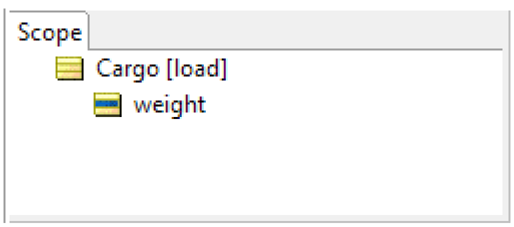
Figure 85: Ruleflow to test two Rulesheets in succession



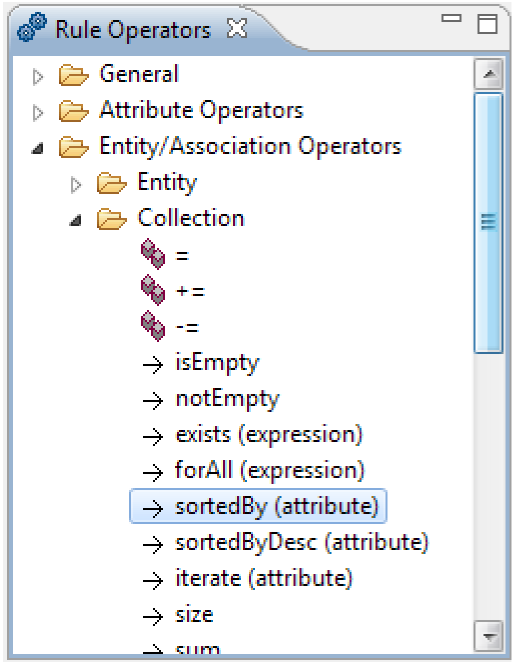
Sorted aliases

You can create a special kind of alias in the Scope section of a Rulesheet. The technique uses the specialized Sequence operator `->next` against a Sorted Alias (a special cached Sequence) inside a filter expression. The Rulesheet is set into a Ruleflow that iterates to bind the alias in each successive invocation to the next element in the sequence.

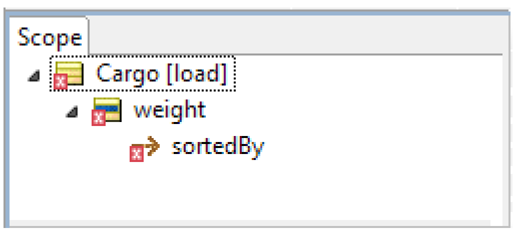
The following example shows a Rulesheet based on the `Cargo` Vocabulary. We brought the `Cargo` entity and its `weight` attribute into the scope:



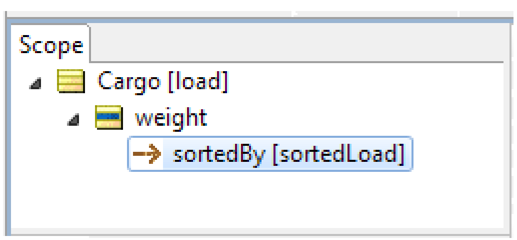
The operators `sortedBy` and `sortedByDesc` enable sorting ascending or descending order of the numeric or alphabetic values of the attribute in the set of data. Note that an attribute with a Boolean data type is not valid for this operation.



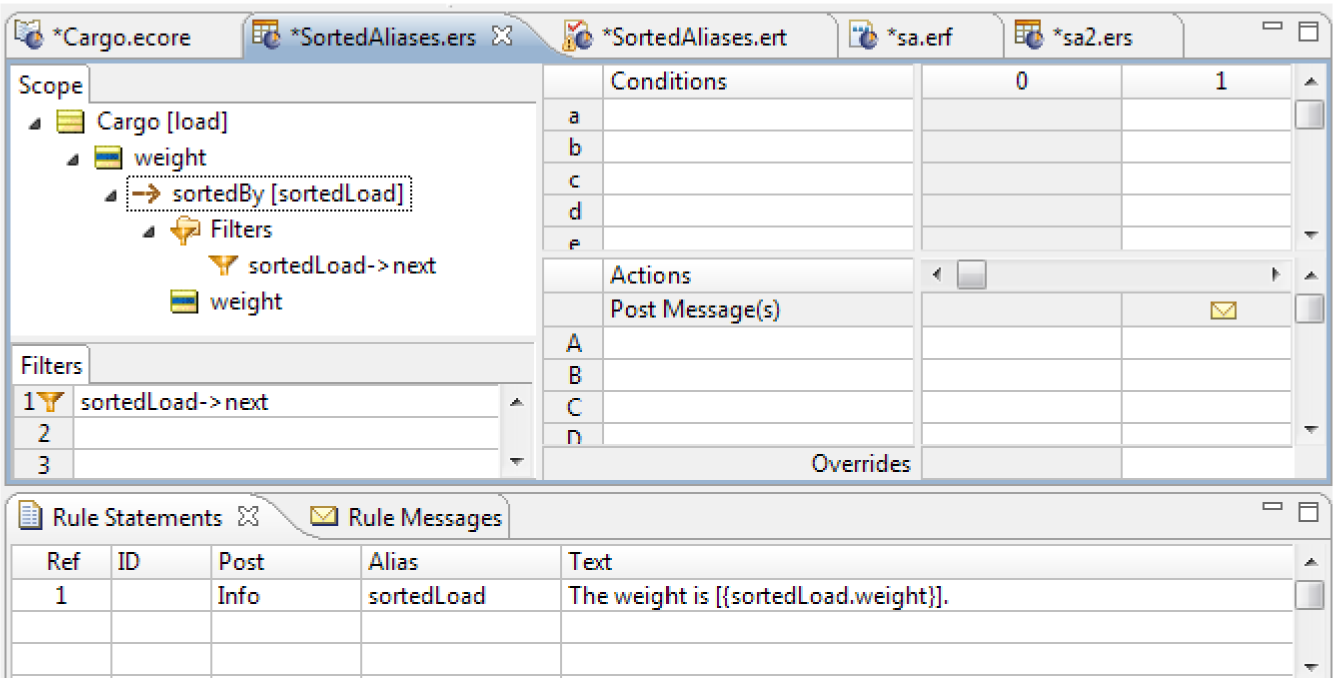
Dragging the `sortedBy` operator and dropping it (you cannot type it in) on the attribute `weight` places it in the scope yet shows an error:



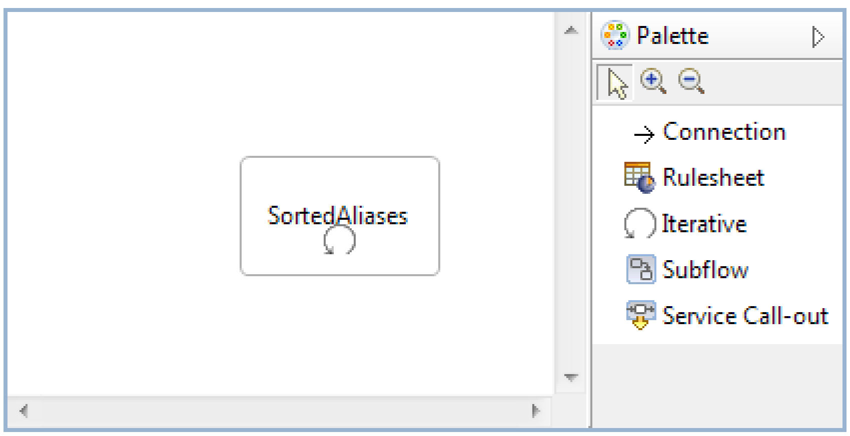
The error message notes that a sorted alias node requires an alias name. When we enter an alias name, the scope is complete.



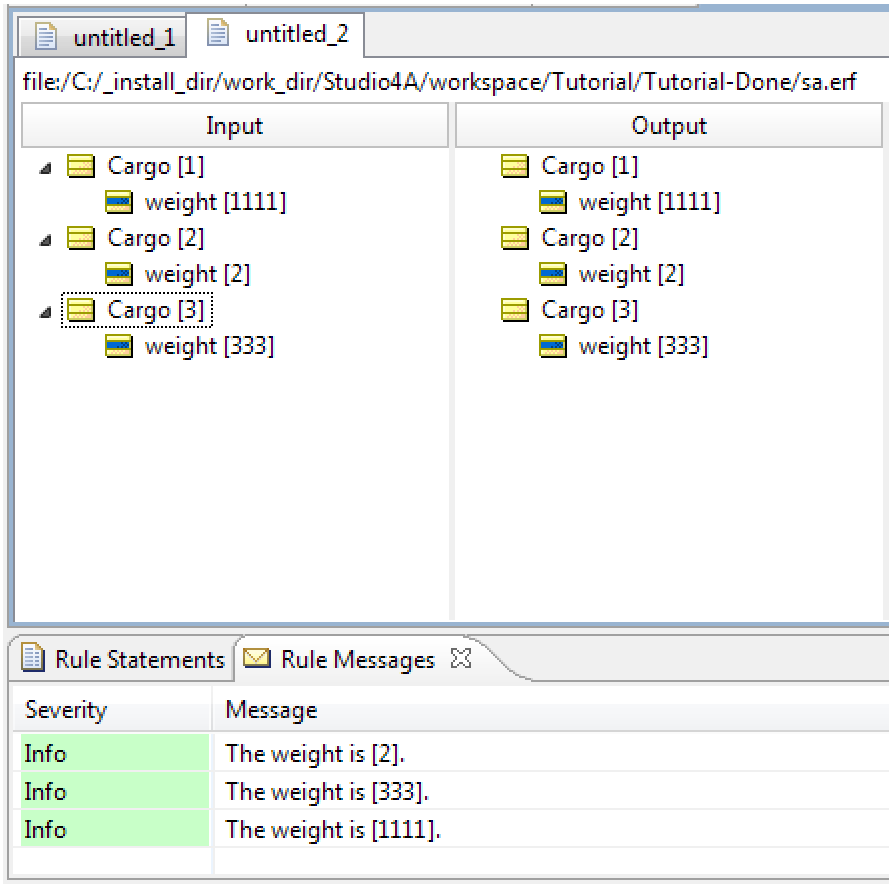
We add a filter expression to establish that, when we iterate through the list, each pass will present the next sequential item in the sorted set. We defined this by dragged `sortedBy` from the scope to filter line 1, and then appended the `->next` operator. We added a rule statement based on sorted load that echoes the weight so we can see the results in our tests.



We saved the Rulesheet and created a Ruleflow, adding in our Rulesheet. Then we dragged an **Iterative** operation to the Rulesheet in the Ruleflow and saved it.



We created a Ruletest with a few Cargo items, each with a weight that we expect to sequence numerically when we run the test. Each iteration posts a message, and that message (the corresponding Rule Statement) contains the embedded attribute load weight. Since each member of the load collection will “trigger” the nonconditional rule, and even though the elements will be processed in no particular order, we expect to see a set of resulting messages with load weight in order. Running the tests repeatedly outputs the weights in ascending order every time.



If we change the operator to `sortByDesc`, the results are shown in *descending* order by weight, as expected.

Singletons

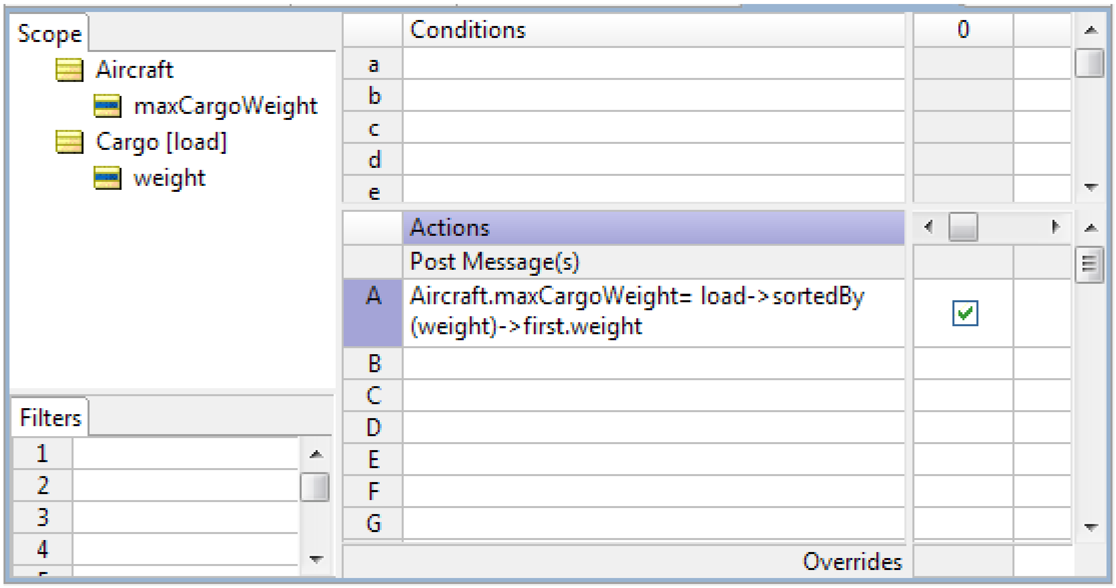
Singletons are collection operations that iterate over a set to extract one arithmetic value - the first, the last, the trend, the average, or the element at a specified position. We saw this behavior when the `sortedAlias` found the first and last element in an iterative list (as well as the elements in between) in the given order.

To examine this feature, we bring the `Aircraft` entity and its `maxCargoWeight` into the scope as well as `Cargo` (with the alias `load`) and its attribute `weight`. The nonconditional action we enter is, literally:

"Show me the maximum cargo weight by examining all the cargo in the load, sorting them by weight from small to large, and returning the smallest one first."

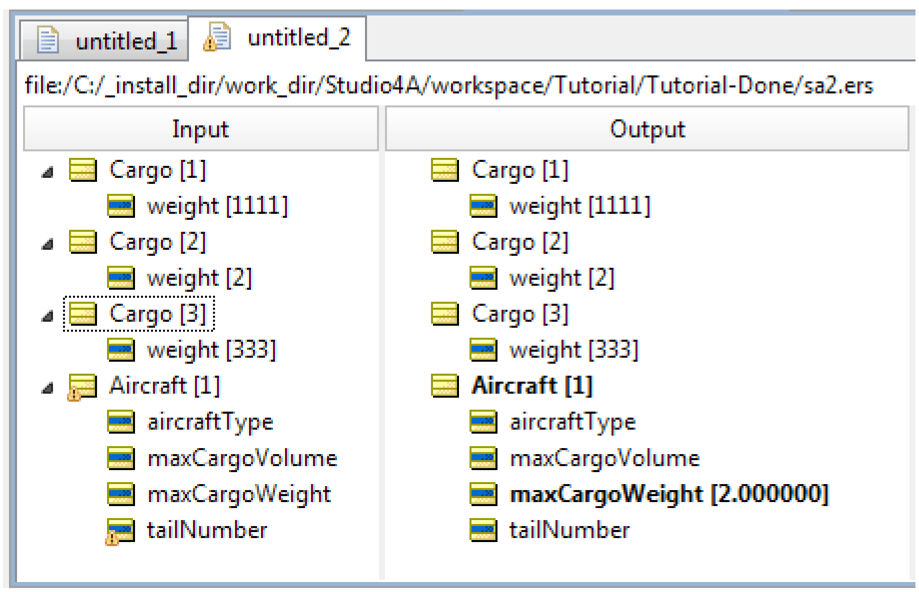
That is entered as:

```
Aircraft.maxCargoWeight=load->sortedBy(weight)->first.weight
```



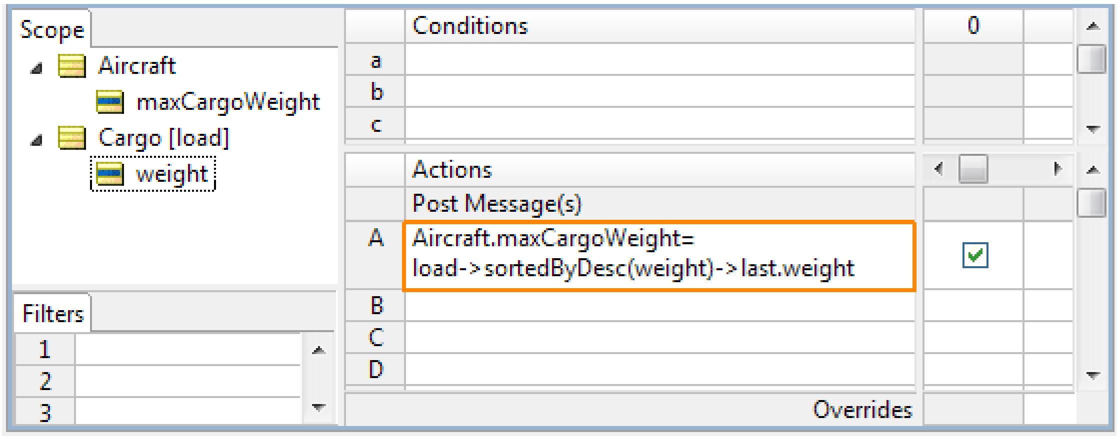
When we extend the test we used for sorted aliases, we need to add an `Aircraft` with `maxCargoWeight` to show the result of the test. The result is as expected - the lightest item passed the test.

Figure 86:



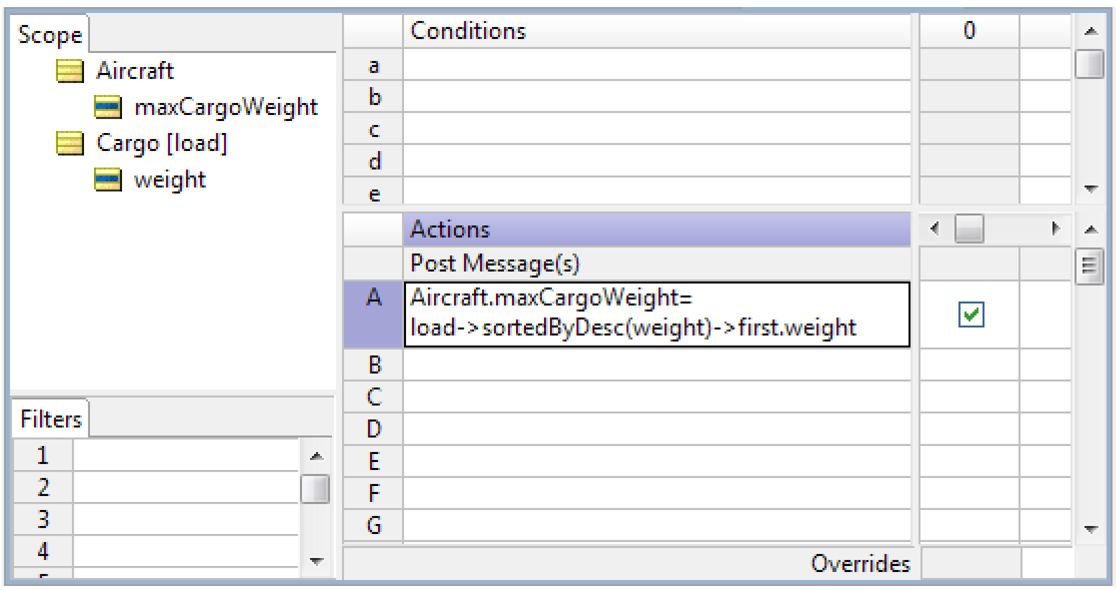
The same result is output when we modify the rule to select last item when we sort the items by descending weight.

Figure 87:



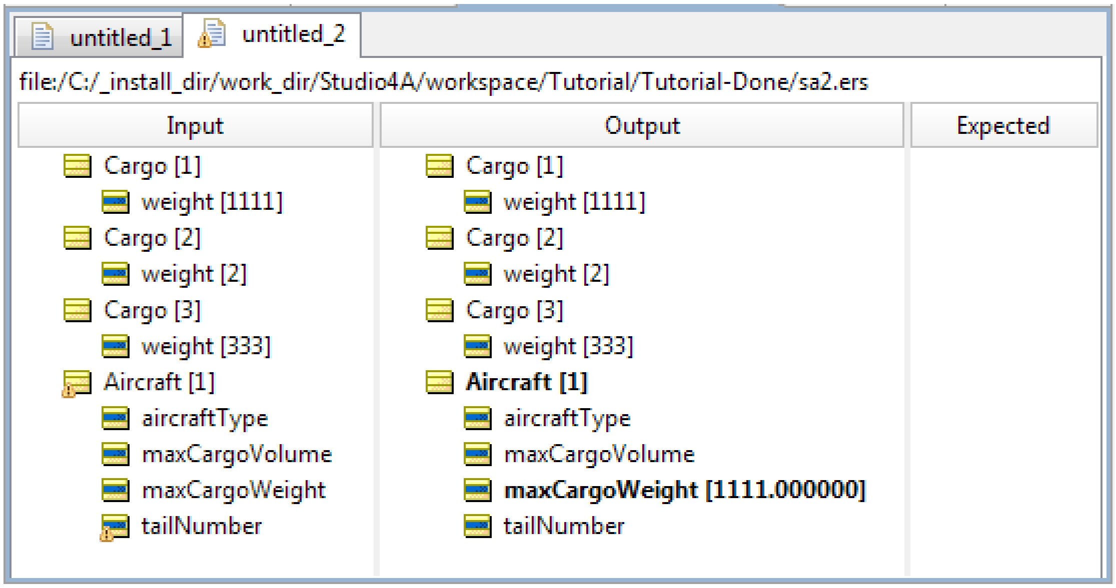
Now we reverse the test to select the first item when we sort the items by descending weight...

Figure 88:



...and the heaviest item is output.

Figure 89:



Note: Singletons do not operate against an iterative Ruleflow as was required by Sorted Aliases. The tests apply directly to the Rulesheet.

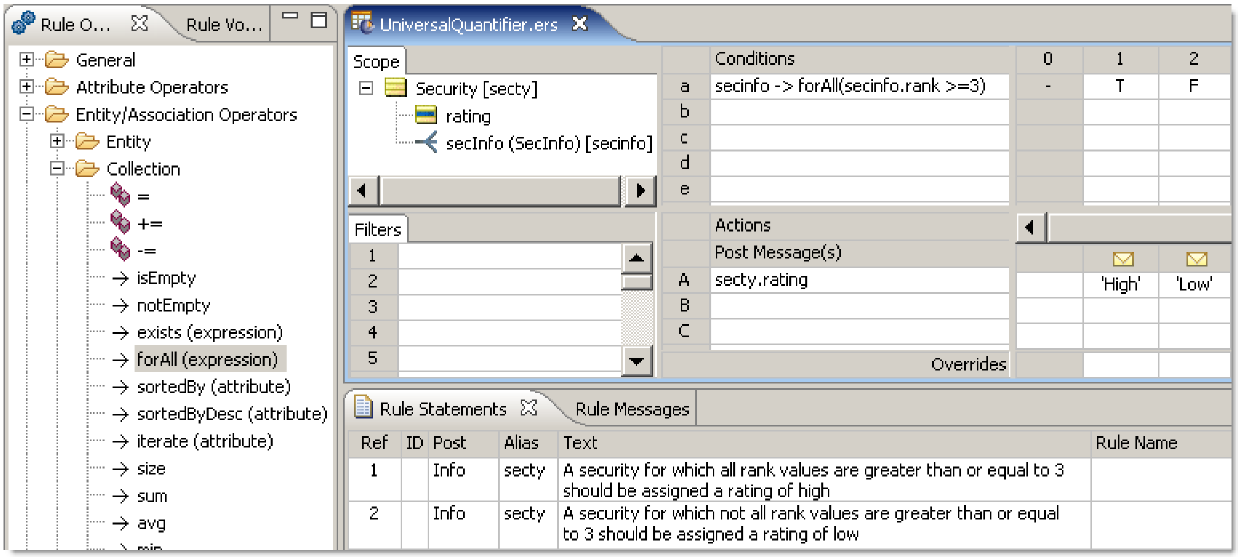
Special collection operators

There are two special collection operators available in `Studio`'s Operator Vocabulary that allow us to evaluate collections for specific Conditions. These operators are based on two concepts from the predicate calculus: the *universal quantifier* and the *existential quantifier*. These operators return a result about the collection, rather than about any particular element within it. Although this is a simple idea, it is actually a very powerful capability – some decision logic cannot be expressed without these operators.

Universal quantifier

The meaning of the universal quantifier is that a condition enclosed by parentheses is evaluated (i.e., its "truth value" is determined) *for all* instances of an entity or collection. This is implemented as the `->forAll` operator in the Operator Vocabulary. We will demonstrate this operator with an example created using the Vocabulary from our security ranking model. Note that these operators act on collections, so all the examples shown will declare aliases in the **Scope** section.

Figure 90: Rulesheet with Universal Quantifier ("for all") Condition



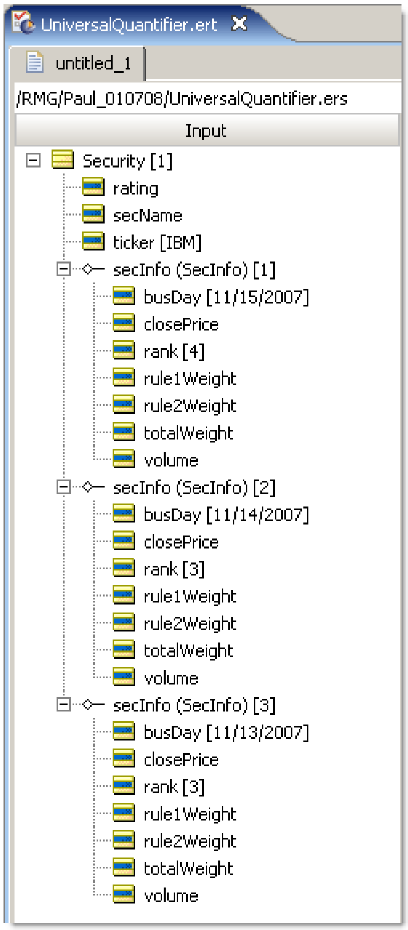
In this figure, we see the Condition

```
secinfo ->forAll(secinfo.rank >= 3)
```

The exact meaning of this Condition is that for the collection of `SecInfo` elements associated with a `Security` (represented and abbreviated by the alias `secInfo`), evaluate if the expression in parentheses (`secinfo.rank >= 3`)is true **for all** elements. The result of this Condition is Boolean because it can only return a value of true or false. Depending on the outcome of the evaluation, a value of either `High` or `Low` will be assigned to the `rating` attribute of our `Security` entity and the corresponding Rule Statement will be posted as a message to the user.

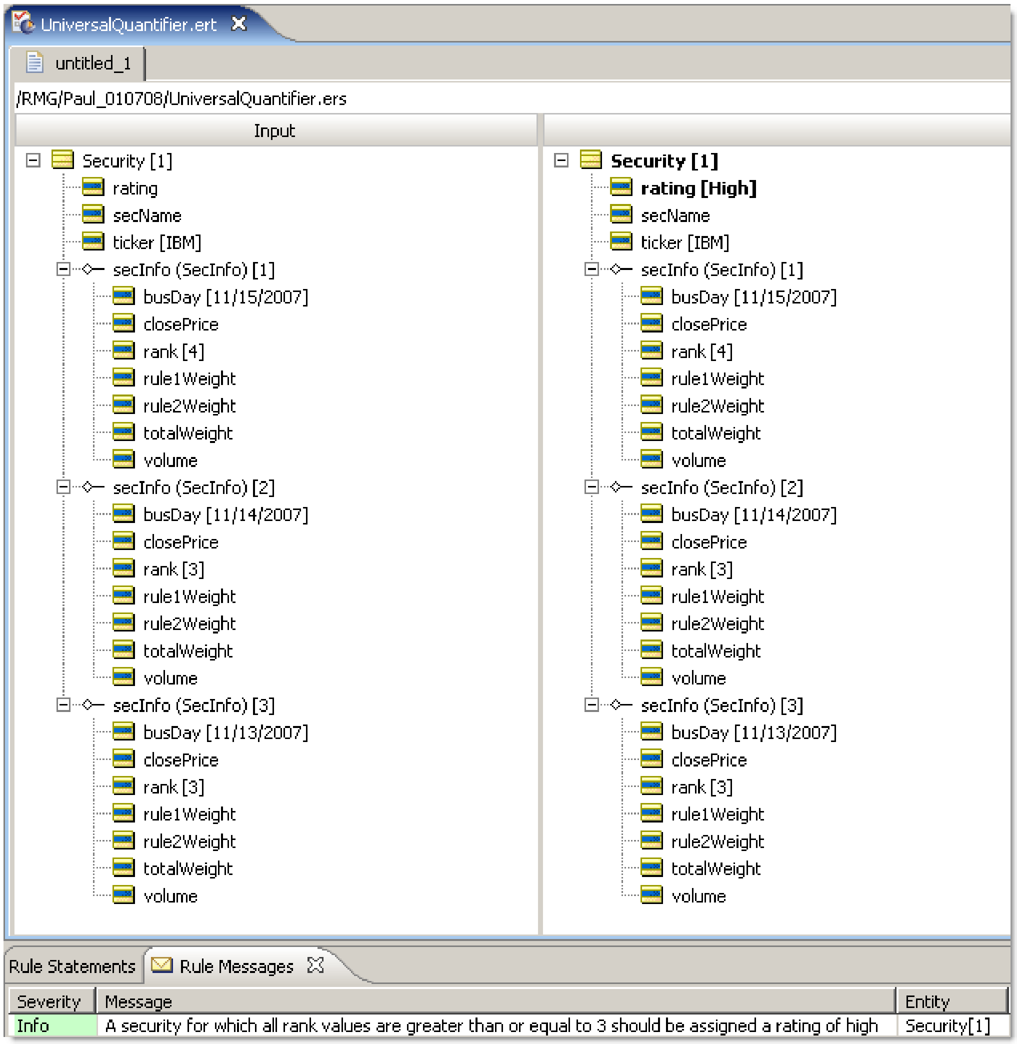
The following figure shows a *Ruletest* constructed to test our "for all" Condition rules.

Figure 91: Ruletest for Testing "for all" Condition Rules



In this *Ruletest*, we are evaluating a collection of three `SecInfo` elements associated with a `Security` entity. Since the `rank` value assigned in each `SecInfo` object is at least 3, we should expect that our "for all" Condition will evaluate to `true` and a rating value of `High` will be assigned to our `Security` object when the *Ruletest* is run through *Corticon Server*. This outcome is confirmed in the *Ruletest* results, as shown:

Figure 92: Ruletest for "for all" Condition Rules

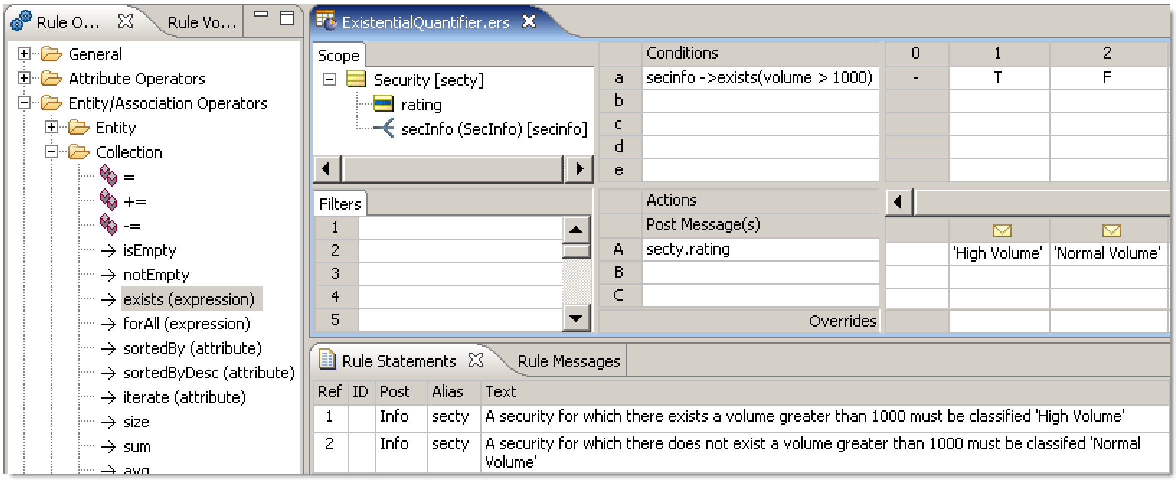


Existential quantifier

The other special operator available is the existential quantifier. The meaning of the existential quantifier is that *there exists at least one* element of a collection for which a given condition evaluates to true. This logic is implemented in the *Rulesheet* using the `->exists` operator from our Operator Vocabulary.

As in our last example, we can construct a *Rulesheet* to determine the `rating` value for a `Security` entity by evaluating a collection of associated `SecInfo` elements with the existential quantifier. In this new example, we will use `volume` rather than `rank` to determine the `rating` value for the security. The *Rulesheet* for this example is shown in the following figure: Rulesheet with Existential Quantifier (exists) Condition.

Figure 93: Rulesheet with Existential Quantifier ("exists") Condition



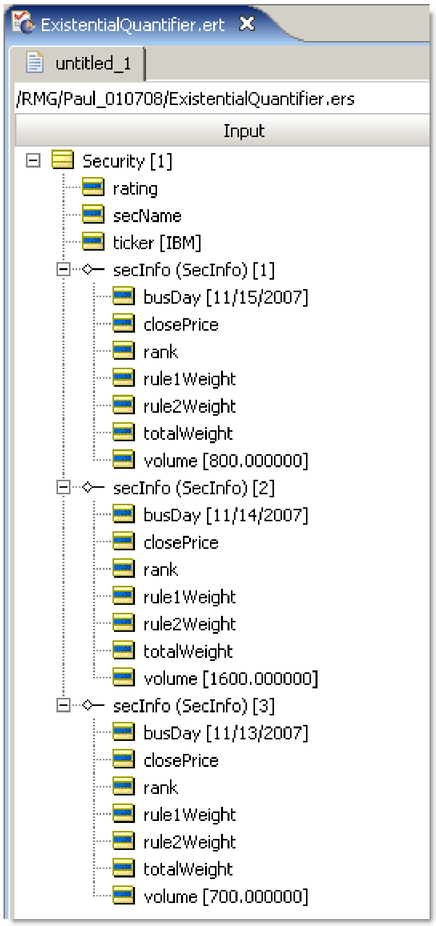
In this *Rulesheet*, we see the Condition

```
secinfo ->exists(secinfo.volume >1000)
```

Notice again the *required* use of an alias to represent the collection being examined. The exact meaning of the Condition in this example is that for the collection of `SecInfo` elements associated with a `Security` (again represented by the `secinfo` alias), determine if the expression in parentheses (`secinfo.volume > 1000`) holds **true** for *at least one* `Secinfo` element. Depending on the outcome of the `exists` evaluation, a value of either `High Volume` or `Normal Volume` will be assigned to the `rating` attribute of our `Security` object and the corresponding Rule Statement will be posted as a message to the user.

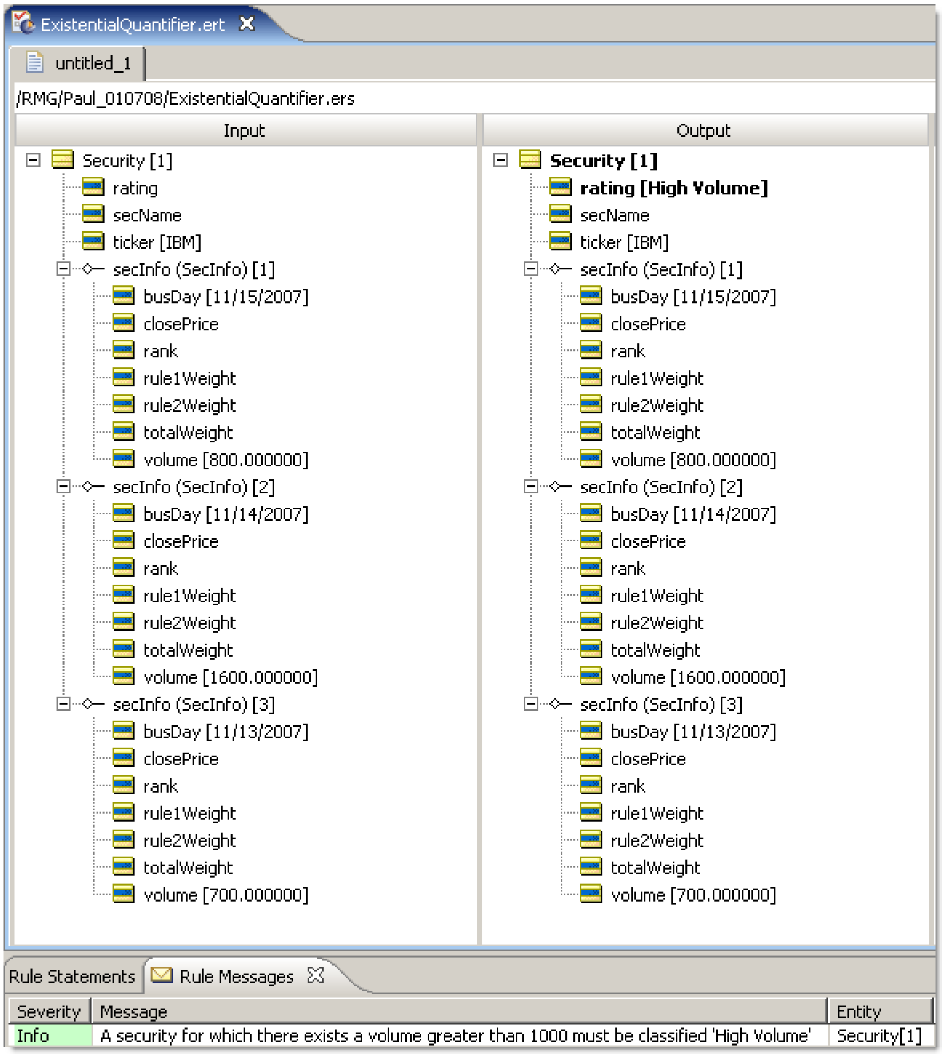
The following figure shows a Ruletest constructed to test our `exists` Condition rules.

Figure 94: Ruletest for Testing `exists` Condition Rules



Once again, we evaluate a collection of 3 `SecInfo` elements associated with a single `Security` entity. Since the `volume` attribute value assigned in at least one of the `SecInfo` objects (`secInfo[2]`) is greater than 1000, we should expect that our `exists` Condition will evaluate to **true** and a `rating` value of `High Volume` will be assigned to our `Security` object when the *`Ruletest`* is run through *`Corticon Server`*. This outcome is confirmed in the *`Ruletest`* shown in the following figure.

Figure 95: Ruletest for `exists` Condition Rules



Another example using the existential quantifier

Collection operators are powerful parts of the CRL – in some cases they may be the only way to implement a particular business rule. For this reason, we provide another example.

Business problem: An auto insurance company has a business process for handling auto claims. Part of this process involves determining a claim's validity based on the information submitted on the claim form. For a claim to be classified as valid, both the driver and vehicle listed on the claim must be covered by the policy referenced by the claim. Claims that are classified as invalid will be rejected, and will not be processed for payment.

From this short description, we extract our primary business rule statement:

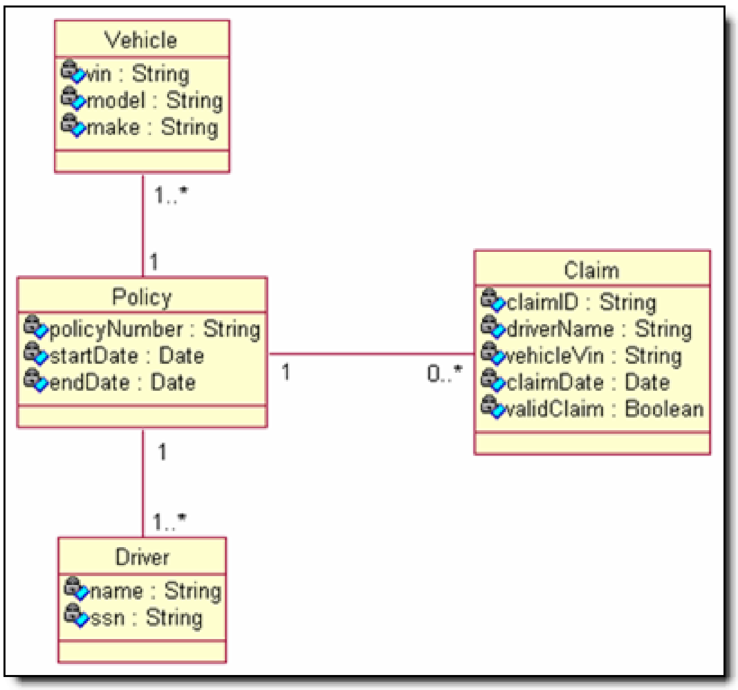
1. A claim is valid if the driver and vehicle involved in a claim are both listed on the policy against which the claim is submitted.

In order to implement our business rule, we propose the **UML Class Diagram** shown below. Note the following aspects of the diagram:

- A Policy may cover one or more Drivers
- A Policy may cover one or more Vehicles
- A Policy may have zero or more Claims submitted against it.
- The Claim entity has been denormalized to include `driverName` and `vehicleVin` .

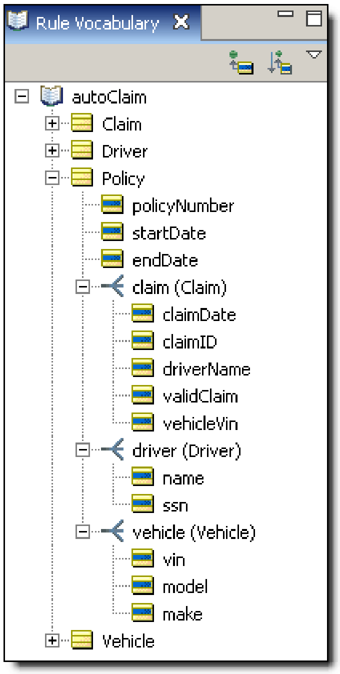
Note: Alternatively, the Claim entity could have referenced `Driver.name` and `Vehicle.vin` (by adding associations between Claim and both Driver and Vehicle), respectively, but the denormalized structure is probably more representative of a real-world scenario.

Figure 96: UML Class Diagram



This model imports into Corticon Studio as the Vocabulary shown in Figure 97 on page 109

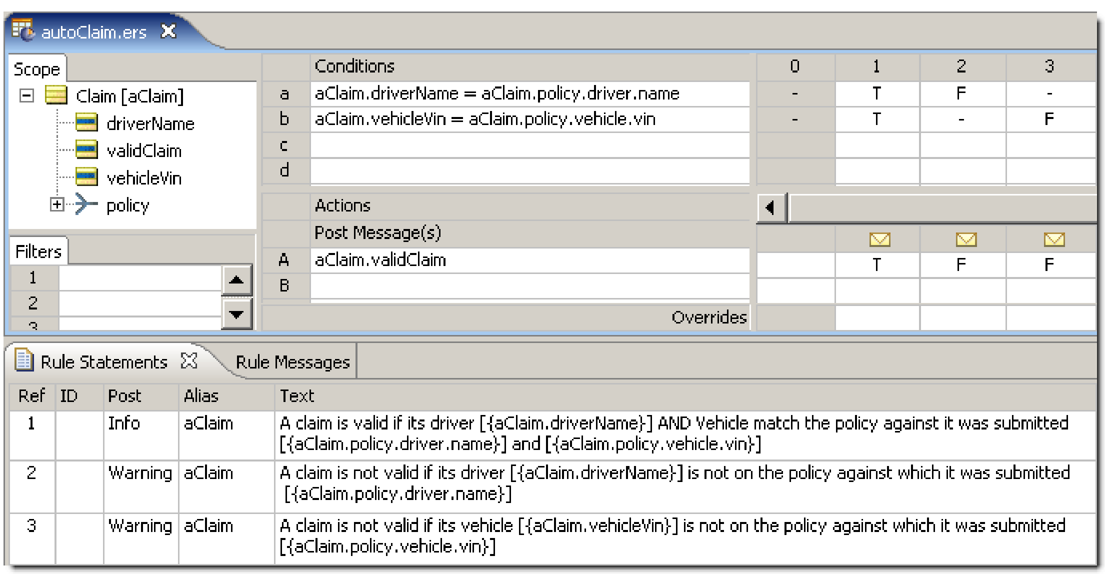
Figure 97: Vocabulary



Model the following rules in Corticon Studio as shown in Figure 98 on page 109

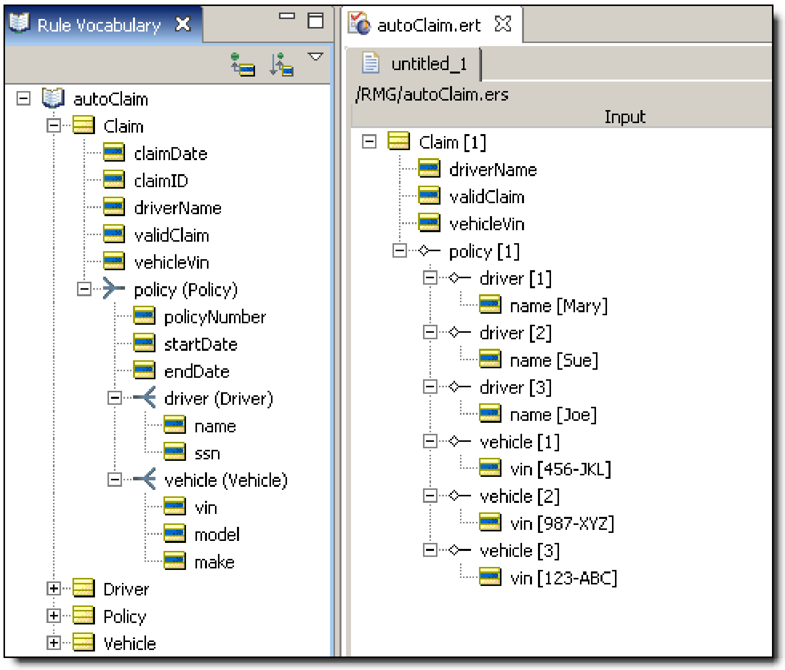
1. For a claim to be valid, the driver's name and vehicle ID listed on the claim must also be listed on the claim's policy.
2. If either the driver's name or vehicle ID on the claim is not listed on the policy, then the claim is not valid.

Figure 98: Rules Modeled in Corticon Studio



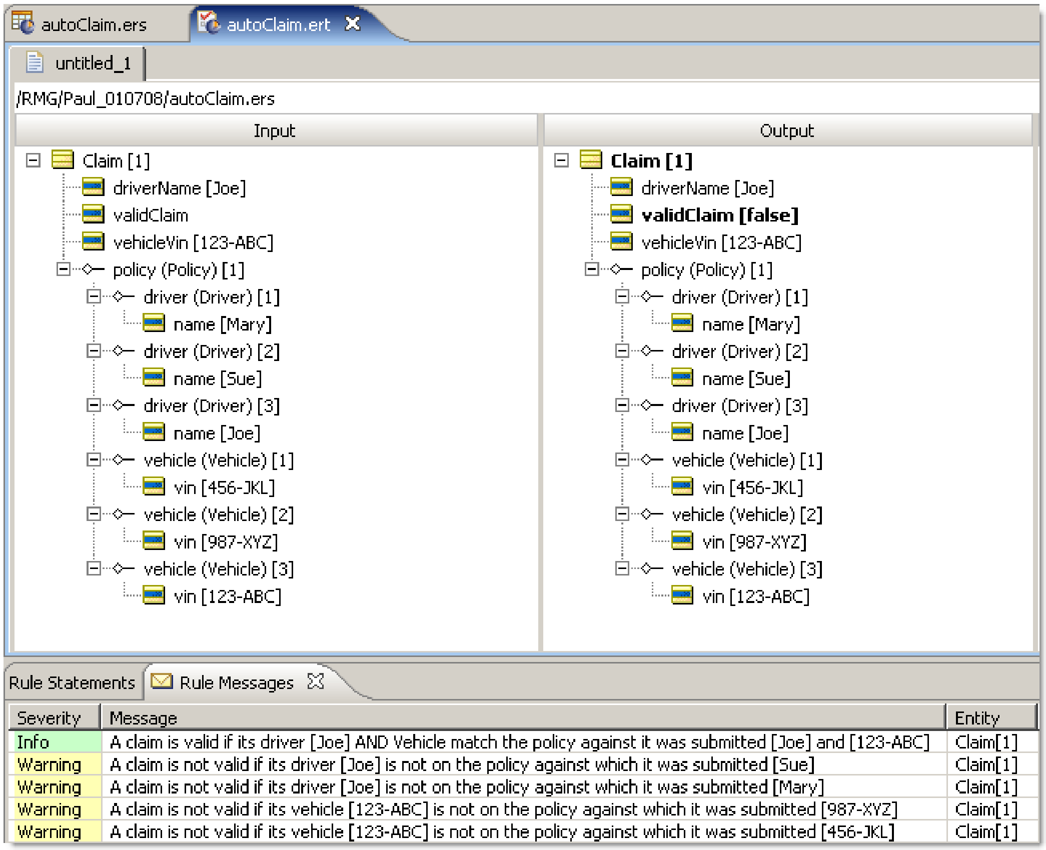
This appears very straightforward. But a problem arises when there are multiple drivers and/or vehicles listed on the policy–in other words, when the policy contains a collection of drivers and/or vehicles. Our Vocabulary permits this scenario because of the cardinalities we assigned to the various associations. We demonstrate this problem with the Ruletest in Figure 99 on page 110

Figure 99: Ruletest



Notice in Figure 100 on page 111 that there are 3 drivers and 3 vehicles listed on (associated with) a single policy. When we run this Ruletest, we see the results:

Figure 100: Ruletest

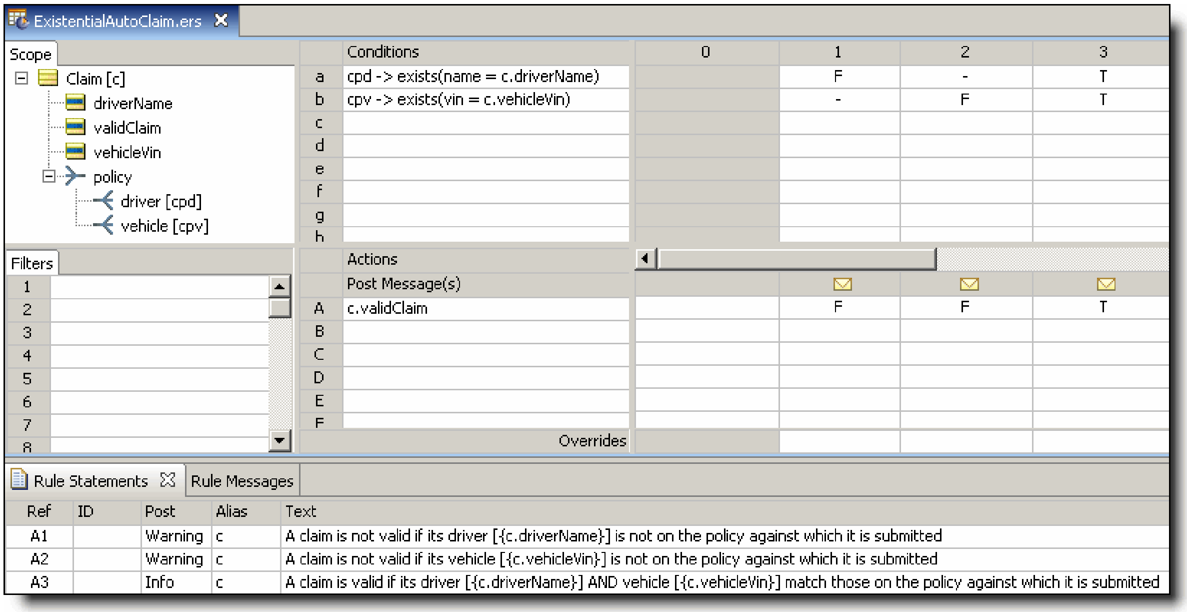


As we see from the Ruletest results, the way Corticon Studio evaluates rules involving comparisons of multiple collections means that the `validClaim` attribute may have inconsistent assignments – sometimes `true`, sometimes `false` (as in this Ruletest). It can be seen from the table below that, given the Ruletest data, 4 of 5 possible combinations evaluate to `false`, while only one evaluates to `true`. This conflict arises because of the nature of the data evaluated, not the rule logic, so Studio's Conflict Check feature does not detect it.

Claim. driverName	Claim.policy. driver.name	Claim. vehicleVin	Claim.policy. vehicle.vin	Rule 1 fires	Rule 2 fires	Rule 3 fires	validClaim
Joe	Joe	123-ABC	123-ABC	X			True
Joe	Sue				X		False
Joe	Mary				X		False
		123-ABC	987-XYZ			X	False
		123-ABC	456-JKL			X	False

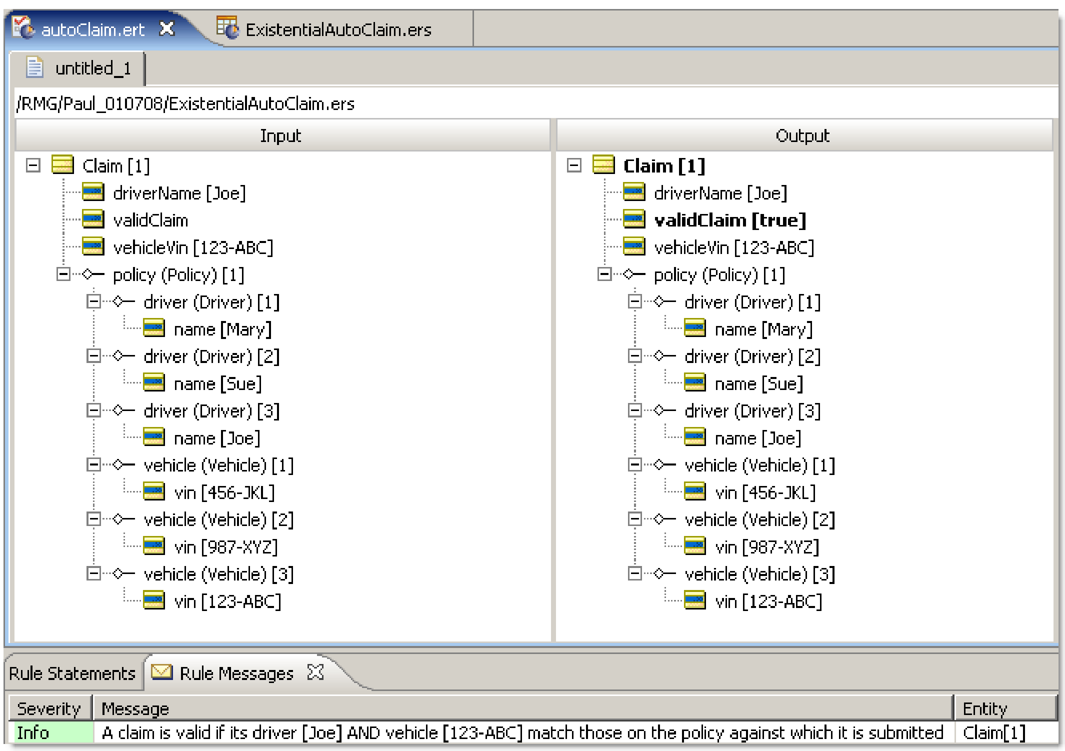
Let's use the existential quantifier to rewrite these rules:

Figure 101: Rules Rewritten Using Extential Quantifier.



This logic tests for the existence of matching drivers and vehicles within the two collections. If matches exist within both, then the `validClaim` attribute evaluates to true, otherwise `validClaim` is false.

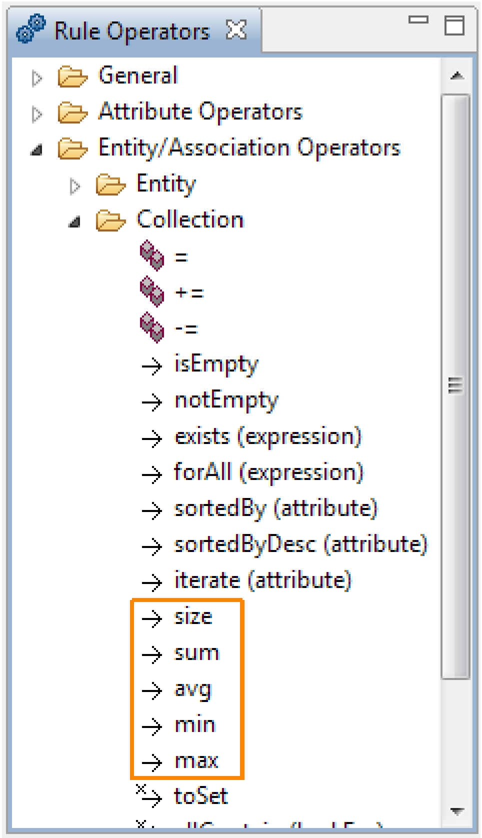
Let's use the same Ruletest data as before to test these new rules. The results are shown below:



Notice that only one rule has fired, and that `validClaim` has been assigned the value of true. This implementation achieves the intended result.

Aggregations that optimize database access

A subset of collection operators are known as *aggregation operators* because they evaluate a collection to compute one value. These aggregation operators are as highlighted:



When these aggregations are applied through the Enterprise Data Connector in a Rulesheet set to **Extend to Database**, the performance impact against large tables can be minimized by performing non-conditional actions that force the calculations onto the database. For an example of this, see Optimizing Aggregations that Extend to Database on page 199

Test yourself - questions – collections

Note: Try this test, and then go to **Answers:** Collections on page 278 to correct yourself.

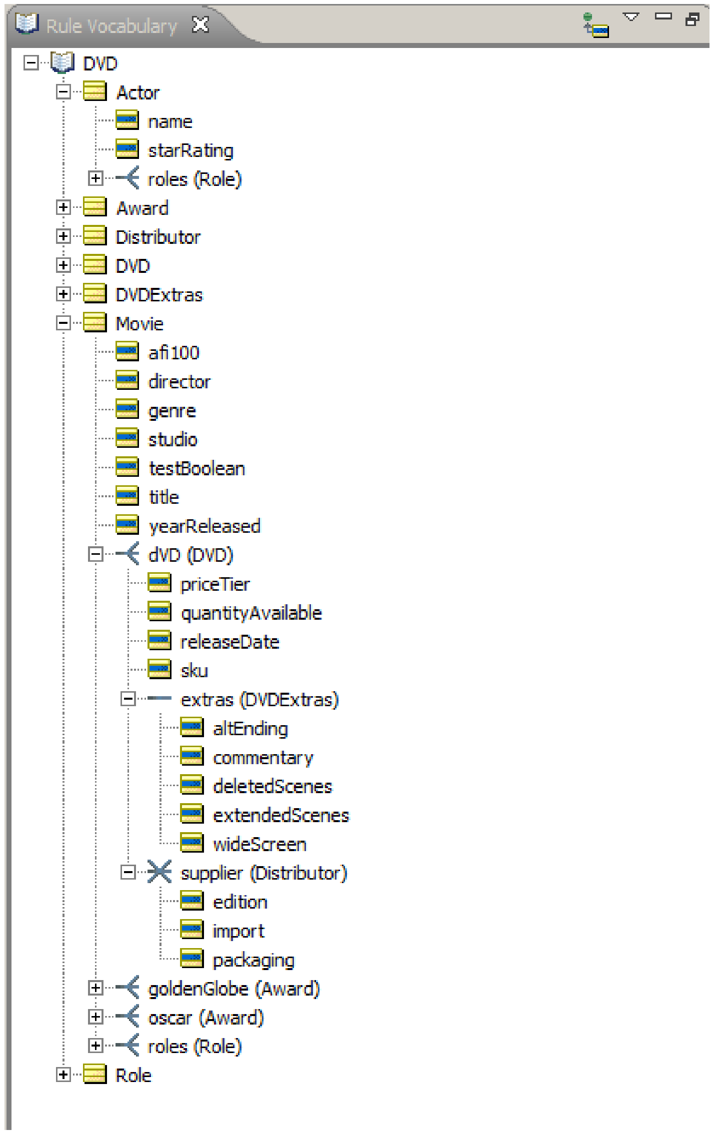
1. Children of a Parent entity are also known as ____________ of a collection.
2. True or False. All collections must have a parent entity
3. True or False. Root-level entities may form a collection
4. True or False. A collection operator must operate on a collection alias.
5. List 3 Collection operators and describe what they do.
6. Which reference contains usage details and examples for every collection operator?

7. Write a Rule Statement that is equivalent to the syntax `Order.total = items.price->sum`
8. In the syntax in question 7, which term is the collection alias?
9. If `items` is an alias representing the `LineItem` entities associated with an `Order` entity, then what would you expect the cardinality of this association to be?
10. Is `Order.lineItem.price->sum` an acceptable replacement for the syntax in Question 7? Why or why not?
11. If you are a Vocabulary designer and want to prevent rule authors from building rules with `LineItem.order` terms, what can you do to prevent it?
12. When collection operators are NOT used in a *`Rulesheet`*, aliases are (circle all that apply)

Optional	Mandatory	Colorful	Convenient

13. If a Nonconditional rule states `LineItem.price = 100`, and my Input Testsheet contains 7 `LineItem` entities, then a collection of data is processed by this rule. Is a collection alias required? Why or why not?
14. Which collection operator is known as the Universal Quantifier?
15. Which collection operator is known as the Existential Quantifier?

For questions 16-18, refer to the following Vocabulary



16. Write expressions for each of the following phrases:
    - **a.** If an actor has had more than 3 roles…
    - **b.** If a movie has not been released on DVD…
    - **c.** If a movie has at least one DVD with deleted scenes…
    - **d.** If a movie won at least one Golden Globe
    - **e.** If the movie had more than 15 actors…
    - **f.** If there's at least 100 copies available of a movie…
    - **g.** If there's less than 2 copies available of a movie…
    - **h.** If the DVD can be obtained from more than 1 supplier…
17. Which entities could be grandchildren of Movie?
18. Which entites could be children of Role?
19. Describe the difference between `->forAll` and `->exists` operators.

20. Describe the difference between `->notEmpty` and `->isEmpty` operators.
21. Why are aliases required to represent collections?

5

Rules containing calculations & equations

Rules that contain equations and calculations are really no different than any other type of rule. Calculation-containing rules may be expressed in any of the sections of the *Rulesheet*.

For details, see the following topics:

Terminology

First we will introduce some terminology that will be used throughout this chapter. In the most basic expression

```
A = B
```

We define A to be the *Left-hand Side* (LHS) of the expression, and B to be the *Right-hand Side* (RHS). The equals sign is an *Operator*, and is included in the Operator Vocabulary in Corticon Studio. But even so simple an expression has its complications. For example, does this expression compare the value of A to B in order to take some action, or does it instead assign the value of B to A ? In other words, is the equals operator performing a *comparison* or an *assignment*? This is a common problem in programming languages, and a common solution is to use two different operators to distinguish between the two meanings. For example the symbol == may signify a comparison operation, whereas := may signify an assignment.

In Studio, such special syntax is unnecessary because the *Rulesheet* itself helps to clarify the logical intent of the rules. For example, typing A=B into a *Rulesheet*'s Condition row (and pressing **Enter**) automatically causes the Values set {T,F} to appear in the rule column cell drop-down lists. This indicates that the rule modeler has written a comparison expression, and Studio expects a value of true or false to result from the comparison. A=B, in other words, is treated as a test – is A equal to B or isn't it?

On the other hand, when A=B is entered into an Action or Nonconditional row (Actions rows in Column 0), it becomes an assignment. In an assignment, the RHS of the equation is evaluated and its value is assigned to the LHS of the equation. In this case, the value of B is simply assigned to A. As with other Actions, we have the ability to activate or deactivate this Action for any column in the decision table (numbered columns in the *Rulesheet*) simply by "checking the box" that automatically appears when the Action's cell is clicked.

In the *Rule Language Guide*, the equals operator (=) is described separately in both its assignment and comparison contexts.

Operator precedence

Operator precedence, or the order in which Corticon Studio evaluates multiple operators in an equation, is defined by the table in "Arithmetic operator precedence" of the *Rule Language Guide*. This table specifies for example, that 2*3+4 evaluates to 10 and not 14 because the multiplication operator * has a higher precedence than the addition operator +. It is a good practice, however, to include clarifying parentheses even when Corticon Studio does not require it. This equation would be better expressed as (2*3)+4. Note the addition of parentheses here does not change the result. When expressed as 2*(3+4), however, the result becomes 14.

Datatype compatibility and casting

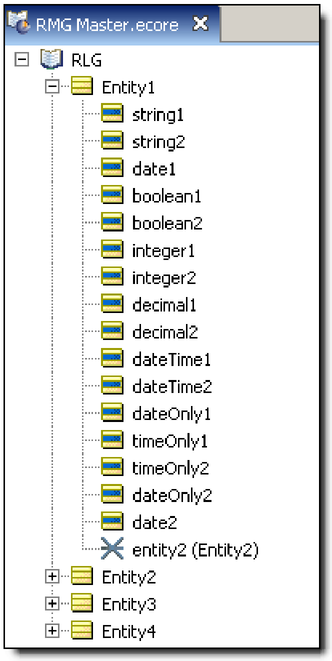
An important prerequisite of any comparison or assignment operation is data type compatibility. In other words, the data type of the equation's LHS (the data type of A) must be compatible with whatever data type results from the evaluation of the equation's RHS (the data type of B). For example, if both attributes A and B are Decimal types, then there will be no problem assigning the Decimal value of attribute B to attribute A.

Similarly, a comparison between the LHS and RHS makes no real sense unless both refer to the same kinds of data. How does one compare lemon (a String) to July 4, 2004 (a Date)? Or false (a Boolean) to 247.82 (a Decimal)?

In general, the data type of the LHS must match the data type of the RHS before a comparison or assignment can be made. (The exception to this rule is the comparison or assignment of an Integer to a Decimal. A Decimal can safely contain the value of an Integer without using any special casting operations.) Expressions that result in inappropriate data type comparison or assignment should turn red in Studio.

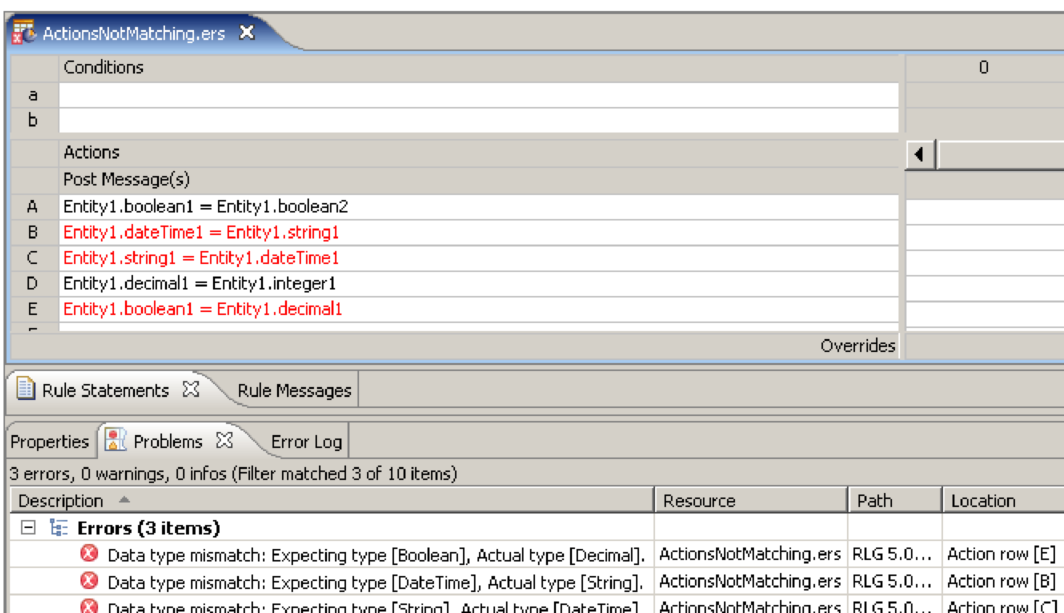
In the examples that follow, we will use the generic Vocabulary from the *Rule Language Guide*, since the generic attribute names indicate their data types:

Figure 102: Generic Vocabulary from the Rule Language Guide



The following figure shows a set of Action rows that illustrate the importance of data type compatibility in assignment expressions:

Figure 103: Datatype Mismatches in Assignment Expressions



Let's examine each of the Action rows to understand why each is valid or invalid.

A – this expression is valid because the data types of the LHS and RHS sides of the equation are compatible (they're both Boolean).

B – this expression is invalid and turns red because the data types of the LHS and RHS sides of the equation are incompatible (the LHS resolves to a DateTime and the RHS resolves to a String).

C – this expression is invalid and turns red because the data types of the LHS and RHS sides of the equation are incompatible (the LHS resolves to a String and the RHS resolves to a DateTime).

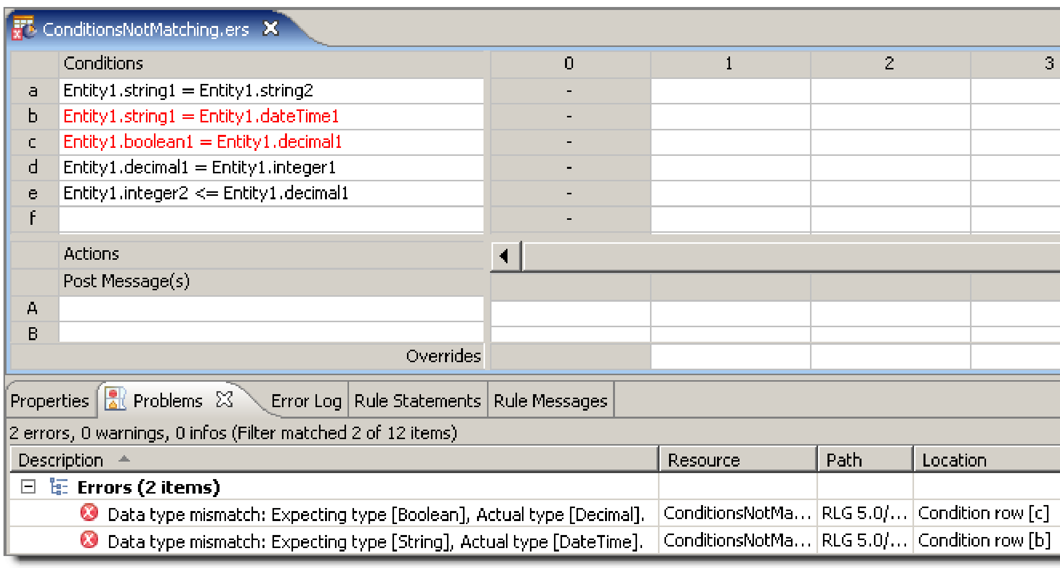
D – this expression is valid because the data types of the LHS and RHS sides of the equation are compatible *even though they are different!* This is an example of the one exception to our general rule regarding data type compatibility: Decimals can safely hold Integer values.

E – this expression is invalid and turns red because the data types of the LHS and RHS sides of the equation are incompatible (the LHS resolves to a Boolean and the RHS resolves to a Decimal). Here, the tool tip provides essentially the same information.

Note that the **Problems** window contains explanations for the red text shown in the `Rulesheet`.

The following figure shows a set of Conditional expressions that illustrate the importance of data type compatibility in comparisons:

Figure 104: Datatype Mismatches in Comparision Expressions



Let's examine each of these Conditional expressions to understand why each is valid or invalid:

a – This comparison expression is valid because the data types of the LHS and RHS sides of the equation are compatible (they're both Strings). Note that Corticon Studio confirms the validity of the expression by recognizing it as a comparison and automatically entering the Values set `{T,F}` in the Values column.

b – This comparison expression is invalid because the data types of the LHS and RHS sides of the equation are incompatible (the LHS resolves to a String and the RHS resolves to a DateTime). Note that, in addition to the red text, Corticon Studio emphasizes the problem by not entering the Values set `{T,F}` in the Values column.

c – This comparison expression is invalid because the data types of the LHS and RHS sides of the equation are incompatible (the LHS resolves to a Boolean and the RHS resolves to a Decimal).

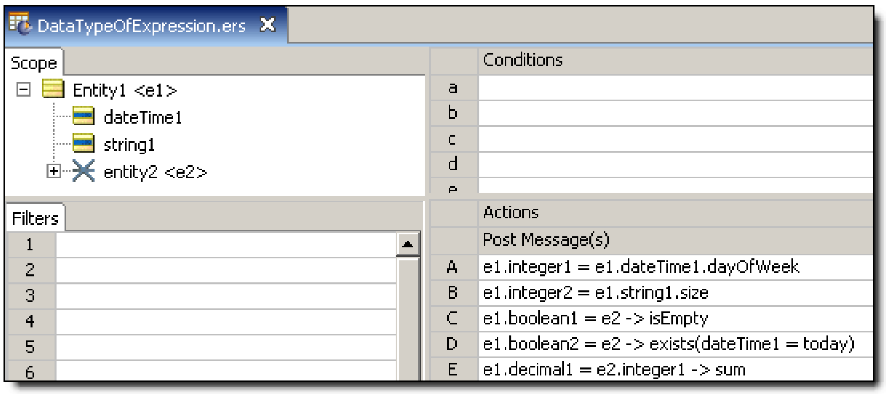
d – This comparison expression is valid because the data types of the LHS and RHS sides of the equation are compatible. This is another example of the one exception to our general rule regarding data type compatibility: Decimals may be safely compared to Integer values.

e – This comparison expression is valid because the data types of the LHS and RHS sides of the equation are compatible. Like example 4, this illustrates the one exception to our general rule regarding data type compatibility: Decimals may be safely compared to Integer values. Unlike an assignment, however, whether the Integer and Decimal types occupy the LHS or RHS of a comparison is unimportant.

Datatype of an expression

It is important to emphasize that the idea of a data type applies not only to specific attributes in the Vocabulary, but to entire expressions. Our examples above have been simple, and the data types of the LHS or the RHS of an equation simply correspond to the data types of those single attributes. But the data type to which an expression resolves may be a good deal more complicated.

Figure 105: Examples of Expression Datatypes



Again, we will examine each assignment to understand what is happening:

A – The RHS of this equation resolves to an Integer data type because the `.dayOfWeek` operator "extracts" the day of the week from a DateTime value (in this case, the value held by attribute `date1`) and returns it as an Integer between 1 and 7. Since the LHS also has an Integer data type, the assignment operation is valid.

B – The RHS of this equation resolves to an Integer because the `.size` operator counts the number of characters in a String (in this case the String held by attribute `string1`) and returns this value as an Integer. Since the LHS also has an Integer data type, the assignment operation is valid.

C – The RHS of this equation resolves to a Boolean because the `->isEmpty` collection operator examines a collection (in this case the collection of `Entity2` children associated with parent `Entity1`, represented by collection alias `e2`) and returns `true` if the collection is empty (has no elements) or `false` if it isn't. Since the LHS also has a Boolean data type, the assignment operation is valid.

D – The RHS of this equation resolves to a Boolean because the `->exists` collection operator examines a collection (in this case, `e2` again) and returns `true` if the expression in parentheses is satisfied at least once, and `false` if it isn't. Since the LHS also has a Boolean data type, the assignment operation is valid.

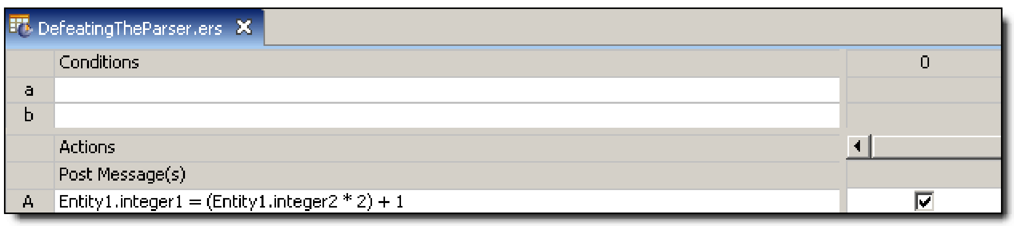
E – the RHS of this equation resolves to an Integer because the `->sum` collection operator adds up the values of all occurrences of an attribute (in this case, `integer2`) in a collection (in this case, `e2` again). Since the LHS has a Decimal data type, the assignment operation is valid. This is the lone case where type casting occurs automatically.

Note: The .dayOfWeek operator and others used in these examples are described fully in the *Rule Language Guide*

Defeating the parser

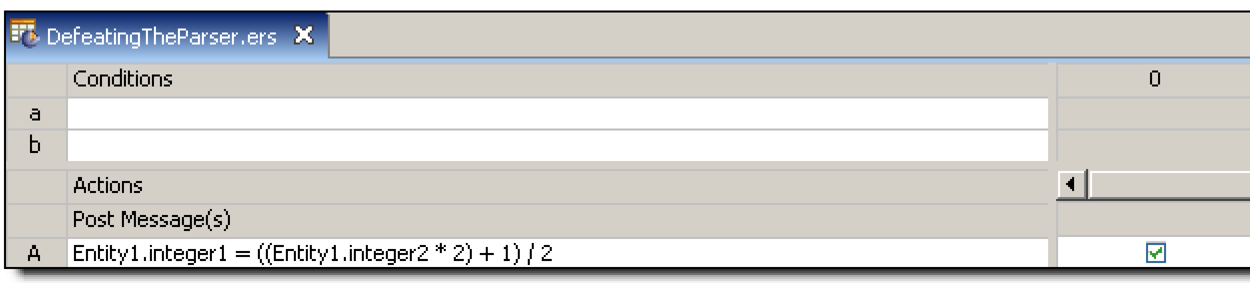
The part of Corticon Studio that checks for data type mismatches (along with all other syntactical problems) is the Parser. The Parser exists to ensure that whatever is expressed in a *Rulesheet* can be correctly translated and compiled into code executable by Corticon Studio's *Ruletest* as well as by the Corticon Server. Because this is a critical function, much effort has been put into the Parser's accuracy and efficiency. But rule modelers should understand that the Parser is not perfect, and can't anticipate all possible combinations of the rule language. It is still possible to "slip one past" the Parser. Here is an example:

Figure 106: LHS and RHS Resolve to Integers



In the figure above, we see an assignment expression where both LHS and RHS return Integers under all circumstances. But making a minor change to the RHS throws this result into confusion:

Figure 107: Will the RHS Still Resolve to an Integer?



The minor change of adding a division step to the RHS expression has a major effect on the data type of the RHS. Prior to modification, the RHS always returns an Integer, but an *odd* Integer! When we divide an odd Integer by 2, a Decimal always results. The Parser is smart, but not smart enough to catch this problem.

When the rule is executed, what happens? How does the *Server* react when the rule instructs it to force a Decimal value into an attribute of type Integer? The *Server* responds by truncating the Decimal value. For example if `integer2` has the value of `2`, then the RHS returns the Decimal value of `2.5`. This value is truncated to `2` and then assigned to `integer1` in the LHS.

When we focus on this rule here, alone and isolated, it's relatively easy to see the problem. But in a complex *Rulesheet*, it may be difficult to uncover this sort of problem. Your only clue to its existence may be numerical test results that do not match the expected values. To be safe, it's usually a good idea to ensure the LHS of numeric calculations has a Decimal data type so no data is inadvertently lost through truncation.

Manipulating datatypes with casting operators

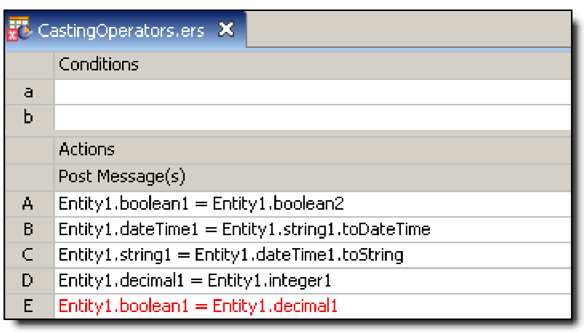
A special set of operators is provided in the *Studio*'s Operator Vocabulary that allows the rule modeler to control the data types of attributes and expressions. These casting operators are described below:

Table 2: Table: Special Casting Operators

Casting Operator	Applies to data of type…	Produces data of type…
`.toInteger`	Decimal, String	Integer
`.toDecimal`	Integer, String	Decimal
`.toString`	Integer, Decimal, DateTime, Date, Time	String
`.toDateTime`	String, Date, Time	DateTime
`.toDate`	DateTime	Date
`.toTime`	DateTime	Time

Returning to Datatype Mismatches in Comparision Expressions, we use these casting operators to correct some of the previous problems:

Figure 108: Using Casting Operators

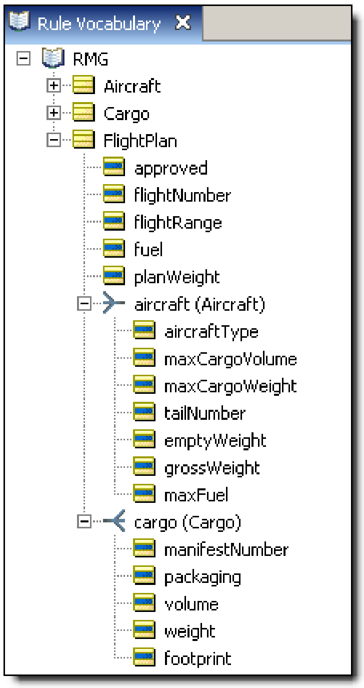


Casting operators have been used in Nonconditional rules N.2 and N.3 to make the data types of the LHS and RHS match. Notice however, that no casting operator exists to cast a Decimal into a Boolean data type.

Supported uses of calculation expressions

To make our examples more interesting and allow for a bit more complexity in our rules, we have extended the basic Tutorial Vocabulary (`Cargo.ecore`) to include a few more attributes. The extended Vocabulary is shown below:

Figure 109: Basic Tutorial Vocabulary Extended



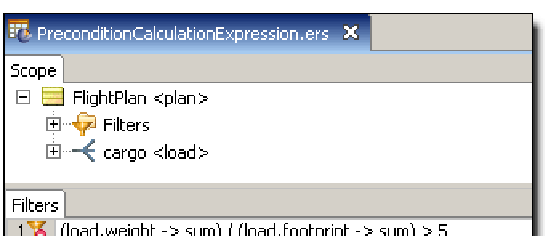
The new attributes are described in the table below:

Table 3: Table: Table of New Attributes Added to the Basic Tutorial Vocabulary

Attribute	Data type	Description
`Aircraft.emptyWeight`	Decimal	The weight of an Aircraft with no fuel or cargo onboard.
`Aircraft.grossWeight`	Decimal	The maximum amount of weight an Aircraft can safely lift, equal to the sum of cargo and fuel weights.
`Aircraft.maxfuel`	Decimal	The maximum amount of fuel an Aircraft can carry.
`Cargo.footprint`	Decimal	The floor space, measured in square feet, taken up by this Cargo.
`FlightPlan.approved`	Boolean	Indicates whether the FlightPlan has been approved or "cleared" for operation.
`FlightPlan.planWeight`	Decimal	The total amount of all Aircraft and Cargo weights for this FlightPlan.
`FlightPlan.flightRange`	Decimal	The distance the Aircraft is expected to fly.
`FlightPlan.fuel`	Decimal	The amount of fuel actually loaded on the Aircraft assigned to this FlightPlan.

Calculation as a comparison in a precondition

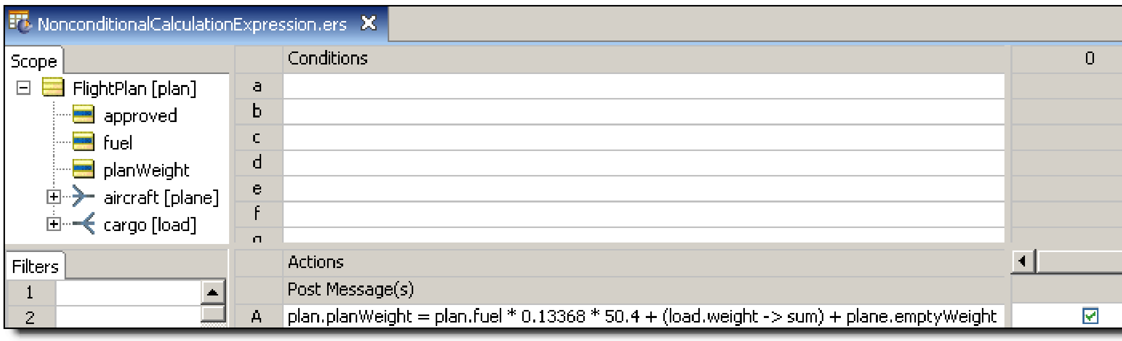
Figure 110: A Calculation in a Preconditional Expression



In this figure, a numeric calculation is used as a comparison in the Filters section of the *Rulesheet*. The LHS of the expression essentially calculates the average pressure exerted by the total cargo load on the floor of the aircraft (sum of the cargo weights divided by the sum of the cargo containers' footprints). This result is compared to the RHS, which is simply the literal value `5`. We might expect to see this type of calculation in a set of rules that deals with special cargos where a lot of weight is concentrated in a small area. This might, for example, require the use of special aircraft with sturdy, reinforced cargo bay floors. Such a Filter expression might be the first step in handling cargos that satisfy this special criterion.

Calculation as an assignment in a noncondition

Figure 111: A Calculation in a Nonconditional Expression



The example shown in this figure uses a calculation in the RHS of the assignment to derive the total weight carried by an Aircraft on the FlightPlan, where the total weight equals the weight of the fuel plus the weight of all Cargos onboard plus the empty weight of the Aircraft itself. The portion

```
plan.fuel * 0.13368 * 50.4
```

converts a fuel load measured in gallons into a weight measured in pounds – conversion factors of 0.13368 cubic feet per gallon and 50.4 pounds per cubic foot of jet fuel have been used here. This portion is then added to:

```
load.weight -> sum
```

which, of course, is equal to the sum of all Cargo weights loaded onto the Aircraft associated with this FlightPlan. The final sum of the fuel, cargo, and Aircraft weights is assigned to the FlightPlan's `planWeight`. Note the parentheses used here are not required – the calculation will produce the same result without them – they have been added for improved clarity.

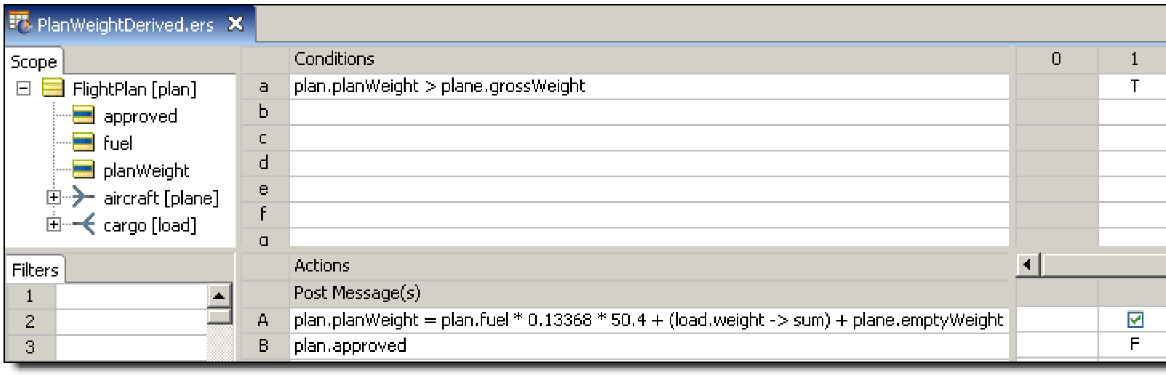
Calculation as a comparison in a condition

Once `planWeight` has been derived by the Nonconditional calculation in the figure below, it may be used immediately elsewhere in this or subsequent *Rulesheet*s.

Note: "Subsequent Rulesheets" means Rulesheets executed later in a Ruleflow. The concept of a Ruleflow is discussed in the *Quick Reference Guide*.

An example of such usage appears in the following figure:

Figure 112: `planWeight` Derived and Used in Same Rulesheet

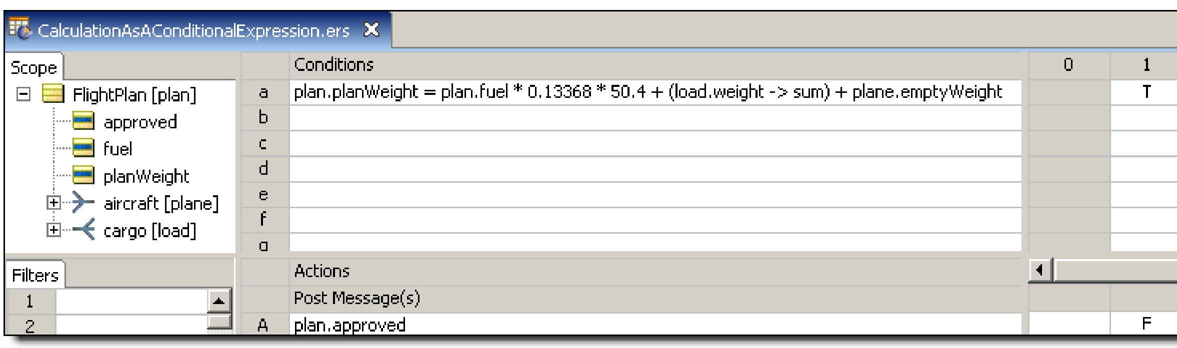


In Condition row a, `planWeight` is compared to the aircraft's `grossWeight` to make sure the aircraft is not overloaded. An overloaded aircraft must not be allowed to fly, so the `approved` attribute is assigned a value of `false`.

This has the advantage of being both clear and easy to reuse – the term `planWeight`, once derived, may be used anywhere to represent the data produced by the calculation. It is also much simpler and cleaner to use a single attribute in a rule expression than it is a long, complicated equation.

But this does not mean that the equation cannot be modeled in a Conditional expression, if preferred. The example shown in the figure below places the calculation in the LHS of the Conditional comparison to derive `planWeight` and compare it to `grossWeight` all in the same expression.

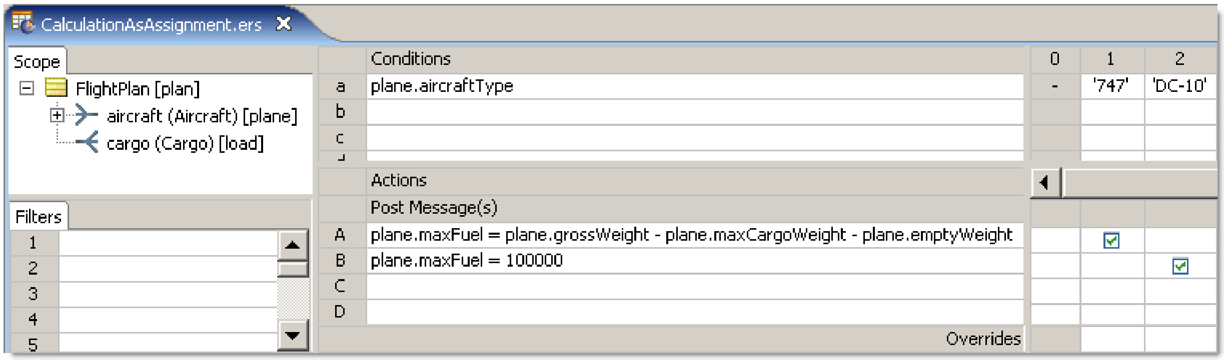
Figure 113: A Calculation in a Conditional Expression



This approach might be preferable if the results of the calculation were not expected to be reused, or if adding an attribute like `planWeight` to the Vocabulary were not possible. Often, attributes like `planWeight` are very convenient "intermediaries" or "holders" to carry calculated values that will be used in other rules in a *`Rulesheet`*. In cases where such attributes are conveniences only, and are not used by external applications consuming a *`Rulesheet`*, they may be designated as "extended transient" attributes in the Vocabulary, which causes their icons to change from blue/yellow to orange/yellow. More details on extended transient attributes are included in Extended transients in the vocabulary of this manual.

Calculation as an assignment in an action

Figure 114: A Calculation in an Action Expression



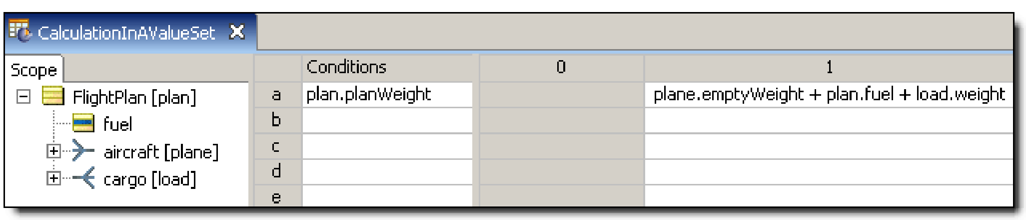
This figure shows two rules that each make an assignment to `maxFuel`, depending on the type of aircraft. In rule 1, the `maxFuel` load for 747s is derived by subtracting `maxCargoWeight` and `emptyWeight` from `grossWeight`. In rule 2, `maxFuel` for DC-10s is simply assigned the literal value `100,000`.

Unsupported uses of calculation expressions

Calculations in value sets and column cells

The Conditional expression shown below is not supported by Studio, even though it does not turn red. Some simpler equations may actually work correctly when inserted in the Values cell or a rule column cell, but it's a dangerous habit to get into because more complex equations generally do not work. It's best to express equations as shown in the previous sections.

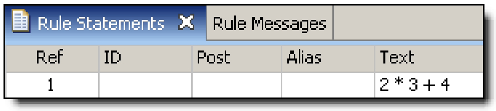
Figure 115: Calculation in a Values Cell and Column



Calculations in rule statements

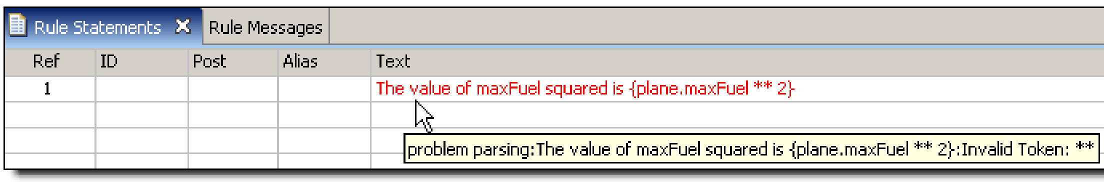
While it is possible to embed *attributes* from the Vocabulary inside Rule Statements, it is not possible to embed equations or calculations in them. Operators and equation syntax not enclosed in curly brackets `{..}` are treated like all other characters in the Rule Statement – nothing will be calculated. If the Rule Statement shown in the following figure is posted by an Action in rule 1, the message will be displayed exactly as shown; it will not calculate a result of any kind.

Figure 116: Calculation in a Rule Statement



Likewise, including equation syntax *within* curly brackets along with other Vocabulary terms is also not permitted. Doing so may cause your text to turn red, as shown:

Figure 117: Embedding a Calculation in a Rule Statement



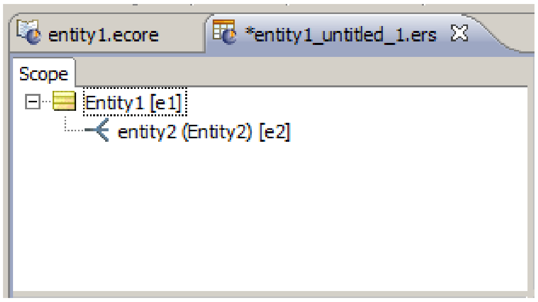
However, even if the syntax does not turn red, you should still not attempt to perform calculations in Rule Statements – it may cause unexpected behavior. When red, the tool tip should give you some guidance as to why the text is invalid. In this case, the exponent operator (`**`) is not allowed in an embedded expression.

Test yourself - questions – rules containing calculations and equations

Note: Try this test, and then go to **Answers:** Rules containing calculations & equations on page 279 to correct yourself.

1. What are the two possible meanings of the equals operator =? In which sections of the Rulesheet is each of these meanings applicable?
2. What is the result of each of the following equations?
   - **a.** 10 + 20 / 5 – 4 ___
   - **b.** 2 * 4 + 5 ___
   - **c.** 10 / 2 * 6 – 8 ___
   - **d.** 2 ** 3 * (1 + 2) ___
   - **e.** -5 * 2 + 5 * 2 ___
3. Is the following assignments expression valid? Why or why not? Entity1.integer1 = Entity1.decimal1

4. What is the data type of each of the following expressions based on the scope shown below?

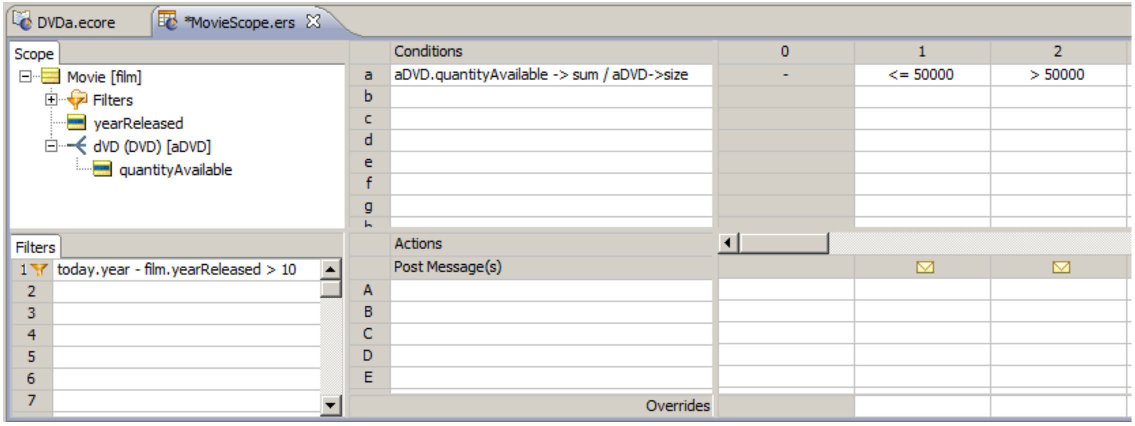


   - e1.dateTime1.year ___
   - e1.string1.toUpper ___
   - e2 -> forAll (integer1 = 10) ___
   - e2.decimal1 -> avg ___
   - e1.boolean1 ___
   - e1.decimal1 > e1.decimal2 ___
   - e2.string2.contains('abc') ___

5. Write "valid" or "invalid" for each of the following assignments
   - e1.decimal1 = e2.integer1 ___
   - e2.decimal2 = e2.string2 ___
   - e1.integer1 = e2.dateTime1.day ______
   - e2.integer1 = e2 -> size ___
   - e1.boolean2 = e2 -> exists (string1 = 'abc') ______
   - e2.boolean2 = e1.string1.toBoolean ______
   - e1.boolean2 = e2 -> isEmpty ______

6. The part of Corticon Studio that checks for syntactical problems is called the __________.
7. True or False. If an expression typed in Corticon Studio does not turn red, then the expression is guaranteed to work as expected.

   Referring to the following illustration, answer questions 8 through 10:



8. What does Filters row 1 test?
9. What does Conditions row "a" test? Is there a simpler way to accomplish this same thing using a different operator available in the `Corticon` Rule Language?
10. Write a Rule Statement for rule column 1. (Assume that the only action required for this rule is to post a Warning message as shown.)
11. True or False. The following sections of the Rulesheet accept equations and calculations:
    - Scope ___
    - Rule Statements ___
    - Condition rows ___
    - Action rows ___
    - Column 0 ___
    - Condition cells ___
    - Action cells ___
    - Filters ___

6

Rule dependency: Chaining and looping

For details, see the following topics:

What is rule dependency?

Dependencies between rules exist when a Conditional expression of one rule evaluates data produced by the Action of another rule. The second rule is said to be "dependent" upon the first.

Forward chaining

The first step in learning to use looping is to understand how it differs from normal inferencing behavior of executing rules, whether executed by Corticon Studio or Corticon Server. When a *`Rulesheet`* is compiled (either by Corticon Studio during a *`Ruletest`* execution, or by Corticon Server during deployment), a *dependency network* for the rules is automatically generated. Corticon Studio and Corticon Server uses this network to determine the order in which rules fire in runtime. For example, in the simple rules below, the proper dependency network is 1 2 3 4.

1. If value = A, then set value = B
2. If value = B, then set value = C
3. If value = C, then set value = D
4. If value = D, then set value = B

This is not to say that all three rules will always *fire* for a given test – clearly a test with `B` as the initial value will only cause rules 2, 3, and 4 to fire. But the dependency network ensures that rule 1 is always *evaluated* before rule 2, and rule 2 is always *evaluated* before rule 3, and so on. This mode of *`Rulesheet`* execution is called **Optimized Inferencing**, meaning the rules execute in the optimal sequence determined by the dependency network generated by the compiler. **Optimized Inferencing** is the default mode of rule processing for all *`Rulesheet`*s.

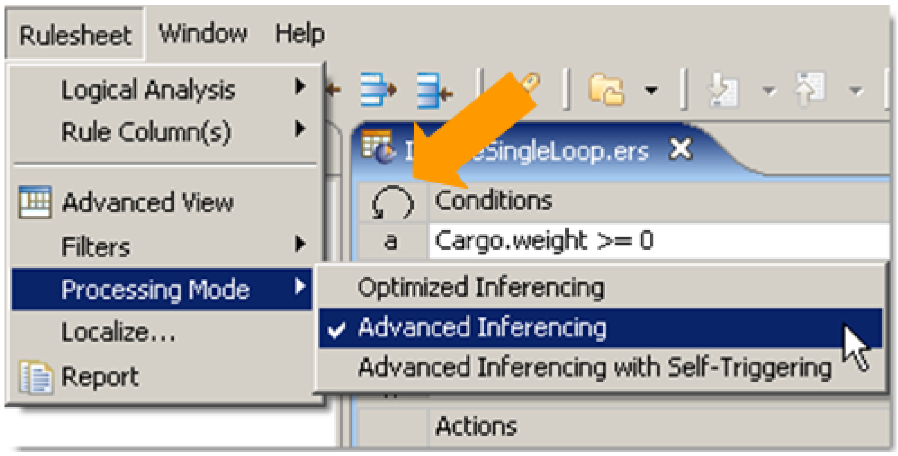
Optimized Inferencing processing is a powerful capability that enables the rule modeler to "break up" complex logic into a series of smaller, less complex rules. Once broken up into smaller or simpler rules, the logic will be executed in the proper sequence automatically, based on the dependencies determined by the compiler.

An important characteristic of Optimized Inferencing processing: the flow of rule execution is single-pass, meaning a rule in the sequence is evaluated once and never revisited, even if the data values (or data "state") evaluated by its Conditions change over the course of rule execution. In our example above, this effectively means that rule execution ceases after rule 4. Even if rule 4 fires (with resulting value = `B`), the second rule **will not** be revisited, re-evaluated, or re-fired even though its Condition (If value = `B`) would be satisfied by the current value (state). We can *force* rule 2 to be re-evaluated only if a one of Corticon Studio's looping processing modes is enabled for the *`Rulesheet`*. Remember, just because sequential processing occurs automatically does not mean looping will occur too. Looping and its enablement are discussed next.

Rulesheet processing: modes of looping

Occasionally, we *want* rules to be re-evaluated and re-fired (if satisfied). This scenario requires the `Corticon` rule engine to make multiple passes through the same `Rulesheet`. We call this behavior advanced inferencing, and to enable it in `Rulesheet` execution, we must set `Rulesheet` processing mode to **Advanced Inferencing** by selecting **Rulesheet > Processing Mode > Advanced Inferencing** from the `Studio` menubar, as shown:

Figure 118: Selecting Advanced Inferencing Processing Mode for a Rulesheet



Also note the circular icon to the immediate left of the Conditions header (see **orange arrow**).

If the rule engine is permitted to loop through the rules above, the following events occur:

Given a value of `A` as the initial data, the Condition in rule 1 will be satisfied and the rule will fire, setting the value to `B`. The 2nd rule's Condition is then satisfied, so the value will advance (or be reset) to `C`, and so on, until the value is once again `B` after the 4th rule fires. Up to this point, the rule engine is exhibiting standard, Optimized Inferencing behavior.

Now here's the new part: the value (state) has changed since the 2nd rule last fired, so the rule engine will re-evaluate the Condition, and, finding it satisfied, will fire the 2nd rule again, advancing the value to `C`. The 3rd rule will also be re-evaluated and re-fired, advancing the value to `D`, and so on. This sequence is illustrated in the following figure.

Figure 119: Loop Iterations

step #	Input value	Rule fired	Output value	Loop Iteration
1	A	1	B	
2	B	2	C	
3	C	3	D	
4	D	4	B	
5	B	2	C	1
6	C	3	D	
7	D	4	B	
8	B	2	C	2
9	C	3	D	
10	D	4	B	
...	...	...	...	...

Here's the key to understanding looping: when a looping processing mode is enabled, rules will be continually re-evaluated and re-fired in a sequence determined by their dependency network as long as data state has changed since their last firing. Once data state no longer changes, looping will cease.

Notice that the last column of the table indicates the number of loop iterations – the first loop does not begin until rule 2 fires for the *second* time. The first time through the rules (steps 1-4) does not count as the first loop iteration because the loop does not actually start until step 5.

Types of loops

Infinite Loops

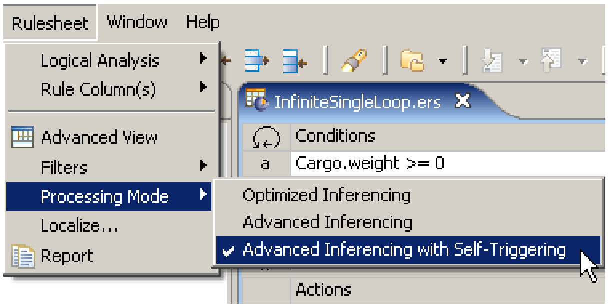
In the example above, looping between rules 2, 3 and 4 continues indefinitely because there is nothing to stop the cycle. Some loops, especially those inadvertently introduced, are not self-terminating. Because these loops will not end by themselves, they are called infinite loops. Infinite loops can be especially vexing to a rule modeler because it isn't always apparent when a *`Rulesheet`* has entered one. A good indication, however, is that rule execution takes longer than expected to complete! A special control is provided to prevent infinite loops. This control is described in the Terminating Infinite Loops section, below.

Trivial Loops

Single-rule loops, or loops caused by rules that depend logically on themselves, are also known as "trivial loops". We consider single-rule loops to be a special kind of loop because they consist of just a single rule that successively revisits, or "triggers", itself.

To enable the self-triggering mode of looping, we must select **Rulesheet > Processing Modes > Advanced Inferencing with Self-Triggering** from ***Studio***'s menubar, as shown in

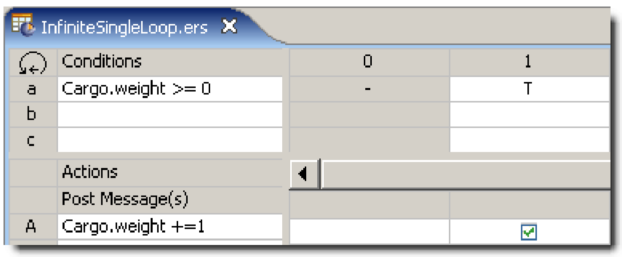
Figure 120: Selecting Advanced Inferencing with Self-Triggering Processing Mode for a Rulesheet



Notice the the icon to the left of the Conditions header - it contains the additional tiny arrow, which indicates self-triggering is active.

Here's an example of a loop created by a self-triggering rule:

Figure 121: Example of an Infinite Single-Rule Loop



Let's trace this rule to make sure we understand how it works.

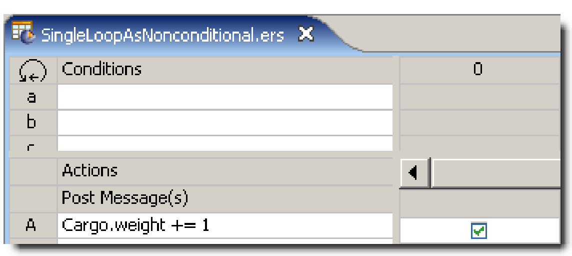
When `Cargo.weight` has a value equal to or greater than `0`, then rule 1 fires and the value of `Cargo.weight` is incremented by `1`. Data state has now *changed*, in other words, the value of at least one of the attibutes has changed. In this case, it's the value of `Cargo.weight` which has changed.

Because it was rule 1 execution that *caused* the data state change, and since self-triggering is enabled, the same rule 1 will be re-evaluated. Now, if the value of `Cargo.weight` satisfied the rule initially, it certainly will do so again, so the rule fires again, and self-triggers again. And so on, and so on. This is also an example of an infinite loop, because no logic exists in this rule to prevent it from continuing to loop and fire.

An Exception to Self-Triggering

Self-triggering logic can also be modeled in Column 0 of the *`Rulesheet`*, as shown:

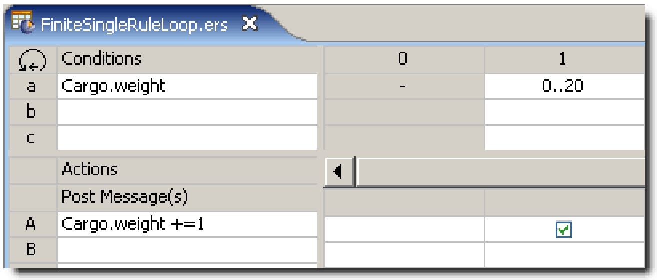
Figure 122: Example of an Infinite Loop created by a Self-Triggering Rule



But this figure is also a good example of why it might be appropriate to disable self-triggering processing: we only want the `weight` to increment once, not enter into an infinite loop, which it would otherwise do, unconditionally! This is a special case where we have intentionally prevented this rule from iterating, even though self-triggering is enabled. This rule will execute only once, regardless of looping processing mode.

Another example of a loop caused by self-triggering rule, but one which is not infinite, is shown below. The behavior described only occurs when Rulesheet processing mode is set to **Advanced Inferencing with Self-Triggers** :

Figure 123: Example of a Finite Single-Rule Loop



In the figure above, the rule continues to fire until `Cargo.weight` reaches a value of `21`, whereupon it fails to satisfy the Condition, and firing ceases. The loop terminates with `Cargo.weight` containing a final value of `21`.

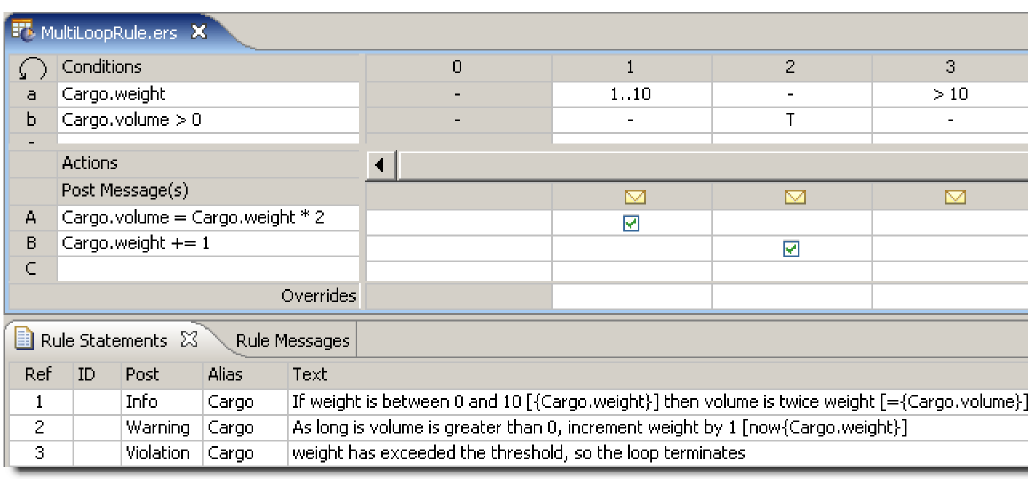
It's important to note that in all three examples, an initial `Cargo.weight` value of `0` or higher was necessary to "activate" the loop. A negative (or null) value, for example, would not have satisfied the rule's Condition and the loop would not have begun at all.

Multi-rule Loops

As the name suggests, multi-rule loops exist when 2 or more rules are mutually dependent. As with single-rule loops, the *Rulesheet* containing the looping rules must be configured to process them. This is accomplished as before. Choose ***Rulesheet*** **> Processing Mode > Advanced Inferencing** from the *Studio* menubar, as shown previously in Selecting Advanced Inferencing Processing Mode for a Rulesheet.

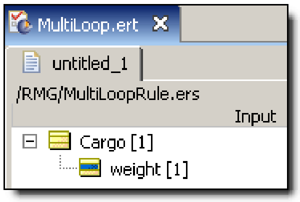
Here's an example of a multi-rule logical loop:

Figure 124: Example of a Finite Multi-Rule Loop



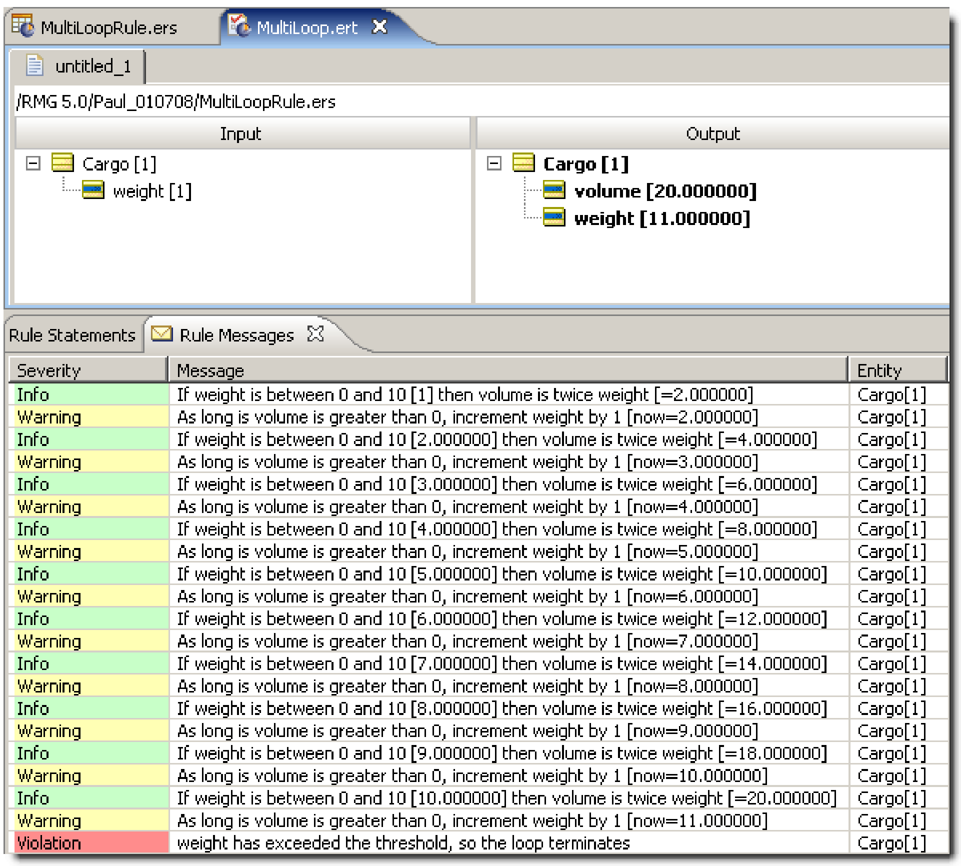
In the figure above, rule 2 is dependent upon rule 1, and rule 1 is dependent upon rule 2. We've also added a rule 3, which does not participate in the 1—2 loop, but will generate a nice `Violation` message when the 1—2 loop finally terminates. Note, rule 3 does not *cause* the 1—2 loop to terminate, it just *announces* that the loop has terminated. Let's test these rules and see how they behave. In **Ruletest for the Multi-rule Rulesheet**, we see a simple *`Ruletest`*.

Figure 125: Ruletest for the Multi-rule Rulesheet



We're providing a starting value of `Cargo.weight` just to get the loop going. According to the Condition in rule 1, this value needs to be between 1 and 10 (inclusive).

Figure 126: Ruletest for the Multi-rule Rulesheet



When intentionally building looping rules, it is often helpful to post messages with embedded attribute values (as shown in the Rule Statements section of Figure 124 on page 138) so we can trace the loop's operation and verify it is behaving as expected. It should be clear to the reader that the *`Ruletest`* shown in **Ruletest for the Multi-rule Rulesheet** contains the expected results.

Looping controls in Corticon Studio

To handle the various aspects of rule looping, Corticon Studio provides several mechanisms for identifying and controlling looping behavior.

Although we've only shown simple examples so far, looping rules can get much more complicated. Sometimes, rules have mutual dependencies by accident – we didn't intend to include loops when we built the *Rulesheet*. It is for this reason that all loop processing is disabled by default (in other words, the default *Rulesheet* processing mode is Optimized Inferencing, which does not permit revisiting rules that have already been evaluated once). We must manually enable the loop processing mode of our choice to cause the loops to execute. This is the strongest, most fool-proof mechanism for preventing unexpected looping behavior – simply keep loop processing disabled.

Identifying loops

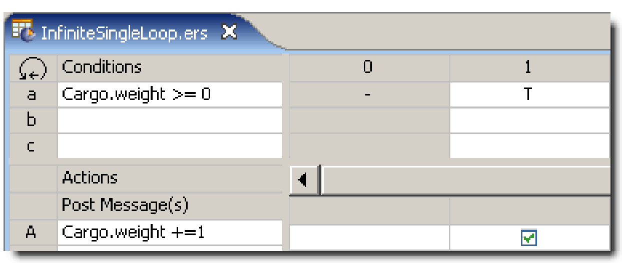
Assuming we haven't intentionally incorporated looping logic in our *Rulesheet*, then we need a way to discover if unintentional loops occur in our rules.

The loop detection tool

To help identify inadvertent loops, Corticon Studio provides a **Check for Logical Loops** tool in the Corticon Studio toolbar. The tool contains a powerful algorithm that analyzes dependencies between rules on the same *Rulesheet*, and reports discovered loops to the rule modeler. For the Loop Detector to notice mutual dependencies, a *Rulesheet* must have looping enabled using one of the choices described earlier.

Clicking the **Check for Logical Loops** icon displays a window that describes the mutual dependencies found on the *Rulesheet*. To illustrate loop detection, we will use a few of the same examples from before.

Figure 127: Example of an Infinite Single-Rule Loop



When applied to a `Rulesheet` containing just the single-rule loop shown in this figure, the **Check for Logical Loops** tool displays the following window:

Figure 128: Checking for Logical Loops in a Rulesheet

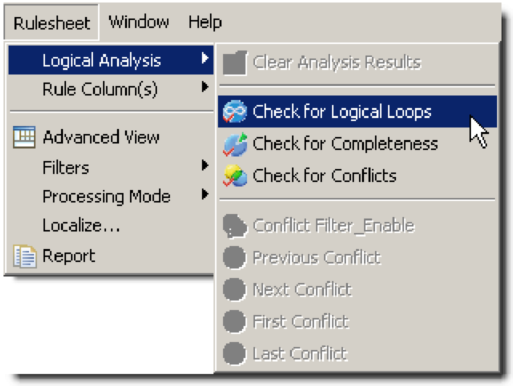
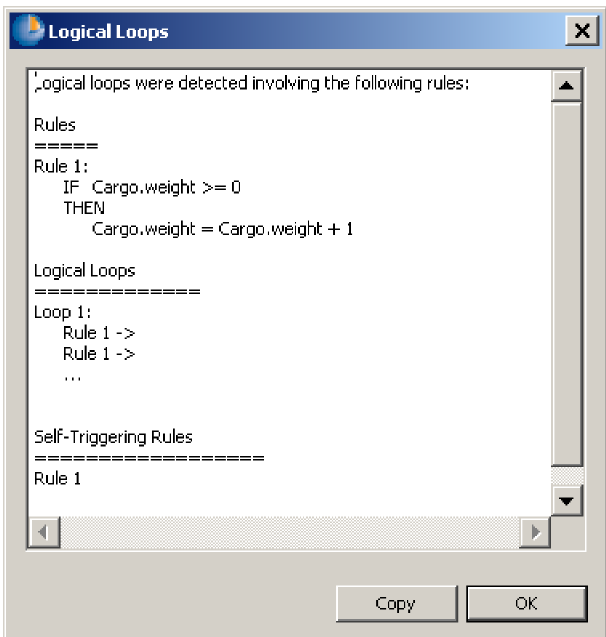


Figure 129: A Single-Rule Loop Detected by the Check for Logical Loops Tool



The Check for Logical Loops tool first lists rules where mutual dependencies exist. Then, it lists the distinct, independent loops in which those rules participate, and finally it lists where self-triggering rules exist (if any). In this simple single-rule loop example, only one rule contains a mutual dependency, and only one loop exists in the `Rulesheet`.

Note that the Check for Logical Loops tool does not automatically fix anything, it simply points out to us that our rules have loops, and gives us an opportunity to remove or modify the offending logic.

Removing loops

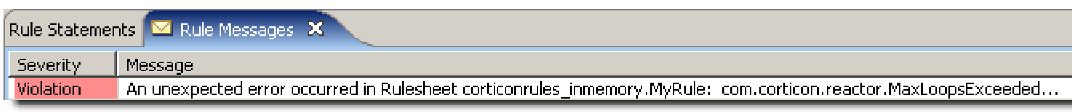
If the Check for Logical Loops tool detects loops, we can take one of several corrective actions:

- If no loops are what you want, then click **Rulesheet > Processing Mode** and de-select whichever of the two looping options is currently selected. Once done, the **Check for Logical Loops** tool will no longer detect loops and the software will no longer process them.
- If loops are what you want, then take measures to ensure that none of the loops can be infinite. Normally, this means adding conditional logic to one of the looping rules to make sure that the rule can't be satisfied indefinitely. This is similar to the bounding of Condition 1 in Example of a Finite Multi-Rule Loop using a Values set of `0..20`. Once `Cargo.weight` reaches `21`, the rule's Condition will no longer be satisfied and the loop will terminate.
- If some loops are good and some are not, then remove the inter-dependencies in the unwanted loops and ensure the selected loops are not infinite.

Terminating infinite loops

By definition, infinite loops won't terminate by themselves. Therefore, *`Corticon`* provides a "safety valve" setting described in the *`Server` Integration & Deployment Guide*. `com.corticon.reactor.rulebuilder.maxloop` is a property that caps the number of iterations allowed before the system automatically terminates a loop. The default setting is 100, meaning that a loop is allowed to iterate up to 100 times normally. Once the number of loops exceeds the `maxloop` setting, then the system automatically terminates the loop and generates a `Violation` error message. This means that the final number of loop iterations will be 101: 100 normal iterations plus the final iteration that causes the `Violation` message to appear and the loop to terminate. The `Violation` message is shown below:

Figure 130: Maxloop Exceeded Violation Message



If you are comfortable writing looping rules, and want the software to be able to loop more than 100 times, be sure to reset this property to a higher value. Keep in mind that the more iterations the system performs, the longer rule execution may take. If the *`Rulesheet`*s you intend to deploy require high iteration counts, be sure to inform your deployment manager so he/she can configure the target *`Corticon Server`* with a higher `maxloop` cap.

Looping example

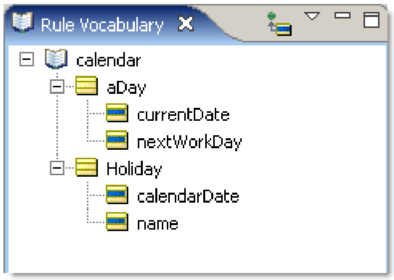
Problem

For any given day of the week, determine the next working day. Take into consideration weekends and holidays.

Solution

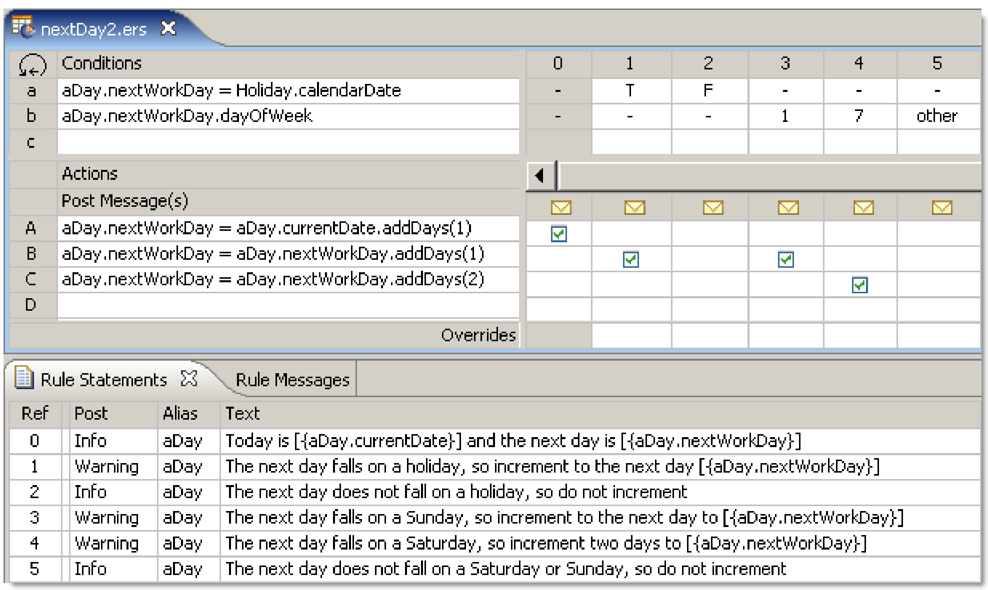
Implemented correctly in Corticon Studio, these rules should start with a given input date, and increment as necessary until the next workday is identified (workday defined here as any day *not* Saturday, Sunday, or a national holiday). A simple Vocabulary that supports these rules is shown in Example of a Finite Single-Rule Loop.

Figure 131: Sample Vocabulary for Holiday Rules



Next, the rules are implemented in the `Rulesheet` shown in the following figure:

Figure 132: Sample Rulesheet for Determining Next Workday



Let's step through this `Rulesheet`.

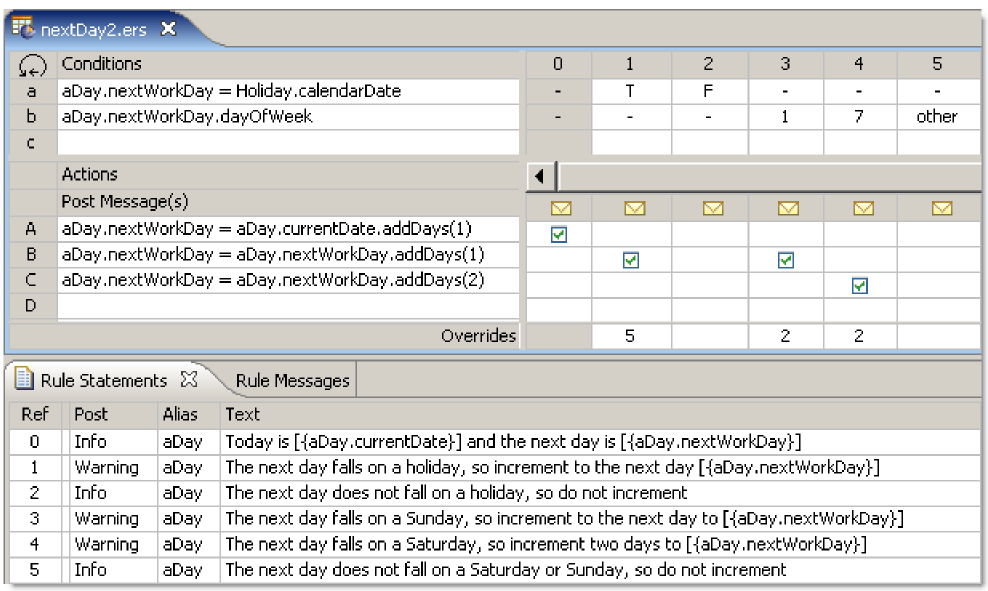
1. First, notice that the Scope section wasn't used. We're using a very simple Vocabulary with short entity names and no associations, so aliases aren't necessary. Furthermore, none of our rules use collection operations, so aliases representing collections aren't required either.
2. The first rule executed is the Nonconditional equation (in Condition/Action column 0) setting `nextWorkDay` equal to `currentDate` plus one day.
3. Rule 1 (in column 1) checks to see if the DateTime of the `nextWorkDay` matches any of the holidays defined in one or more `Holiday` entities. If it does, then the Action row B increments `nextWorkDay` by one day and posts a warning message.
4. Rule 3 checks to see if the `nextWorkDay` falls on a Sunday. Notice that this rule uses the `.dayOfWeek` operator, which is described in full detail in the *Rule Language Guide*. If the day of the week is Sunday (in other words, `.dayOfWeek` returns a value of `1`), then the Action increments `nextWorkDay` by <u>one</u> day and posts a Warning message.
5. Rule 4 checks to see if the `nextWorkDay` falls on a Saturday. If the day of the week is Saturday (in other words, `.dayOfWeek` returns a value of `7`), then the Action row C increments `nextWorkDay` by <u>two</u> days and posts a Warning message. By incrementing 2 days, we skip an extra iteration because we know Sunday is also a non-workday!

Do not forget to check for conflicts – they do exist in this *`Rulesheet`*. However, we will make the assumption that a holiday never falls on a weekend.

Note: Resolution of the conflicts is straightforward, so we won't address that in detail here. One conflict – that between rules 1 and 4 - is left unresolved because we have assumed that a holiday never falls on a weekend. See Logical Analysis chapter more a complete discussion of conflict and other logical problems.

A modified *`Rulesheet`* displays the overrides added to resolve the conflicts in the following figure:

Figure 133: Holiday Rules with Ambiguities Resolved by Overrides


nextDay2.ers

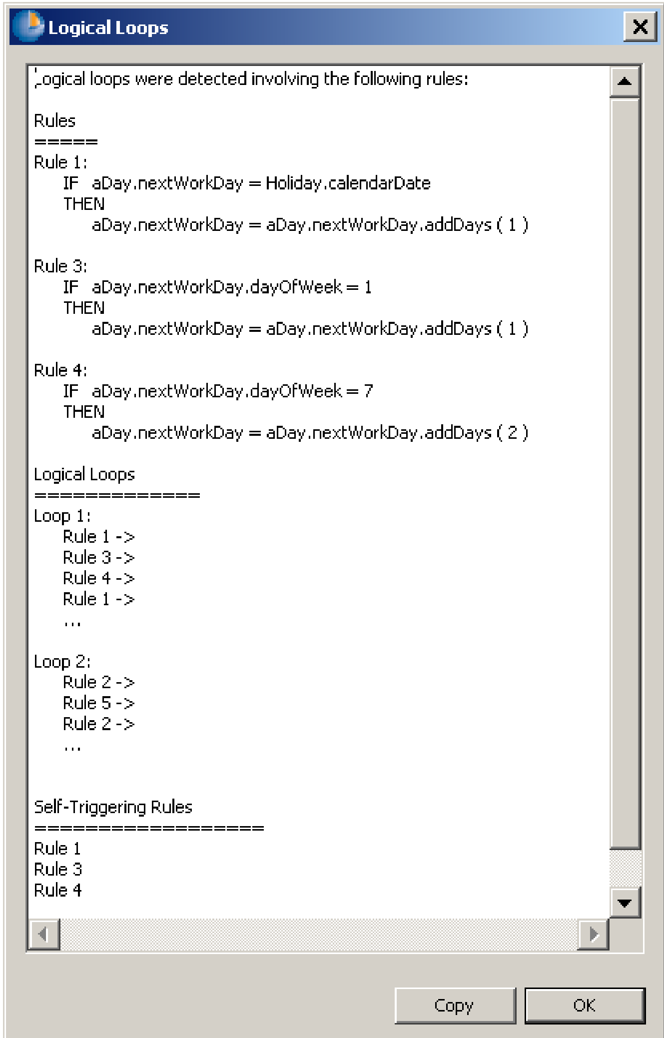
	Conditions	0	1	2	3	4	5
a	aDay.nextWorkDay = Holiday.calendarDate	-	T	F	-	-	-
b	aDay.nextWorkDay.dayOfWeek	-	-	-	1	7	other
c							
	Actions						
	Post Message(s)	✉	✉	✉	✉	✉	✉
A	aDay.nextWorkDay = aDay.currentDate.addDays(1)	☑					
B	aDay.nextWorkDay = aDay.nextWorkDay.addDays(1)		☑		☑		
C	aDay.nextWorkDay = aDay.nextWorkDay.addDays(2)					☑	
D							
	Overrides		5		2	2	

Rule Statements | Rule Messages

Ref	Post	Alias	Text
0	Info	aDay	Today is [{aDay.currentDate}] and the next day is [{aDay.nextWorkDay}]
1	Warning	aDay	The next day falls on a holiday, so increment to the next day [{aDay.nextWorkDay}]
2	Info	aDay	The next day does not fall on a holiday, so do not increment
3	Warning	aDay	The next day falls on a Sunday, so increment to the next day to [{aDay.nextWorkDay}]
4	Warning	aDay	The next day falls on a Saturday, so increment two days to [{aDay.nextWorkDay}]
5	Info	aDay	The next day does not fall on a Saturday or Sunday, so do not increment

Using the same rules as before, let's click the **Logical Loop Checker** icon in the Corticon Studio toolbar. The following window opens:

Figure 134: Results of Logical Loop Check

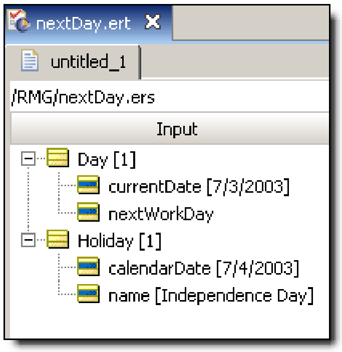


This window first identifies exactly which rules are involved in loops. Secondly, the window outlines the specific attribute interactions that create the loops.

Now that we fully understand the looping logic present in our Rulesheet, let's create a `Ruletest` to verify that the loops operate as intended and produce the correct business results.

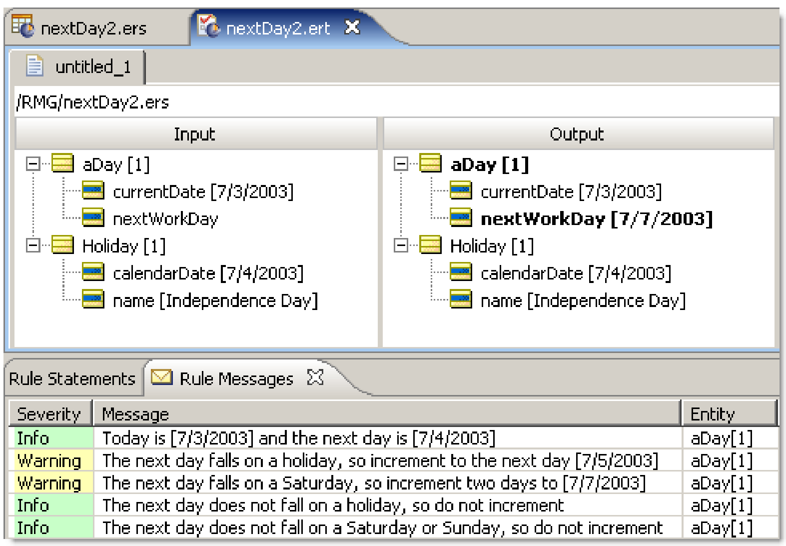
Ruletest for Holiday Rules

Figure 135: Ruletest for Holiday Rules



Given the fact that July 4^{th}, 2003 falls on a Friday, we expect `nextWorkDay` to contain a final value of July 7^{th}, 2003 – a Monday – when the loops terminate. Running the *`Ruletest`*:

Figure 136: Ruletest for Holiday Rules



As you can see in this figure, the expected result is obtained.

Using conditions as a processing threshold

We want to distinguish looping, which involves revisiting, re-evaluating, and possible re-firing rules, and which requires you to enable one of the looping modes discussed above, from another behavior that may appear similar on the surface.

You have almost certainly noticed Corticon's inherent ability to process multiple test scenarios at once. For example, a rule written using the Vocabulary term `Cargo.weight` will be evaluated (and potentially fired) for every instance of `Cargo` encountered during execution. If a *`Ruletest`* contains 4 `Cargo` entities, then the rule engine will test the rule's conditions with each of them. If any of the `Cargo` entities satisfy the rule's conditions, then the rule will fire. This could mean that the rule fires once, twice, or up to four times, depending on the actual data values of each `Cargo`. We know from our prior discussion of Scope that a rule will evaluate *all* data that shares the same scope as the rule itself.

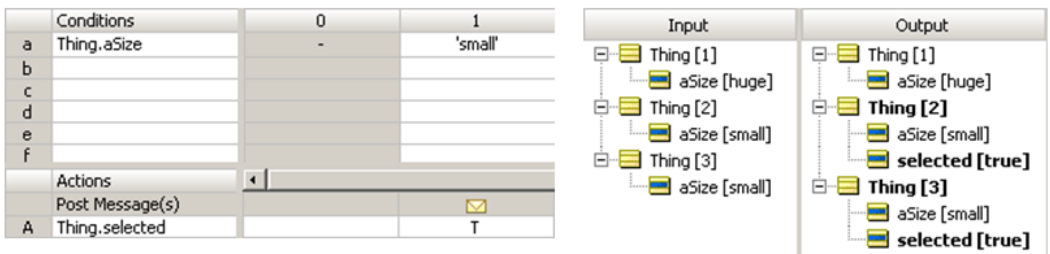
This iterative behavior is a natural part of the Corticon rule engine design – there's nothing special we need to do to enable it or "turn it on". Note, that this behavior is different from the modes of looping discussed above because the `Cargo.weight` rule is not re-evaluated for a given piece of data. Rule execution is still single-pass. It is just that it makes a single pass through *each* of the 4 `Cargo` entities.

The advantage of this natural iteration is that we don't need to force it – the rule engine will automatically process all data that shares the same scope as the rule. If the *`Ruletest`* contains 4 Cargos, the rule will be evaluated 4 times. If the *`Ruletest`* contains 4000 Cargos, the rule will be evaluated 4000 times. We don't write the rule any differently in Corticon Studio.

But this advantage can also be a disadvantage. What if we *want* rule execution to stop part-way through its evaluation of a given set of entity data (which we call a "binding"). What if, after finding a `Cargo` that satisfies the rule among the set (binding) of `Cargo` entities, we want to *stop* evaluation mid-stream? In normal operation, this is not possible.

Here's a simple example.

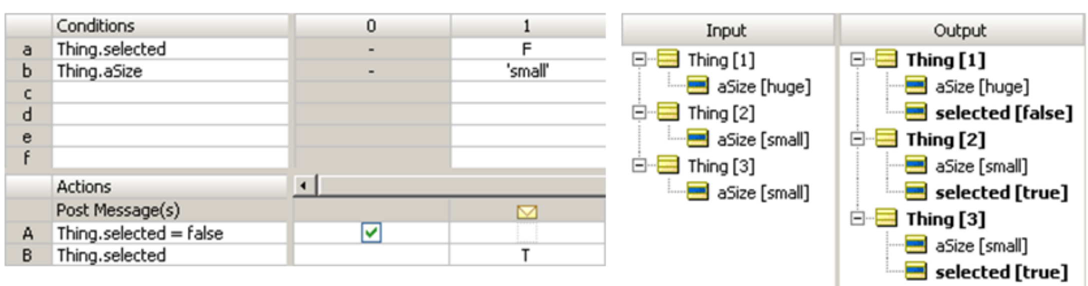
Figure 137: Rulesheet and Ruletest, no threshold condition, CaPT disabled



In the example above, no threshold condition, CaPT disabled, we see a simple rule that sets `thing.selected = true` for all `thing.aSize = 'small'`. Notice in the adjacent *`Ruletest`*, that each small `Thing` is selected. `Thing[2]` and `Thing[3]` are both small, so they are both selected by the rule. The rule has evaluated all three Things, but finding only two that satisfy the rule's condition, only fires twice. This iteration happened automatically.

What if we wanted rule execution to stop after finding the first `Thing` that satisfies the rule? In other words, allow the rule engine to fire for `Thing[2]` but stop processing *before*firing for `Thing[3]`. Is that possible? You might think the following *`Rulesheet`* would accomplish this goal.

Figure 138: Rulesheet and Ruletest, threshold condition added, CaPT disabled



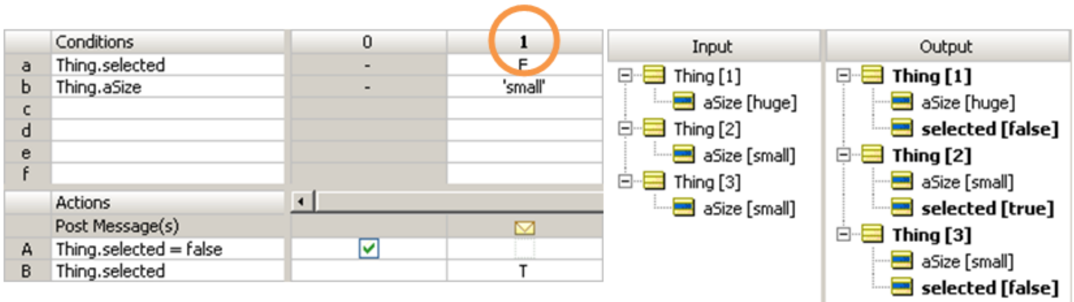
The example in this figure includes two changes: 1) `Thing.selected` is defaulted to `false` in the Nonconditional rule (Action row `A0`). And 2) a second Condition row checks for `Thing.selected = false` as part of rule 1. This is called a "threshold" condition.

You might be temped to think that when `Thing[2]` fires the rule, its value of selected (re-set to `true`) would be sufficient to stop further evaluation and execution of `Thing[3]`. But as we see in the adjacent *`Ruletest`*, that this isn't the case. The reason is that `Thing[3]` is an entirely separate entity within the binding, and is entitled to its own evaluation of rule 1 regardless of what happended with `Thing[2]`. The addition of the threshold condition accomplished nothing.

A special feature in Corticon Studio, called **Use Condition as Processing Threshold** (abbreviated as CaPT), allows us to interrupt processing of the binding. You activate this option by selecting the rule column involved, then from the Corticon Studio menu bar, choose **Rulesheet > Rule Columns(s) > Use Condition as Processing Threshold**.

Once selected, CaPT causes the rule column header to display in bold type, as shown below, circled in orange:

Figure 139: Rulesheet and Ruletest, threshold condition added, CaPT enabled



When CaPT is activated, it breaks out of the automatic binding iteration whenever an instance in the binding fails to satisfy the threshold condition. In this case, `Thing[2]`, having just fired rule 1, no longer satisfies the threshold condition, and causes rule execution to stop before evaluating `Thing[3]`. If we re-ran this *`Ruletest`*, we might see `Thing[3]` evaluated first, in which case rule execution would stop before evaluating `Thing[2]`.

Within a binding, sequence of evaluation of elements is *random* and may change from execution to execution. There is nothing about the binding that enforces an order or sequence among the bound elements.

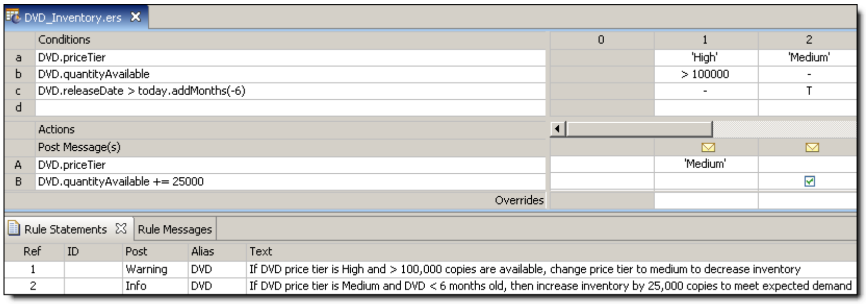
Test yourself - questions – Rule dependency: chaining and looping

Note: Try this test, and then go to **Answers:** Rule dependency: dependency & inferencing on page 280 to correct yourself.

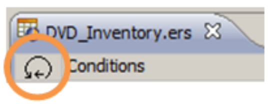
Use the following Vocabulary to answer the next questions.

1. What is the main difference between inferencing and looping?
2. A loop that does not end by itself is known as an __________ loop.
3. A loop that depends logically on itself is known as a single-rule or __________ loop.
4. True or False. The **Check for Logical Loops** tool in Corticon Studio will always find mutual dependencies in a *`Rulesheet`* if they are present.
5. True or False. The **Check for Logical Loops** tool in Corticon Studio can fix inadvertent loops.

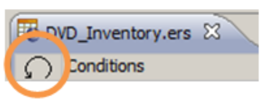
   Referring to the following illustration, answer questions 6 through 8



6. Given these two rules, is it necessary for the *Rulesheet* to use the Inferencing mode shown? Why or why not?
7. Is there any potential harm in having this *Rulesheet* configured to Advanced Inferencing with Self-Triggering? Why or why not?
8. If the *Rulesheet* as shown above were tested with a DVD having a price tier of `High`, quantity available of `150,000`, and release date within the past 6 months, what would be the outcome of the test?
9. This icon indicates which type of inferencing is enabled for this *Rulesheet*?



10. This icon indicates which type of inferencing is enabled for this *Rulesheet*?



11. A ________________ determines the sequence of rule execution and is generated when a *Rulesheet* is _______________.

7

Filters & preconditions

Conditional expressions modeled in the **Filters** section of a *`Rulesheet`* can behave in two different ways: as filters alone or as filters *plus* preconditions. Both behaviors are explained and illustrated in this chapter.

Henceforth, we will refer to any conditional expression entered in the **Filters** window of a *`Rulesheet`* generically as a Filter, regardless of its strict mode of behavior. This will help us to differentiate the expression itself from its specific behaviors.

This chapter uses the automotive insurance Vocabulary example first introduced in the Collections chapter.

Note: This topic is directly related to the Rulesheet's **Database Filter** functionality. When you are set to Integrqataion and Deployment mode, the Enterprise Data Connector (EDC) enables this toggle. When cleared, it is a filter that is applied to the data currently in working memory. When checked, the filter is a database query that can retrieve data from the database which is then added to working memory.

For details, see the following topics:

- What is a filter?
- What is a precondition?
- Using collection operators in a filter
- Multiple filters on collections
- Filters that use OR
- Test yourself - questions – filters & preconditions

What is a filter?

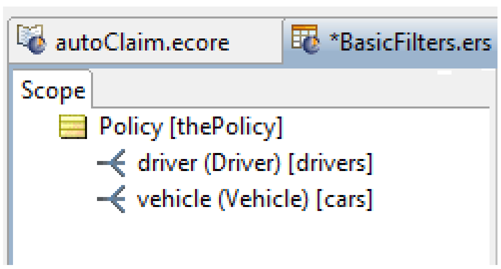
A Filter expression acts to limit or reduce the data in working memory to only that subset whose members satisfy the expression. A Filter *does not* permanently remove or delete any data; it simply *excludes* data from evaluation by other rules in the same *Rulesheet*.

We often say that data satisfying a Filter expression "survives" the Filter. Data that does not survive the Filter is said to be "filtered out". Data that has been filtered out is *ignored* by other rules in the same *Rulesheet*.

A Filter expression, no matter what its full behavior, only affects data in its own *Rulesheet* – *Rulesheet*s are unaffected by Filter expressions in other *Rulesheet*s.

As an example, look at the *Rulesheet* sections shown in the following two figures:

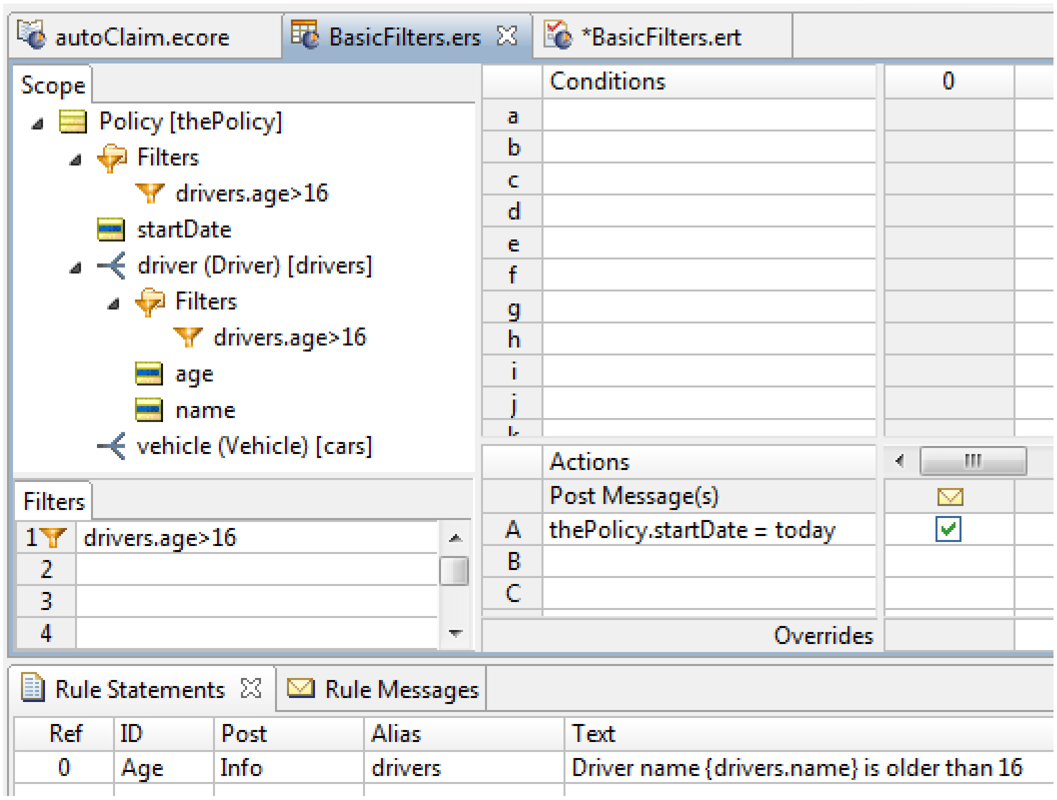
Figure 140: Aliases Declared



The **Scope** window in this figure defines aliases for a root-level `Policy` entity, a collection of `Driver` entities related to that `Policy`, and a collection of `Vehicle` entities related to that `Policy`, named `thePolicy`, `drivers`, and `cars`, in that order.

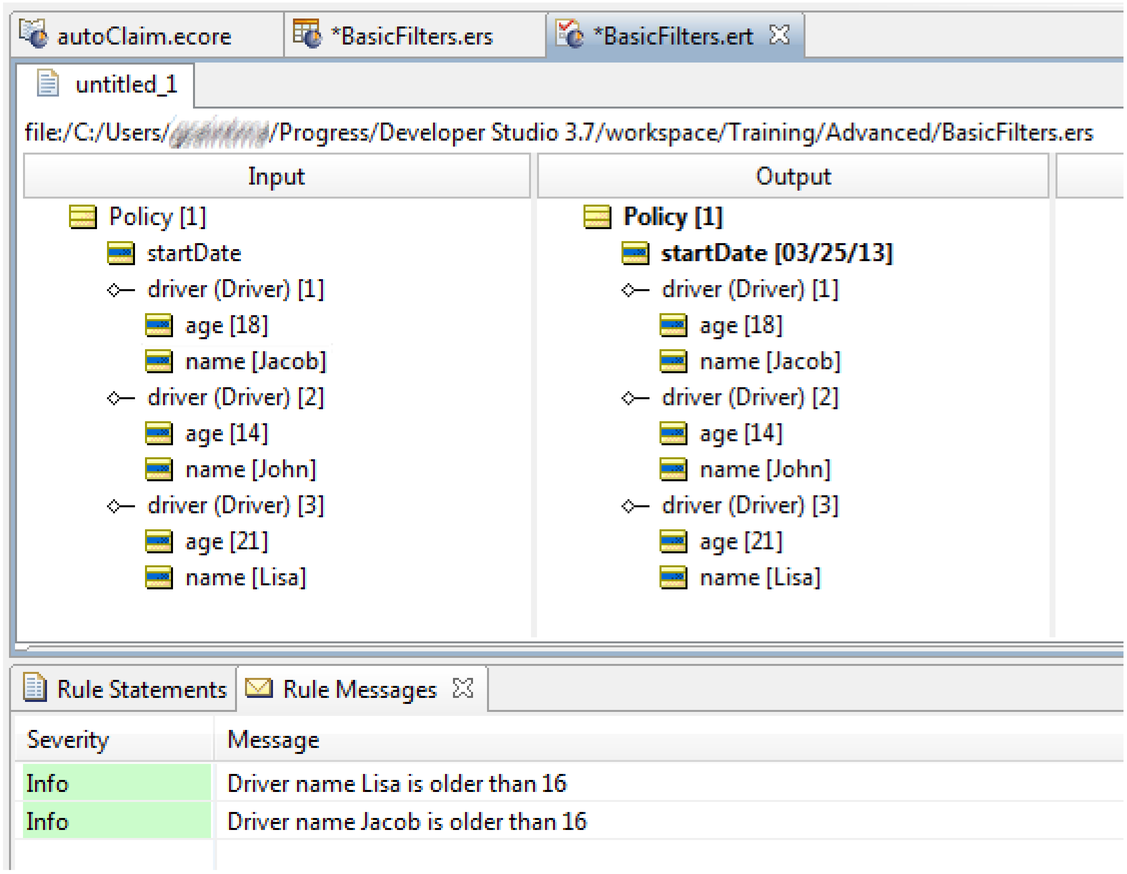
To start with, we will write a simple Filter and observe its default behavior. In the simple scenario below, the Filter expression reduces the set of data acted upon by the Nonconditional rule (column 0), which in this case merely posts the Rule Statement as a message.

Figure 141: Rulesheet to Illustrate Basic Filter Behavior



Our result is not unexpected: for every element in the collection (every `Driver`) whose `age` attribute is greater than `16`, we see a posted message in the *`Ruletest`*, as shown below:

Figure 142: Ruletest to test Filter Behavior



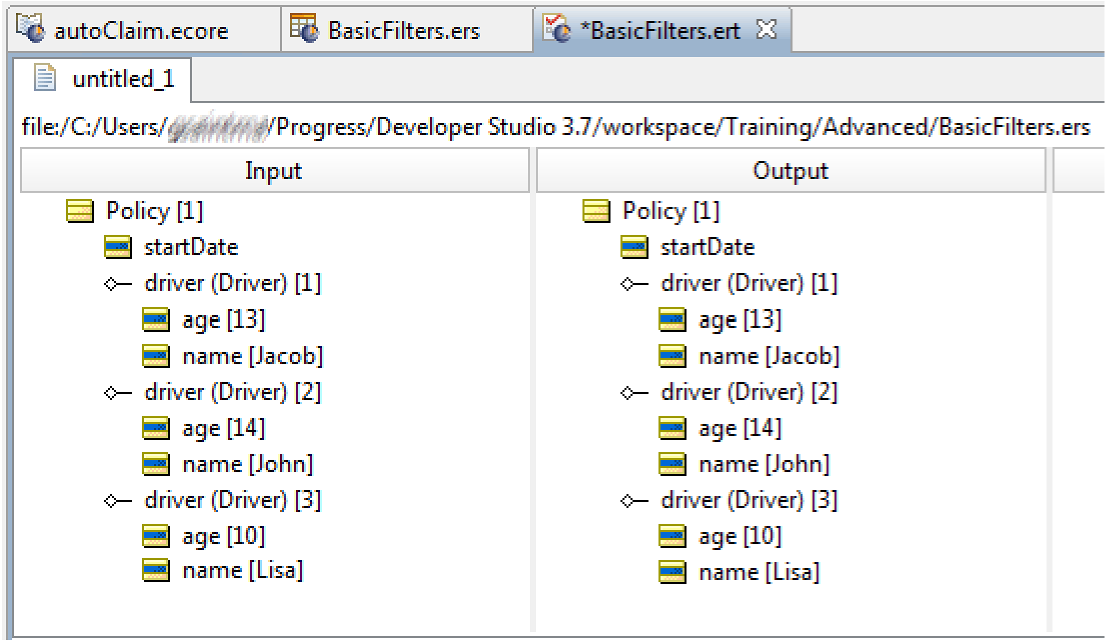
The policy is issued because there are drivers over 16. But because only `Jacob` and `Lisa` are older than 16, Rule Messages are posted only for them.

Full filters

By default, each Filter you write acts as a *full* filter. This means not only will the data not satisfying the Filter be filtered out of subsequent evaluations, but in cases where this data comprises a collection where no elements survive the Filter, *the parent entity will also be filtered out*!

Here is the Testsheet with three juvenile drivers:

Figure 143: Ruletest for Full Filter



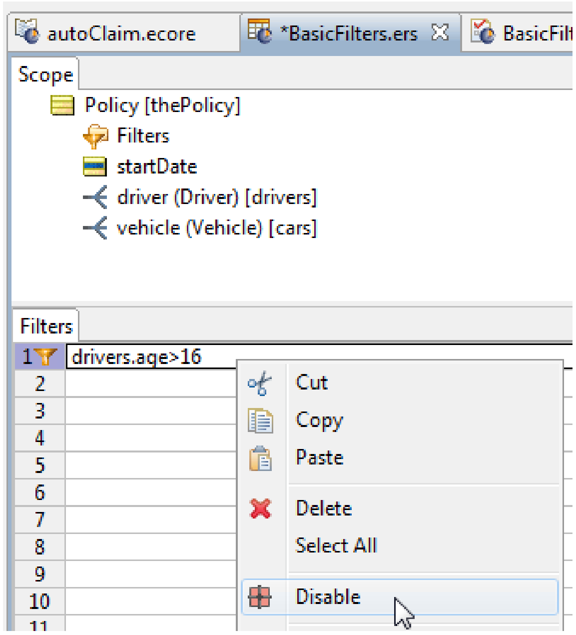
Notice two important things about this *`Ruletest`*'s results: first, none of the `Driver` entities in the Input are older than 16, which means none of them survives the Filter. Second, because the parent `Policy` entity does not contain at least one `Driver` which satisfies the Filter, then the parent `Policy` itself also fails to survive the Filter. If no `Policy` entity survives the Filter, then rule Column 0 has no data upon which to act, so no `Policy` is assigned a `startDate` equal to `today`. The Testsheet's Output, shown in the figure above, confirms the behavior.

Why would we want a Filter to behave this way? Perhaps because, if these are the only drivers seeking a policy, there must be at least one driver of legal age to warrant issuing a policy. While you will probably find that the full filter behavior is generally what you want when filtering your data, it might be too strict in other situations. If other rules on the *`Rulesheet`* act or operate on `Policy`, then a maximum filter gives you a very easy way to specify and control *which* `Policy` entities are affected.

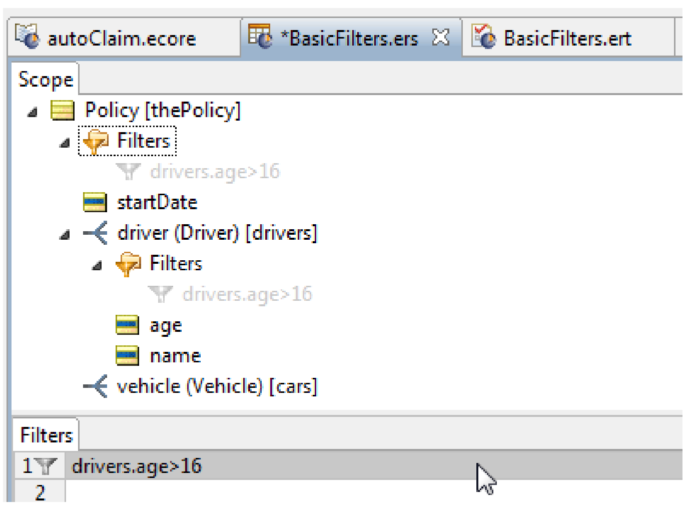
Note: Full filtering, or *maximum* filtering, is also the original behavior of Filters (and of their Precondition/Filter counterparts in prior releases of *`Corticon Studio`*), so for backward compatibility purposes with older models written with these expectations, we have kept it this way as new versions of *`Corticon`* have been released over the years. We wouldn't want to change an important behavior like this and have older Rule Sets begin acting completely differently from their authors' intentions.

Disabling a Full Filter

In testing you will find times when you might want to remove one filter. Instead of deleting the filter, you can simply *disable* it by right-clicking the rule and then choosing **Disable**, as shown:



Once disabled all applications of the filter are rendered in gray, as shown:



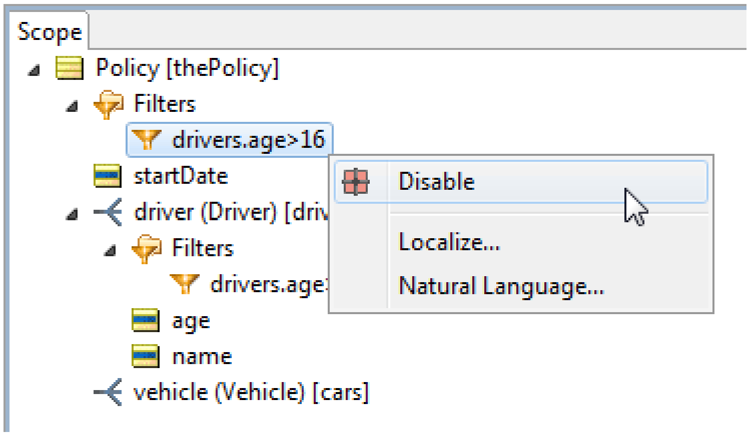
A disabled full filter is really no filter at all. You can perform the corresponding action to again **Enable** the filter.

Limiting filters

There are occasions, however, when the all-or-nothing behavior of a full Filter is unwanted because it is too strong. In these cases, we want to apply a Filter to specified elements of a collection but still keep the selected entities even if none of the children survive the Filter.

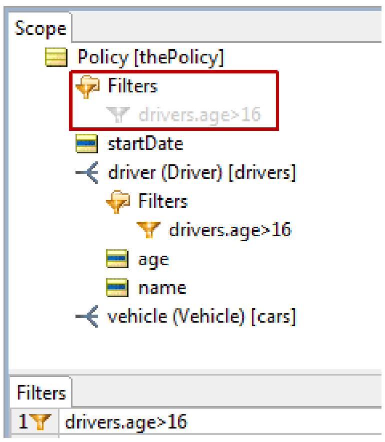
To turn a Filter expression into a limiting Filter, right-click on a Filter in the scope section and select **Disable** from the menu, as shown:

Figure 144: Selecting to limit a filter



This causes that specific filter position to no longer apply, indicated in gray:

Figure 145:



Notice that the filter is still enabled, and that it will still be applied at the `Driver` level. We have ***limited*** the filter.

Use case for limiting filters

The preceding example was basic. Let's explore a some more complex example of limited filters.

Consider the case where there is a rule component designed to process customers and orders.

A Customer has a 1 to many relationship with an Order.

The rule component has 2 objectives, one to process customers and the second to process orders.

If we define a filer that tests for a GOLD status on an order we can have four logical iterations of how the filter could be applied to the ruleset.

```
Case 1: filter is not applied at all.
Case 2: filter is applied to all customers and all orders.
Case 3: filter is only applied to customers.
Case 4: filter is only applied to orders.
```

A business statement for these cases could be as follows:

```
   Case 1: Process all customers and all orders.
   Case 2: Process only GOLD status orders and only customers that have a GOLD
 status order.
   Case 3: Process only customers that have a GOLD status order and all orders
 of a processed customer.
   Case 4: Process all customers and only GOLD status orders.
```

For filter modeling, the filter expression could be written as `Customer.order.status = 'GOLD'` The modeling consideration for the cases are:

```
   Case 1: Filter is not entered (or filter disabled, or filter disabled at
 both Customer and Customer.order levels in the scope).
   Case 2: Filter is entered with no scope modifications (enabled at both
 Customer and Customer.order levels in the scope).
   Case 3: Filter is entered and then disabled at the Customer.order level in
 the scope.
   Case 4: Filter is entered and then disabled at the Customer level in the
 scope.
```

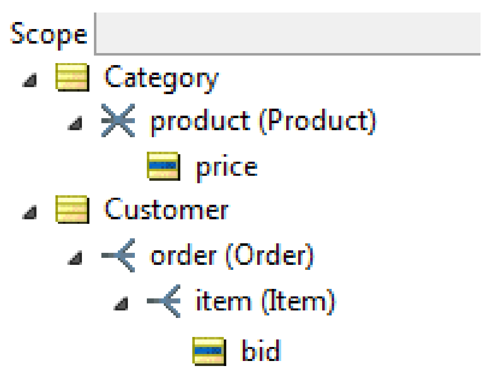
You see how one filter can apply limits to the full filter to achieve the preferred profile of what survives the filter and what gets filtered out.

Next, let's look at more complex set of limiting filters.

Example of limiting filters

Consider the following Rulesheet Scope of a Vocabulary:

Figure 146: Scope in a Rulesheet that will be filtered

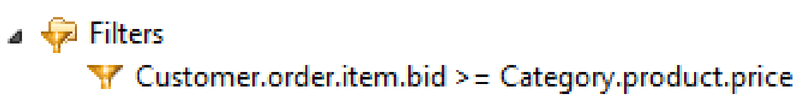


Consider the filter to be applied to data:

`Customer.order.item.bid >= Category.product.price`

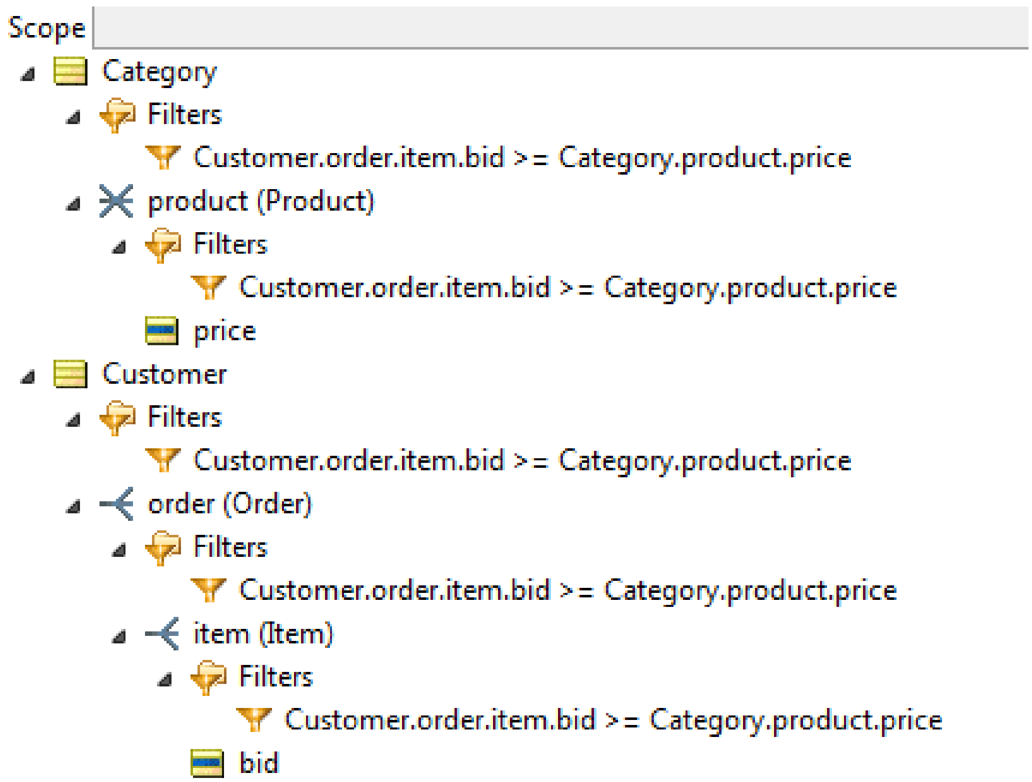
This is shown in the Rulesheet's **Filters** section as:

Figure 147: Definition of a filter



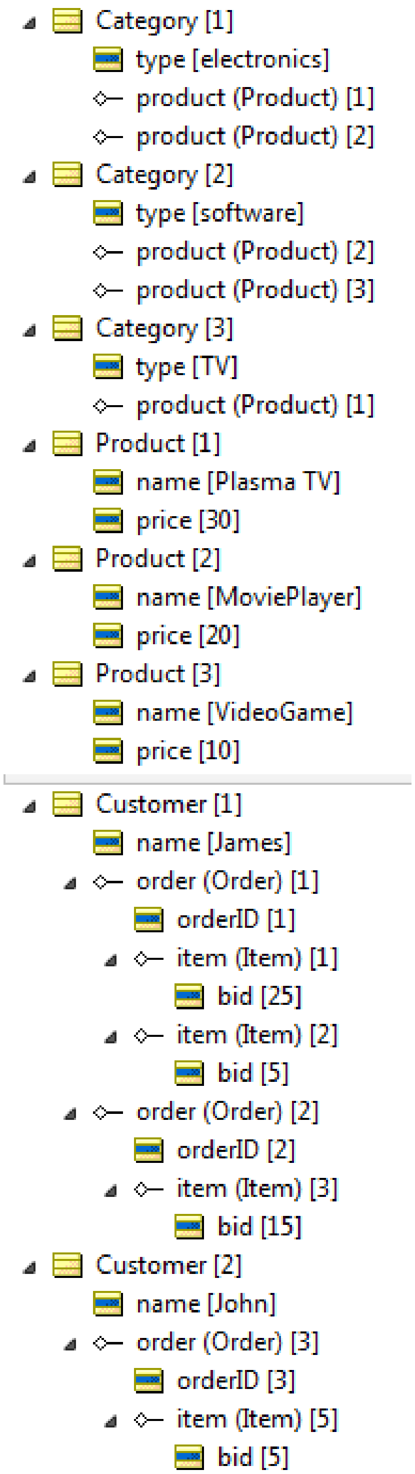
The resulting filter application applies at several levels, as shown:

Figure 148: Application of the filter to the Scope's tree structure



A Ruletest Testsheet might be created as follows:

Figure 149:



This data tree contains five entity types (`Customer`, `Order`, `Item`, `Category`, `Product`).

This filter is evaluated as follows:

```
Customer[1],Order[1],Item[1],Category[1],Product[1] false
Customer[1],Order[1],Item[1],Category[1],Product[2] true
```

```
Customer[1],Order[1],Item[1],Category[2],Product[2] true
Customer[1],Order[1],Item[1],Category[2],Product[3] true
Customer[1],Order[1],Item[1],Category[3],Product[1] false
Customer[1],Order[1],Item[2],Category[1],Product[1] false
Customer[1],Order[1],Item[2],Category[1],Product[2] false
Customer[1],Order[1],Item[2],Category[2],Product[2] false
Customer[1],Order[1],Item[2],Category[2],Product[3] false
Customer[1],Order[1],Item[2],Category[3],Product[1] false
Customer[1],Order[2],Item[3],Category[1],Product[1] false
Customer[1],Order[2],Item[3],Category[1],Product[2] false
Customer[1],Order[2],Item[3],Category[2],Product[2] false
Customer[1],Order[2],Item[3],Category[2],Product[3] true
Customer[1],Order[2],Item[3],Category[3],Product[1] false
Customer[2],Order[3],Item[5],Category[1],Product[1] false
Customer[2],Order[3],Item[5],Category[1],Product[2] false
Customer[2],Order[3],Item[5],Category[2],Product[2] false
Customer[2],Order[3],Item[5],Category[2],Product[3] false
Customer[2],Order[3],Item[5],Category[3],Product[1] false
```

The tuples that evaluate to `true` are:

```
Customer[1],Order[1],Item[1],Category[1],Product[2]
Customer[1],Order[1],Item[1],Category[2],Product[2]
Customer[1],Order[1],Item[1],Category[2],Product[3]
Customer[1],Order[2],Item[3],Category[2],Product[3]
```

The entities that survive the filter are:

```
Customer[1]
Customer[1],Order[1]
Customer[1],Order[2]
Customer[1],Order[1],Item[1]
Customer[1],Order[2],Item[3]
Category[1]
Category[2]
Category[1],Product[2]
Category[2],Product[2]
Category[2],Product[3]
```

The Scope section of the Rulesheet expands as follows:

Notice how the filter is applied towards each discrete entity referenced in the expression:

- When the filter is applied to `Customer`, then the survivor of the filter is `Customer[1]`, if not applied then `{Customer[1], Customer[2]}` survive the filter.
- When the filter is applied to `Customer.order` then the surviving tuples are `{Customer[1], Order[1]}` and `{Customer[1],Order[2]}`. When **not** applied then it is the same (because there was no `Order` child of `Customer[1]` that did not survive the filter).
- When the filter is **not** applied at the `Customer` level as well as the `Customer.order` level, then all `Customer.order` tuples survive the filter with result `{Customer[1],Order[1]}, { Customer[1],Order[2]}, {Customer[2],Order[3]}`
- When the filter is applied to `Customer.order.item` then the surviving tuples are `{Customer[1],Order[1],Item[1]}` and `{Customer[1],Order[2],Item[3]}`. When `not` applied (at this level but at higher levels) then the surviving tuples will be `{Customer[1],Order[1],Item[1]}, {Customer[1],Order[1],Item[2]}, {Customer[1],Order[2],Item[3]}`
- When the filter is applied to `Category` then the surviving entities are `Category[1], Category[2]`. When **not** applied then `Category[1], Category[2], Category[3]`.

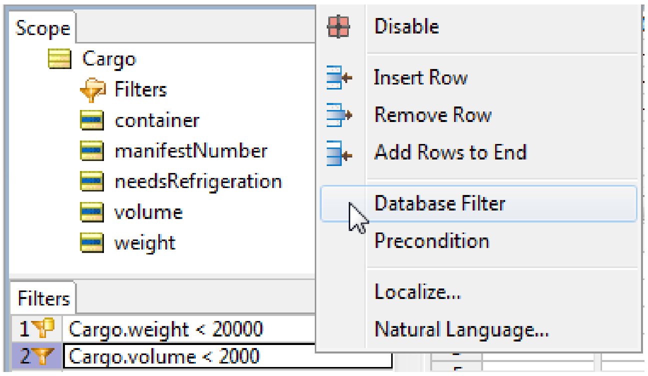
- When the filter is applied to the `Category.product` level then the surviving tuples will be `{Category[1], Product[2]}, {Category[2], Product[2]}, {Category[2], Product[3]}`

You see how a filter applied (at each level) determines which entities are processed when a rule references a subset of the filter's entities. With the *limiting filters* feature, the filter may or may not be applied to each entity referenced by the filter.

Database filters

When set to Integration and Deployment mode, a filter provides a toggle for **Database Filter**, as shown:

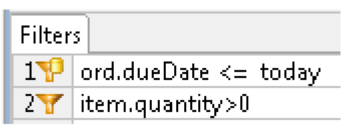
Figure 150: Setting a database filter



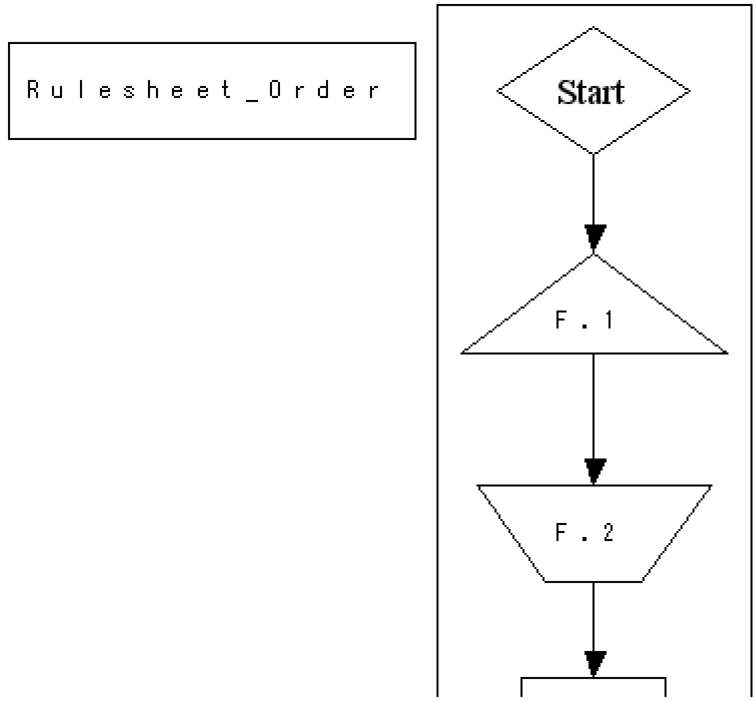
When the option is cleared, the filter is applied only to data currently in working memory.

When checked, the filter becomes a *database query* that will retrieve data from the database, and then add the retrieved data to working memory.

A database filter is distinguished by a database cylinder decoration as shown, where filter 1 is a database filter and filter 2 is a local filter:



When you choose **Rulesheet > Logical Analysis > Execution Sequence Diagram**, the graphic that is generated distinguishes a database filter from local filter by its shape:



In this example, **F.1**, the database query, is displayed within a triangle while **F.2**, the local filter, is displayed within an inverted trapezoid (a quadrilateral with parallel horizontal bases and legs that converge downward.)

What is a precondition?

If you're comfortable with the limiting and full behaviors of a Filter expression, then its precondition behavior is even easier to understand. While reading this section, keep in mind that *Filters always act as either limiting or full filters, but they can **also** act as preconditions* if you enable that behavior as described in this section. If you think of filtering as a *mandatory* behavior but a precondition as an extra, *optional* behavior, then you will be in good shape later. Also, it may be helpful to think of the precondition behavior, if enabled, taking effect *after* the filtering step is complete.

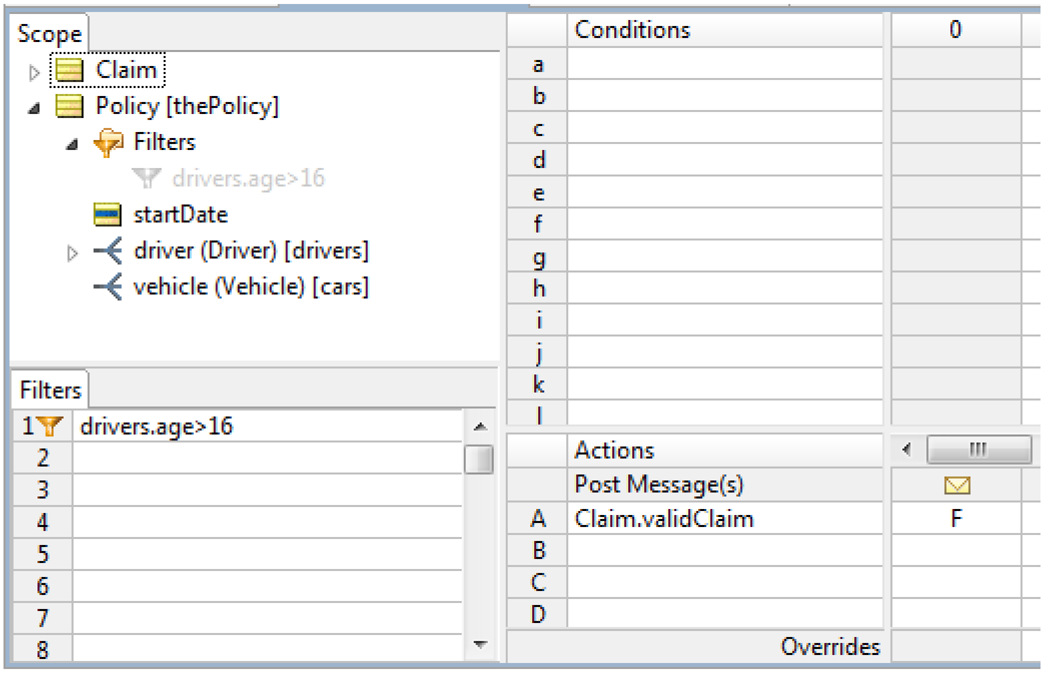
Precondition behavior of a Filter ensures that execution of a *`Rulesheet`* **stops** unless *at least one* piece of data survives the Filter. If execution of a *`Rulesheet`* stops because no data survived the Filter, then execution moves on to the next *`Rulesheet`* (in the case where the *`Rulesheet`* is part of a *`Ruleflow`*). If no more *`Rulesheets`* exist in the *`Ruleflow`*, then execution of the entire *`Ruleflow`* is complete.

In effect, a Filter with precondition behavior enabled acts as a "gatekeeper" for the entire *`Rulesheet`* - if no data survived the Filter, then the *`Rulesheet`*'s "gate stays closed" and no additional rules on that *`Rulesheet`* will be evaluated or executed *no matter what*.

If however, data survived the Filter, then the "gate opens" and the surviving data can be used in the evaluation and execution of other rules on the same *`Rulesheet`*.

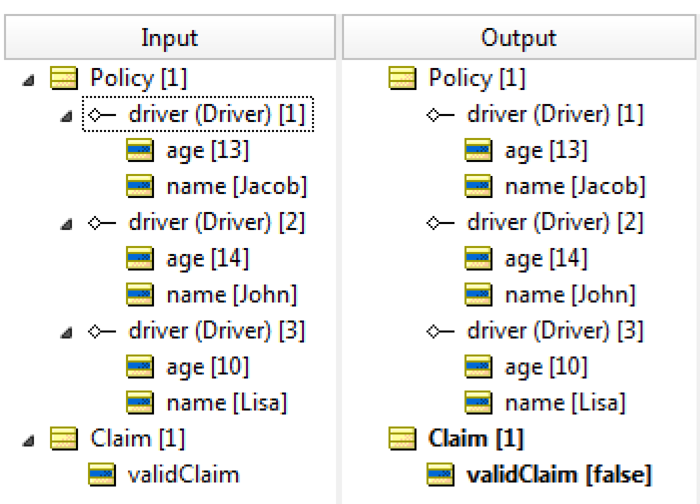
The precondition behavior of a Filter is significant because it allows us to control *Rulesheet* execution regardless of the scope used in the rules. Take for example the *Rulesheet* shown in the following figure. The Filter in row 1 is acting in its standard default mode of full filter. This means that `Driver` entities in the collection named `drivers` and the collection's parent entity `Policy` are both affected by this Filter. Only those elements of `drivers` older than 16 will survive, and at least one must survive for the parent `Policy` also to survive.

Figure 151: Input Rulesheet for Precondition



But how does this affect the `Claim` in Nonconditional row A (of rule column `0`)? `Claim`, as a root-level entity, is safely *outside of the scope* of our Filter, and therefore unaffected by it. Nothing the Filter does (or doesn't do) has any effect on what happens in Action row `A` – the two logical expressions are completely independent and unrelated. As a result, `Claim.validClaim` will always be `false`, even when none of the elements in `drivers` are older than 16. A quick *Ruletest* verifies this prediction:

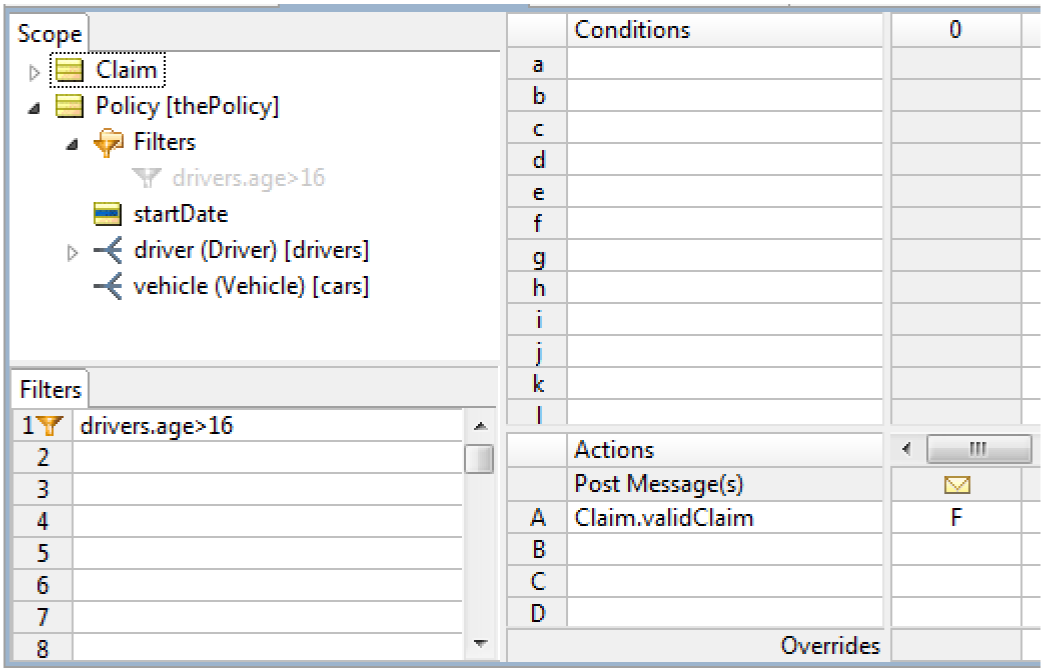
Figure 152: Rulesheet for an Action Unaffected by a Filter



But what if the business intent of our rule is to update `Claim` based on the evaluation of `Policy` and its collection of `Drivers`? What if the business intent *requires* that the `Policy` and `Claim` really be related in some way? How do we model this?

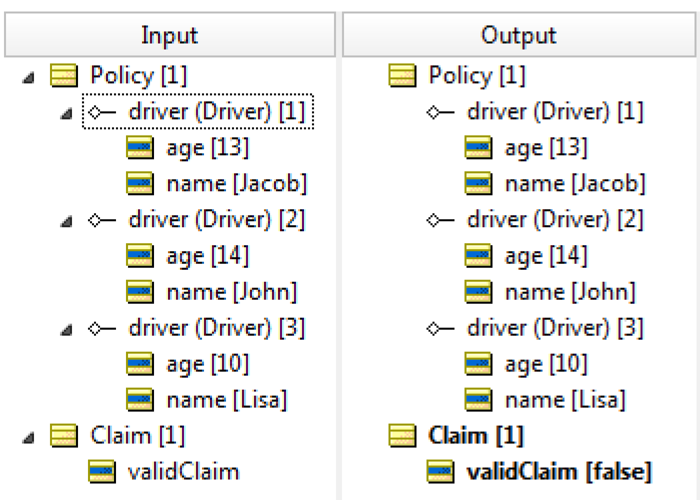
Before "true" precondition behavior was introduced in Studio 4.0, our only practical option was to mandate an actual physical association between `Policy` and `Claim`, then incorporate that association into the scope of our Filter and rules. For example:

Figure 153: Rulesheet for Precondition



Notice that `Claim` is no longer a root-level entity – we have associated it with `Policy` and given the *associated* `Claim` an alias `aClaim`. It is the alias, not the root-level entity, that's used in Nonconditional row `A`. So, when no elements of `drivers` are older than 16, the full filter ensures the parent `Policy` entity does not survive. And since the `Policy` does not survive the filter, its associated `Claim` does not survive, either. Here's an example of this:

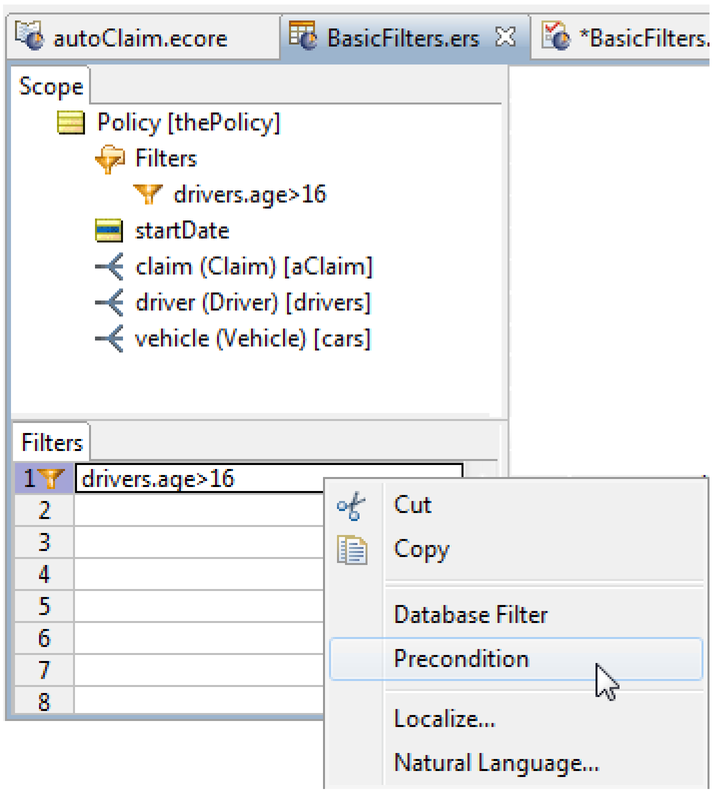
Figure 154: Ruletest for Precondition



The net effect is that `validClaim` can only be `false` when one or more drivers is older than 16, which is what we want. But obtaining this result required us to "monkey around" with our data -- and, possibly our Vocabulary, data model, and database schema as well -- to associate `Claim` with `Policy`. Sometimes we as rule modelers have this freedom and flexibility. Often, we do not. If we don't, then we need an alternative method for controlling the execution of subsequent rules without relying on "unnatural" or artificial data and/or data model manipulations. Here's where the precondition behavior is useful.

Using the same example as in above, right-click on Filter row 1 and select **Precondition**.

Figure 155: Selecting Precondition Behavior from the Filter Right-Click Popup Menu



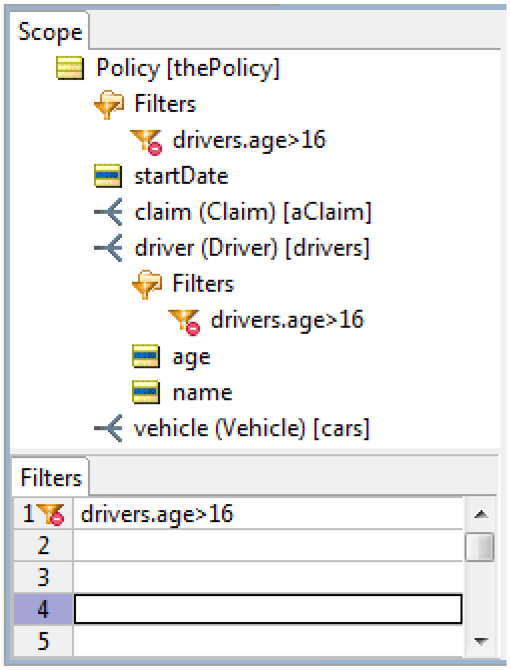
Note that the two options **Precondition** and **Limiting Filter** are mutually exclusive: turning one on turns the other off. A Filter cannot be both a Precondition AND a limiting Filter because at least one piece of data ALWAYS survives a limiting filter, so a Precondition would never stop execution.

Selecting **Precondition** causes the following:

- The yellow funnel icon in the **Filter** window receives a small red circle symbol
- The yellow funnel icons in the **Scope** window receive small red circle symbols

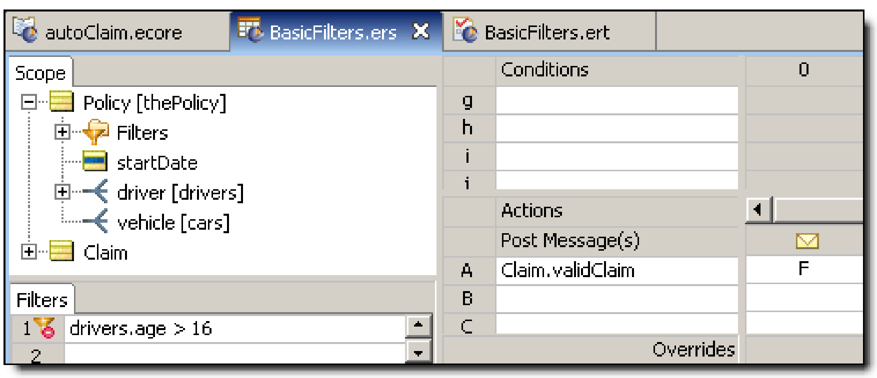
The following figure shows a Filter in **Precondition** mode.

Figure 156: A Filter in Precondition Mode



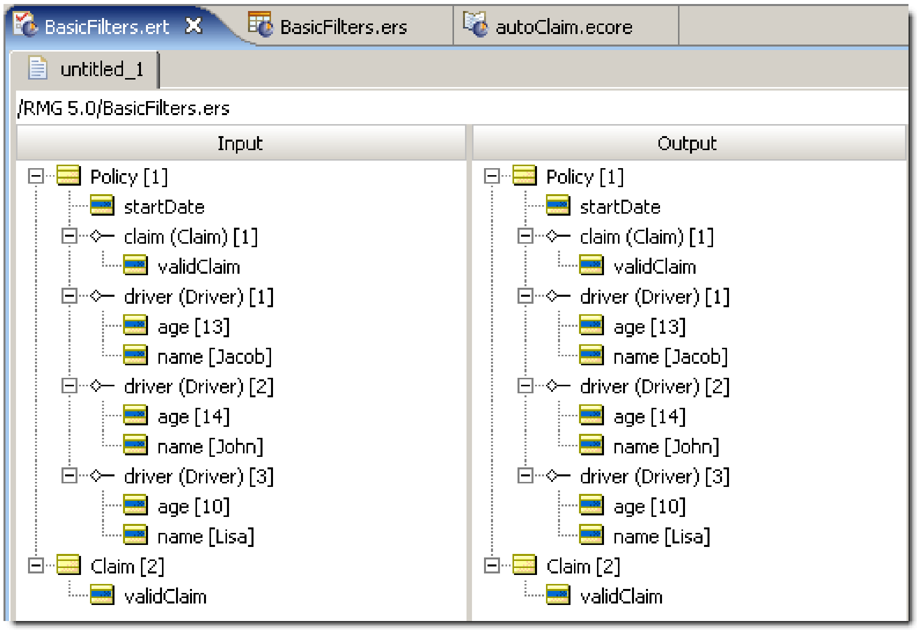
As described before, the precondition behavior of the Filter will cause *Rulesheet* execution to stop whenever no data survives the Filter. So in the original case where `Policy` and `Claim` were unassociated, a Filter in precondition mode, as shown:

Figure 157: Rulesheet with a Filter in Precondition Mode



accomplishes our business intent without artificially changing our Vocabulary or underlying data model. A final proof is provided in the following figure:

Figure 158: Testsheet for a Filter in Precondition Mode



Summary of filter and preconditions behaviors

- A Filter just reduces the available data for other rules in the `Rulesheet` to use. Filters produce shades of gray - all data, some data, or no data may result from a filter.
- A Filter in **Precondition** mode stops `Rulesheet` execution if no data survives the filter. Preconditions produce black and white results: either data survives the filter and the precondition allows `Rulesheet` execution to continue, or no data survives and the precondition forces `Rulesheet` execution to stop.
- Filter expressions always acts as a filter. By default, they act as filters *only*. If you also need true precondition behavior, then setting the Filter to **Precondition** mode will enable precondition behavior while keeping filter behavior.

Performance implications of the precondition behavior

A rule fires whenever data sharing the rule's scope exists that satisfies the rule's conditions. In other words, to fire any rule, the rule engine must first collect the data that shares the rule's scope, and then check if any of it satisfies the rule's conditions. So even in a `Rulesheet` where no rules actually fire, the rule engine may have still needed to work hard to come to that conclusion. And hard work requires time, even for a high-performance rule engine like `Corticon`'s.

A Filter expression acting only as a filter never stops `Rulesheet` execution; it simply limits the amount of data used in rule evaluations and firings. In other words, it *reduces the set of data that is evaluated* by the rule engine, but it does not actually stop the rule engine's *evaluation* of remaining rules. Even if a filter successfully filters out all data from a given data set, the rule engine will still evaluate this empty set of data against the available remaining rules. Of course, no rules will fire, but the evaluation process still occurs and still takes time.

Filter expressions also acting as preconditions change this. Now, if no data survives the filter (remember, Filter expressions always act as filters even when also acting as preconditions) then *Rulesheet* execution stops in its tracks. No additional evaluations are performed by the rule engine. That *Rulesheet* is done and the rule engine begins working on the next *Rulesheet*. This can save time and improve engine performance when the *Rulesheet* contains many additional rules that would have been at least evaluated were the expression in filter-only mode (the default mode).

Using collection operators in a filter

In the following examples, all Filter expressions use their default Filter-only behavior. As we discussed in the Rule Writing Techniques chapter, the logic expressed by the following three `Rulesheet`s provides the same result:

Figure 159: A Condition/Action rule column with 2 Conditional rows

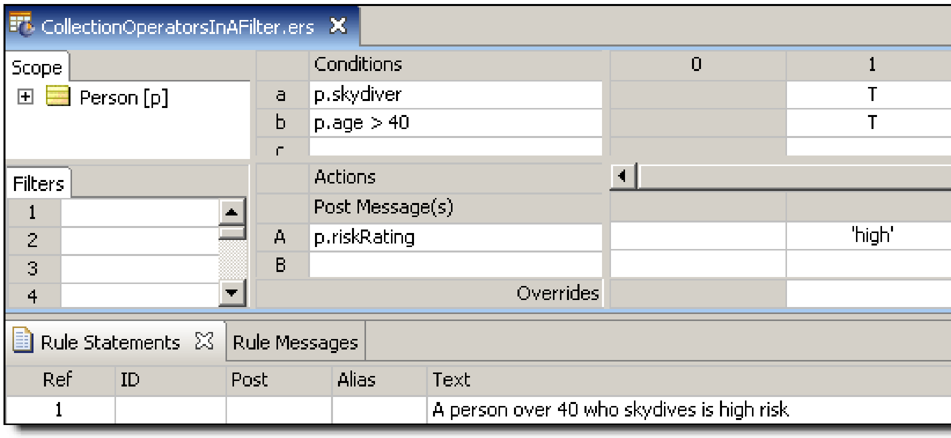


Figure 160: Rulesheet with one Condition row moved to Filters row

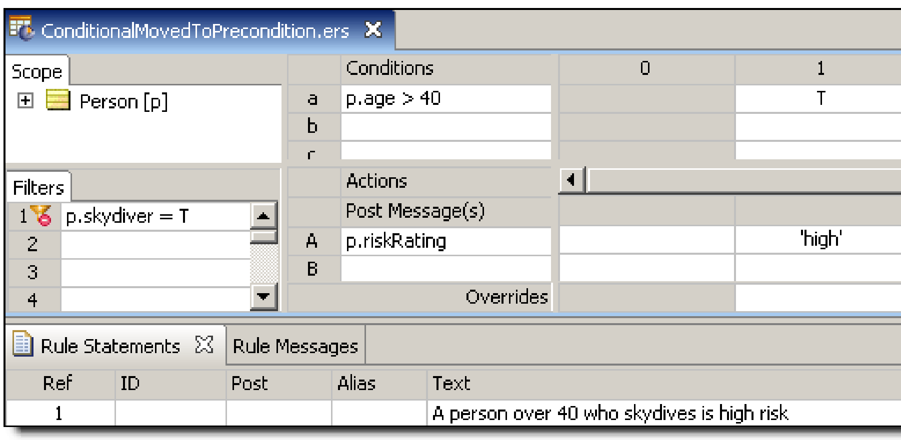
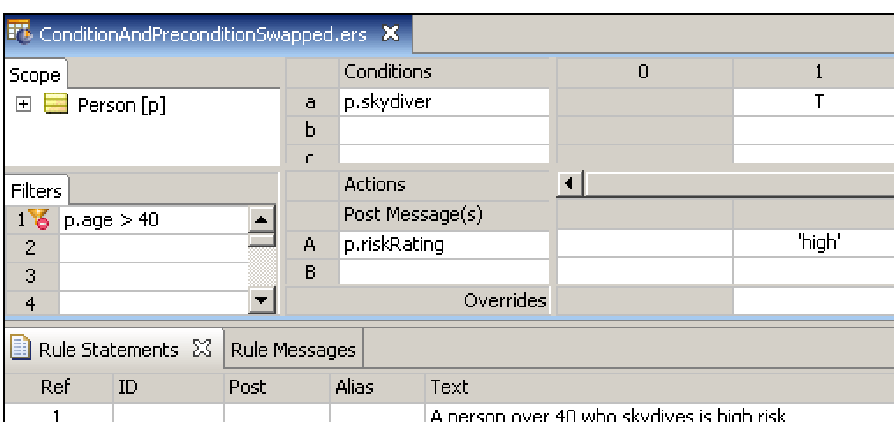


Figure 161: Rulesheet with Filter and Condition rows swapped



Even though expressions in the Filters section of the *Rulesheet* are evaluated before Conditions, the results are the same. This holds true for all rule expressions that do not involve collection operations (and therefore do not need to use aliases – we have used aliases in this example purely for convenience and brevity of expression): conditional statements, whether they are located in the Filters or Conditions sections, are **AND**'ed together. Order does not matter.

In other words, to use the logic from the preceding example:

```
If person.age > 40 AND person.skydiver = true, then person.riskRating = 'high'
```

Because it does not matter which conditional statement is executed first, we could have written the same logic as:

```
If person.skydiver = true AND person.age > 40, then person.riskRating = 'high'
```

This independence of order is similar to the commutative property of multiplication: 4 x 5 = 20 and 5 x 4 = 20. Aliases work perfectly well in a declarative language (like *Studio*'s) because regardless of the order of processing, the outcome is always the same.

Location matters

Unfortunately, order independence does **not** apply to conditional expressions that include collection operations. In the following *Rulesheet*s, notice that one of the conditional expressions uses the collection operator `->size`, and therefore must use an alias to represent the collection `Person`.

Figure 162: Collection Operator in Condition row

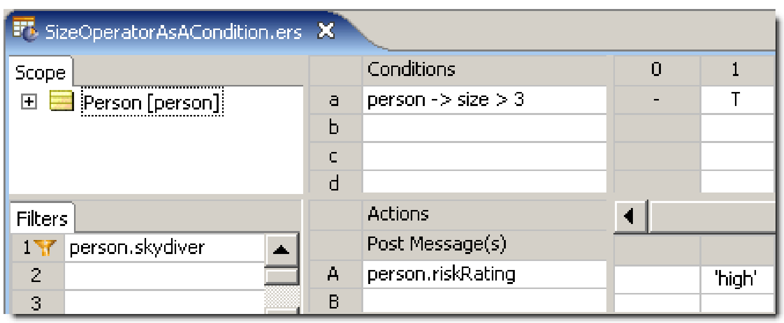
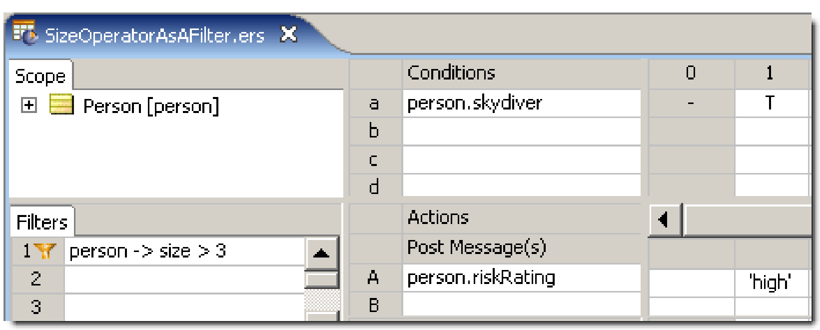
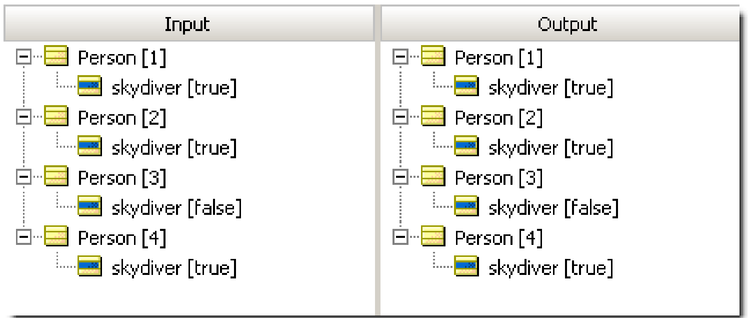


Figure 163: Collection Operator in Filter row



The *Rulesheet*s appear identical with the exception of the location of the two conditional statements. But do they produce identical results? Let's test the *Rulesheet*s to see, testing **Collection Operator in Condition row** first:

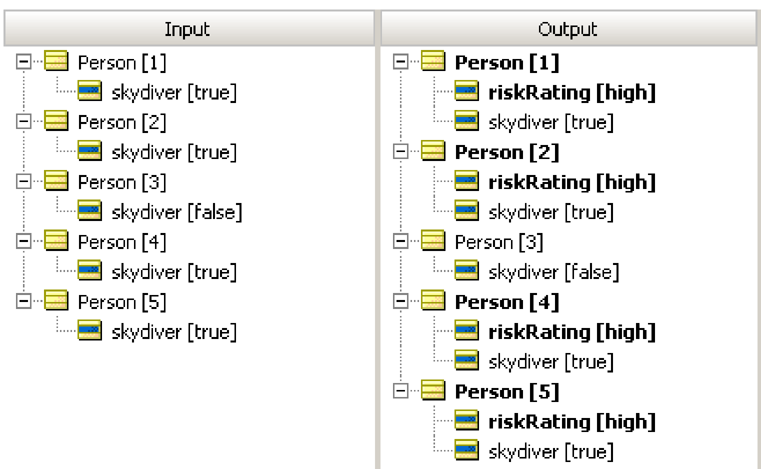
Figure 164: Ruletest with 3 Skydivers



What happened here? Because Filters are always applied first, our *Rulesheet* initially "screened" or "filtered out" the elements of collection `person` whose skydiver value was `false`. Think of the Filter as allowing only skydivers to "pass through" to the rest of the *Rulesheet*. The Conditional rule then checks to see if the number of elements in collection `person` is more than 3. If it is, then ALL `person` elements in the collection *that pass through the filter* (in other words, all skydivers) receive a `riskRating` value of '`high`'. Because our first *Ruletest* included only 3 skydivers, the collection fails the Conditional rule, and no value is assigned to `riskRating` for any of the elements, skydiver or not.

Let's modify the *Ruletest* and rerun the rules:

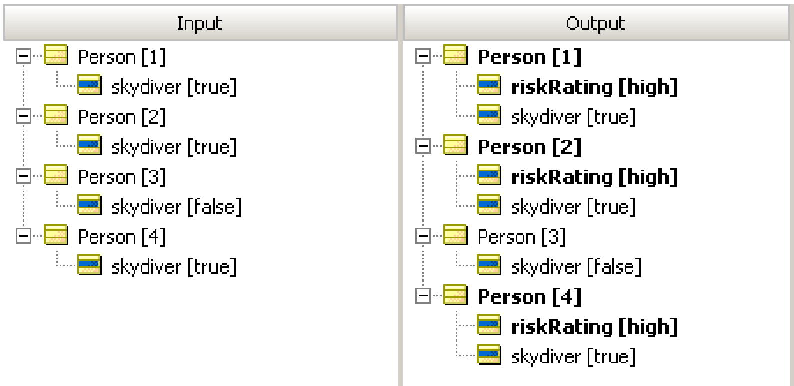
Figure 165: Ruletest with 4 Skydivers



It's clear from this run that our rules fired correctly, and assigned a `riskRating` of `'high'` to all skydivers for a collection containing more than 3 skydivers.

Now let's test the *Rulesheet* in Collection Operator in Filter row, where the rule containing the collection operation is in the Filters section.

Figure 166: Ruletest with 3 Skydivers



What happened this time? Because Filters apply first, the `->size` operator counted the number of elements in our `person` collection, regardless of who skydives and who does not. Here, the Filter allows any collection – *and the whole collection* – of more than 3 persons to "pass through" to the Conditions section of the *Rulesheet*. Then, the Conditional rule checks to see if any of the elements in collection `person` skydive. Each `person` who skydives receives a `riskRating` value of `high`. Even though our *Ruletest* included only 3 skydivers, the collection contains 4 persons and therefore passes the Preconditional filter. Any `skydiver` in the collection then has its `riskRating` assigned a value of `high`.

It's important to point out that the *Rulesheet*s in Collection Operator in Condition row and Collection Operator in Filter row really implement two different business rules. When we built our *Rulesheet*s, we neglected to write the plain-language business rule statements (violating our methodology!). The rule statements for these two *Rulesheet*s would look like this:

1. All skydivers in groups of more than 3 **skydivers** must be assigned a riskRating of 'high'
2. All skydivers in groups of more than 3 **persons** must be assigned a riskRating of 'high'

The difference here is subtle but important. In the first rule statement, we are testing for skydivers within groups that contain more than 3 *skydivers*. In the second, we are testing for skydivers within groups of more than 3 *people*.

Multiple filters on collections

Let's construct a slightly more complicated example by adding a third conditional expression to our rule.

Figure 167: Rulesheet with 2 Conditions

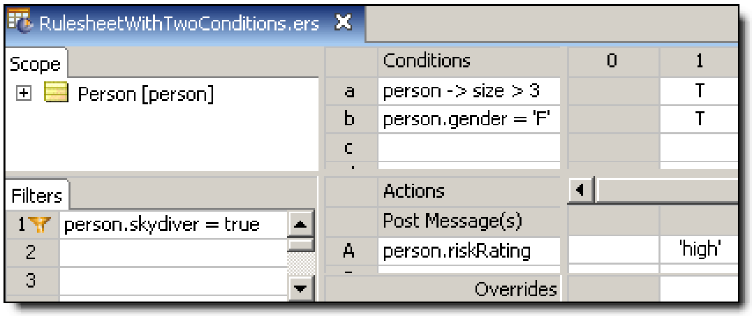
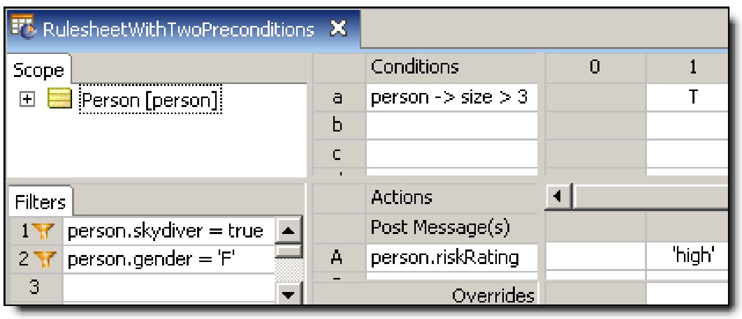
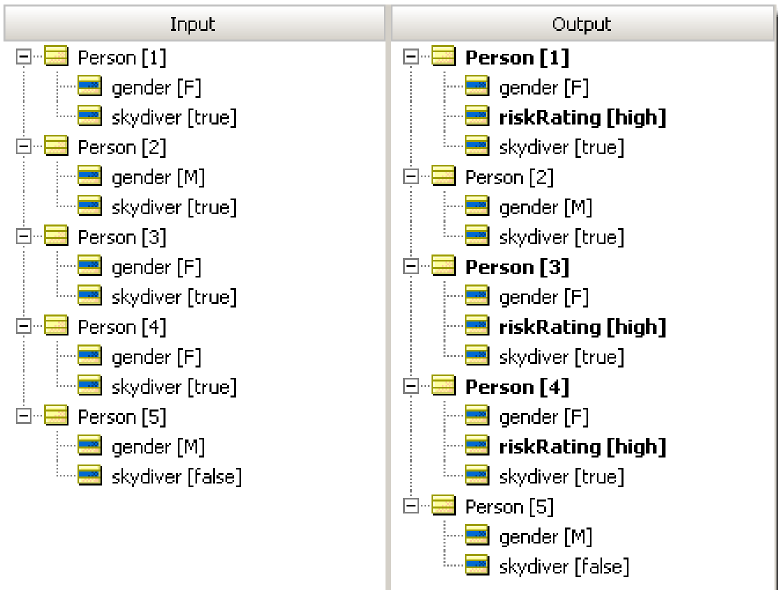


Figure 168: Rulesheet with 2 Filters



Once again, our *Rulesheet*s differ only in the location of a Conditional expression. In the first rulesheet above, the gender test is modeled in the second Conditional row, whereas in the other rulesheet (Rulesheet with 2 Filters), it's implemented in the second Filter row. Does this difference have an impact on rule execution? Let's build a *Ruletest* and use it to test the *Rulesheet* in **Rulesheet with 2 Conditions** first.

Figure 169: Ruletest



As we see in this figure, the combination of a Condition that uses a collection operator (the size test) with another Condition that does not (the gender test) produces an interesting result. What appears to have happened is that, for a collection of more than 3 skydivers, all females in that group have been assigned a `riskRating` of 'high';. Step-by-step, here is what the *Server* did:

1. The Filter screened the collection of Persons (represented by the alias `person`) for `skydivers`.
2. If there are more than 3 "surviving" elements in `person` (i.e., `skydivers`), then all females in the filtered collection are assigned a `riskRating` value of `high`. It may be helpful to think of the *Server* checking to make sure there are more than 3 surviving elements, then "cycling through" those whose gender is female, and assigning `riskRating` one element at a time.

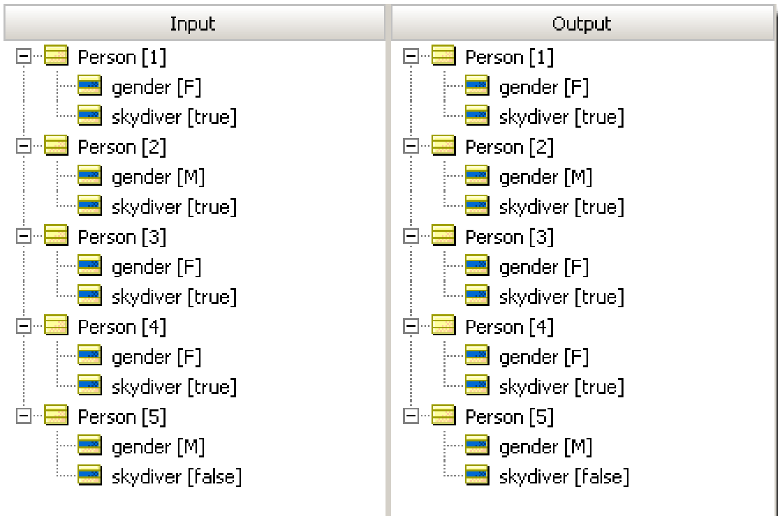
Expressed as a plain-language rule statement, our *Rulesheet* implements the following rule statement:

1.	All female skydivers in a group of more than 3 skydivers must be assigned a riskRating value of high

It's important to note that Conditions **do not** have the same filtering effect on collections that Filter expressions do, and the order of Conditions in a rule has *no effect whatsoever* on rule execution.

Now that we understand the results in Ruletest, let's see what our second Rulesheet produces.

Figure 170: Ruletest

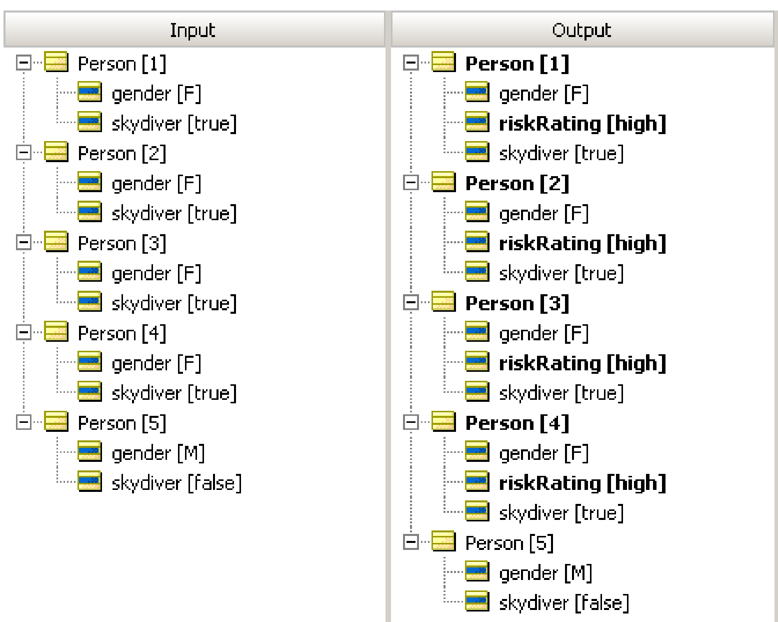


This time, no `riskRating` assignments were made to any element of collection `person`. Why? Because multiple Filters are logically **AND**'ed together, forming a compound filter. In order to survive the compound filter, elements of collection `person` must be both skydivers **AND** female. Elements that survive this compound filter pass through to the "size test" in the Condition/Action rule, where they are counted. If there are more than 3 remaining, then all surviving elements are assigned a `riskRating` value of `high`. Rephrased, our *`Rulesheet`* implements the following rule statement:

1.	All female skydivers in a group of more than 3 female skydivers must be assigned a riskRating of high

Just to confirm we understand how the *`Server`* is executing this *`Rulesheet`*, let's modify our *`Ruletest`* and rerun:

Figure 171: Ruletest



Ruletest now includes 4 female skydivers, so, if we understand our rules correctly, we expect all 4 to pass through the compound filter and then satisfy the size test in the Conditions. This should result in all 4 surviving elements receiving a `riskRating` of `high`. Ruletest confirms our understanding is correct.

Filters that use OR

Just as compound filters can be created by writing multiple Preconditions, filters can also be constructed using the special word `or` directly in the *`Rulesheet`*. See the *Rule Language Guide* for an example.

Test yourself - questions – filters & preconditions

Note: Try this test, and then go to **Answers:** Preconditions & filters on page 281 to correct yourself.

1. True or False. All expressions modeled in the Filters section of the *`Rulesheet`* behave as filters.
2. True or False. All expressions modeled in the Filters section of the *`Rulesheet`* behave as preconditions.
3. True or False. Some rules may be unaffected by Filters expressions on the same *`Rulesheet`*.
4. When 2 conditional expressions are expressed as 2 Filter rows, they are logically ______ together.

or'ed	and'ed	replaced	duplicated

5. True or False. A Filter row is a stand-alone rule that can be assigned its own Rule Statement
6. A null collection is a collection that:
   a. has a parent but no children
   b. has children but no parent
   c. has no parent and no children
   d. has a parent and children
7. An empty collection is a collection that:
   a. has a parent but no children
   b. has children but no parent
   c. has no parent and no children
   d. has a parent and children

8. A Filter expression is equivalent to a Conditional expression as long as it includes ______ collection operators in the expression.

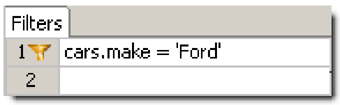
some	all	no	at least one

9. True or False. To join two Filters with an or operator, you must use the word or in between expressions.

10. By default, all Filter expressions are _____________ filters

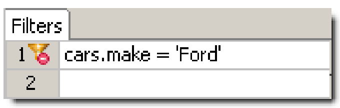
limiting	coffee	full	extreme

11. The Filter expression shown below has which behavior(s)?



limiting filter	full filter	precondition	noncondition

12. The Filter expression shown below has which behavior(s)?

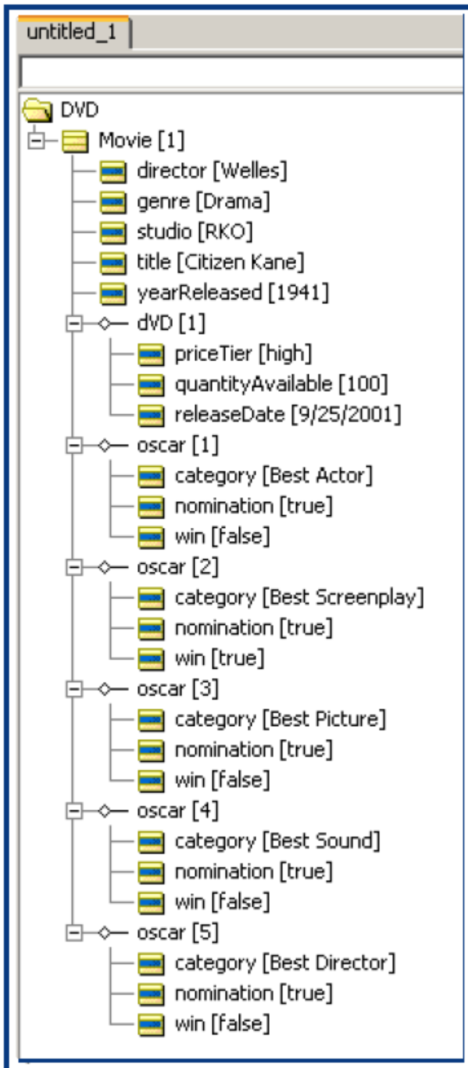


limiting filter	full filter	precondition	noncondition

13. What happens when a Filter expression, acting as a precondition, is not satisfied?
    - **a.** The expression is ignored and *`Rulesheet`* execution continues
    - **b.** The *`Rulesheet`* is re-executed from the beginning
    - **c.** The last *`Rulesheet`* is executed
    - **d.** The next *`Rulesheet`* is executed
    - **e.** All *`Rulesheet`* execution stops
    - **f.** Execution of that *`Rulesheet`* stops

14. Which Filters behaviors may be active at the same time?
    - **a.** Full filter and precondition
    - **b.** Limiting filter and precondition
    - **c.** Limiting and full filter
    - **d.** Precondition may only act alone

15. For the sample data shown below, determine which data survives the Filter for each question. Enter the entity number (the number in square brackets) for each survivor in the appropriate column. Assume the collection `Movie` has alias `movies`, `Movie.dvd` has alias `dvds`, and `Movie.oscar` has alias `oscars`. Full filters are shown in regular type and limiting filters are shown in **bold type**. None behave as Preconditions.



Precondition/Filter Expressions	Movie	DVD	Oscar
example: movies.studio = 'RKO'	1	1	1,2,3,4,5
a. dvd.priceTier = 'high'			
b. oscars -> size > 4			
c. oscars.win = T			
d. oscars.nomination			
e. oscars.win or oscars.category = 'Best Actor'			
f. oscars.win and oscars.category = 'Best Actor'			
g. dvd.quantityAvailable > 100			
h. oscars -> exists(win = T)			
i. movies.yearReleased.yearsBetween (today) > 50			
j. dvd -> notEmpty			
k. movies -> isEmpty			
l. dvd.releaseDate > '1/1/2000'			
m. movies.genre <> 'Drama'			
n. oscars -> forAll(win = T)			
o. oscars -> size > 2			

8

Recognizing and modeling parameterized rules

For details, see the following topics:

- Parameterized rule where a specific attribute is a variable (or parameter) within a general business rule
- Parameterized rule where a specific business rule is a parameter within a generic business rule
- Populating an AccountRestriction table from a sample user interface
- Test yourself - questions – recognizing and modeling parameterized rules

Parameterized rule where a specific attribute is a variable (or parameter) within a general business rule

During development, **patterns** may emerge in the way business rules define relationships between Vocabulary terms. For example, in our sample FlightPlan application, a recurring pattern might be that all aircraft have limits placed on their maximum takeoff weights. We might notice this pattern by examining specific business rules captured during the business analysis phase:

1. 747 aircraft must not exceed maximum cargo weight of 200,000 lbs
2. DC-10 aircraft must not exceed maximum cargo weight of 150,000 lbs

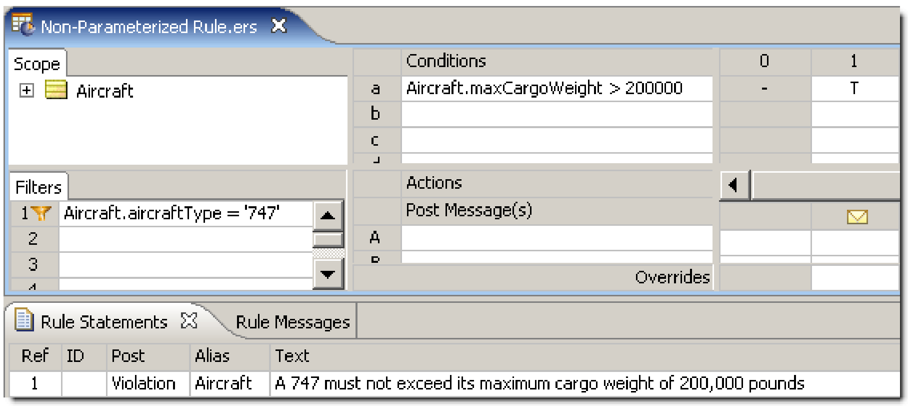
These rules are almost identical; only a few key parts – *parameters* – are different. Although aircraft type (`747` or `DC-10`) and max cargo weight (`200,000` or `150,000` lbs) are different in each rule, the basic form of the rule is the same. In fact, we can generalize the rule as follows:

Where the parameters **X** and **Y** can be organized in table form as shown below:

Aircraft type X	Maximum cargo weight Y
747	200,000
DC-10	150,000

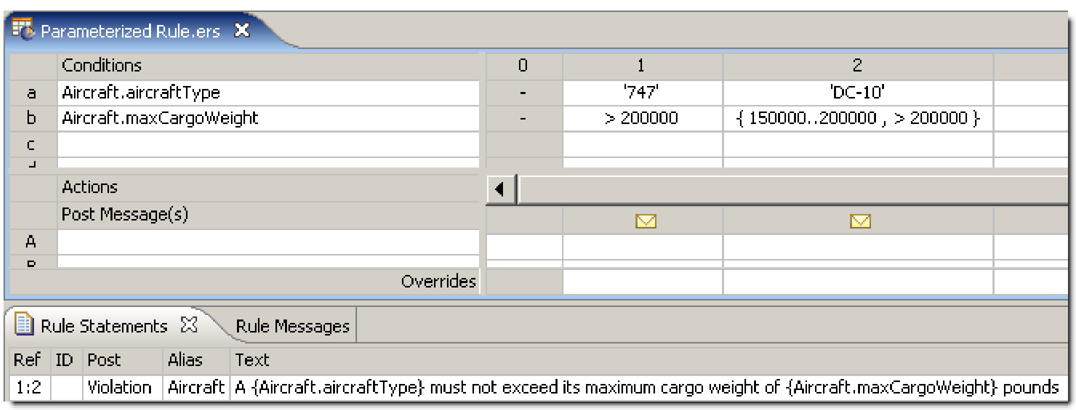
It is important to recognize these patterns because they can drastically simplify rule writing and maintenance in Corticon Studio. As shown in the following figure, we could build these two rules as a pair of *`Rulesheet`*s, each with a Filter expression that filters data by `aircrafttype`.

Figure 172: Non-Parameterized Rule



But there is a simpler and more efficient way of writing these two rules that leverages the concept of parameterization. The following figure illustrates how this is accomplished:

Figure 173: Parameterized Rules



Notice how both rules are modeled on the same *`Rulesheet`*. This makes it easier to organize rules that share a common pattern and maintain them over time. If the air cargo company decides to add new aircraft types to its fleet in the future, the new aircraft types can simply be added as additional columns.

Also notice the business rule statements in the Rule Statements section. By entering `1:2` in the **Ref** column and inserting attribute names into the rule statement, the same statement can be reused for both rule columns. The syntax for inserting Vocabulary terms into a rule statement requires the use of `{..}` curly brackets enclosing the term. See the *Rule Language Guide* for more details on embedding dynamic values in Rule Statements.

In addition to collecting parameterized rules on the same *`Rulesheet`*, other things can be done to improve rule serviceability. In the **Trade Allocation** sample application that accompanies the *`Business Rules Management`* installation, two parameterized rules are accessible directly from the application's user interface – the user can update these parameters without entering the Corticon Studio because they are stored externally. When the application runs, Corticon Studio accesses the parameter table to determine which rules should fire.

Parameterized rule where a specific business rule is a parameter within a generic business rule

The previous section illustrated the simplest examples of parameterized rules. Other subtler examples occur frequently. For example, let's return to the **Trade Allocation** sample application included in the Corticon Studio installation.

A recurring pattern in **Trade Allocation** might be that specific accounts prohibit or restrict the holding of specific securities for specific reasons. We might notice this pattern by examining specific business rules captured during the business analysis phase:

1. The Airbus Account must not hold securities issued by its competitors.
2. The Puritan Pensions Account must not hold securities issued by companies in the Tobacco industry.
3. The SafeHaven Investments Account must not hold securities of less than investment grade quality (less than Bbb)

The first specific rule might be motivated by another, general rule that states:

4. A client's account must not invest in its competition

The general rule explains why Airbus places this specific restriction on its account holdings – Boeing is a competitor. The second rule is very similar in that it also defines an account restriction for a security attribute (the issuer's industry classification), even though the rule has a different motivation. (A client's investments must not conflict with its ethical guidelines?)

There may be many other business rules that share a common structure, meaning similar entity context and scope. This pattern allows us to define a generic business rule:

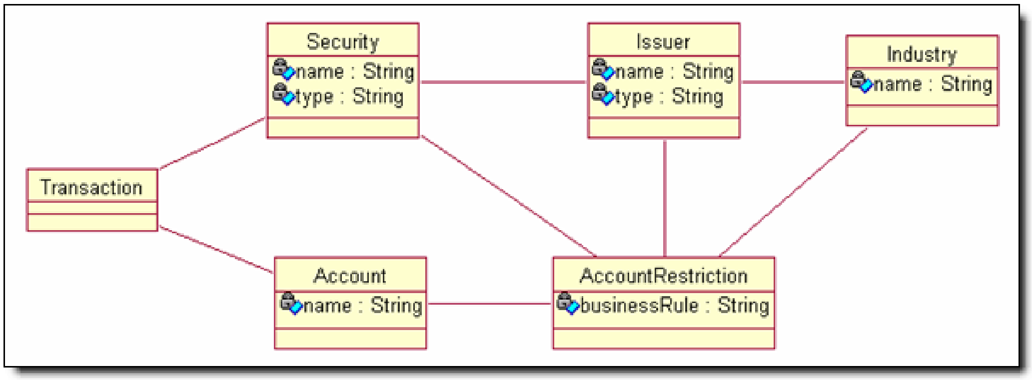
5. An **Account** may restrict holding a **type of Security** for a **specific reason**

Or, rewritten as a constraint:

6. An Account must not hold a type of Security for a specific reason

Absent a method for accommodating many similar rules as a single, generalized case, we need to enter each specific rule separately into a *Rulesheet*. This makes the task of capturing, optimizing, testing, and managing these rules more difficult and time-consuming than necessary. In the example of **Trade Allocation**, an Account Restriction (as a Vocabulary term) might be associated with Account (as the "holder" or "owner" of the restriction), as well as other entities shown in the following figure. For illustration purposes, the Vocabulary is shown as a UML class diagram.

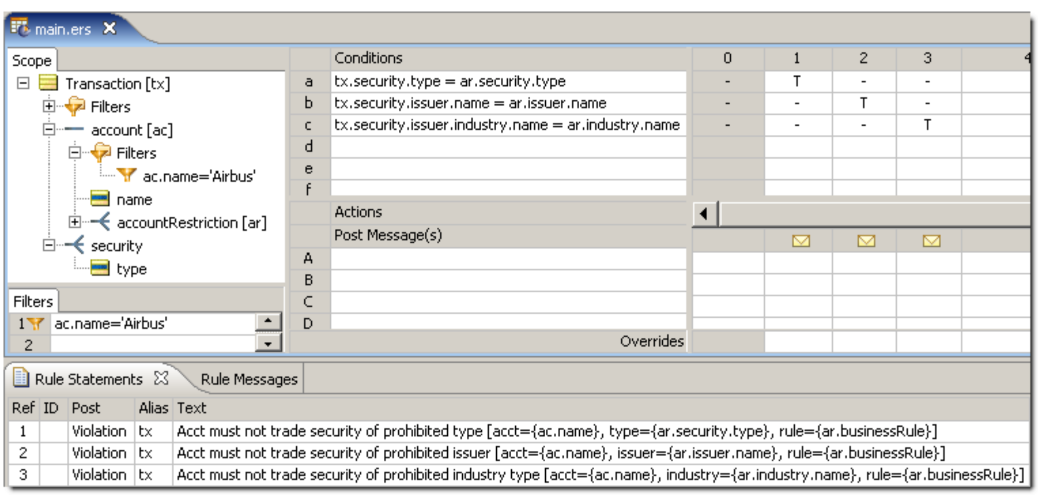
Figure 174: UML Class Diagram of Sample Vocabulary



With this Vocabulary, the following *Rulesheet* can be defined:

Entity/Attribute	Generic business rule
`Security.type`	An account must not hold a security of a restricted type
`Issuer.name`	An account must not hold a security issued by a restricted company
`Industry.name`	An account must not hold a security issued by a company in a restricted industry

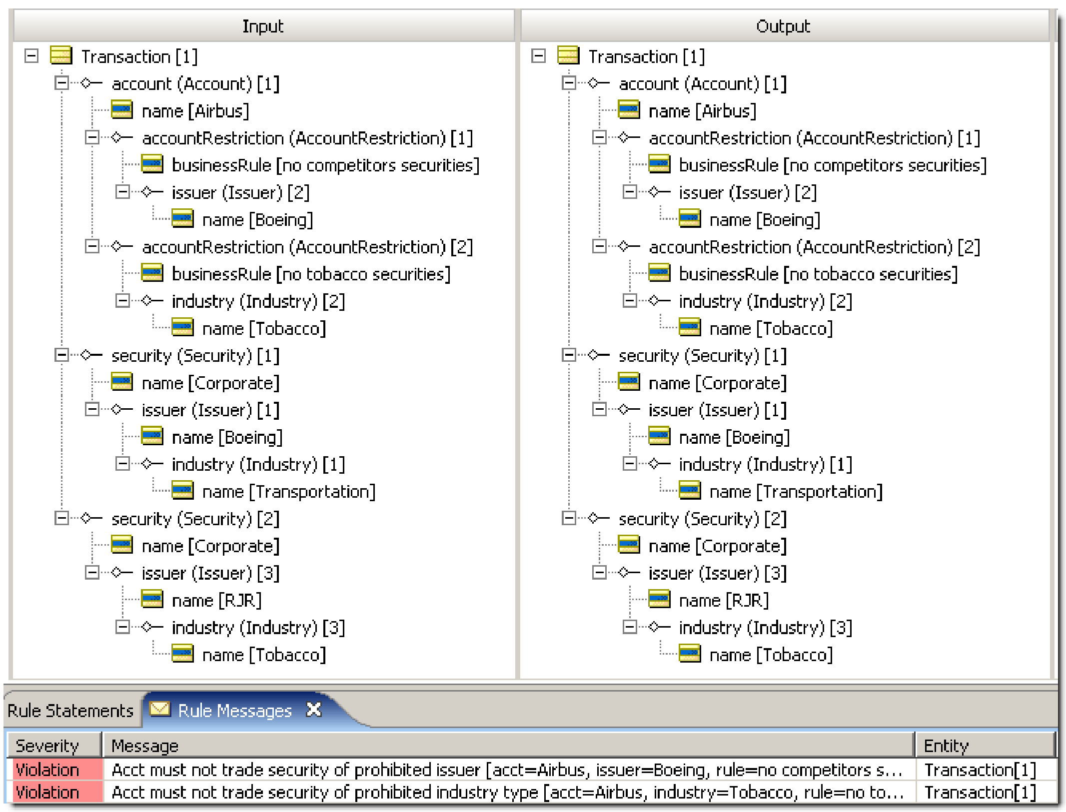
Figure 175: Parameterized Rule Example



Note that `Transaction` is the scope for this *`Rulesheet`* because all included rules apply only to `Securities` related to a specific `Account` and contained in the same `Transaction`. (See the topic Rule Scope and Context for an in-depth explanation of Scope.) Also, note that the rule statements have been written as *generic* rules, with parameters appended to identify the *specific* examples involved in rule execution. This provides the user with a more complete explanation of which rule fired and why it fired.

The following *`Ruletest`* tests the second and third rule statements. A single transaction contains one account, `Airbus`, which has two account restrictions: `no competitor securities` and `no tobacco industry` securities. Two securities are included in the transaction, one for `Boeing` (a competitor) and one for `RJR` (a company in the tobacco industry). Running the *`Ruletest`* in the following figure, we see:

Figure 176: Ruletest



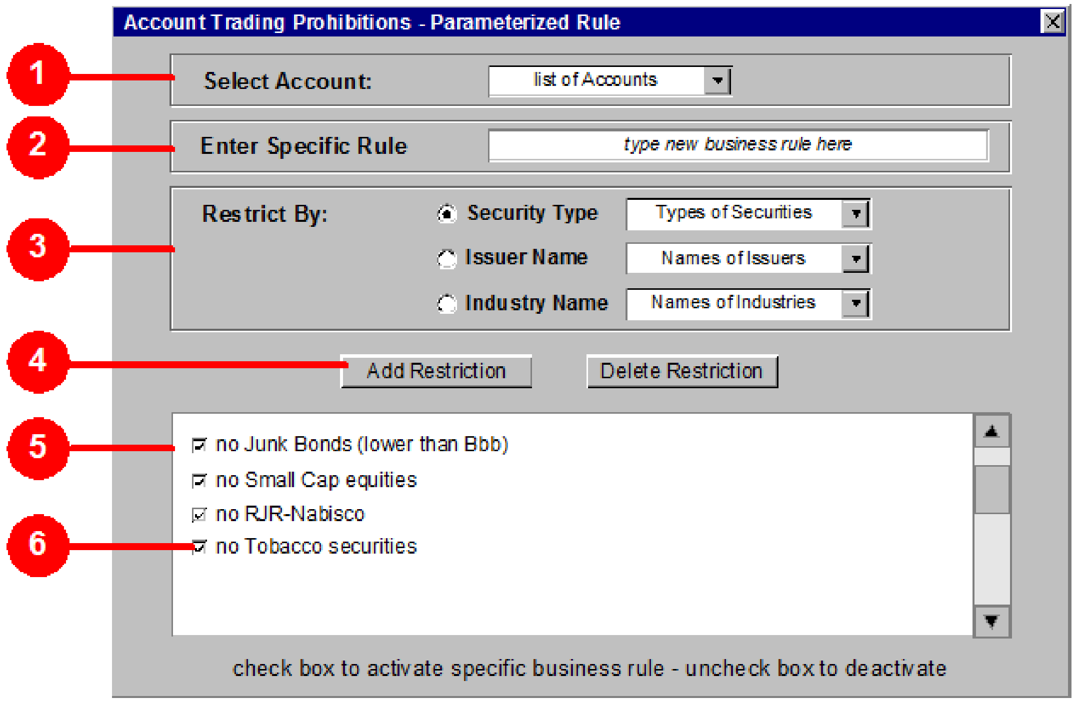
Note the Violation messages posted as a result of the rules firing.

Populating an AccountRestriction table from a sample user interface

Parameterizing rules can improve reuse and simplify maintenance. In fact, maintenance of some well-defined rule patterns can be further simplified by enabling users to modify them *external* to Corticon Studio altogether. A user may define and maintain specific rules that follow the generic rule pattern (analogous to an *instance* of a generic rule *class*) using a graphical interface or database table built for this purpose.

The following is a sample user interface that could be constructed to manage parameterized rules that share similar patterns. Note, this sample interface is discussed here only as an example of a parameterized rule maintenance application. It is not provided as part of the Corticon Studio installation.

Figure 177: Sample GUI Window for Populating a Rule's Parameter Table



1. The user selects an Account for which the Account Restriction will be created. Referring back to our example, the user would select `Airbus` from the list box.
2. The user enters a specific business rule that provides the motivation for the Account Restriction. The prior example used `no competitor securities` and `no tobacco securities`.
3. The user selects the type of restriction being created. Our example used `issuer.name` and `industry.name`.
4. Once all components of the Account Restriction are entered and selected, clicking *Add Restriction* creates the restriction by populating the AccountRestriction table in an external database.

AccountRestriction table				
Account	**Security.type**	**Issuer.name**	**Industry.name**	**Business Rule**
Airbus	---	Boeing	---	No competitor securities
Airbus	---	---	Tobacco	No tobacco securities

5. After adding a restriction, it appears in the lower scrolling text box. Selecting the Business Rule in the scrolling text box and clicking *Delete Restriction* will remove it from the box and from the table.
6. The checkbox indicates an active or inactive Business Rule. This allows the user to deactivate a rule without deleting it. In practice, another attribute could be added to the `AccountRestriction` entity called `active`. A Precondition might filter out inactive rules to prevent them from firing during runtime.

   WARNING!

   Whenever you decide to maintain rule parameters outside of Corticon Studio, you run the risk of introducing ambiguities or conflicts into your *`Rulesheet`*. The Conflict Checker may not help you discover these problems since some of the rule data isn't shown in Corticon Studio. So always try to design your parameter maintenance forms and interfaces to prevent ambiguities from being introduced.

Test yourself - questions – recognizing and modeling parameterized rules

Note: Try this test, and then go to **Answers:** Recognizing & modeling parameterized rules on page 282 to correct yourself.

1. When several rules use the same set of Conditions and Actions, but different values for each, we say that these rules share a common ________________.

2. Another name for the different values in these expressions is ______________ .

3. True or False. When several rules share a pattern, the best way to model them is as a series of Boolean Conditions.

4. What's a potential danger of maintaining rule parameters outside of a Corticon Studio *`Rulesheet`*?

5. Write a generalized rule that identifies the pattern in the following rule statements:

- Platinum customers buy $100,000 or more of product each year
- Gold customers buy between $75,000 and less than $100,000 of product each year
- Silver customers buy more than $50,000 and less than $75,000 of product each year
- Bronze customers buy between $25,000 and $50,000 of product each year

6. In the rules listed above, what are the parameters?

7. Describe the ways in which these parameters can be maintained. What are the advantages and disadvantages of each option?

9

Writing Rules to access external data

Corticon rules can read from and write to an RDBMS. This feature is named Enterprise Data Connector or "EDC", and is sometimes referred to as Direct Database Access or "DDA".

Note: Documentation topics on EDC:

- The tutorial, *Using Enterprise Data Connector (EDC)*, provides a focused walkthrough of EDC setup and basic functionality.
- *Writing Rules to access external data* chapter in the *Rule Modeling Guide* extends the tutorial into scope, validation, collections, and filters.
- *Relational database concepts in the Enterprise Data Connector (EDC)* in the *Integration and Deployment Guide* discusses identity strategies, key assignments, catalogs and schemas, database views, table names and dependencies, inferred values, and join expressions.
- *Implementing EDC* in the *Integration and Deployment Guide* discusses the mappings and validations in a Corticon connection to an RDBMS.
- *Deploying Corticon Ruleflows* in the *Integration and Deployment Guide* describes the Deployment Console parameters for Deployment Descriptors and compiled Decision Services that use EDC.
- *Vocabularies: Populating a New Vocabulary: Adding nodes to the Vocabulary tree view* in the *Quick Reference Guide* extends its subtopics to detail all the available fields for Entities, Attributes, and Associations.

You should work through the procedural style of the *Corticon Tutorial: Using Enterprise Data Connector (EDC)* to experience configuring and using this new feature.

Note: The functionality described in this chapter requires that Studio is running in the **Integration & Deployment** mode, and, for your Server, that you have a license file that enables EDC. Contact Progress Software technical support or your Progress Software representative for additional details.

Overview

When you create a Vocabulary, you use the properties of the Entities, Attributes, and Associations to define Rulesheets, Ruletests, and Ruleflows. Everything is local -- any data required by the rules is either entered as Ruletest input or is generated by the rules during execution.

Corticon EDC lets you define mappings to a database so that rules can "reach out" to access (query) a database directly, and then retrieve what it needs "on-the-fly" during execution, thus enriching the information available to the rules.

As useful as this capability is to the technical people responsible for Rulesheet deployment and integration, a rule modeler might ask "what's the cost to me?" In designing EDC, Corticon made this capability as transparent to the rule modeler as possible. In other words, we don't want the rule modeler to worry about where the data "fed" to Corticon Server is stored, how it's retrieved and assembled, or how it is sent. We want the rule modeler to be concerned with getting the rules right, and let everything else follow from there. This is consistent with our declarative approach to rule modeling – modeling rules that express what to do, not how to do it.

This chapter focuses on the aspects of rule modeling that are affected by the Corticon Enterprise Data Connector.

For details, see the following topics:

A Scope refresher

As we have seen throughout this manual, the concept of scope is key to any solid understanding of rule design and execution. Using scope in a rule helps define or constrain which data is included in rule processing, and which data is excluded. If a rule uses the Vocabulary term `FlightPlan.cargo.weight` then we know that those `FlightPlan` entities without associated `Cargo` entities will be ignored. How can we act upon a `FlightPlan.cargo.weight` if the FlightPlan doesn't have an associated cargo? Obviously we can't – and neither can Corticon Studio or Server.

But we also know that Vocabulary root-level entities – `FlightPlan`, for example – have the broadest scope possible. In fact, they have universal or global scope, meaning they include every entity. This means that a rule using root-level `FlightPlan` acts upon every instance of `FlightPlan`, including `Cargo.flightPlan`, `Aircraft.flightPlan`, or any other role using `FlightPlan` that may exist in our Vocabulary.

When we add the ability for the Corticon Server and Studio to dynamically retrieve data from a database, rule scope determines which data to retrieve. This is exactly the same concept as Studio determining which data in an Input Ruletest to process and which to ignore based upon a rule's scope. So if we write rules using root-level `FlightPlan`, then the Studio will process all FlightPlans present in the Input Ruletest during rule execution.

But with EDC's Direct Database Access, the amount of test data is no longer limited to that contained in a single Input Ruletest – it is limited by the size of the connected database. Rules using root-level `FlightPlan` (or any other root-level entity) will force the Server or Studio to retrieve ALL `FlightPlan` entities (records) from the database. If the database is very large, then that will mean a large amount of data is retrieved. For this reason, **database access for root-level rules is turned off by default**. This ensures that we do not accidentally force the Server to perform extremely large and time-consuming data retrievals from the database unless we explicitly require it.

Validation of database properties

When EDC is enabled, the Vocabulary elements - Entity, Attribute, and Association - each have additional properties that can be entered by the user, or inferred from database metadata. Corticon EDC validates these Vocabulary-to-Database mappings, and displays error conditions in a window. There are three aspects to the database validation function:

Dynamic Validation

Corticon Studio validates against imported database metadata as property values change in a Vocabulary. For example, for a database-persistent entity, if you specify a table name that does not exist in the database metadata, the system posts a validation message in the **Problems View**. Studio creates other error and warning validation messages depending on the severity of the issue detected, such as:

- Property values are explicitly contradicted by database metadata.
- You select a property value, then re-import metadata, only to find that the selected value no longer exists in the database schema.

Warnings are also created for "soft" errors, such as:

- If you designate a Vocabulary entity as datastore persistent, the system is unable to infer which database table best matches the entity name, dynamic validation issues a warning message.
- If the system is unable to unambiguously determine the join expression for a given association, the association is flagged as a warning until you select one of the allowable values.

Note: Dynamic validation is always performed against the imported copy of database metadata. You must ensure that metadata is imported into the Vocabulary whenever the database schema is modified.

On-Demand Validation

In addition to dynamic validation, Corticon Studio provides the Vocabulary menu action **Validate Mappings** so that you can validate the Vocabulary, as a whole, against the database schema. Unlike dynamic validation, on-demand validation is performed against the actual schema so it is considered. the definitive test of Vocabulary mappings.

Validation at Deployment

Corticon Server leverages on-demand validation functionality whenever a decision service is deployed. If Corticon Server detects a problem, it throws an exception and prevents deployment.

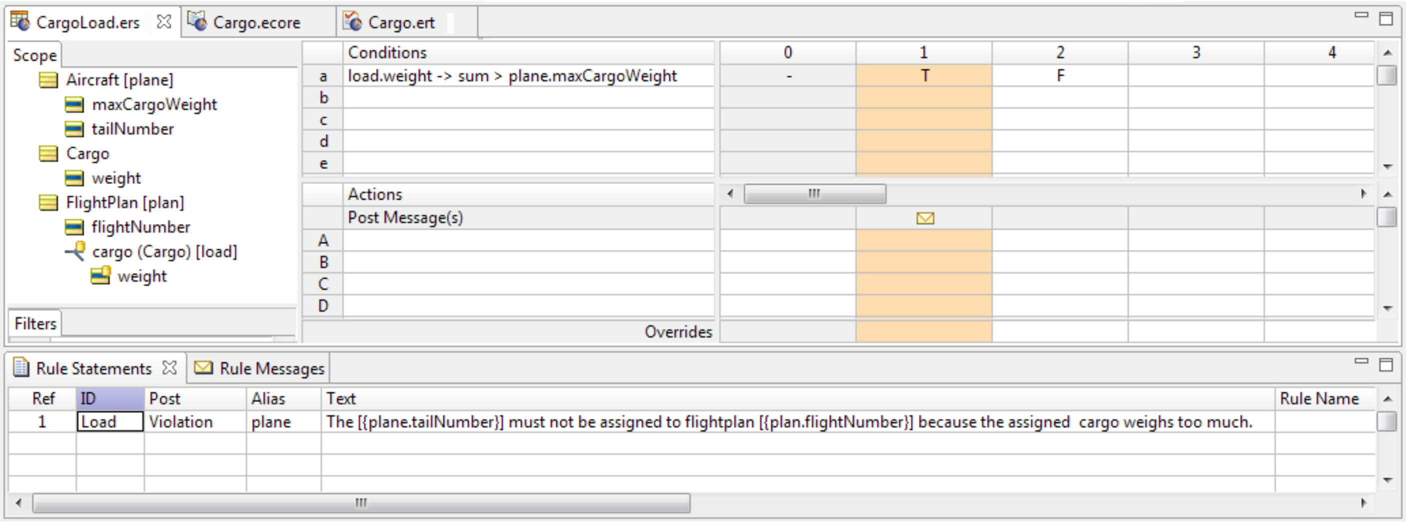
Note: Deployment-time validation check can be turned off in a server property file in (`CcServer.properties`) to “force” deployment despite mapping errors if circumstances warrant it.

Enabling Database Access for Rules using root-level Entities

Since database access for rules using root-level terms is disabled by default, we need to know how to enable it for those circumstances when we do want it. We call this “extending” a root-level entity to the database. To illustrate, we’ll use a simple rule based on the same `Cargo.ecore` used elsewhere in this manual, the *Corticon Studio: Basic Rule Modeling Tutorial*, and the *Corticon Tutorial: Using Enterprise Data Connector (EDC)*.

The process of connecting this particular Vocabulary to an external database is discussed in detail in the EDC tutorial. While the mechanics of this connection may not be of much interest (or importance) to a rule modeler, we do need to be comfortable with Studio Test behavior when connected to an external database in order to focus on what *extending to a database* really means to our rule expressions.

Figure 178: Sample Rulesheet

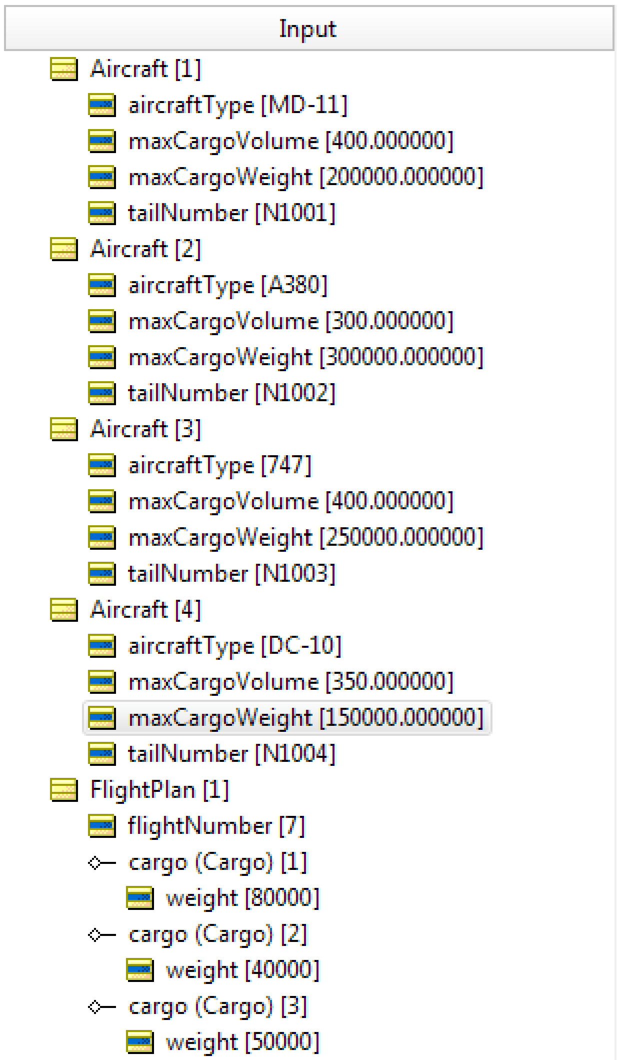


The Rulesheet shown above simply adds up (sums) the collection (you may want to review the Collections chapter for a refresher on this operator or collections in general) of `Cargo` weights associated with a `FlightPlan` (`load.weight`) and compares this to the `maxCargoWeight` of the root-level `Aircraft`. Our intention is to perform this comparison for every available `Aircraft`, so we have used the root-level `Aircraft` in our Conditional expression. Any `Aircraft` whose `maxCargoWeight` is inadequate will be identified with a posted `Violation` message.

Testing the Rulesheet with Database Access disabled

Testing this Rulesheet without database access is a simple matter of building an Input Ruletest with all necessary data. An example of this is a Ruletest we create against the `Cargo.ecore` named `CargoLoad.ert`. Its input data is as shown:

Figure 179: Sample Input Ruletest



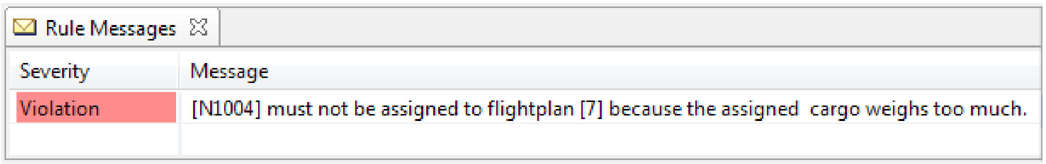
Looking at this Input Ruletest, we see a single `FlightPlan` with its collection of `Cargo` this collection is what we're representing with the alias `load` in our Rulesheet's Scope section. Each `Cargo` has a `weight` value entered.

The four root-level `Aircraft` entities are also shown. Each one has a `maxCargoWeight`, which will be compared to the sum of `load.weight` during rule execution.

Given what we know about rule scope, we can confidently predict that the test data provided in this Input Ruletest will be processed by the Rulesheet because it contains the same scope!

In the following figure, we've executed the Test and see that it functioned as expected. Since `load.weight` sums to `170000` lbs. and the `Aircraft` with `tailNumber` `N1004` can only carry `150000` pounds, we receive a `Violation` message for that `Aircraft` and that `Aircraft` alone. All other `Aircraft` have `maxCargoWeight` values of `200000` or more, so they fail to fire the rule.

Figure 180: Ruletest Violation Message



So far, this behavior is exactly what we have come to expect from rules – they process data of the same scope.

Save the `CargoLoad.ert` Ruletest

Testing the Rulesheet with Database Access enabled

First, let's update the database we have been using in the EDC tutorial to prepare for the features that we want to demonstrate. The Ruletest we just created, `CargoLoad.ert`, has the aircraft data we want including the primary key, the `tailNumber`. It actually extends the tutorial's data with one added row. But it has cargo info we want to keep aside for now.

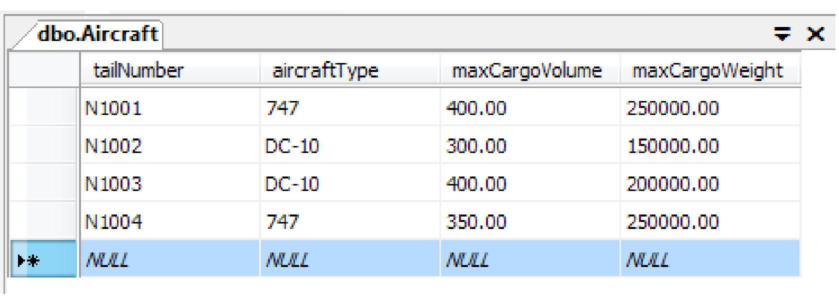
We'll copy the Ruletest, drop those unwanted inputs, and then update the database.

Note: The procedure for connecting and mapping a Vocabulary to an external database, and setting an Input Ruletest to access that database in **Read Only** and **Read/Update** modes is described fully in the *EDC Tutorial*.

To load the aircraft data:

1. In the Rule Project Explorer, copy and paste the `CargoLoad.ert` file. Name the copy `AircraftLoader.ert`.
2. Open `AircraftLoader.ert`.
3. In the Input area, click on `FlightPlan`, and then press **Delete**.
4. Select the menu option **Ruletest > Testsheet > Database Access > Read/Update**.
5. Select the menu command **Ruletest > Testsheet > Run Test**.

Look at the `Aircraft` table in the database. You see the updated values and the new row:



dbo.Aircraft

tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	250000.00
N1002	DC-10	300.00	150000.00
N1003	DC-10	400.00	200000.00
N1004	747	350.00	250000.00
NULL	NULL	NULL	NULL

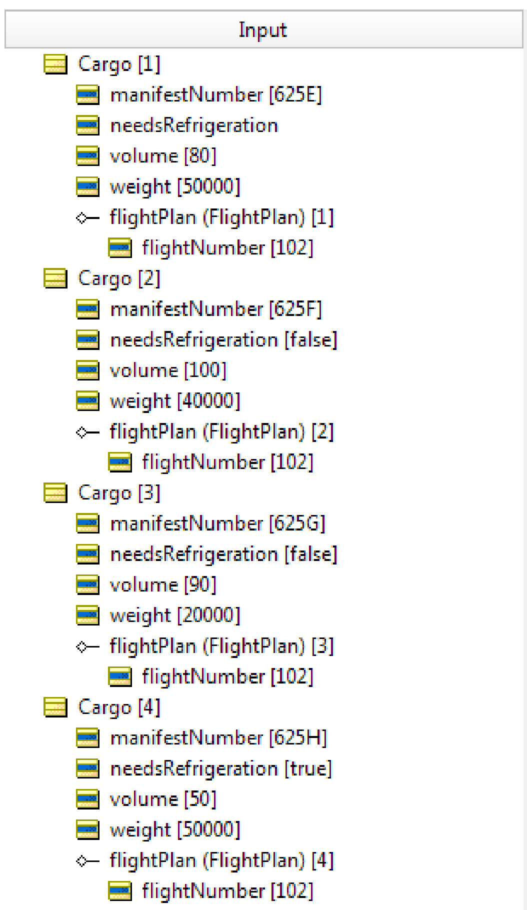
To make the test effective, we need to add some heavy cargo to one of the flight plans. Here, we created four SQL query lines to add four new Cargo manifests to one flight:

```
INSERT INTO Cargo.dbo.Cargo
      (manifestNumber,RflightPlanAssoc_flightNumber,
       needsRefrigeration,container,volume,weight)
     VALUES ('625E',102,null,null,80,50000);
INSERT INTO Cargo.dbo.Cargo
      (manifestNumber,RflightPlanAssoc_flightNumber,
       needsRefrigeration,container,volume,weight)
     VALUES ('625F',102,0,null,100,40000);
INSERT INTO Cargo.dbo.Cargo
      (manifestNumber,RflightPlanAssoc_flightNumber,
       needsRefrigeration,container,volume,weight)
     VALUES ('625G',102,0,null,90,20000);
INSERT INTO Cargo.dbo.Cargo
      (manifestNumber,RflightPlanAssoc_flightNumber,
       needsRefrigeration,container,volume,weight)
     VALUES ('625H',102,1,null,50,50000);
```

Copy the text in the codeblock and paste it into a new SQL Query in your database, and execute it.

Note: You could also create a Ruletest, `CargoLoader`, with these values and the associated `flightPlan`, entering the values as shown, and then running the test in **Read/Update** mode:

Figure 181: Using a Ruletest to add Cargo rows to the connected external database



The `Cargo` table now shows that there are eight items, five of which are assigned to one flight:

Figure 182: Cargo Table from Database

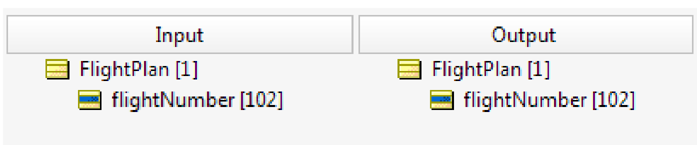
manifestNumber	container	needsRefrigera...	volume	weight
625A	NULL	NULL	10	1000
625B	oversize	False	40	1000
625C	NULL	False	20	30000
625D	NULL	True	10	1000
625E	NULL	NULL	80	50000
625F	NULL	False	100	40000
625G	NULL	False	90	20000
625H	NULL	True	50	50000

Note: We are not evaluating container requirements in these exercises.

Setting up the test

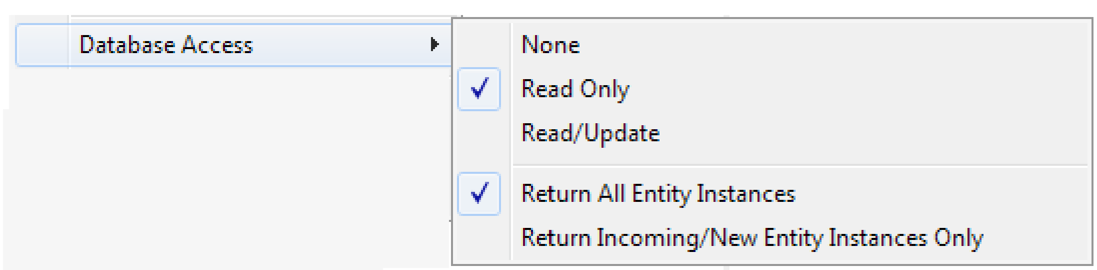
Let's create a new Ruletest that uses the test subject we created earlier, the `CargoLoad.ers` Rulesheet: CargoLoad Rulesheet. We will create a new Input Ruletest that just takes the `FlightPlan` entity from the scope, and then enter the `flightNumber` value `102`. When we run the test, the Output is identical to the input and there are no messages. That seemed to do nothing:

Figure 183: Ruletest of FlightPlan Seed Data



Notice that the only data necessary to provide in the Input Ruletest is a `FlightPlan.flightNumber` value – since this attribute serves as the primary key for the `FlightPlan` table, Studio has all the “seed data” it needs to retrieve the associated `Cargo` records from the `Cargo` database table. In addition to retrieving the `load.weight` collection, we also needed all `Aircraft` records from the `Aircraft` table. But this didn’t happen – no `Aircraft` records were retrieved, so the rule’s comparison couldn’t be made, so the rule couldn’t fire. We should have expected this since we have already learned that database access for root-level terms is disabled by default.

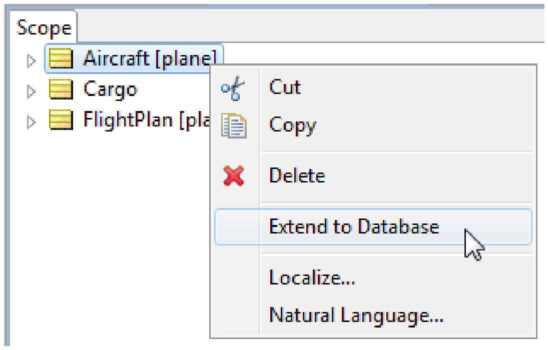
Let's set the Ruletest to read data from the database and return everything that it finds. Toggle the menu options in the **Ruletest > Testsheet** menu as shown:



When we run the test again, the output is still the same as the input and there are no messages.

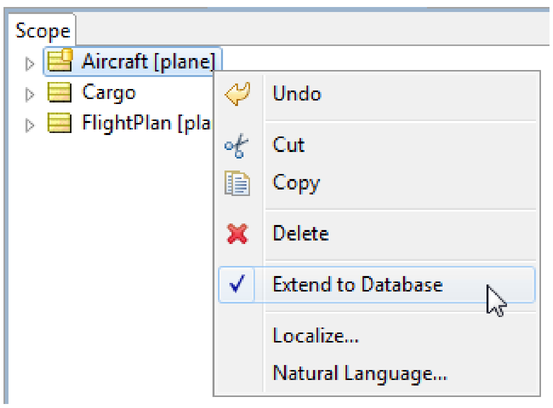
Extend to Database

What we want to do now is set the Rulesheet to **Extend to Database**, and then see how it impacts the test. On the `CargoLoad.ers` Rulesheet, right-click `Aircraft` in the **Scope** area, and then select **Extend to Database**, as shown:



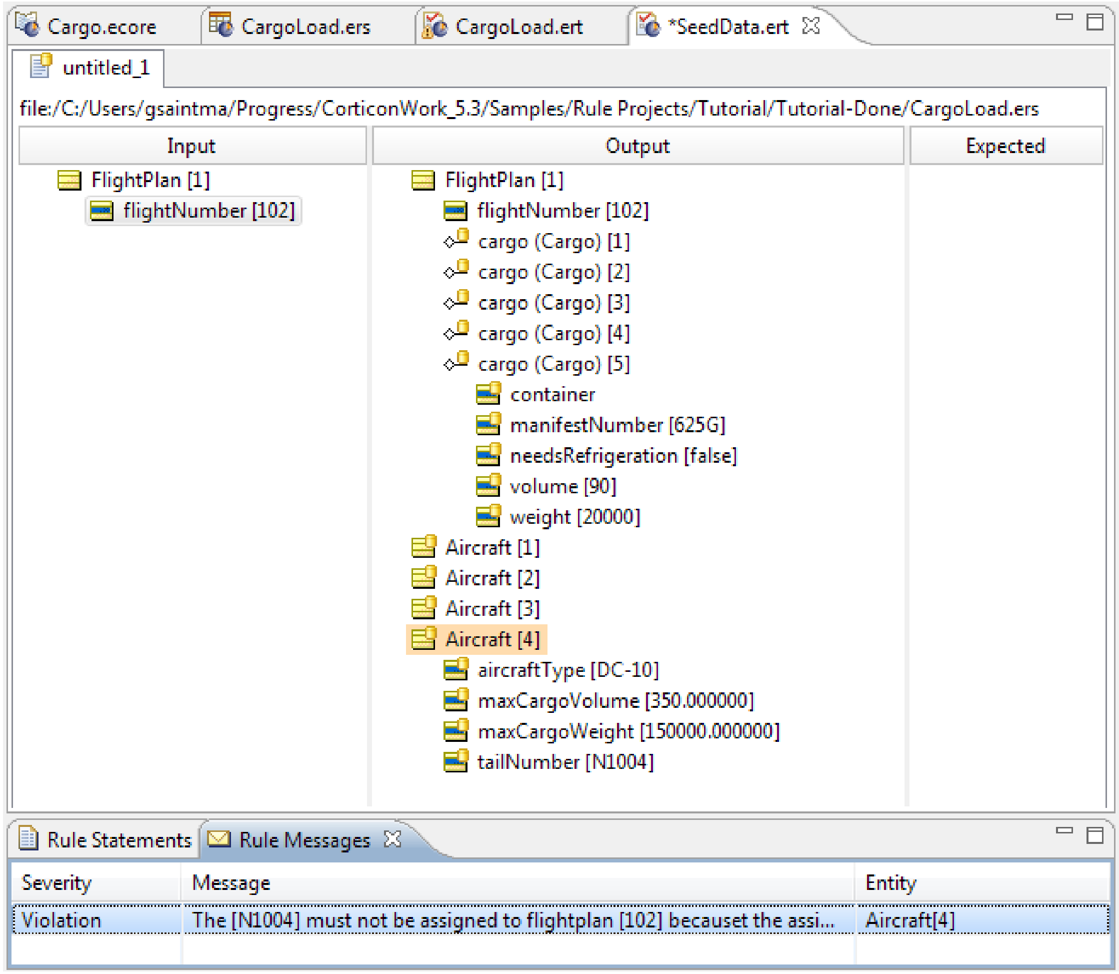
Once selected, the option shows a checkmark, and the `Aircraft` icon in the **Scope** is decorated with a database icon, as shown:

Figure 184: A Root-Level Scope Row Extended to the Database



Save your Rulesheet to ensure that these changes take effect. Now, retest the same Input Ruletest shown in Input Ruletest with Seed Data. The results are as follows:

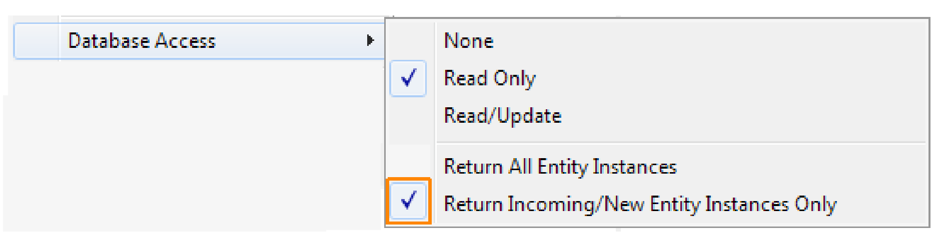
Figure 185: Results Ruletest Showing a Successful Extend-to-Database Retrieval



These results are much different! Corticon has successfully retrieved all `Aircraft` records, performed the summation of all the cargo in the given flightplan, and identified an `Aircraft` record that fails the test. Given this set of sample data, it is the `Aircraft` with `tailNumber N1004` that receives the `Violation` message.

Returning all instances can be overwhelming

While this rich relational data retrieval is good to see, we are only have four planes and five packages in the flight plan. What if we have 1,000 planes and hundreds of thousands of packages every day? That amount of data would be overwhelming. So what we can do is constrain the return data to just relevant new information by toggling the Ruletest's return option to **Return Incoming/New Entity Instances Only**, as shown:

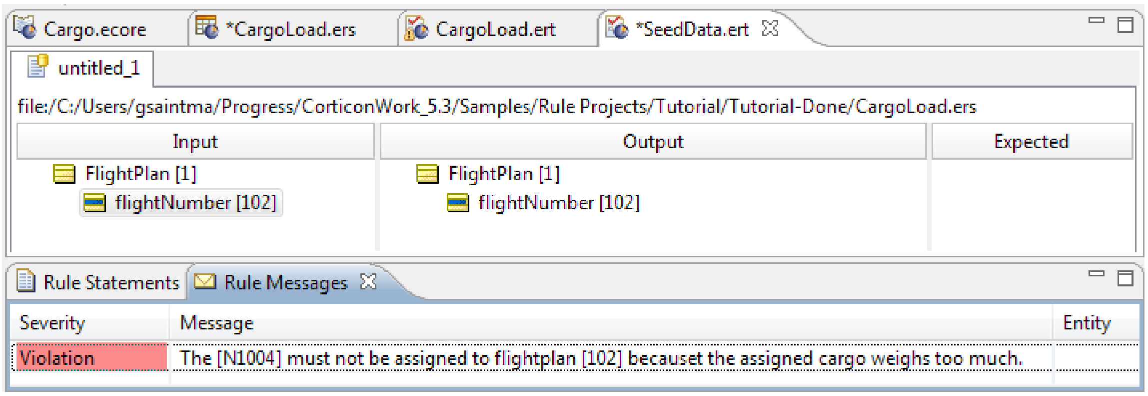


The data that returns is drawn only from those entities that were:

- Directly used in the rules.
- Present in the request message.
- Generated by the rules (if any).

Note: This option can be set in Deployment Descriptor file (`.cdd`), or as a parameter in the 9-parameter version of `addDecisionService` method in the Server API scripts.

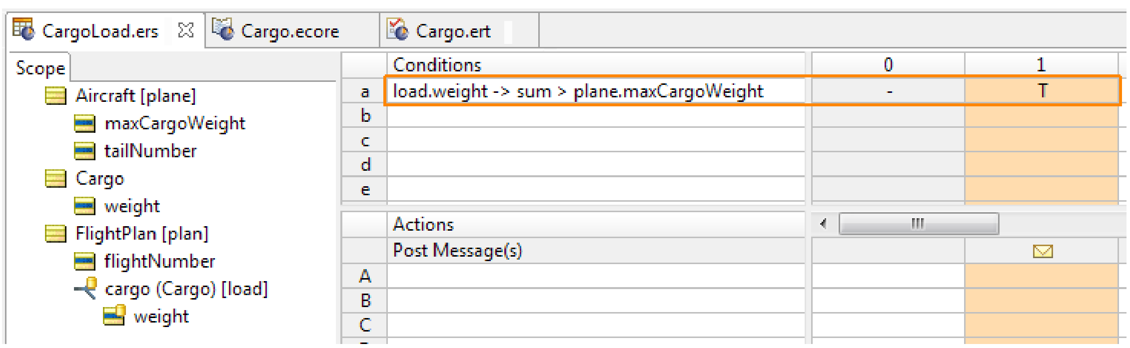
When you run the Ruletest now, the output is unchanged yet we got the Violation message as to which plane cannot be assigned that flight plan.



That result is concise, providing what could be all we really wanted to know in this test.

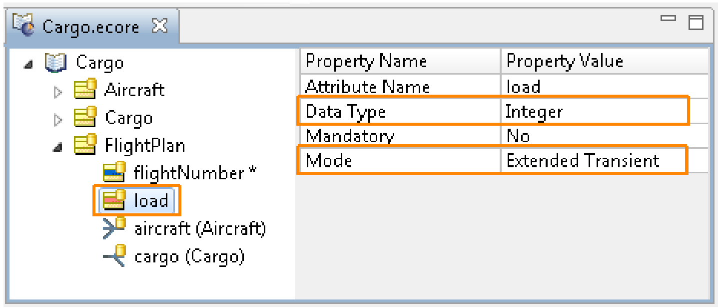
Optimizing Aggregations that Extend to Database

Our rulesheet used a condition statement that did a calculation and a difference, calling a statement when it evaluated as `true`, as shown:

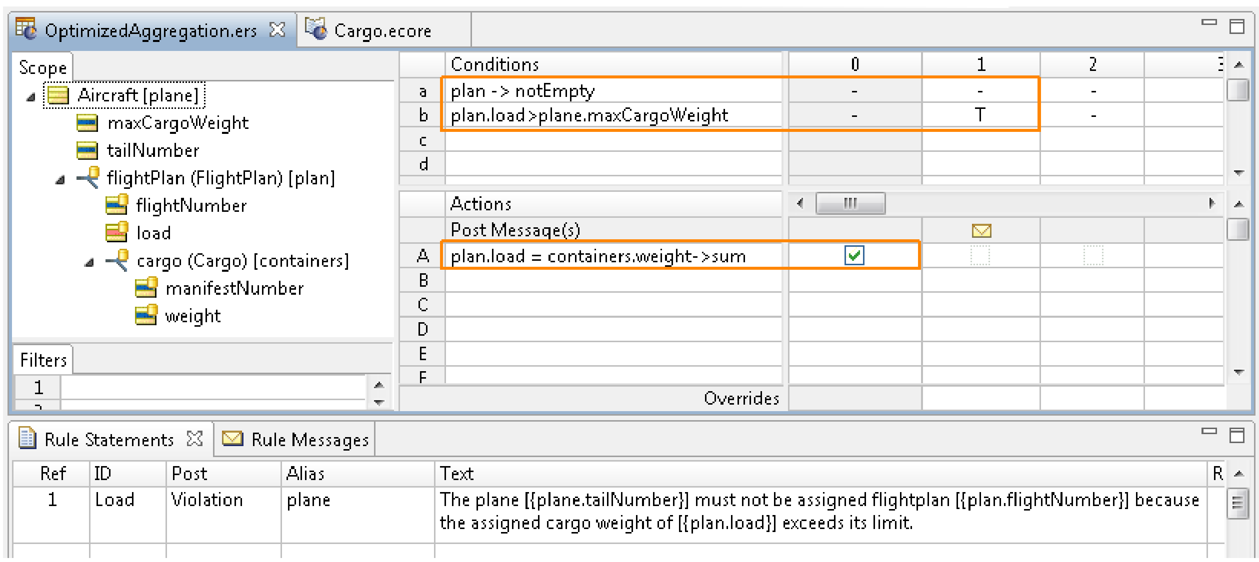


As written, `load.weight ->sum > plane.maxCargoWeight`, the condition will copy all the relevant cargo records into Corticon's memory to perform its `sum`, and then evaluate whether total weight is greater that the plane's capacity. As we are extended to database, the number of values could be large. Corticon lets you optimize such calculations for non-conditional (column 0) actions.

You can recast the conditions by creating an attribute in the `FlightPlan` entity to store a calculation. Here, we created the `load` attribute, and then set its properties so that the Data Type, `Integer` is the same as the `weight` data it will aggregate, and setting the Mode to `Extended Transient` as this is data that will be just used locally:



You could rewrite the conditions and actions to create a non-conditional rule followed by a conditional test of the computed result, as follows:



This optimizes the performance by calculating `load` on the database-side, and then evaluating the `load` against `maxCargoWeight` in memory.

Note: This feature applies to all Collection operators that are **aggregation** operators: `sum`, `avg`, `size`, `min`, and `max`. See Aggregations that optimize database access on page 113 for more information about these Collection operators.

Precondition and Filters as query filters

When the Enterprise Data Connector is in use, Scope rows in a Rulesheet can act as queries to an external database. When an alias definition is designated as **Extend to Database**, the scope of the alias is assumed to include all database records in the Entity's corresponding table. But we often want or need to qualify those queries to further constrain the data returned to Server or Studio. You can think of conditional clauses written in the Preconditions/Filters section of the Rulesheet as placing constraints on these queries. If you are familiar with structured query languages (SQL), then you may recognize these constraints as "`WHERE` clauses" in a SQL query.

If you are not familiar with SQL, that's OK. Review the Filters& Preconditions chapter of this manual to learn more about how a **Precondition/Filter** expression serves to reduce or "filter" the data in working memory so that only the data that satisfies the expression "survives" to be evaluated and processed by other rules on the same Rulesheet. EDC simply extends working memory to an external database; the function of the Precondition/Filter expression remains the same.

For performance reasons, it is often desirable to perform a complete query -- including any `WHERE` clauses -- inside the database before returning the results set (the data) to Studio or Server. An unconstrained or unfiltered results set from an external database may be very large, and takes time to transfer from the database to Studio or Server. Once the results set has entered Studio's or Server's working memory, then Preconditions/Filters expressions serve to reduce (or filter) the results set further before rules are applied. But if we believe the unfiltered results set will take too much time to transfer, then we may decide to execute the Preconditions/Filters expressions inside the database query, thereby reducing the results set prior to transmission to Studio or Server. This may make the entire database access process faster.

Filter Query qualification criteria

When **any** of the following are true, the Precondition/Filter expression ***does not qualify*** as a Query Filter:

1. If it does not contain at least one alias which has been extended to the database.
2. If it contains any attributes of Boolean datatype. Boolean datatypes are implemented inconsistently in commercial RDBMS, and cannot be included in Query Filters.
3. If it uses an operator not supported by databases (see list below)
4. If it references more than one alias not extended to database.

Operators supported in Query Filters

Query Filters are Corticon Rule Language expressions which are performed in the database. As such, the operators used in these expressions must be compatible with the database's native query language, which is always based on some form of SQL. Not all Corticon Rule Language operators have comparable functions in SQL. Those operators supported by standard SQL and therefore also permitted in Query Filters are shown in the table below:

Table 4: Operators supported by Query Filters

Operator Name	Operator Syntax	Datatypes Supported
Equal To (comparison)	=	DateTime, Decimal, Integer, String
Not Equal To	<>	DateTime, Decimal, Integer, String
Less Than	<	DateTime, Decimal, Integer, String
Greater Than	>	DateTime, Decimal, Integer, String

Operator Name	Operator Syntax	Datatypes Supported
Less Than or Equal To	`<=`	DateTime, Decimal, Integer, String
Greater Than or Equal To	`>=`	DateTime, Decimal, Integer, String
Absolute Value	`.absval`	Decimal, Integer
Character Count	`.size`	String
Convert to Upper Case	`.toUpper`	String
Convert to Lower Case	`.toLower`	String
Substring	`.substring`	String
Equal To (comparison)	`.equals`	String
Collection is Empty	`->isEmpty`	Collection
Collection is not Empty	`->notEmpty`	Collection
Size of Collection	`->size`	Collection
Sum	`->sum`	Collection
Average	`->avg`	Collection
Minimum	`->min`	Collection
Maximum	`->max`	Collection
Exists	`->exists`	-

Note: The Collection operators listed above must be used directly on the extended-to-database alias in order to qualify as a Query Filter. If the collection operator is used on an associated child alias of the extended-to-database alias, then the expression is processed in memory.

Using multiple filters in Filter Queries

One or more filters can be set as a database filter. When multiple filters are set as database filters, Corticon logically combines them with the `AND` operator to form one database query.

Note: If the database filters have different entity/alias references they will not be logically combined into one query. Each filter will execute in processing order. To determine which expression gets processed first, generate an execution sequence diagram by choosing **Rulesheet > Rulesheet > Execution Sequence Diagram** from Studio's menubar.

Consider the filters:

- `Customer.age > 18`
- `Customer.status = 'GOLD'`

the result is one database query:

```
Select * from Customer where age > 18 and status = "GOLD"
```

However, when the two filters are:

- `Customer.age > 18`
- `Order.total > 1000`

the result is two database queries (because Customer and Order are not logically related):

```
Select * from Customer where age > 18
```

```
Select * form Order where total > 1000
```

When the database filter contains more than one database entity/alias -- a compound filter -- it still acts as a single query. For example:

- `Order.bid >= Item.price`

which results in the query:

```
Select * from Order o,Item i where o.bid > i.price
```

When there are multiple filters related to one or more of the entities in a compound filter, they are combined with the `AND` operator For example, consider the filters:

- `Order.bid >= Item.price`
- `Order.status = 'VALID'`
- `Item.qty > 0`

which results in the query:

```
Select * from Order o,Item I where o.bid > i.price and o.status = "VALID"
and i.qty > 0
```

Test yourself - questions – Writing rules to access external data

Note: Try this test, and then go to **Answers:** Logical analysis & optimization on page 283 to correct yourself.

1. Rule scope determines which _____________ is processed during rule execution.
2. Why is root-level database access disabled by default?
3. When a Scope row is shown in bold text, what do we know about that entity's database access setting?
4. True or False. Only root-level entities can be extended to a database.

5. Which documents explain in more detail how the Direct Database Access feature works?
6. In general, does a rule author need to care about where actual data is stored, how it is retrieved, or how it is sent to the rules when creating Rule Sets?
7. Are there any exceptions to the general rule you defined in the question above?

10

Logical analysis and optimization

For details, see the following topics:

Testing, analysis, and optimization

Corticon Studio provides the rule modeler with tools to test, validate, and optimize rules and *Rulesheet*s prior to deployment. Before proceeding, let's define these terms.

Scenario testing

Scenario testing is the process of comparing *actual* decision operation to *expected* operation, using data scenarios or test cases. The Corticon Studio `Ruletest` provides the capability to build test cases using real data, which may then be "fed" to a set of rules for evaluation. The actual output produced by the rules may then be compared to the output we expect those rules to produce. If the actual output matches the expected output, then we may have *some* degree of confidence that the decision is performing properly. Why only *some* confidence and not *complete* confidence will be addressed over the course of this chapter.

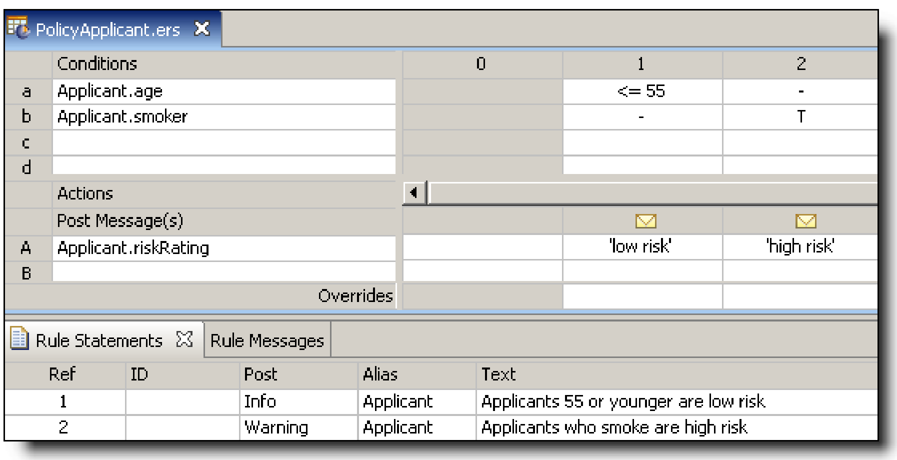
Rulesheet analysis & optimization

Analysis and optimization is the process of examining and correcting or improving the logical construction of `Rulesheet`s, *without* using test data. As with testing, the analysis process verifies that our rules are functioning correctly. Testing, however, does nothing to inform the rule builder about the execution efficiency of the `Rulesheet`s. Optimization of the rules ensures they execute most efficiently, and provide the best performance when deployed in production.

The following example illustrates the point:

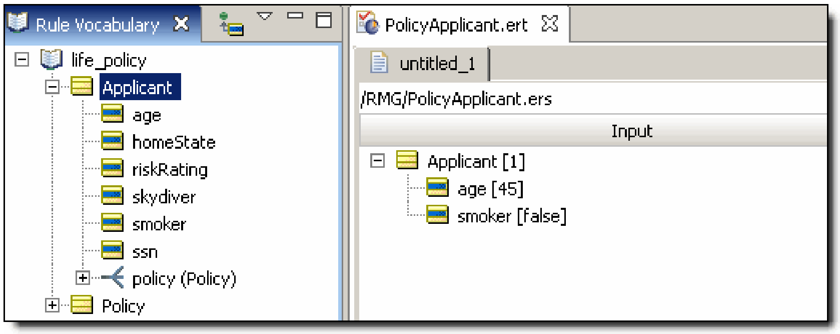
Two rules are implemented to profile life insurance policy applicants into two categories, high risk and low risk. These categories might be used later in a business process to determine policy premiums.

Figure 186: Simple Rules for Profiling Insurance Policy Applicants



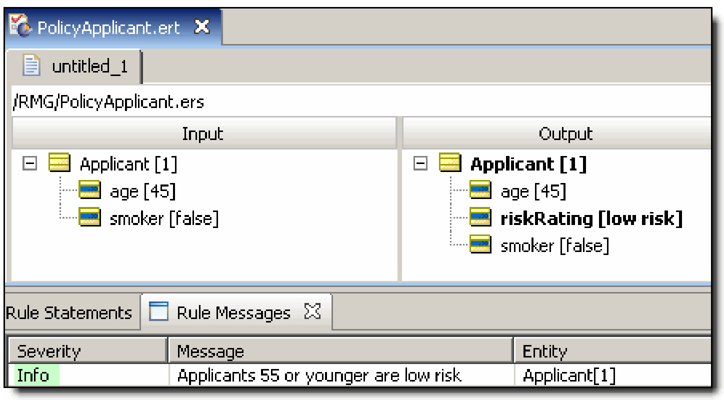
To test these rules, we create a new scenario in a `Ruletest`, as shown in A Test Scenario Created in a Ruletest.

Figure 187: A Test Scenario Created in a Ruletest



In this scenario, we have created a single example of `Person`, a non-smoker aged 45. Based on the rules we just created, we expect that the Condition in Rule 1 will be satisfied *(People aged 55 or younger…)* and that the person's `riskRating` will be assigned the value of `low`. To confirm our expectations, we run the `Ruletest` :

Figure 188: Ruletest



As we see in that figure, our expectations are confirmed: Rule 1 fires and `riskRating` is assigned the value of `low`. Furthermore, the `.post` command displays the appropriate rule statement. Based on this single scenario, can we say conclusively that these rules will operate properly for other possible scenarios; i.e., for all instances of `Person`? How do we answer this critical question?

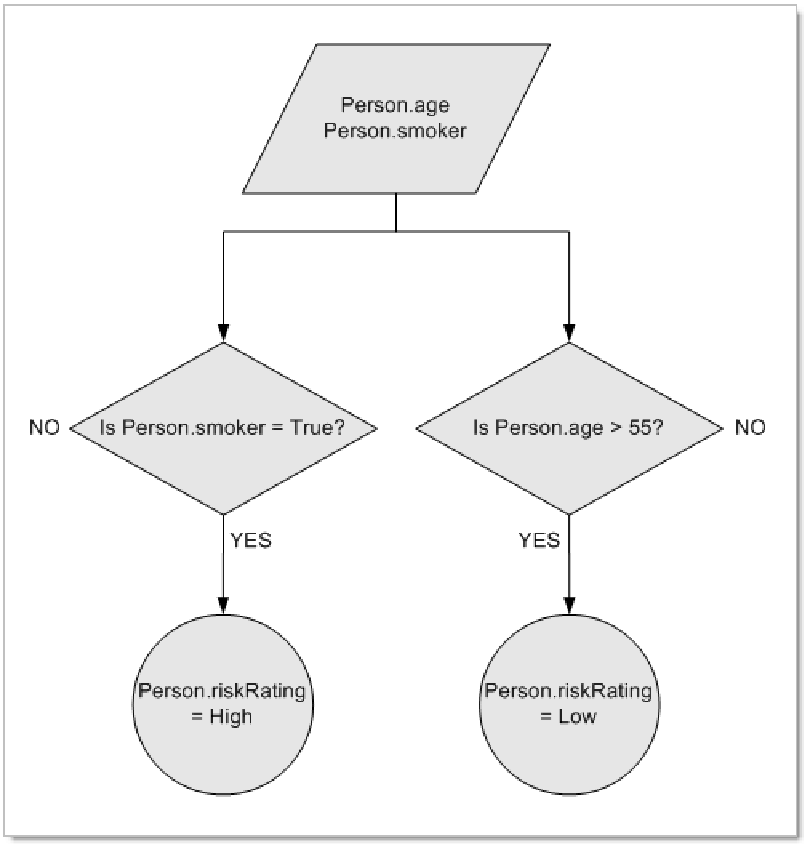
Traditional means of analyzing logic

The question of proper decision operation for all possible instances of data is fundamentally about analyzing the logic in each set of rules. Analyzing each individual rule is relatively easy, but business decisions are rarely composed of just a single rule. More commonly, a decision is composed of dozens or hundreds of rules, and the ways in which the rules interact can be very complex. Despite this complexity, there are several traditional methods for analyzing sets of rules to discover logical problems.

Flowcharts

A flowchart that captures these two rules might look like the following:

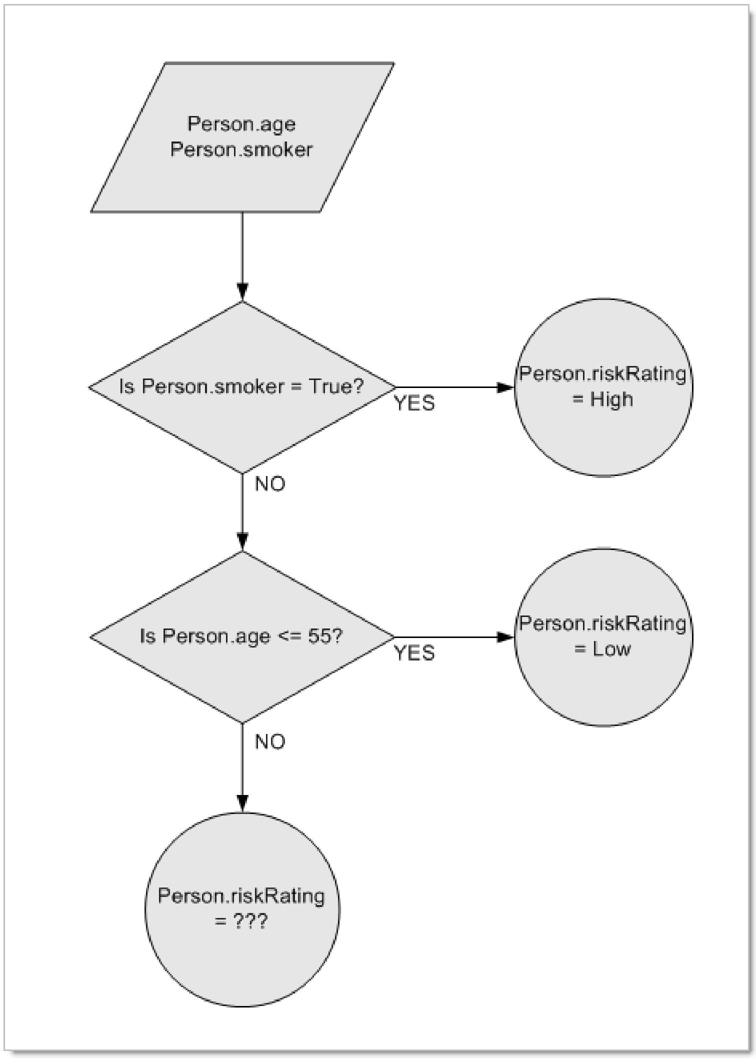
Figure 189: Flowchart with 2 Rules



Upon closer examination, the flowchart reveals two problems with our rules: what Action(s) should be taken if either test fails, in other words, if `Person.age>55` or if `Person.smoker=false`? The rules built in Simple Rules for Profiling Insurance Policy Applicants do not handle these two cases. But there is also a third, subtler problem here: what happens if **both** Conditions are satisfied, specifically when `Person.age<=55` **and** `Person.smoker=true`? When `Person.age<=55`, we want `Person.riskRating` to be given the value of `low`. But when `Person.smoker=true`, we want `Person.riskRating` to be given the value of `high`.

We have discovered a dependency between our rules – they are not truly separate and independent evaluations because they both assign a value to the same attribute. So the flowchart we began with turns out to be an incorrect graphical representation of our rules, because the decision flow does not truly follow two parallel and independent paths. Let's try a different flowchart:

Figure 190: Flowchart with 2 Dependent Rules



In the flowchart in Flowchart with 2 Dependent Rules, we have acknowledged an interdependence between the two rules, and have arranged them accordingly. However, a few questions still exist. For example, why did we choose to place the smoker rule *before* the age rule? By doing so we are giving the smoker rule an implicit priority over the age rule because any smoker will immediately be given a `riskRating` value of `High` regardless of what their `age` is. Is this what the business intends, or are we as modelers making unjustified assumptions?

We call this a problem of **logical conflict**, or **ambiguity** because it's simply not clear from our two rules, as they have been written, what the correct outcome should be. Does one rule take priority over the other? *Should* one rule take priority over the other? This is, of course, a business question, but the rule writer must be aware of the dependency problem and resulting conflict in order to ask the question in the first place. Also, notice that there is still no outcome for a non-smoker older than 55. We call this a problem of **logical completeness** and it must be taken into consideration, no matter which flowchart we use.

The point we are making is that discovery of logical problems in sets of rules using the flowcharting method is very difficult and tedious, especially as the number and complexity of rules in a decision (and the resulting flowcharts) grows.

Test databases

The use of a test database is another common method for testing rules (or any kind of business logic, for that matter). The idea is to build a large number of test cases, with carefully chosen data, and determine what the correct system response should be for each case.

Then, the test cases are processed by the logical system and output is generated. Finally, the *expected* output is compared to the *actual* output, and any differences are investigated as possible logical bugs.

Let's construct a very small test database with only a few test cases, determine our expected outcomes, then run the tests and compare the results. We want to ensure that our rules execute properly for all cases that might be encountered in a "real-life" production system. To do this, we must create a set of cases that includes **all** such possibilities.

In our simple example of two rules, this is a relatively straightforward task:

Table 5: Table: All Combinations of Conditions in Table Form

condition	Smoker (smoker = true)	Non-Smoker (smoker = false)
Age <= 55		
Age > 55		

In this table, we have assembled a matrix using the Values sets from each of the Conditions in our rules. By arranging one set of values in rows, and the other set in columns, we create the Cross Product (also known as the *direct product* or *cross product*) of the two Values sets, which means that every member of one set is paired with every member of the other set. Since each Values set has only two members, the Cross Product yields 4 distinct possible combinations of members (2 multiplied by 2). These combinations are represented by the *intersection* of each row and column in the table above. Now let's fill in the table using the expected outcomes from our rules.

Rule 1, the age rule, is represented by row 1 in the table above. Recall that rule 1 deals exclusively with the age of the applicant and is not impacted by the applicant's smoker value. To put it another way, the rule produces the same outcome *regardless* of whether the applicant's smoker value is `true` or `false`. Therefore, the action taken when rule 1 fires (`riskRating` is assigned the value of `low`) should be entered into both cells of row 1 in the table, as shown:

Figure 191: Rule 1 Expected Outcome

condition	Smoker (smoker = true)	Non-Smoker (smoker = false)
Age <= 55	**low**	**low**
Age > 55		

Likewise, rule 2, the smoker rule, is represented by column 1 in the table above, **All Combinations of Conditions in Table Form** . The action taken if rule 2 fires (`riskRating` is assigned the value of `high`) should be entered into both cells of column 1 as shown:

Figure 192: Rule 2 Expected Outcome

condition	Smoker (smoker = true)	Non-Smoker (smoker = false)
Age <= 55	**low, high**	**low**
Age > 55	**high**	

The table format illustrates the fact that a complete set of test data should contain four distinct cases (each cell corresponds to a case). Rearranging, our test cases and expected results can be summarized as follows:

Figure 193: Test Cases Extracted from Cross Product

Test case	age	smoker	Expected outcome
1	`<= 55`	`true`	`low, high`
2	`<= 55`	`false`	`low`
3	`> 55`	`true`	`high`
4	`> 55`	`false`	

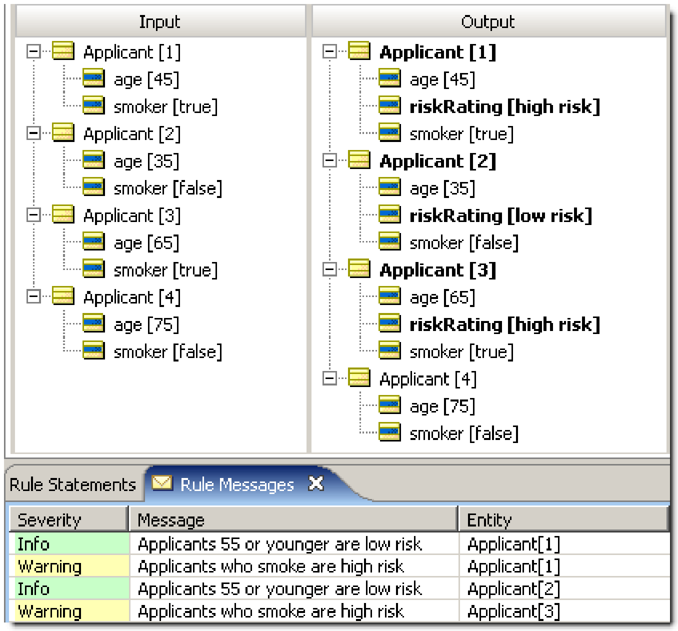
The table format also highlights two problems we encountered earlier with flowcharts. In the figure **Rule 2 Expected Outcome**, row 1 and column 1 intersect in the upper left cell (this cell corresponds to test case #1 in the figure above). As a result, each rule tries to assert its own action – one rule assigns a `low` value, and the other rule assigns a `high` value. Which rule is correct?

Logically speaking, they both are. But if the rule analyst had a *business* preference, it was certainly lost in the implementation. As before, we simply can't tell by the way the two rules are expressed. Logical conflict reveals itself once more.

Also notice the lower right cell (corresponding to test case #4) – it is empty. The combination of `age>55` **AND** non-smoker (`smoker=false`) produces no outcome because neither rule deals with this case – the logical incompleteness in our business rules reveals itself once more.

Before we deal with the logical problems discovered here, let's build a `Ruletest` in Studio that includes all four test cases in the figure above.

Figure 194: Inputs and Outputs of the 4 Test Cases



Let's look at the test case results in the figure above. Are they consistent with our expectations? With a minor exception in case #1, the answer is yes. In case #1, `riskRating` has been assigned the value of `high`. But also notice the rule statements posted: case #1 has produced two messages which indicate that both the age rule and the smoker rule fired as expected. But since `riskRating` can hold only one value, the system non-deterministically (at least from our perspective) assigned it the value of `high`.

So if using test cases works, what is wrong with using it as part of our Analysis methodology? Let's look at the assumptions and simplifications made in the previous example:

1. We are working with just two rules with two Conditions. Imagine a rule pattern comprising three Conditions – our simple 2-dimensional table expands into three dimensions. This may still not be too difficult to work with as some people are comfortable visualizing in three dimensions. But what about four or more? It is true that large, multi-dimensional tables can be "flattened" and represented in a 2-D table, but these become very large and awkward very quickly.
2. Each of our rules contains only a single Conditional parameter limited to only two values. Each also assigns, as its Action, a single parameter which is also limited to just two values.

When the number of rules and/or values becomes very large, as is typical with real-world business decisions, the size of the Cross Product rapidly becomes unmanageable. For example, a set of only six Conditions, each choosing from only ten values produces a Cross Product of 10^6, or one *million* combinations. Manually analyzing a million combinations for conflict and incompleteness is tedious and time-consuming, and still prone to human error.

In many cases, the potential set of cases is so large, that few project teams take the time to rigorously define all possibilities for testing. Instead, they often pull test cases from an actual database populated with real data. If this occurs, conflict and incompleteness may never be discovered during testing because it is unlikely that every possible combination will be covered by the test data.

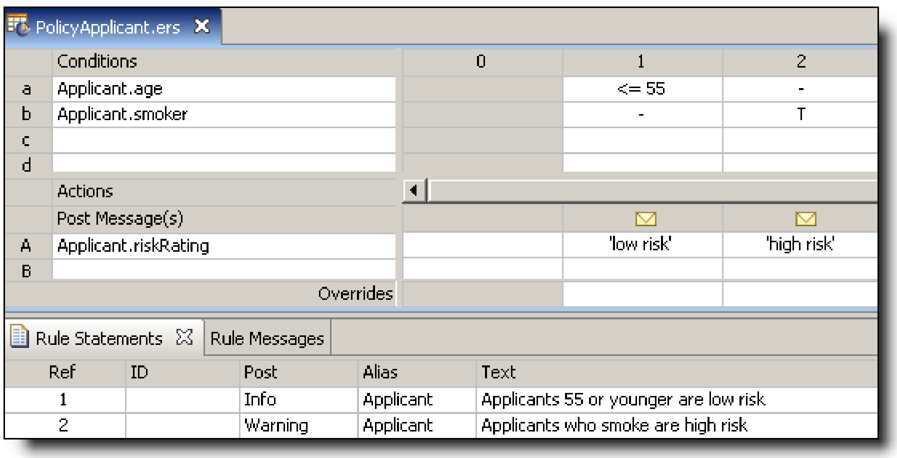
Using Corticon Studio to validate rulesheets

Now, having demonstrated how to test rules with real cases (as performed in Inputs and Outputs of the 4 Test Cases) as well as having discussed two manual methods for developing these test cases, it is time to demonstrate how Corticon Studio performs conflict and completeness checking automatically.

Expanding rules

Returning to our original rules (reproduced from Simple Rules for Profiling Insurance Policy Applicants):

Figure 195: Simple Rules for Profiling Insurance Policy Applicants

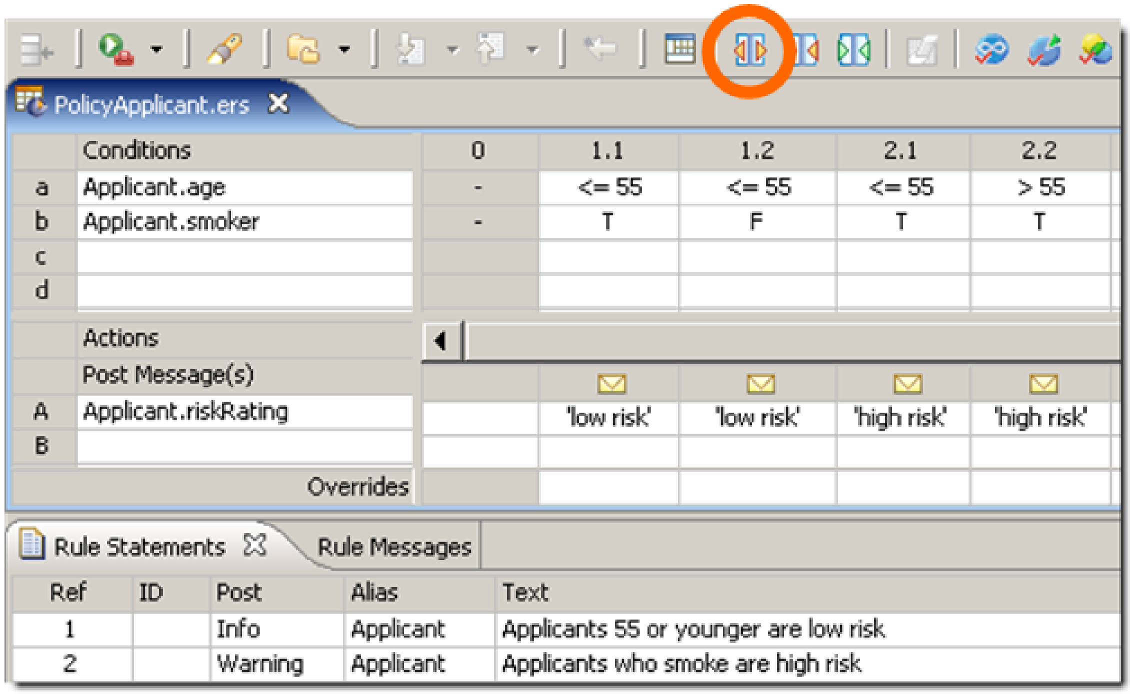


As illustrated by the table in Rule 1 Expected Outcome, rule 1 (the age rule) is really a combination of two *sub-rules*; we specified an age value for the first Condition but did not specify a smoker value for the second Condition. Because the smoker Condition has two possible values (`true` and `false`), the two sub-rules can be stated as follows:

1. Applicants aged 55 or younger **AND** who do not smoke are assigned a risk rating of low risk
2. Applicants aged 55 or younger **AND** who do smoke are assigned a risk rating of low risk

Corticon Studio makes it easy to view sub-rules for any or all columns in a Rulesheet. By clicking the **Expand Rules** button on the toolbar, or simply double-clicking the column header, Corticon Studio will display sub-rules for any selected column. If no columns are selected, then all sub-rules for all columns will be shown. Sub-rules are labeled using Decimal numbers: rule 1 below has two sub-rules labeled 1.1 and 1.2. Sub-rules 1.1 and 1.2 are equivalent to the upper left and upper right cells in Rule 1 Expected Outcome.

Figure 196: Expanding Rules to Reveal Components

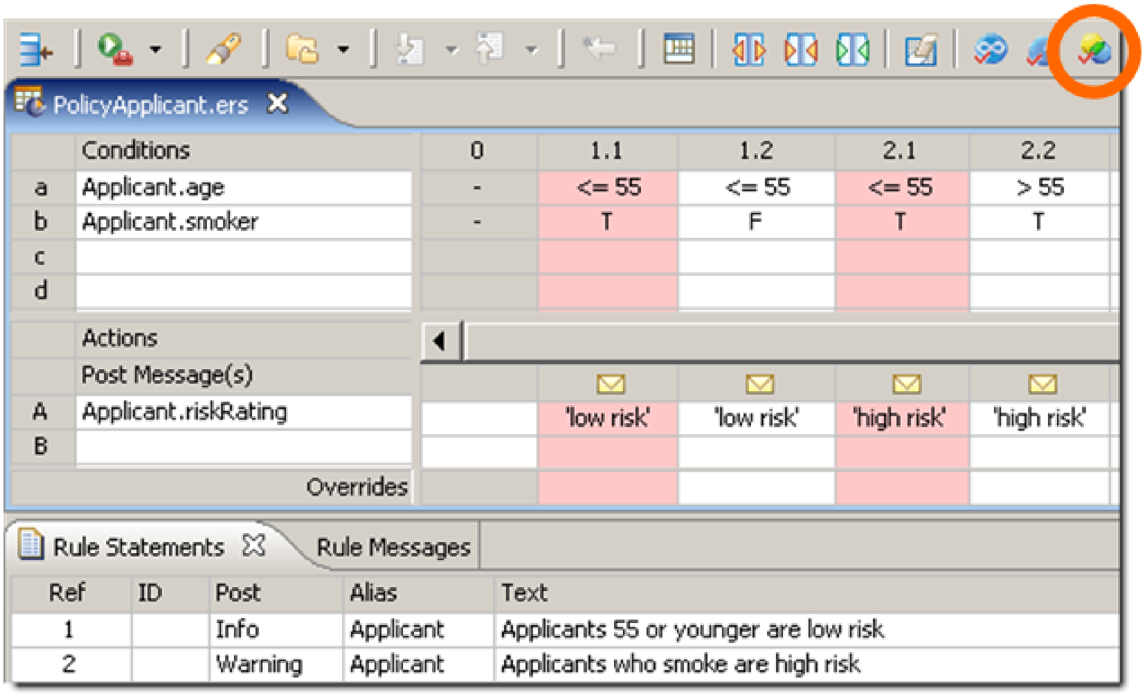


As we pointed out before, the outcome is the same for each sub-rule. Because of this, the sub-rules can be summarized as the general rules shown in column 1 of Simple Rules for Profiling Insurance Policy Applicants. We also say that the two sub-rules "collapse" into the rules shown in column 1. The 'dash' symbol in the smoker value of column 1 indicates that the actual value of smoker does not matter to the execution of the rule – it will assign `riskRating` the value of `low` no matter what the smoker value is (as long as `age <= 55`, satisfying the first Condition). Looking at it a different way, only those rules with dashes in their columns have sub-rules, one for each value in the complete value set determined for that Condition row.

The conflict checker

With our two rules expanded into four sub-rules as shown in Expanding Rules to Reveal Components), most of the Cross Product is displayed for us. Click the **Check for Conflicts**  button in the toolbar.

Figure 197: A Conflict Revealed by the Conflict Checker

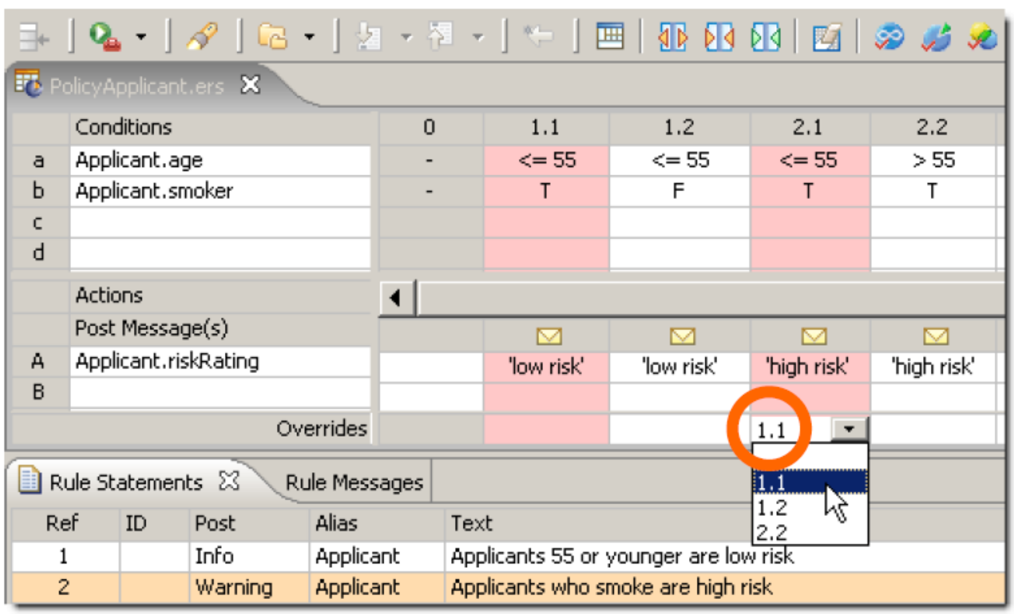


The mechanics of stepping through and resolving each conflict has been covered in detail in the *Corticon Studio - Basic Tutorial*. Here, our intent is to correlate the results of the automatic conflict check with the problems we identified first with the flowchart method, then later with test cases. Sub-rules 1.1 and 2.1, the sub-rules highlighted in pink and yellow in Figure 197 on page 215, correspond to the intersection of column 1 and row 1 of Rule 2 Expected Outcome or test case #1 in Test Cases Extracted from Cross Product. But note that Corticon Studio does not instruct the rule writer how to resolve the conflict – it simply alerts the rule writer to its presence. The rule writer, ideally someone who knows the business, must decide how to resolve the problem. The rule writer has two basic choices:

1. Change the Action(s) for one or both rules. We could change the Action in sub-rule 1.1 to match 2.1 or vice versa. Or we could introduce a new Action, say riskRating = medium, as the Action for both 1.1 and 2.1. If either method is used, the result will be that the Conditions and Actions of sub-rule 1.1 and 2.1 are *identical*. This removes the conflict, but introduces redundancy, which, while not a logical problem, can reduce processing performance in deployment. Removing redundancies in `Rulesheet`s is discussed in the Optimization section of this chapter.

2. Use an **Override**. Think of an override as an exception. To override one rule with another means to instruct the Corticon Server to fire *only one* rule even when the Conditions of both rules are satisfied. Another way to think about overrides is to refer back to the discussion surrounding the flowchart in Flowchart with 2 Dependent Rules. At the time, we were unclear which decision should execute first – no priority had been declared in our rules. But it made a big difference how we constructed our flowchart and what results it generated. To use an override here, we simply select the number of the sub-rule *to be overridden* from the drop-down box at the bottom of the column of the *overriding* sub-rule, as shown circled in the following figure. This is expressed simply as "sub-rule 2.1 overrides 1.1". It is incorrect to think of overrides as defining execution sequence. An override does not mean "fire rule 2.1 **then** fire rule 1.1." It means "fire rule 2.1 and **do not** fire rule 1.1".

Figure 198: Override Entered to Resolve Conflict



An override is essentially another business rule, which should to be expressed somewhere in the *Rule Statements* section of the Rulesheet. To express this override in plain English, the rule writer might choose to modify the rule statement for the *overridden* rule:

> 1. Applicants aged 55 or younger are assigned a low risk rating *unless* they smoke, in which case they are assigned a high risk rating.

This modification successfully expresses the effect of the override.

If ever in doubt as to whether you have successfully resolved a conflict, simply click the **Check for Conflicts** button again. The affected sub-rules should not highlight as you step through any remaining ambiguities. If all ambiguities have been resolved, you will see the following window:

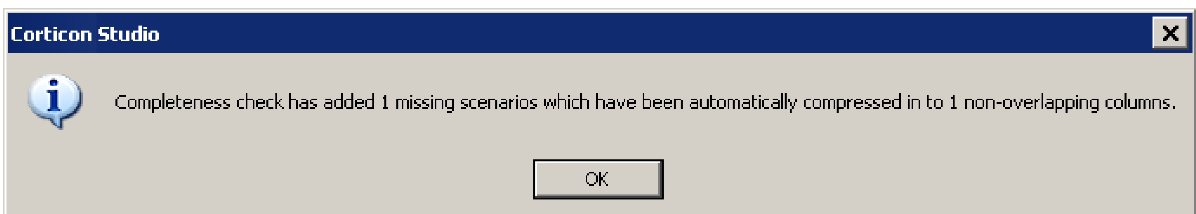
Figure 199: Conflict Resolution Complete

The completeness checker

While our rules are expanded, let's check for incompleteness. Again, the mechanics of this process are described in the *Corticon Studio tutorial: Basic Rule Modeling*. Our discussion here will be limited to correlating results with the previous manual methods of logical analysis.

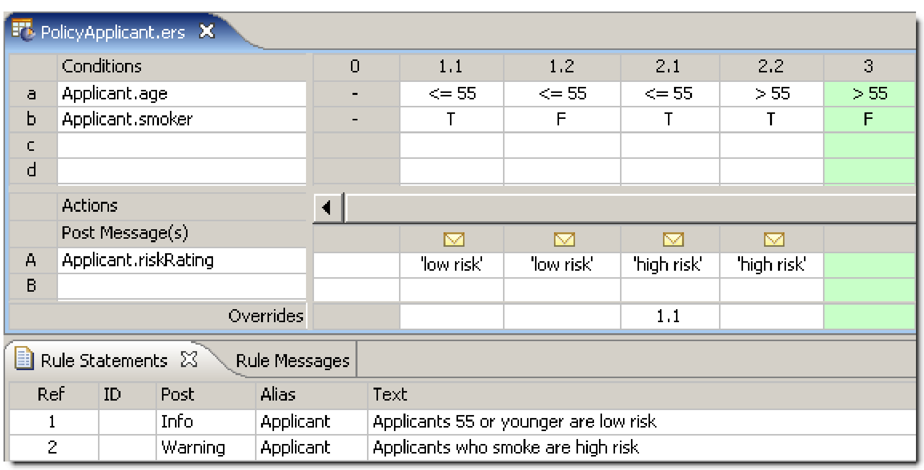
Clicking the **Check for Completeness** button, the message window is displayed:

Figure 200: Completeness Check Message Window



Clicking **OK** to dismiss the message window, we see that the Completeness Check has produced a new column (3), shaded in green:

Figure 201: New Rule Added by Completeness Check



This new rule, the combination of `age>55` **AND** `smoker=false` corresponds to the intersection of column 2 and row 2 in Rule 2 Expected Outcome and test case #4 in Test Cases Extracted from Cross Product. The Completeness Checker has discovered our missing rule! To do this, the Completeness Checker employs an algorithm which calculates all mathematical combinations of the Conditions' values (the Cross Product), and compares them to the combinations defined by the rule writer as other columns (other rules in the Rulesheet). If the comparison determines that some combinations are missing from the Rulesheet, these combinations are automatically added to the *`Rulesheet`*. As with the Conflict Check, the Action definitions of the new rules are left to the rule writer. The rule writer should also remember to enter new plain-language **Rule Statements** for the new columns so it is clear what logic is being modeled. The corresponding rule statement might look like this:

2.	An applicant older than 55 who does not smoke is profiled as medium risk

Automatically Determining the Complete Values Set

As values are manually entered into column cells in a Condition row, Corticon Studio automatically creates and updates a set of values, which for the given datatype of the Condition expression, is complete. This means that as you populate column cells, the list of values in the drop-down boxes you select from will grow and change.

In the drop-down box, you will see the list of values you have entered, plus null if the attribute or expression can have that value. But this list displayed in the drop-down is not the *complete* list – Corticon Studio maintains the complete list "under the covers" and only shows you the elements which you have manually inserted.

This automatically generated complete value list serves to feed the Completeness Checker with the information it needs to calculate the Cross Product and generate additional "green" columns. Without complete lists of possible values, the calculated Cross Product itself will be incomplete.

Automatic Compression of the New Columns

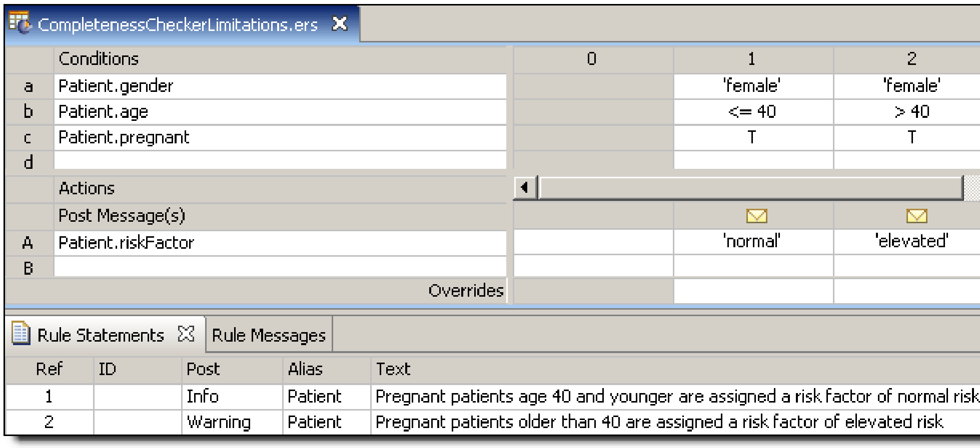
Another important aspect of the Completeness Checker's operation is the automatic compression it performs on the resulting set of missing Conditions. As we see from the message displayed in Completeness Check Message Window, the algorithm not only identifies the missing rules, but it also compresses them into *non-overlapping* columns. Two important points about this statement:

1. The compression performed by the Completeness Checker is a different kind of compression from that performed by the Compression Tool introduced in the Optimization section of this chapter. The optimized columns produced by the Completeness Check contain *no redundant sub-rules* (that's what non-overlapping means), whereas the Compression Tool will intentionally inject redundant sub-rules in order to create dashes wherever possible. This creates the optimal visual representation of the rules.
2. The compression performed here is designed to reduce the results set (which could be extremely large) into a manageable number while simultaneously introducing no ambiguities into the Rulesheet (which might arise due to redundant sub-rules being assigned different Actions).

Limitations of the Completeness Checker

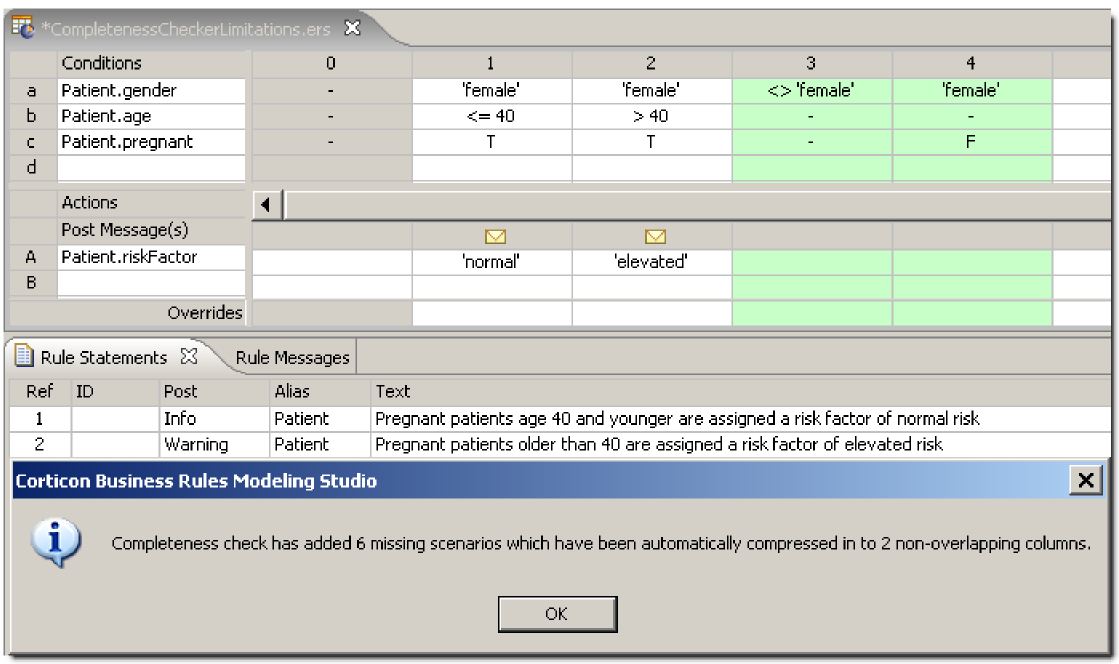
The Completeness Checker is powerful in its ability to discover missing combinations of Conditions from your `Rulesheet`. However, it is not smart enough to determine if these combinations make *business sense* or not. The example in the following figure shows two rules used in a health care scenario to screen for high-risk pregnancies:

Figure 202: Example Prior to Completeness Check



Now, we will click on the Completeness Checker:

Figure 203: Example after Completeness Check



Notice that columns 3-4 have been automatically added to the `Rulesheet`. But also notice that column 3 contains an unusual Condition: `gender <> female`. Because the other two Conditions in column 3 have dash values, we know it contains component or sub-rules. By double-clicking on column 3's header, its sub-rules are revealed:

Figure 204: Non-Female Sub-Rules Revealed

3.1	3.2	3.3	3.4
<> 'female'	<> 'female'	<> 'female'	<> 'female'
<= 40	<= 40	> 40	> 40
T	F	T	F

Because our *`Rulesheet`* is intended to identify high-risk pregnancies, it would not seem necessary to evaluate non-female (i.e., male) patients at all. And if male patients are evaluated, then we can say with some certainty that the scenarios described by sub-rules 3.1 and 3.3 – those scenarios containing pregnant males – are truly unnecessary. While these combinations may be members of the Cross Product, they are clearly not combinations that can occur in real life. If other rules in an application prevent combinations like this from occurring, then sub-rules 3.1 and 3.3 may also be unnecessary here. On the other hand, if no other rules catch this faulty combination earlier, then we may want to use this opportunity to raise an error message or take some other action that prompts a re-examination of the input data.

Renumbering Rules

Continuing with the previous pregnancy example, let's assume that we agree that sub-rules 3.1 and 3.3 are impossible, and so may be safely ignored. However, we decide to keep sub-rules 3.2 and 3.4 and assign Actions to them. For this example, we will just post violation messages to them.

However, when we try to enter Rule Statements for sub-rules 3.2 and 3.4, we discover that Rule Statements can only be entered for general rules (whole-numbered columns), not sub-rules. To convert column 3, with its four sub-rules, into four whole-numbered general rules, select **Rulesheet >Rule Column(s)>Renumber Rules** from the **Studio** menubar.

Figure 205: Sub-Rules Renumbered and Converted to General Rules

CompletenessCheckerLimitations.ers

	Conditions	0	1	2	3	4	5	6	7
a	Patient.gender	-	'female'	'female'	<> 'female'	<> 'female'	<> 'female'	<> 'female'	'female'
b	Patient.age	-	<= 40	> 40	<= 40	<= 40	> 40	> 40	-
c	Patient.pregnant	-	T	T	T	F	T	F	F
d									
	Actions								
	Post Message(s)								
A	Patient.riskFactor		'normal'	'elevated'					
B									
	Overrides								

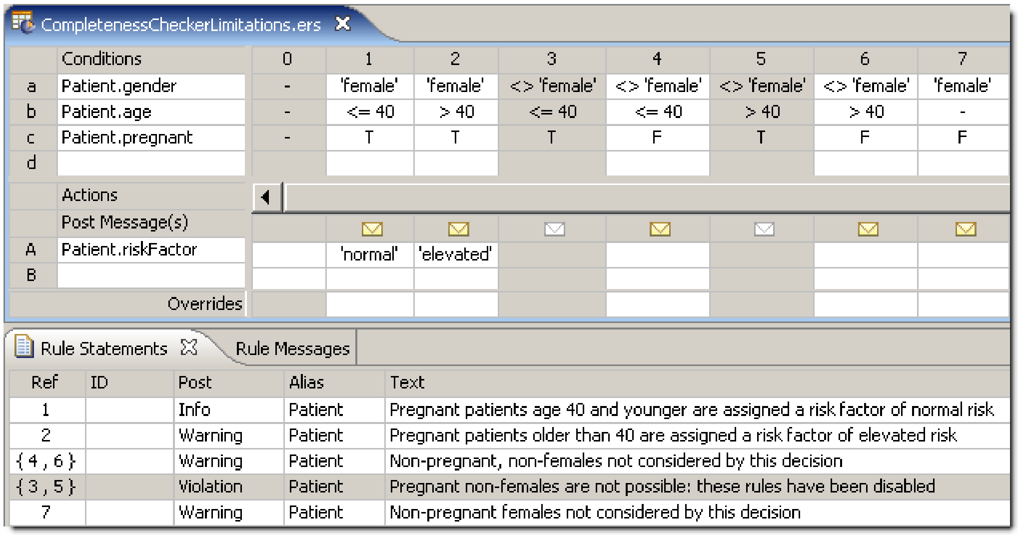
Rule Statements | Rule Messages

Ref	ID	Post	Alias	Text
1		Info	Patient	Pregnant patients age 40 and younger are assigned a risk factor of normal risk
2		Warning	Patient	Pregnant patients older than 40 are assigned a risk factor of elevated risk

Now that the columns have been renumbered, Rule Statements may be assigned to columns 4 and 6, and columns 3 and 5 can be deleted or disabled (if you want to do so).

When impossible or useless rules are created by the Completeness Checker, we recommend disabling the rule columns rather than deleting them. When disabled, the columns remain visible to all modelers, eliminating any surprise (and shock) when future modelers apply the Completeness Check and discover missing rules that you had already found and deleted. And if you disable the columns, be sure to include a Rule Statement that explains why. See the following figure for an example of a fully complete and well-documented `Rulesheet`

Figure 206: Final Rulesheet with impossible rules disabled



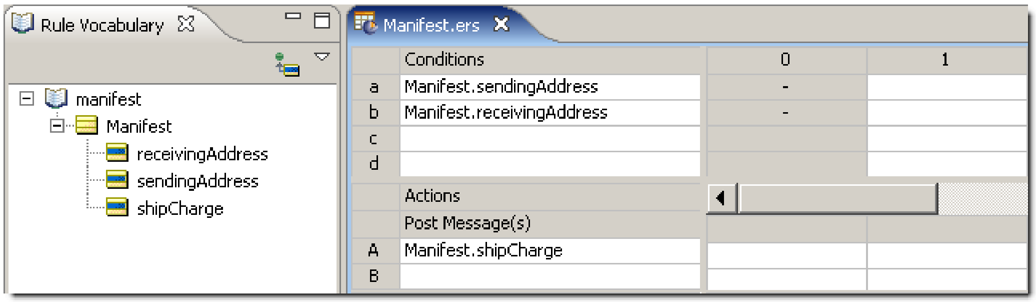
Letting the expansion tool work for you: tabular rules

Business rules, especially those found in operational manuals or procedures, often take the form of tables. Take for example the following table that generates shipping charges between two geographic zones:

Matrix to Calculate Shipping Charges per Pound					
From/To	**zone 1**	**zone 2**	**zone 3**	**zone 4**	**zone 5**
zone 1	`$0.25`	`$0.35`	`$0.45`	`$0.55`	`$0.65`
zone 2	`$0.35`	`$0.25`	`$0.35`	`$0.45`	`$0.55`
zone 3	`$0.45`	`$0.35`	`$0.25`	`$0.35`	`$0.45`
zone 4	`$0.55`	`$0.45`	`$0.35`	`$0.25`	`$0.35`
zone 5	`$0.65`	`$0.55`	`$0.45`	`$0.35`	`$0.25`

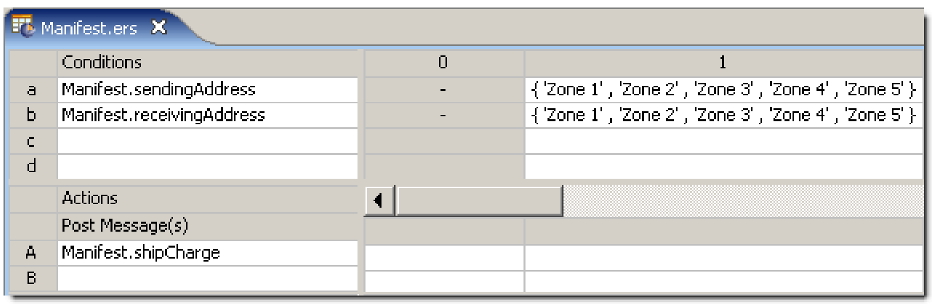
In the following figure, we have built a simple Vocabulary with which to implement these rules. Because each cell in the table represents a single rule, our `Rulesheet` will contain 25 columns (the Cross Product equals 5x5 or 25).

Figure 207: Vocabulary and Rulesheet to Implement Matrix



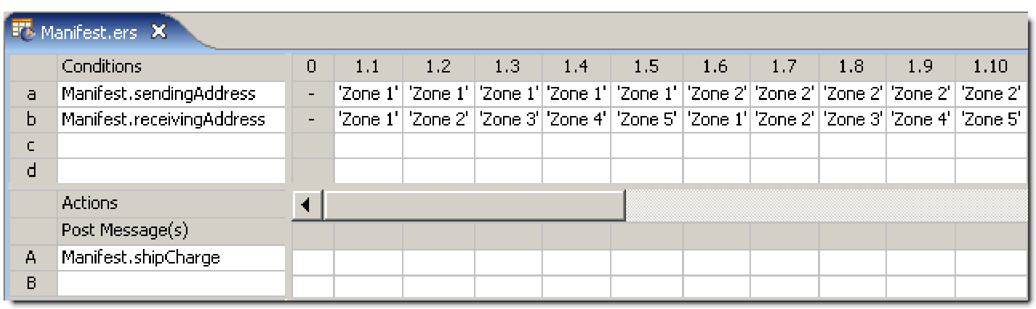
Rather than manually create all 25 combinations (and risk making a mistake), you can use the Expansion Tool to help you do it. This is a three-step process. Step 1 consists of entering the full range of values found in the table in the Conditions cells, as shown:

Figure 208: Rulesheet with Conditions Automatically Populated



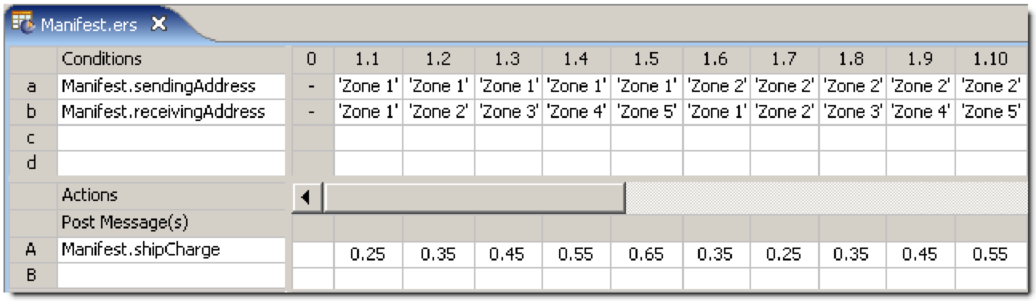
Now, use the Expansion Tool to expand column 1 into 25 non-overlapping columns. We now see the 25 sub-rules of column 1 (only the first ten sub-rules are shown in the folowing figure due to page width limitations in this document):

Figure 209: Rule 1 Expanded to Show Sub-Rules



Each sub-rule represents a single cell in the original table. Now, select the appropriate value of `shipCharge` in the **Actions** section of each sub-rule as shown:

Figure 210: Rulesheet with Actions Populated

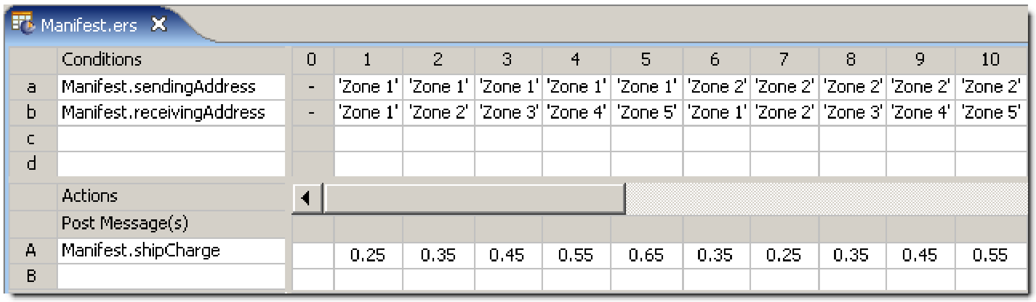


Manifest.ers

	Conditions	0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10
a	Manifest.sendingAddress	-	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 2'	'Zone 2'	'Zone 2'	'Zone 2'	'Zone 2'
b	Manifest.receivingAddress	-	'Zone 1'	'Zone 2'	'Zone 3'	'Zone 4'	'Zone 5'	'Zone 1'	'Zone 2'	'Zone 3'	'Zone 4'	'Zone 5'
c												
d												
	Actions											
	Post Message(s)											
A	Manifest.shipCharge		0.25	0.35	0.45	0.55	0.65	0.35	0.25	0.35	0.45	0.55
B												

In step 3, shown in the following figure, we renumber the sub-rules to arrive at the final Rulesheet with 25 general rules, each of which may now be assigned a Rule Statement.

Figure 211: Rulesheet with Renumbered Rules



Manifest.ers

	Conditions	0	1	2	3	4	5	6	7	8	9	10
a	Manifest.sendingAddress	-	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 2'	'Zone 2'	'Zone 2'	'Zone 2'	'Zone 2'
b	Manifest.receivingAddress	-	'Zone 1'	'Zone 2'	'Zone 3'	'Zone 4'	'Zone 5'	'Zone 1'	'Zone 2'	'Zone 3'	'Zone 4'	'Zone 5'
c												
d												
	Actions											
	Post Message(s)											
A	Manifest.shipCharge		0.25	0.35	0.45	0.55	0.65	0.35	0.25	0.35	0.45	0.55
B												

We will revisit this example in the Optimization section.

Memory management

As you might suspect, the Completeness Checker and Expansion algorithms are memory-intensive, especially as Rulesheets become very large. If Corticon Studio runs low on memory, refer to the *Corticon Studio Installation Guide*, "Changing Corticon Studio Memory Allocation" section, for details on increasing Corticon Studio's memory allotment.

Logical loop detection

Corticon Studio has the ability to both detect and control rule looping. This is important because loops are sometimes inadvertently created during rule implementation. Other times, looping is intentionally introduced to accomplish specific purposes. Both scenarios are discussed in the chapter Rule Dependency: Chaining and Looping.

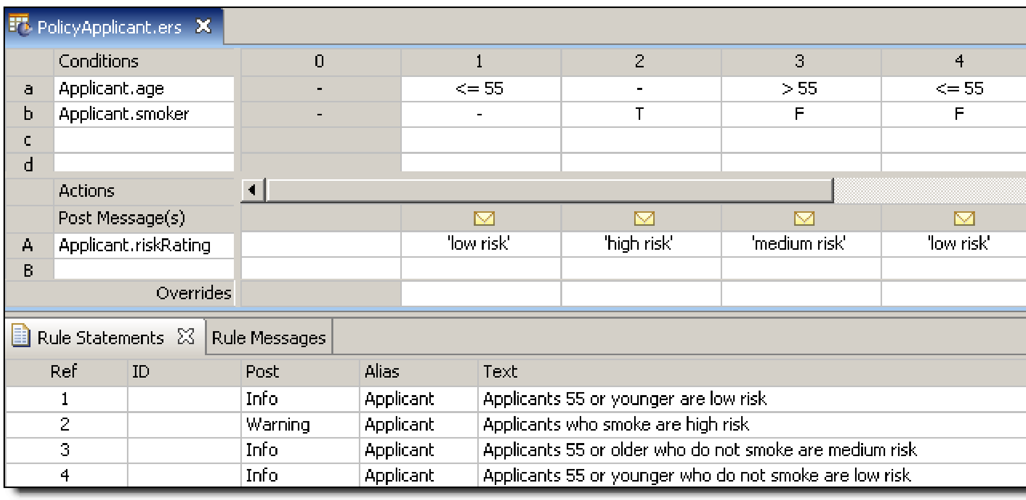
Optimizing rulesheets

The compress tool

Corticon Studio helps improve performance by removing redundancies within *Rulesheet*s. There are two types of redundancies the **Compress Tool** detects and removes:

1. Rule or sub-rule duplication. The Compress Tool will search a *Rulesheet* for duplicate columns (including sub-rules that may not be visible unless the rule columns are expanded), and delete extra copies. Picking up where we left off in New Rule Added by Completeness Check, let's add another rule (column #4), as shown in the following figure:

Figure 212: New Rule (#4) Added



PolicyApplicant.ers

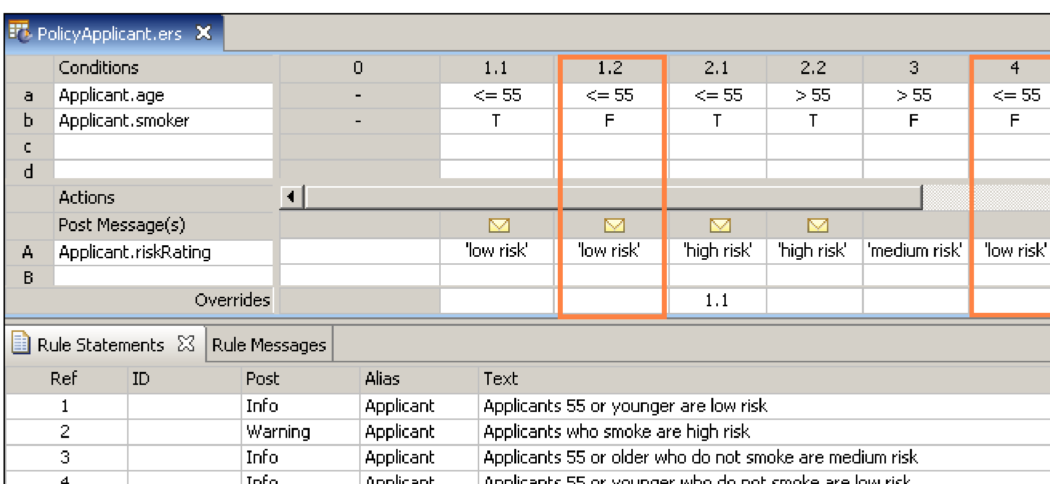
	Conditions	0	1	2	3	4
a	Applicant.age	-	<= 55	-	> 55	<= 55
b	Applicant.smoker	-	-	T	F	F
c						
d						
	Actions					
	Post Message(s)					
A	Applicant.riskRating		'low risk'	'high risk'	'medium risk'	'low risk'
B						
	Overrides					

Rule Statements | Rule Messages

Ref	ID	Post	Alias	Text
1		Info	Applicant	Applicants 55 or younger are low risk
2		Warning	Applicant	Applicants who smoke are high risk
3		Info	Applicant	Applicants 55 or older who do not smoke are medium risk
4		Info	Applicant	Applicants 55 or younger who do not smoke are low risk

While these 4 rules use only 2 Conditions and take just 2 Actions (an assignment to `riskRating` and a posted message), they already contain a redundancy problem. Using the **Expand Tool** this redundancy is visible in the following figure:

Figure 213: Redundancy Problem Exposed



PolicyApplicant.ers

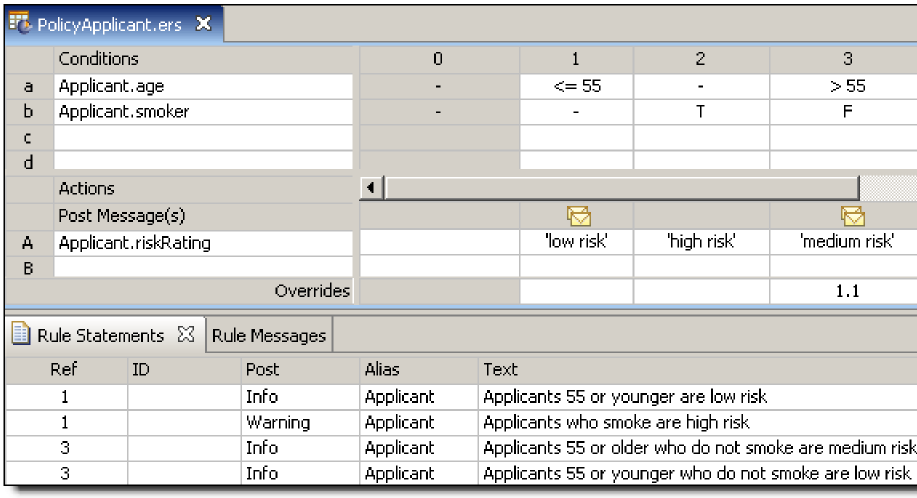
	Conditions	0	1.1	1.2	2.1	2.2	3	4
a	Applicant.age	-	<= 55	<= 55	<= 55	> 55	> 55	<= 55
b	Applicant.smoker	-	T	F	T	T	F	F
c								
d								
	Actions							
	Post Message(s)							
A	Applicant.riskRating		'low risk'	'low risk'	'high risk'	'high risk'	'medium risk'	'low risk'
B								
	Overrides				1.1			

Rule Statements | Rule Messages

Ref	ID	Post	Alias	Text
1		Info	Applicant	Applicants 55 or younger are low risk
2		Warning	Applicant	Applicants who smoke are high risk
3		Info	Applicant	Applicants 55 or older who do not smoke are medium risk
4		Info	Applicant	Applicants 55 or younger who do not smoke are low risk

Clicking on the **Compress Tool** has the effect shown in the following figure:

Figure 214: Rulesheet After Compression



	Conditions	0	1	2	3
a	Applicant.age	-	<= 55	-	> 55
b	Applicant.smoker	-	-	T	F
c					
d					
	Actions				
	Post Message(s)				
A	Applicant.riskRating		'low risk'	'high risk'	'medium risk'
B					
	Overrides				1.1

Ref	ID	Post	Alias	Text
1		Info	Applicant	Applicants 55 or younger are low risk
1		Warning	Applicant	Applicants who smoke are high risk
3		Info	Applicant	Applicants 55 or older who do not smoke are medium risk
3		Info	Applicant	Applicants 55 or younger who do not smoke are low risk

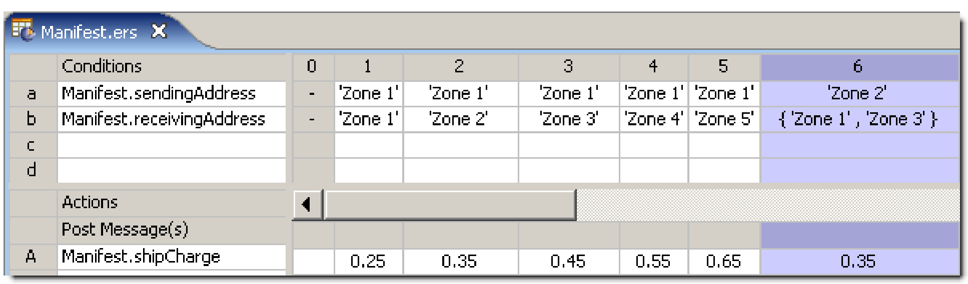
Looking at the compressed `Rulesheet` in this figure, we see that column #4 has disappeared entirely. More accurately, the Compress Tool determined that column 4 was a duplicate of one of the sub-rules in column 1 (1.2) and simply removed it. Looking at the Rule Statement section, we see that the rule statement for rule 4 has been renumbered to match the surviving rule.

Compression does not, however, alter the *text* of the rule statement; that task is left to the rule writer.

It is important to note that the compression does not alter the `Rulesheet`'s logic; it simply affects the way the rules **appear** in the `Rulesheet` – the number of columns, Values sets in the columns, and such. Compression also streamlines rule execution by ensuring that no rules are processed more than necessary.

2. Combining Values sets to simplify and shorten `Rulesheet`s. Recall our shipping charge example. By using the Compress Tool, `Rulesheet` columns are combined wherever possible by creating Values sets in Condition cells. For example, rule 6 in the figure **Compressed Shipping Charge Rulesheet** (highlighted below) is the combination of rule 6 and 8 from Rulesheet with Renumbered Rules.

Figure 215: Compressed Shipping Charge Rulesheet



	Conditions	0	1	2	3	4	5	6
a	Manifest.sendingAddress	-	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 2'
b	Manifest.receivingAddress	-	'Zone 1'	'Zone 2'	'Zone 3'	'Zone 4'	'Zone 5'	{ 'Zone 1' , 'Zone 3' }
c								
d								
	Actions							
	Post Message(s)							
A	Manifest.shipCharge		0.25	0.35	0.45	0.55	0.65	0.35

Value sets in Condition cells are equivalent to the logical operator **OR**. Rule 6 therefore reads:

> 6. A manifest with a Zone 2 sending address **AND** a Zone 1 **OR** Zone 3 receiving address costs $0.35 per pound to ship.

In deployment, The `Server` will execute this new rule 6 faster than the previous rule 6 and 8 together.

Producing characteristic rulesheet patterns

Because Corticon Studio is a visual environment, patterns often appear in the *Rulesheet* that provide insight into the decision logic. Once a rule writer recognizes and understands what these patterns mean, he or she can often accelerate rule modeling in the *Rulesheet*. The Compression Tool is designed to reproduce *Rulesheet* patterns in some common cases.

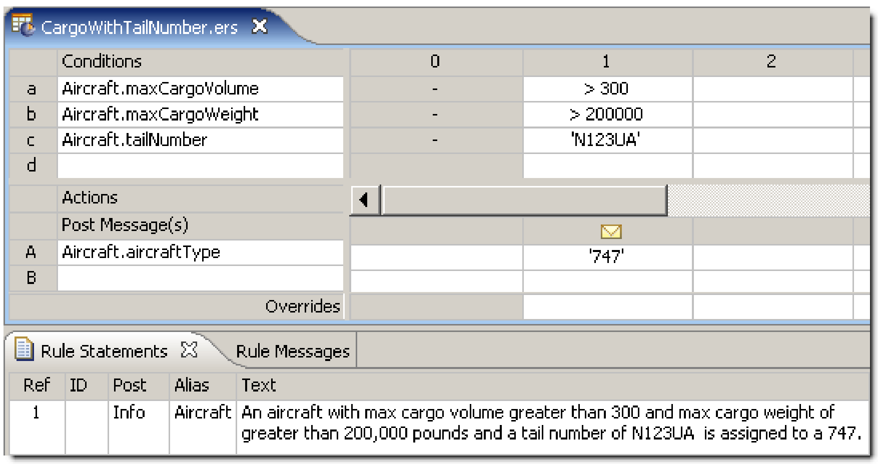
For example, take the following rule statement:

1. An aircraft with max cargo volume greater than 300 **AND** max cargo weight gre 200,000 **AND** tail number of N123UA must be a 747.

2. Otherwise it must be a DC-10.

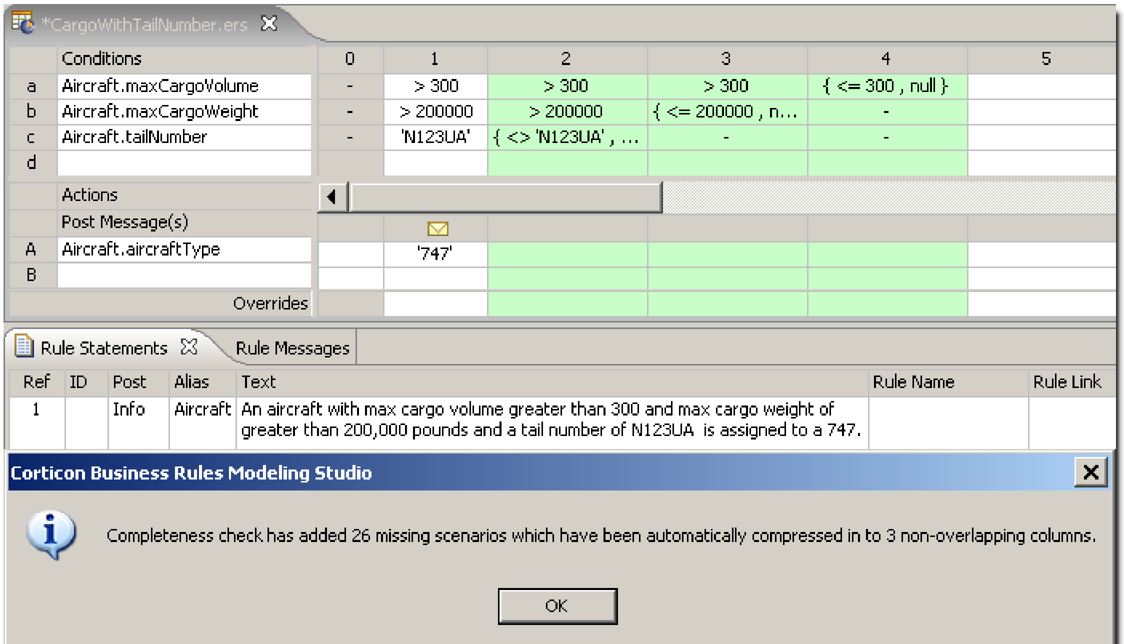
Applying some of the techniques from this manual, we might implement rule 1 as:

Figure 216: Implementing the 747 Rule



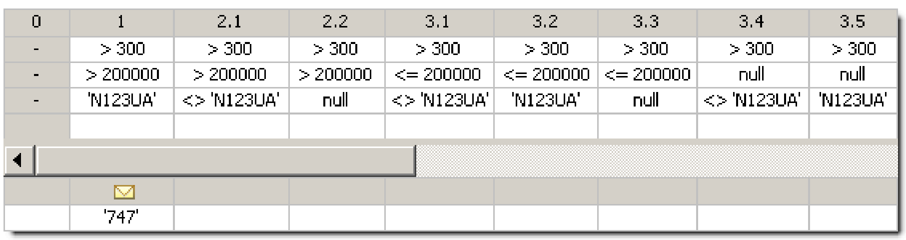
Now, letting the Completeness Checker populate the missing columns:

Figure 217: Remaining Columns Produced by the Completeness Checker



To remind you of the underlying Cross Product used by the Completeness Checker, we will **Expand** the `Rulesheet` momentarily and examine the sub-rules present:

Figure 218: Underlying Sub-Rules Produced by the Completeness Checker

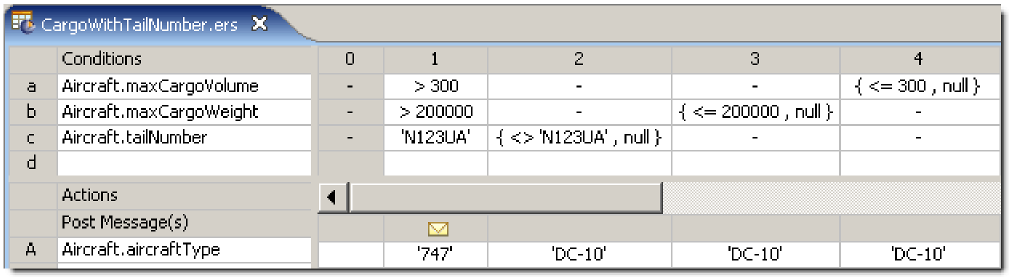


A total of 26 new columns (counting both rules and sub-rules) have been created – exactly what we expect and what the **Completeness Check** message window states.

Note: Three Conditions each with three members in their Values sets yields a Cross Product of 27 combinations (3*3*3 or 3 cubed). Subtracting the combination already present in column 1, we expect 26 new columns to be added.

Now, **Compress** the `Rulesheet` and fill in the Actions for the new columns as shown in Missing Rules with Actions Assigned:

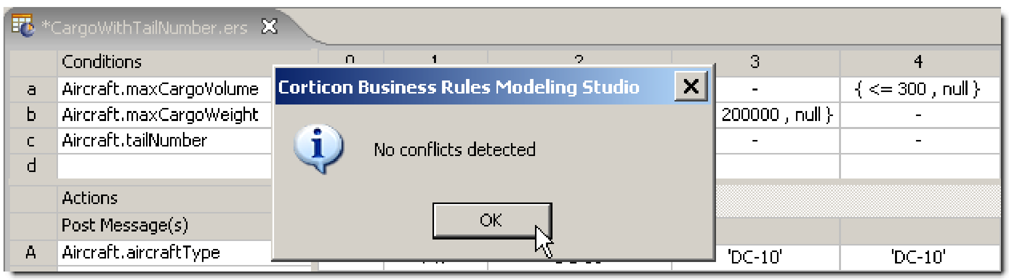
Figure 219: Missing Rules with Actions Assigned



	Conditions	0	1	2	3	4
a	Aircraft.maxCargoVolume	-	> 300	-	-	{ <= 300 , null }
b	Aircraft.maxCargoWeight	-	> 200000	-	{ <= 200000 , null }	-
c	Aircraft.tailNumber	-	'N123UA'	{ <> 'N123UA' , null }	-	-
d						
	Actions					
	Post Message(s)					
A	Aircraft.aircraftType		'747'	'DC-10'	'DC-10'	'DC-10'

Because the added rules are non-overlapping, we can be sure they won't introduce any ambiguities into the `Rulesheet`. To prove this, select the **Conflict Checker**

Figure 220: Proof that no New Conflicts have been Introduced by the Completeness Check



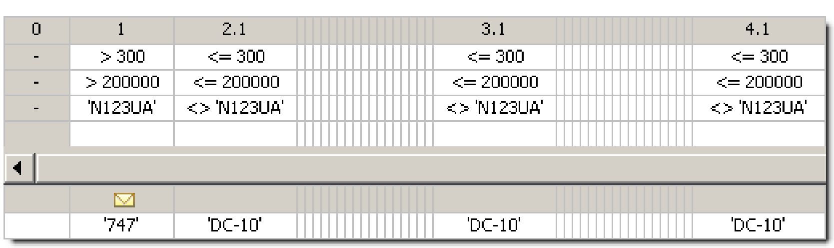
This pattern tells us that the only case where the aircraft type is a 747 is when max cargo volume is greater than 300 **AND** max cargo weight is greater than 200,000 **AND** tail number is N123UA. This rule is expressed in column 1. In all other cases, specifically where max cargo volume is 300 or less **OR** max cargo weight is 200,000 or less **OR** tail number is something other than N123UA will the aircraft type be a DC-10 (or any of the values are null). These rules are expressed in columns 2, 3 and 4, respectively.

The characteristic diagonal line of Condition values in columns 2-4, surrounded by dashes indicates a classic **OR** relationship between the 3 Conditions in these columns. The Compression algorithm was designed to produce this characteristic pattern whenever the underlying rule logic is present. It helps the rule writer to better "see" how the rules relate to each other.

Compression creates sub-rule redundancy

Compressing our example into a recognizable pattern, however, has an interesting side-effect - we have also introduced more sub-rules than were initially present. To see this, simply **Expand** the `Rulesheet` as shown:

Figure 221: Expanding Rules *Following* Compression



0	1	2.1	3.1	4.1
-	> 300	<= 300	<= 300	<= 300
-	> 200000	<= 200000	<= 200000	<= 200000
-	'N123UA'	<> 'N123UA'	<> 'N123UA'	<> 'N123UA'
	'747'	'DC-10'	'DC-10'	'DC-10'

You may be surprised to see a total of 54 sub-rules (columns) displayed (in the figure above) instead of the 26 we had prior to compression. Look closely at the 54 columns and you will see several instances of sub-rule redundancy – of the 18 sub-rules within original columns 2, 3 and 4, almost half are redundant (for example, sub-rules 2.1, 3.1 and 4.1, shown in the figure above, are identical). What happened?

Effect of compression on Corticon Server performance

Why does Corticon Studio have what amounts to two different kinds of compression – one performed by the Completeness Checker and another performed by the Compression Tool? It is because each has a different role during the rule modeling process. The type of compression performed during a Completeness Check is designed to reduce a (potentially) very large set of missing rules into the smallest possible set of non-overlapping columns. This allows the rule writer to assign Actions to the missing rules without worrying about accidentally introducing ambiguities.

On the other hand, the compression performed by the Compression Tool is designed to reduce the number of rules into the smallest set of general rules (columns with dashes), even if the total number of sub-rules is larger than that produced by the Completeness Checker. This is important for three reasons:

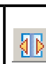
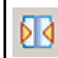
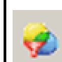
1. The Compression Tool preserves or reproduces key patterns familiar and meaningful to the rule modeler
2. The Compression Tool, by reducing a `Rulesheet` to the smallest number of columns, optimizes the executable code produced by Corticon Server's on-the-fly compiler. Smaller `Rulesheet`s (lower column count) result in faster Corticon Server performance.
3. The Compression Tool, by reducing columns to their most general state (the most dashes), improves Corticon Server performance by allowing it to ignore all Conditions with dash values. This means that when the rule in column 3 of Missing Rules with Actions Assigned is evaluated by Corticon Server, only the max cargo weight Condition is considered – the other two Conditions are ignored entirely because they contain dash values. When rule 3 of Missing Rules with Actions Assigned is evaluated after the **Completeness Check** is applied but *before* the **Compression Tool**, however, both max cargo weight and volume Conditions are considered, which takes slightly more time. So even though both `Rulesheet`s have the same number of columns (four), the `Rulesheet` with more generalized rules (more dashes - Missing Rules with Actions Assigned) will execute faster because the engine is doing less work.

Test yourself - questions – logical analysis and optimization

Note: Try this test, and then go to **Answers:** Logical analysis & optimization on page 283 to correct yourself.

1. What does it mean for two rules to be ambiguous?
2. What does it mean for a `Rulesheet` to be complete?
3. Are all ambiguous rules wrong, and must all ambiguities be resolved before deployment? Why or why not?

4. Are all incomplete `Rulesheet`s wrong, and must all incompletenesses be resolved before deployment? Why or why not?
5. Match the Corticon Studio tool name with its toolbar icon

Conflict Checker	
Compression Tool	
Expansion Tool	
Collapse Tool	
Conflict Filter	
Completeness Checker	

6. Explain the different ways in which an Ambiguity/Conflict between two rules can be resolved.
7. True or False. Defining an override enforces a specific execution sequence of the two ambiguous rules
8. True or False. A Conditions row with an incomplete values set will always result in an incomplete Rulesheet.
9. If a Rulesheet is incomplete due to an incomplete values set, will the Completeness Checker detect the problem? Why or why not?
10. Can a rule column define more than one override?
11. If rule 1 overrides rule 2, and rule 2 overrides rule 3, does rule 1 automatically override rule 3?
12. Are rules created by the Completeness Checker always legitimate?
13. In a rule column, what does a dash (-) character mean?
14. True or False. The Expansion Tool permanently changes the rule models in a `Rulesheet`. If false, how can it be reversed?
15. True or False. The Compression Tool permanently changes the rule models in a `Rulesheet`. If false, how can it be reversed?
16. If a rule has 3 condition rows, and each condition row has a Values set with 4 elements, what is the size of the Cross Product?
17. In above question, is it necessary to assign actions for every set of conditions (i.e., for every column)?
18. If you do not want to assign actions for every column, what can be done to/with these columns?
19. Which Corticon Studio tool helps to improve `Rulesheet` performance?

Expansion Tool	Compression Tool	Completeness Checker	Collapse Tool	Squeeze Tool

20. How is the compression performed by the Completeness Checker different from that performed by the Compression Tool?

21. What's wrong with using databases of test data to discover `Rulesheet` incompleteness?
22. If you expand a rule column and change the Actions for one of the sub-rules, what will Corticon Studio force you to do before saving the changes?
23. What does it mean for one rule to subsume another?

11

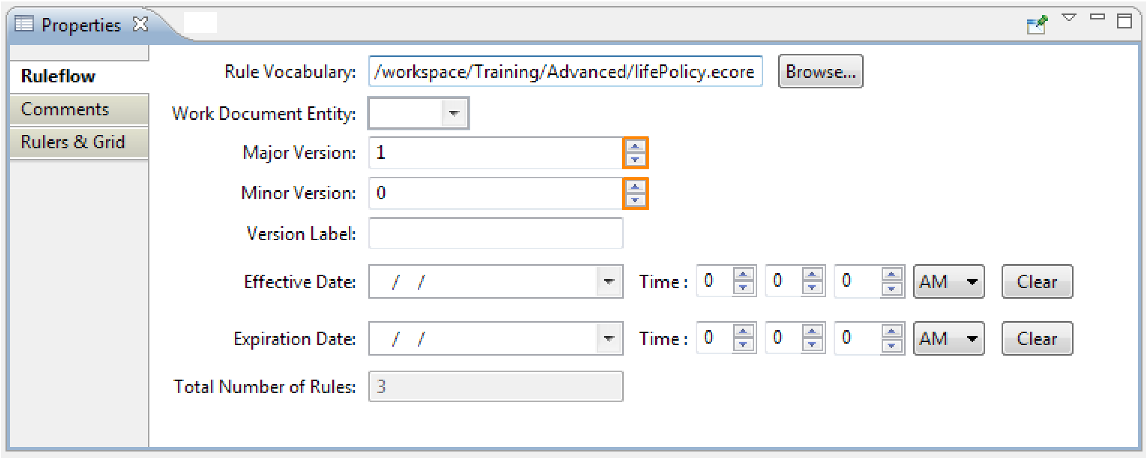
Ruleflow versioning & effective dating

For details, see the following topics:

Setting a Ruleflow version

Major and minor version numbers for `Ruleflow`s are optional. They can be assigned by selecting the menu command **Ruleflow >Properties**, and then clicking on the buttons on the **Major Version** and **Minor Version** lines, as highlighted:

Figure 222: Assigning a Version Number to a Ruleflow



When you use different Version numbers to describe identically named Ruleflows, the Corticon Server keeps them distinguished in its memory, so it can respond correctly to requests for a specified version. In other words, an application or process can use (or "call") different versions of the same Ruleflow depending on certain criteria. The details of how this works at the Server level are discusssed in the topics at *"Decision Service versioning and effective dating" in the Integration & Deployment Guide*.

A plain-text description of this version can be added in the **Comments** tab. Version numbers can be set higher at anytime, but cannot be set lower.

Major and minor versions

Minor and Major version designations are arbitrary and may be adapted to fit the version naming conventions used in different environments. As an example, `Ruleflow` minor versions may be incremented whenever a component Rulesheet is modified. Major `Ruleflow` versions may be incremented when more substantial changes are made to it, such as adding, replacing, or removing a `Rulesheet` from the `Ruleflow`.

Verison numbers can incremented, but not decremented.

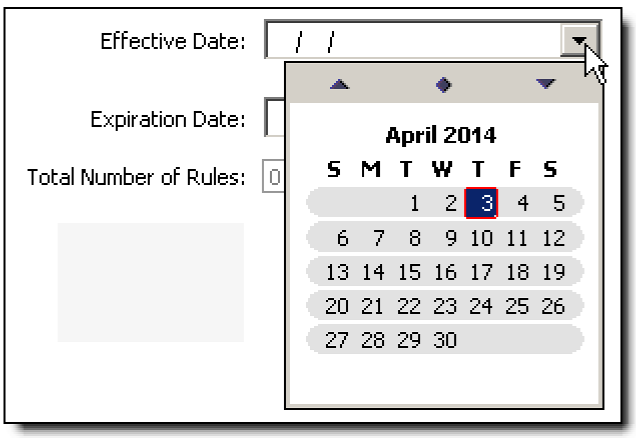
The details of how to invoke a `Ruleflow` by version number, see the topic *"Decision Service versioning and effective dating" in the Integration & Deployment Guide*.

Setting effective and expiration dates

Effective and Expiration dateTimes are optional for `Ruleflow`s and can be assigned singly or in pairs. When we use different Effective and Expiration dateTimes to describe identically named `Ruleflow`s, the `Server` keeps them straight in memory, and responds correctly to requests for the different dates. In other words, an application or process can use different versions of the same `Ruleflow` depending on dateTime criteria. The details of how this works at the `Server` level is technical in nature and is described in the *Server Integration & Deployment Guide*.

Effective and Expiration Dates may be assigned using the same window as above. Clicking on the **Effective Date** or **Expiration Date** drop-down displays a calendar and clock interface, as shown below:

Figure 223: Setting Effective and Expiration Dates



Test yourself - questions – ruleflow versioning & effective dating

Note: Try this test, and then go to **Answers:** Ruleflow versioning & effective dating on page 284 to correct yourself.

1. True or False. If a Ruleflow has an Effective date, it must also have an Expiration date.
2. True or False. If a Ruleflow has an Expiration date, it must also have an Effective date.
3. True or False. Ruleflow Version numbers are mandatory
4. Which `Studio` menu contains the Ruleflow Properties settings?
5. True or False. A Ruleflow Minor or Major Version number may be raised or lowered.
6. True or False. Ruleflow Effective and Expiration dates are mandatory.

12

Localizing Corticon Studio

For details, see the following topics:

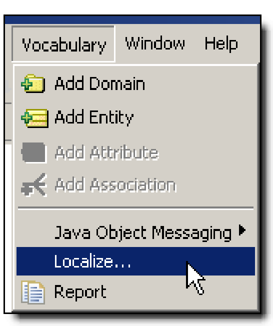
- Localizing the Vocabulary
- Localizing the Rulesheet

Localizing the Vocabulary

A Vocabulary can be localized into several different languages. If a Vocabulary includes multiple locale information, then Corticon Studio displays the locale corresponding to the operating system's current locale.

To localize a Vocabulary, select **Vocabulary>Localize…** from Corticon Studio's menubar as shown below:

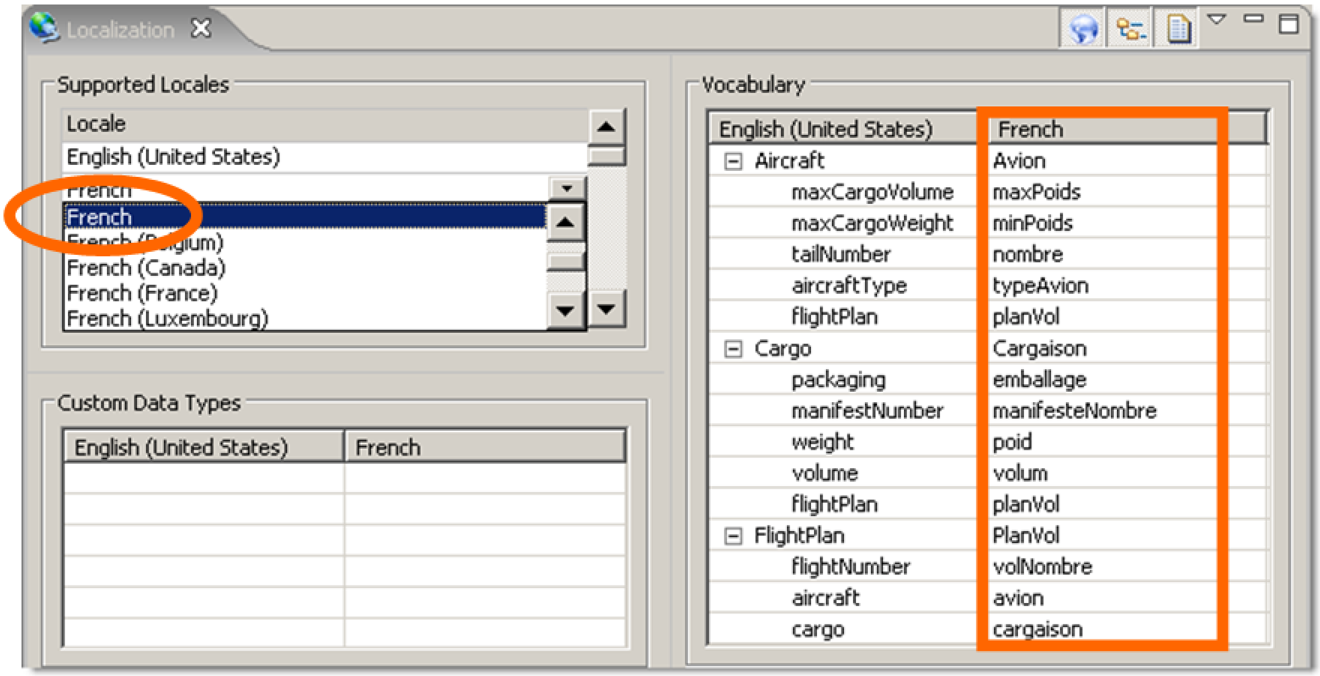
Figure 224: Localize a Vocabulary



Corticon Studio displays the Vocabulary Localization window as shown below.

Notice that we've selected French in the second line of the **Supported Locales** pane, circled below in orange. This choice causes a second column to appear to the right in the **Vocabulary** pane (shown below in an orange rectangle).

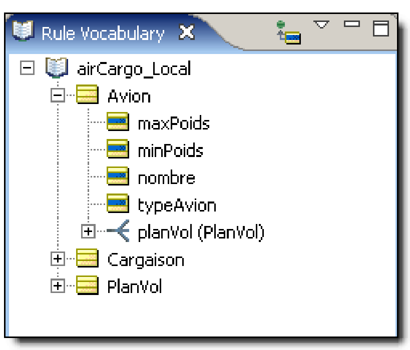
Figure 225: Selecting and populating a second locale



In the **French** column, we must manually add French translations for each Vocabulary term, including association role names. These translated terms must be unique from the English or other base version shown in the left-hand column.

With a localized Vocabulary, now switch your operating system to French locale. Different OS's have different methods for switching locales - consult your OS help or documentation for assistance.

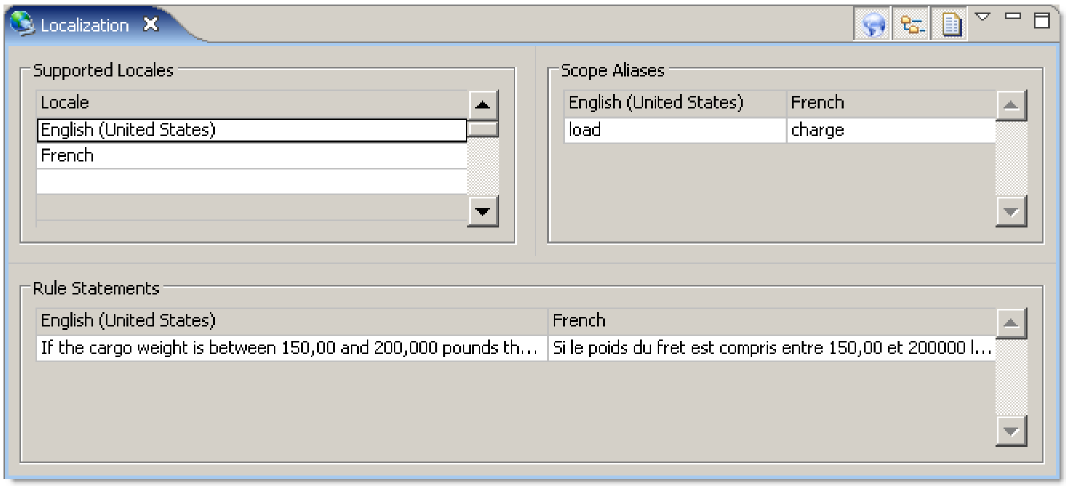
Figure 226: A Vocabulary displaying its French translation



Localizing the Rulesheet

When you create a new *`Rulesheet`* or *`Ruletest`* using a localized Vocabulary, those assets will be localized too. The **Rulesheet >Localize…** menu selection allows you to further localize the *`Rulesheet`* by translating Scope aliases and Rule Statements, as shown below:

Figure 227: A Rulesheet displaying its French translation



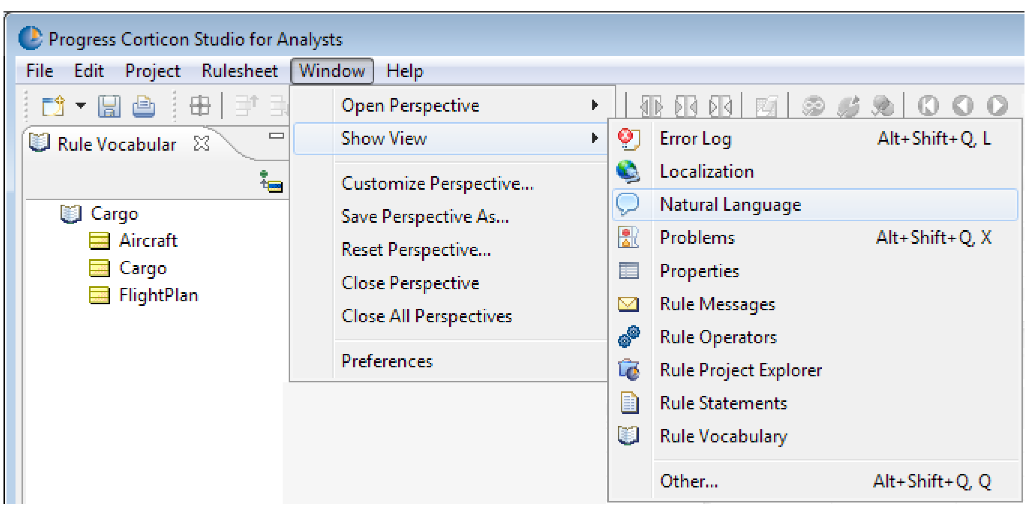
13

Working with rules in natural language

Progress Corticon lets you use Natural Language (NL) words, phrases, and sentences for use as substitute terms in `Rulesheet` Conditions and Actions.

Open the Natural Language window by choosing the menu command **Window>Show View>Natural Language**.

Figure 228: Opening the Natural Language view for a Rulesheet

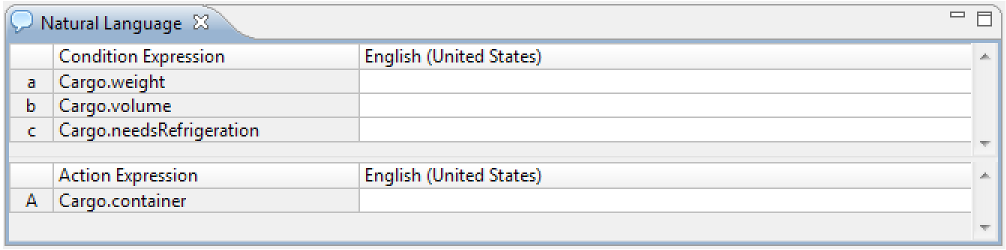


You could also right-click on the Rulesheet to drop down a menu with the **Natural Language** command.

Note: If you do not see the **Natural Language** command listed, you may need to **Reset Perspective** first. This typically only happens if you are upgrading from an earlier version of Corticon Studio.

The Natural Language view displays at the top of the Rulesheet in Corticon Studio:

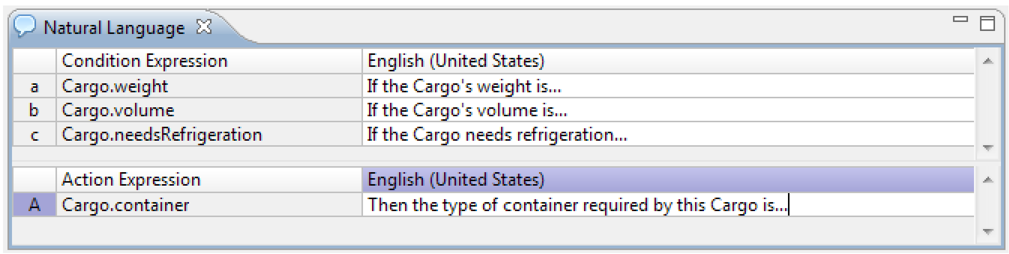
Figure 229: The Natural Language View



If you have other Locales enabled (see Localization chapter for more details), then the Natural Language window will include columns for other languages besides English. This allows you to define Natural Language text for those locales, too.

A populated Natural Language window is shown below. Notice that we've tried to be as clear and descriptive as possible, including the words `If` in Conditions and `Then` in Actions. We've also used ... to indicate that the expression continues in the column cells to the right. Your use of natural language may vary, but we recommend adopting a consistent, clear style.

Figure 230: Populating the Natural Language Window populated with natural language text

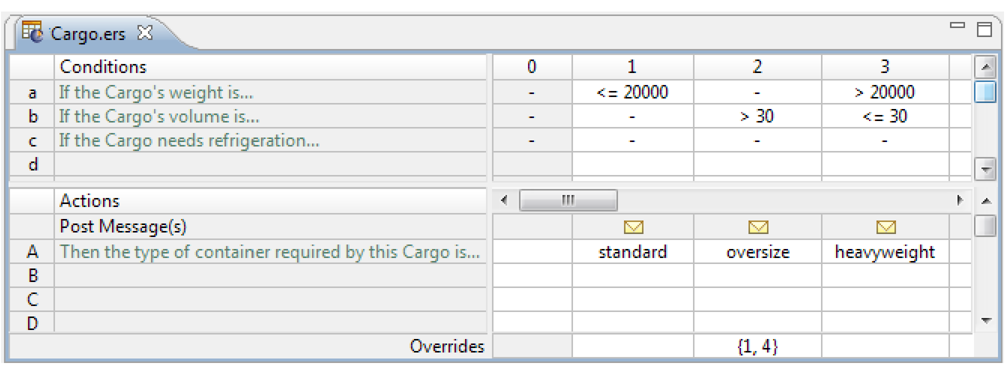


When your natural language expressions are defined, view these entries in place of the standard Condition and Action expressions in the Rulesheet as follows:

1. Close the **Natural Language** view by clicking its **X**.

2. Click the **Natural Language: On** menu button, , to display the Natural Language expressions you entered. Note that while Natural Language is displayed, the text of the Condition and Action rows cannot be edited.

   Figure 231: Rulesheet with natural language text

3. To revert to original, editable expressions, Click the **Natural Language: Off** menu button, .

14

The Corticon reporting framework

Corticon Studio contains a flexible and extensible framework for generating reports from `Studio` files.

Each type of Studio asset uses a built-in XML template that defines the structure of an XML report generated by Studio. Then, a built-in XSLT stylesheet transforms the XML into a regular HTML file viewable with a standard web browser such as Microsoft Internet Explorer or Mozilla Firefox

The *Quick Reference Guide* covers the mechanics of creating reports using the standard templates included with the installation. This chapter describes how to customize reporting templates and formats by taking advantage of the inherent flexibility of XML.

For details, see the following topics:

How Corticon creates reports

When a user selects **Rulesheet >Report**, Corticon Studio automatically:

- generates a XML file using its built-in XML template
- transforms the XML to HTML using its built-in XSLT stylesheet
- displays the HTML file in a web browser
- copies the XML and HTML files to the JVM's temporary directory. In Windows, by default, this is `C:\Users\{user}\Local Settings\Temp`. You can change this by adding the following line to `Studio.ini` located in the `[CORTICON_HOME]\Studio` directory

  ```
  -Djava.io.tmpdir=<your new Reports directory>
  ```

- Use *forward* slashes when writing the path to your new Reports directory, such as `C:/Program Files (x86)/Corticon/ Studio/myReports`

Customizing the XSLT stylesheets

Studio's built-in XSLT stylesheets cannot be viewed or modified. However, you can tell Studio to use an XSLT stylesheet of your own creation if you want to generate custom Studio reports. The files required, including a sample copy of the built-in XSLT stylesheets, are located in `[CORTICON_WORK_DIR]\Samples\Reports\Rulesheet` (or `\Vocabulary`, `\Ruleflow`, `\Ruletest`).

Once you have created your own XSLT stylesheet (or modified the sample provided), copy the entire `[CORTICON_WORK_DIR]\Samples\Reports` directory to the `[CORTICON_HOME]\Studio\eclipse\configuration\com.corticon.brms` (full Corticicon Studio) or `[CORTICON_HOME]\Studio\configuration\com.corticon.brms` (Corticicon Studio for Analysts).

When you close and re-open Corticon Studio, it will "discover" the new XSLT stylesheets and use them to generate reports.

15

Troubleshooting Rulesheets and Ruleflows

In addition to being a convenient way to test your *Rulesheet*s with real business scenarios, the Corticon Studio *Ruletest* facility is also the best way to troubleshoot rule, *Rulesheet*, and *Ruleflow* operation. Corticon *Ruletest* are designed to replicate exactly the data handling, translation, and rule execution by Corticon Server when deployed as a Java component or web service in a production environment.

This means that if your rules function correctly when executed in a Corticon *Ruletest*, you can be confident they will also function correctly when executed by Corticon Server. If they do not, then the trouble is most likely in the way data is sent to Corticon Server – in other words, in the technical integration. This is such a fundamental tenet of rule modeling with Corticon, we'll repeat it again:

If your rules function correctly when executed in a Corticon Studio, they will also function correctly when executed by Corticon Server. If they do not, then the trouble is most likely your client application's integration/invocation with/of Corticon Server.

We offer the following methodology to guide your rule troubleshooting and debugging efforts. The basic technique is known generically as "half-splitting" or "binary chopping", in other words, dividing a decision into smaller logical pieces, then setting aside the known-good pieces systematically until the problem is isolated.

This guide is not intended to be an in-depth cookbook for correcting specific problems since, as an expression language, the Corticon Rule Language offers too many syntactical combinations to address each in any detail.

For details, see the following topics:

Where did the problem occur?

Regardless of the environment the error or problem occurred in, we will always first attempt to reproduce the behavior in Studio. If the error occurred while you were building and testing rules in Corticon Studio, then you're already in the right place. If the error occurred while the rules were running on `Corticon Server` (in a test or production environment), then you will want to obtain a copy of the deployed `Ruleflow` (`.erf` file) and open it, its constituent `Rulesheet`s (`.ers` files) and its Vocabulary (`.ecore` file) in Studio.

Using Corticon Studio to reproduce the behavior

It is always helpful to build and save "known-good" `Ruletest`s (`.ert` files) for the `Corticon Rulesheet`s and `Ruleflow`s you intend to deploy. A known-good `Ruletest` not only verifies your `Rulesheet` or `Ruleflow` is producing the expected results for a given scenario, it also enables you to re-test and re-verify these results at any time in future.

If you do not have a known-good `Ruletest`, you will want to build one now to verify that the `Ruleflow`, as it exists right now, is producing the expected results. If you have access to the actual data set or scenario that produced the error in the first place, it is especially helpful to use it here now. Run the `Ruletest`.

Running a Ruletest in Corticon Studio

When the `Ruletest` is run, does Corticon Studio produce any error messages? By this, we do **not** mean the info, warning, or violation messages that may be posted by normal operation of the rules.

- If you encounter any of the following errors, take the actions described in that section. It may be possible, using the techniques in this manual, to work around a problem by identifying the expression syntax that produces it, and trying to express the logic in a different way. The `Corticon` Rule Language is very flexible and usually allows the same logic to be expressed in many different ways.
- If you do not encounter any of these errors, proceed to the Analyzing Test Results section.

The Null Pointer Exception

Figure 232: A Null Pointer Exception as Shown in a Ruletest

Severity	Message	Entity
Violation	Rule execution has abnormally terminated due to encountering a java.lang.NullPointerException	

A null pointer exception (NPE) almost always indicates a problem with the rule engine, whether it is being employed by the Corticon Studio or by Corticon Server. If you encounter this error, contact Technical Support, and set aside copies of the Vocabulary, `Rulesheet`, and `Ruletest` so we can use them for further troubleshooting.

The Reactor Exception

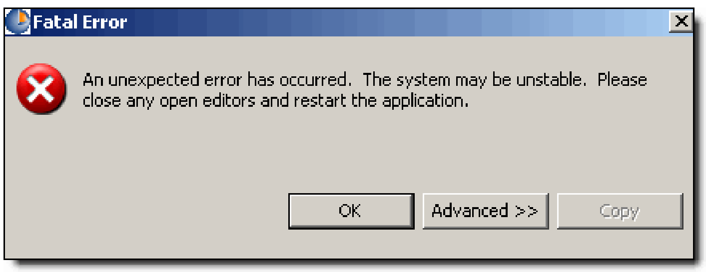
Figure 233: The Reactor Exception Window

This error is also indicative of a possible rule engine problem, but it may also indicate improper or unnecessarily complex language usage in your rule expressions. The Rule Language's flexibility may permit workarounds to the limitation(s) that produced this message; `Progress Corticon` Technical Support should be contacted for further assistance. As with the NPE errors, please save copies of the Vocabulary, `Rulesheet`, and `Ruletest` for use by `Progress Corticon` Support staff.

The Fatal Exception

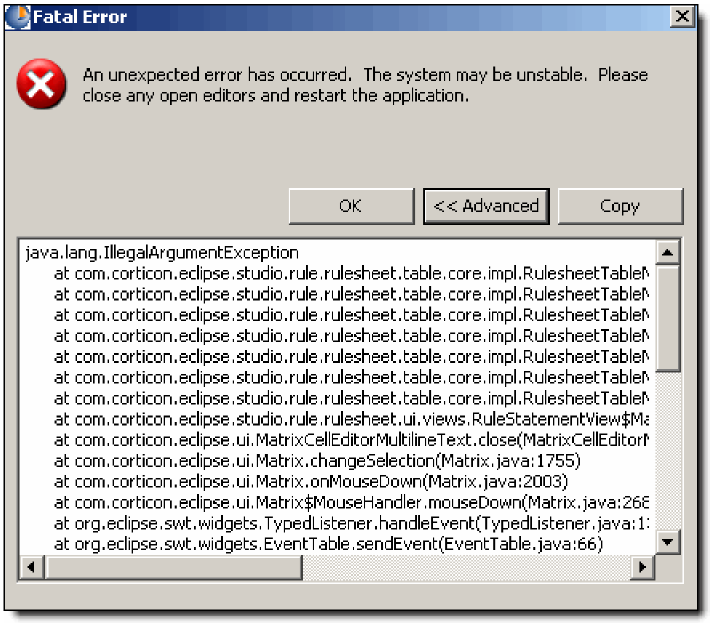
On very rare occasions, you may experience a full crash of the Corticon Studio application. In these cases, you will see a window like the following:

Figure 234: The Fatal Error Window



This type of problem can only be resolved by Corticon. But before contacting *Progress Corticon* Technical Support, click on the Advanced >> button, which will display a window with more information about the cause of the error, as shown below:

Figure 235: The Fatal Error Details Window



Click the **Copy** button (as shown) and paste the text into a text file. Send us this text file along with the standard set of Corticon Studio files (Vocabulary, *Rulesheet*, *Ruletest*) when you report this problem.

Analyzing Ruletest results

This section assumes:

- Your *Ruletest* produced none of the errors listed above, or
- You or *Corticon* Technical Support identified workarounds that overcame these errors

Does the *Rulesheet* produce the expected test results? In other words, does the *actual* output match the *expected* output?

- If so, and you were using the same scenario that caused the original problem, then the problem is not with the rules or with Studio, but instead with the data integration or *Server* deployment. Proceed to Troubleshooting Server Problems
- If not, the problem is with the rules themselves. Continue in this section.

Identifying the breakpoint

To understand why your rules are producing incorrect results, it's important to know where in the Rulesheet or Ruleflow the rules stop behaving as expected. At some point, the rules stop acting normally and start acting abnormally – they "break". Once we identify where the rule breaks, the next step is to determine why it breaks.

Because of a fundamental Corticon Server behavior, the breakpoint will always be a single rule. If the data available to Corticon Server causes **all** of a rule's Conditions to be satisfied and Corticon Server is able to execute **all** of the rule's Actions, then Corticon Server will fire the rule.

An important corollary to this: if any one of the rule's Actions cannot be executed (for reasons to be discussed later), then none of the Actions will be executed – the rule will not fire. Bottom line: a rule will fire in its entirety or not at all. We call this "all or nothing" behavior.

An important tool to help identify the breakpoint is the Ruletest's message box. By choosing values for `Post` and `Alias` columns in the **Rule Messages** window, you can generate a trace or log of the rules that fire during execution. The message box in a Ruletest will display those messages in the order they were generated by Corticon Server. In other words, the order of the messages in the box (top to bottom) corresponds to the order in which the rules were fired by Corticon Server. While messages in the message box can also be sorted by Severity or Entity by clicking on the header of those columns, clicking on the Message column header will always sequence according to the order in which the rules fired. Inserting attribute values into rule statements can also provide good insight into rule operation. But beware; a non-existent entity inserted into a rule statement will prevent the rule from firing, becoming the cause of another failure!

Enabling and disabling individual Condition/Action rows, entire rule columns, Filter rows, and even whole Rulesheets is another powerful way to isolate problems in your ruleset. Right-clicking Condition or Action row headers, column headers, Filter row headers, or Rulesheet boxes in the Ruleflow will display a pop-up menu containing enable/disable options. Disabled rows and columns will be shaded in gray on the Rulesheet, while disabled Rulesheets turn dark gray in the Ruleflow diagram. Be sure to save these changes before running a Ruletest to ensure the changes take effect.

Figure 236: Rulesheet with Rule Column #2 Disabled.

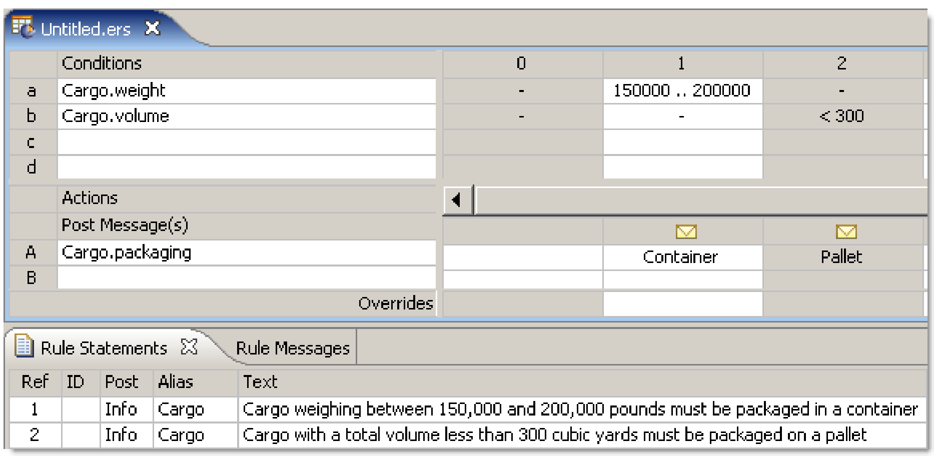
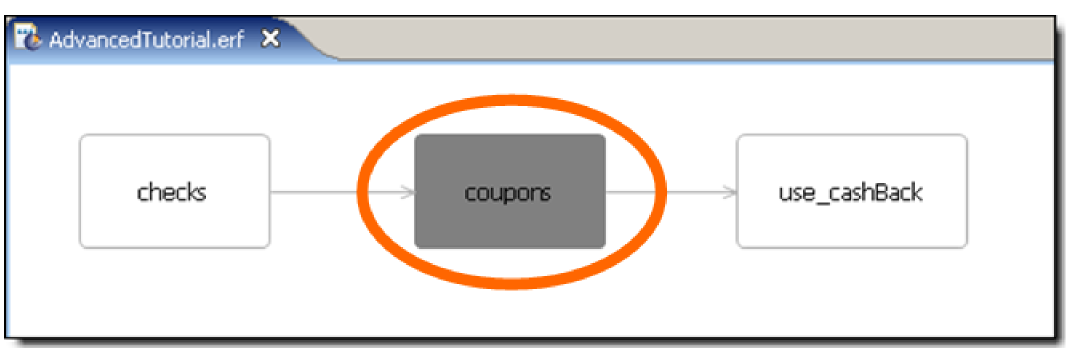


Figure 237: Disabled Rulesheet, Tab Label Circled.



Disable rows, rule columns, and/or Rulesheets until the strange or unexpected behavior stops.

At the breakpoint

At the point at which abnormal behavior begins, what results is the breakpoint rule producing?

- No results at all – the breakpoint rule *should* fire (given the data in the `Ruletest`) but does not. Proceed to the No Results section.
- Incorrect results – the breakpoint rule *does* fire, but without the expected result. Proceed to the Incorrect Results section.

No results

Failure of a rule to produce any results at all indicates the rule is telling the rule engine to do something it can't do. (This assumes, of course, that the rule *should* fire under normal circumstances.) Frequently, this means the engine tries to perform an operation on a term that does not exist or isn't defined at time of rule execution. For example, trying to:

- Increment or decrement an attribute (using the `+=` or `-=` operators, respectively) whose value does not exist (in other words, has a `null` value).
- "Post" a message to an entity that does not exist, either because it was not part of the `Ruletest` to begin with, or because it was deleted or re-associated by prior rules.
- "Post" a message with an embedded term from the Vocabulary whose value does not exist in the `Ruletest`, or was deleted by prior rules.
- Create (using the `.new` operator) a collection child element where no parent exists, either because it was not part of the `Ruletest` to begin with, or because it was deleted or re-associated by prior rules.
- Trying to *forward-chain* -- using the results of one expression as the input to another -- within the same rule. For example, if Action row B in a given rule derives a value that is required in Action row C, then the rule may not fire. Both Actions must be executable independently in order for the rule to fire. If forward-chaining is required in the decision logic, then the chaining steps should be expressed as separate rules.

No partial rule firing

A Condition/Action rule column should never "partially fire", meaning Action 1 is executed but Action 2 is not. If Action A cannot execute, then Action B will not execute either, even if there is nothing wrong with Action B by itself. An Action containing any one of the problems listed above is sufficient to prevent a rule from firing, even if all other Actions in the rule are valid. Understanding this "all or nothing" execution behavior is very important for troubleshooting.

An exception to this rule is the special Nonconditional rule column 0. Each Action row in column 0 counts as its own separate, independent rule, so Action row A may fire even if Action row B does not.

Initializing null attributes

Once the breakpoint rule has been isolated, it is often helpful to copy the relevant logic into another `Rulesheet` for more focused testing. Refer to the *Rule Language Guide* to ensure you have expressed your rules correctly. Be sure to review the usage restrictions for the operator(s) in question.

If, after isolating and verifying the suspicious expression syntax, you are unable to fix the problem, please call `Progress Corticon` Technical Support. As always, be prepared to send us a) the product version used, and b) the set of Corticon files (`.ecore`, `.ers`, `.erf`, and `.ert`) that will allow us to reproduce the problem.

Incorrect results

Once the breakpoint rule has been isolated, it is often helpful to copy the relevant logic into another `Rulesheet` for more focused testing. Refer to the *Rule Language Guide* to ensure you have expressed your rules correctly. Be sure to review the usage restrictions for the operator(s) in question.

If, after isolating and verifying the suspicious expression syntax, you are unable to fix the problem, please call `Progress Corticon` Technical Support. As always, be prepared to send us a) the product version used, and b) the set of Corticon files (`.ecore`, `.ers`, `.erf`, and `.ert`) that will allow us to reproduce the problem.

Troubleshooting Corticon Server problems

This section assumes that Corticon Studio problems, including incorrect rule expression, have been ruled out or fixed in previous sections. When Corticon Server side is involved, the best troubleshooting tool is the log file produced during Corticon Server operation. To configure logging for maximum benefit, follow these steps:

1. Stop Corticon Server.
2. Go to the location of your Corticon Server's `CcConfig.jar` file. In the default Tomcat installation (in Windows), this is `[CORTICON_HOME]\Server\Tomcat\webapps\axis\WEB-INF\lib`
3. Open `CcConfig.jar` with a standard decompression utility (like WinZip or 7-Zip) and look for a text file named `CcCommon.properties`.
   a) Look for the `loglevel` property and change its value to `INFO`
4. Restart Corticon Server and rerun the test.
5. Go to the directory containing log files. In the default Windows Tomcat installation, this is `[CORTICON_HOME]\Server\Tomcat\CcServer\Log` and locate today's log file. Its filename is in `yyyymmdd.log` format.
6. Either delete or rename today's log file. A new log file is created every day, and each Corticon Server transaction adds entries to the day's log. Because daily log files may become quite large, it is useful to start with a fresh log that records only the problematic transaction. The next time Corticon Server processes a transaction, a new log file will be created and entries made in it.

   What is Corticon Server's response?

No response
In most cases, the failure of Corticon Server to produce a log file means that the invocation/request is not even reaching it! The most common causes of a non-responsive (invocation produces no log file entry) Corticon Server include:

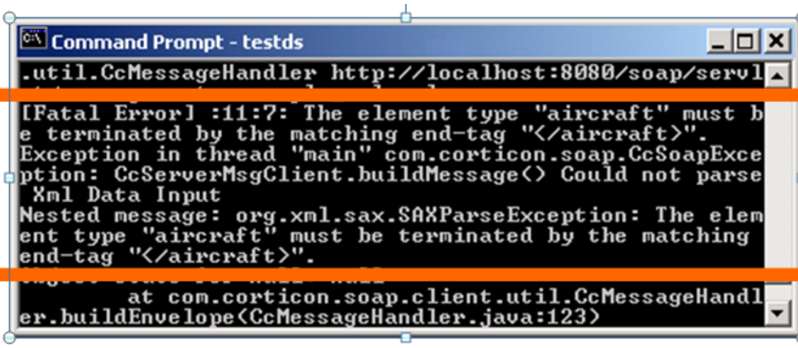
- Incorrect Corticon Server deployment. Review procedures in the *Server Integration & Deployment Guide* to ensure correct deployment.
- Incorrect Corticon Server invocation
  1. Incorrect URL. If using a web services deployment, ensure the SOAP message is addressed correctly, and that no firewalls or port configurations prevent the SOAP message from reaching Corticon Server.
  2. Incorrect API. Review the *Server Integration & Deployment Guide* for complete details on Java APIs available for Corticon Server invocation. A complete *JavaDoc* and sample code set is also provided in the `[CORTICON_HOME]\JavaDoc` directory.
- Even though Corticon Server may not respond to an incorrect invocation, the host server or container (app server, web server, and similar) may respond either at a command line or log level. Check to see if the host server has responded to your invocation.

Response containing errors
The most common causes of erroneous Corticon Server responses include:

- Incorrect messaging
  1. Message payload does not conform to service contract. Compare your SOAP message to the service contract produced by the Deployment Console to ensure compliance. Many third-party tools are available that automatically validate an XML document (in this case, the SOAP message) to its schema (in this case, the WSDL service contract). Notice that if Corticon Server cannot even parse the inbound SOAP message, no entry will be made in

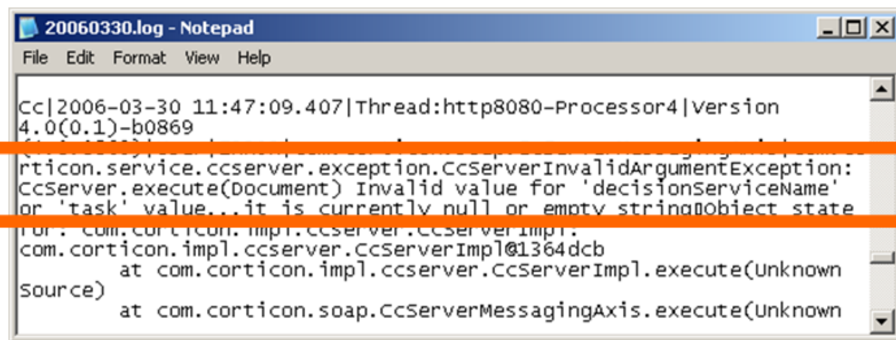
Corticon Server's log. Instead, the error message will be displayed directly in the web server window, as shown below:

Figure 238: Server Window Message Highlighting Incorrect SOAP Request Structure



2. Incorrect or missing Decision Service Name. Ensure the SOAP message's Decision Service Name attribute matches the name of the Decision Service as it was deployed by either a deployment descriptor file or an API method call. See figure below.

   Figure 239: Log Excerpt Highlighting Missing Decision Service Name in SOAP Request



- Corticon Server licensing problem

  1. License not installed. Your `CcLicense.jar` license file must be located in the same directory as your server installation's `CcServer.jar` file. In the default installation, `CcServer.jar` is located in `[CORTICON_HOME]\Server\Tomcat\webapps\axis\WEB-INF\lib`, so ensure your valid license file is there.
  2. If you are using one of the `.war` or `.ear` packages provided in `[CORTICON_WORK_DIR]\Samples`, then be sure that those packages also include valid copies of `CcLicense.jar`.

3. License expired. See your Progress Corticon representative for an updated license file.

Figure 240: Maintenance Message in Server Window

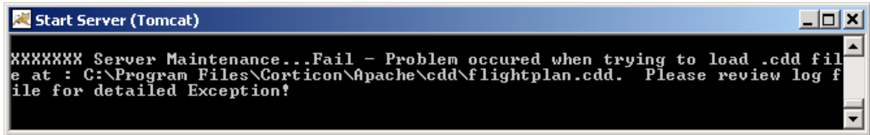
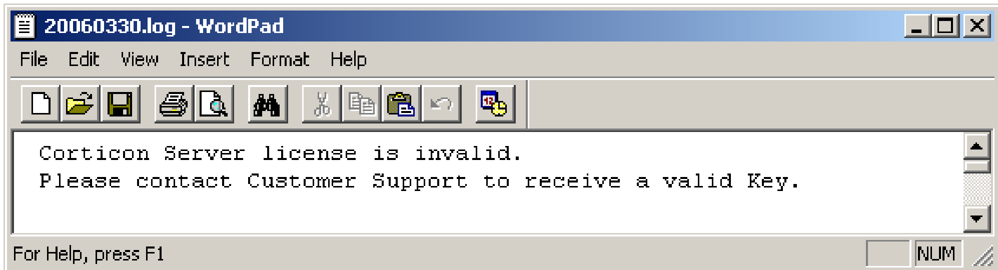
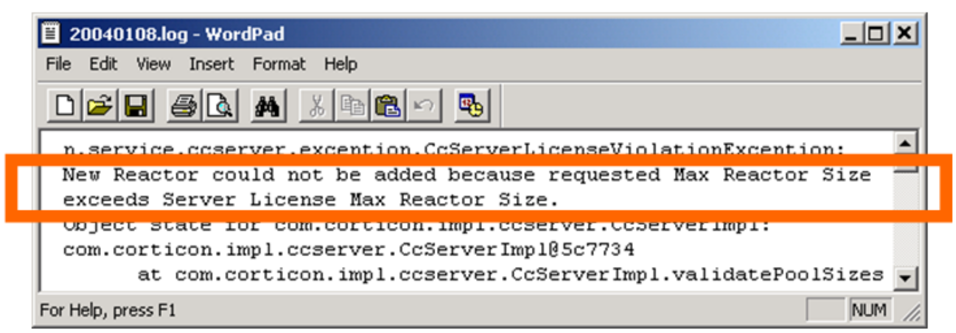


Figure 241: Log Excerpt Highlighting License Expiration Message



4. License capacity exceeded. License capacity is measured in several ways:
   - Number of unique Decision Services that may run concurrently in Corticon Server. Make sure your license can support the total number of unique `.erf` files referenced by deployed `.cdd` files.
   - Number of *copies* of Decision Services the Corticon Server may create to handle transaction volume (known as "Reactors"). Make sure your license can support the max pool size specified in the deployed `.cdd` files. See figure below.
   - Number of rules allowed for all Decision Services deployed. Make sure your license can support the total number of rules contained in all the deployed `.erf` files.
   - IP address (or multiple) allowed. Make sure your license URL address matches the address of your Corticon Server.
   - Number of transactions per time period allowed. Make sure your license can support the total number of transactions per time being requested of Corticon Server.

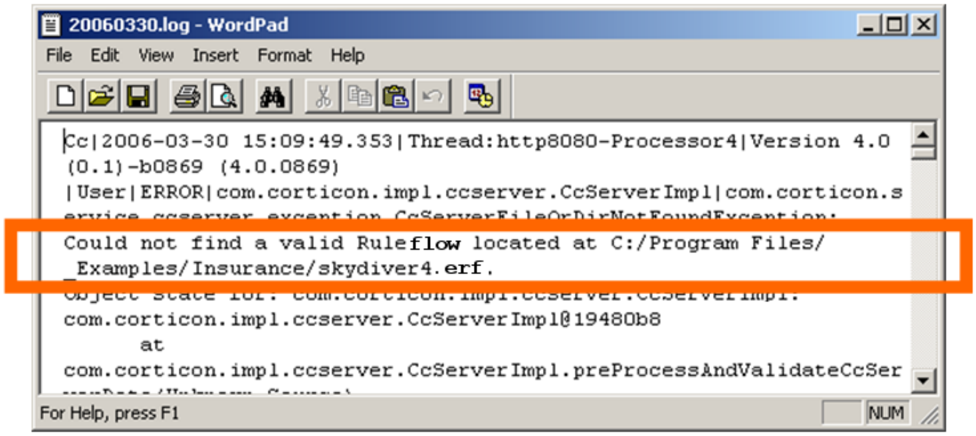
   Figure 242: Log Excerpt Highlighting License Pool Limitation



- Deployment Descriptor (`.cdd`) file problems

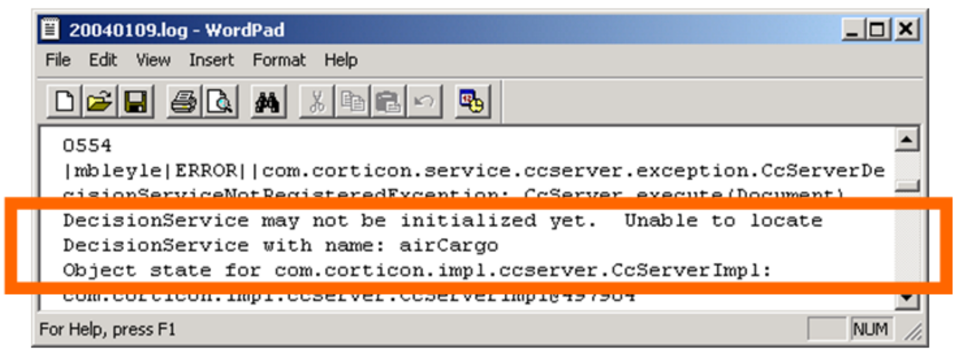
1. Missing `.erf` file. The `.erf` file has been moved and is no longer located in the directory referenced by the `.cdd` file.

   Figure 243: Log Excerpt Highlighting Incorrect .erf Path

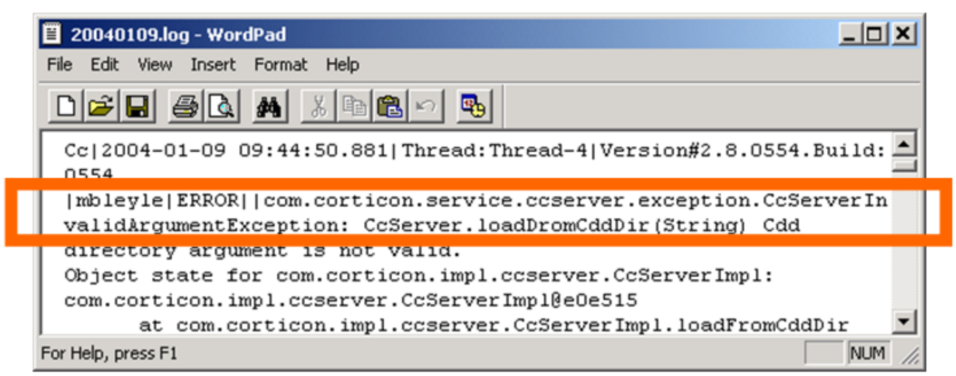
2. Missing `.cdd` file. The `.cdd` file is missing from the `\cdd` directory, or the taskname contained in the SOAP request message does not match any of the tasknames in any of the `.cdd` files deployed to Corticon Server.

   Figure 244: Log Excerpt Highlighting Unknown Decision Service Name

3. Missing `\cdd` directory. The proper location of this directory in your server installation (when installed using default parameters) is `[CORTICON_WORK_DIR]\cdd`

   Figure 245: Log Excerpt Highlighting Missing `\cdd` Directory

- Object translation errors due to incorrect Vocabulary external name mappings

1. External names mapped incorrectly
2. External data types specified incorrectly
3. ALL entities must be mapped, even those where all attributes are transient.

Test yourself - questions – troubleshooting rulesheets and ruleflows

Note: Try this test, and then go to **Answers:** Troubleshooting rulesheets on page 284 to correct yourself.

1. Troubleshooting is based on the principle that *Rulesheet*s behave the same way when tested in Corticon Studio as when executed on ___________.
2. The first step in troubleshooting a suspected rule problem is to reproduce the behavior in a Corticon Studio _________ (test)
3. If the *Rulesheet* executes correctly in Corticon Studio, then where does the problem most likely occur?
4. Which of the following problems requires you to contact *Progress Corticon* Support for help?

Fatal Error	Null Pointer Exception	Reactor Error	Expired License

5. The specific rule where execution behavior begins acting abnormally is called the _____________.
6. True or False. When a rule fires, some of its Actions may execute and some may not.
7. What Corticon Studio tools help you to identify the *Rulesheet*'s breakpoint?
8. A dark gray-colored *Rulesheet* box within a *Ruleflow* indicates that the *Rulesheet* has been __________.
9. A disabled rule:
   a. executes in a Corticon Studio Test but not on the Corticon Server
   b. executes on the Corticon Studio but not in a Corticon Studio Test
   c. executes in both Corticon Studio Tests and on the Corticon Server
   d. executes neither in a Corticon Studio Test nor on the Corticon Server
10. Where is the Corticon Server `loglevel` setting located?
11. To produce a more detailed Corticon Server log file, what `loglevel` setting should be enabled?
12. True or False. The Corticon Server license file needs to be located everywhere the Corticon Server is installed.
13. If you are reporting a possible Corticon Studio bug to Corticon Support, what minimum information is needed to troubleshoot?
14. Which of the following cannot be disabled?

a. a Condition row

b. an Action row

c. a Filter row

d. a leaf of the Scope tree

e. a Noncondition row (i.e., an Action row in Column 0)

f. a rule column

g. a *Rulesheet*

h. a *Ruleflow*

A

Standard boolean constructions

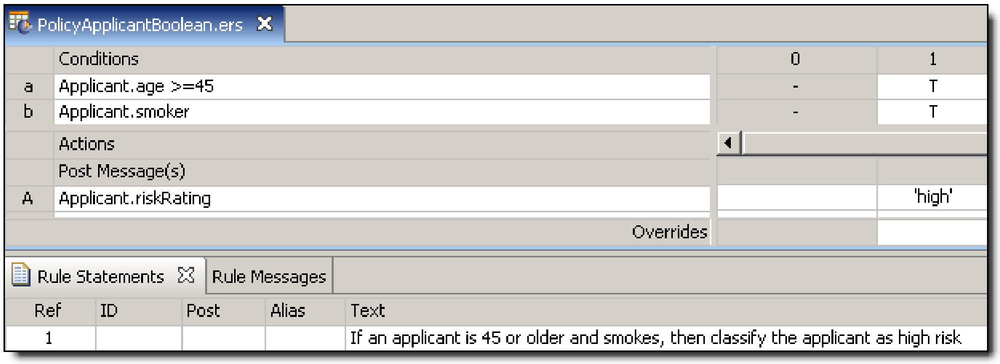
For details, see the following topics:

- AND
- NAND
- OR
- XOR
- NOR
- XNOR

AND

In a decision table, a rule with **AND'**ed Conditions is expressed as a single column, with values for each Condition aligned vertically in that column. For example:

1. If a person is 45 or older and smokes, then classify the person as high risk



In this scenario, each Condition has a set of 2 possible values:

person is 45 or older: `{true, false}`

person is a smoker: `{true, false}`

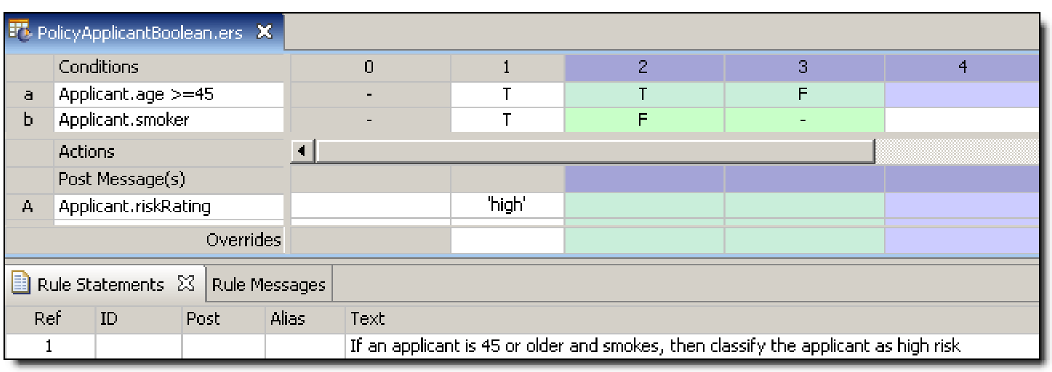
and the outcome may also have two possible values:

person's risk rating: `{low, high}`

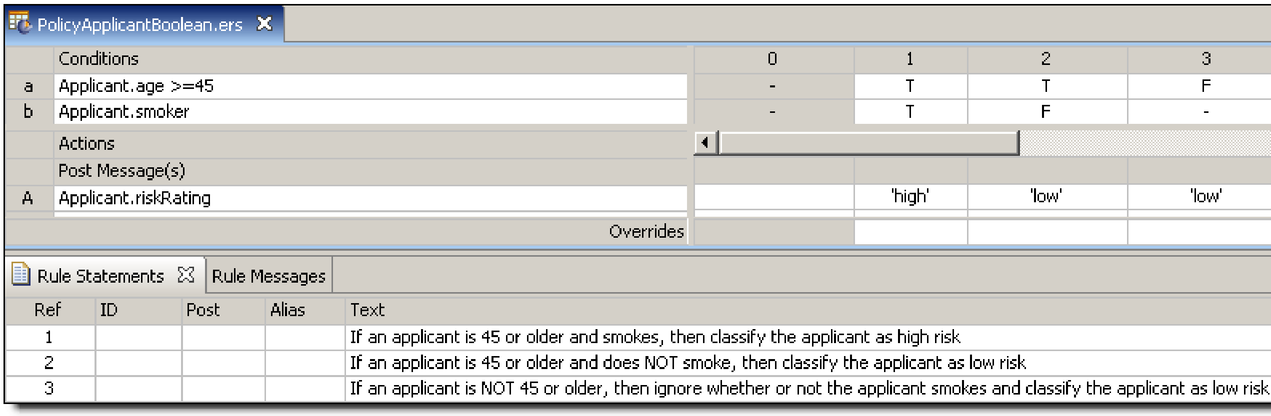
These Conditions and Actions yield the following truth table:

age >= 45	smoker	risk rating
`true`	`true`	`high`
`true`	`false`	
`false`	`true`	
`false`	`false`	

Note that we have only filled in a single value of risk rating, because the business rule above only covers a single scenario: where `age >= 45` and `smoker = true`. A Completeness Check quickly identifies the remaining 3 scenarios:



Completing the truth table and the *`Rulesheet`* requires the definition of 2 additional business rules:



and updating the truth table, we recognize the classic **AND** Boolean function.

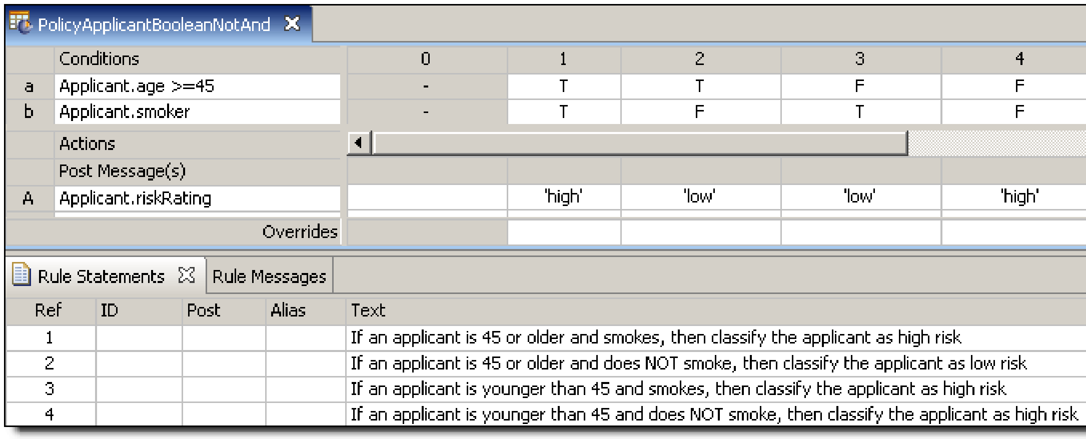
age >= 45	smoker	risk rating
`true`	`true`	`high`
`true`	`false`	`low`
`false`	`true`	`low`
`false`	`false`	`low`

Once the basic truth table framework has been established in the *`Rulesheet`* by the Completeness Checker – in other words, all logical combinations of Conditions have been explicitly entered as separate columns in the *`Rulesheet`* – we can alter the outcomes to implement other standard Boolean constructions. For example, the **NAND** construction has the following truth table:

NAND

age >= 45	smoker	risk rating
`true`	`true`	`low`
`true`	`false`	`high`
`false`	`true`	`high`
`false`	`false`	`high`

Also known as “Not And”, this construction is shown in the following *`Rulesheet`*:



PolicyApplicantBooleanNotAnd

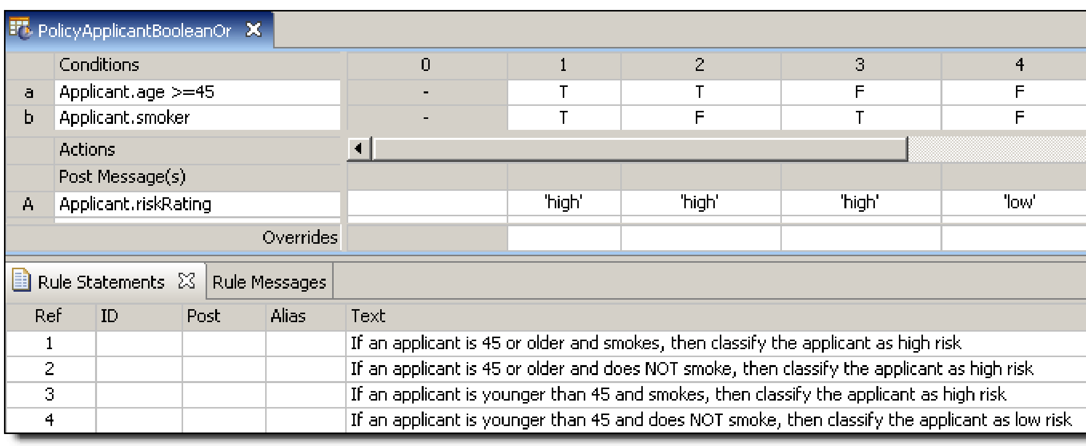
	Conditions	0	1	2	3	4
a	Applicant.age >=45	-	T	T	F	F
b	Applicant.smoker	-	T	F	T	F
	Actions					
	Post Message(s)					
A	Applicant.riskRating		'high'	'low'	'low'	'high'
	Overrides					

Rule Statements | Rule Messages

Ref	ID	Post	Alias	Text
1				If an applicant is 45 or older and smokes, then classify the applicant as high risk
2				If an applicant is 45 or older and does NOT smoke, then classify the applicant as low risk
3				If an applicant is younger than 45 and smokes, then classify the applicant as high risk
4				If an applicant is younger than 45 and does NOT smoke, then classify the applicant as high risk

OR

age >= 45	smoker	risk rating
`true`	`true`	`high`
`true`	`false`	`high`
`false`	`true`	`high`
`false`	`false`	`low`



PolicyApplicantBooleanOr

	Conditions	0	1	2	3	4
a	Applicant.age >=45	-	T	T	F	F
b	Applicant.smoker	-	T	F	T	F
	Actions					
	Post Message(s)					
A	Applicant.riskRating		'high'	'high'	'high'	'low'
	Overrides					

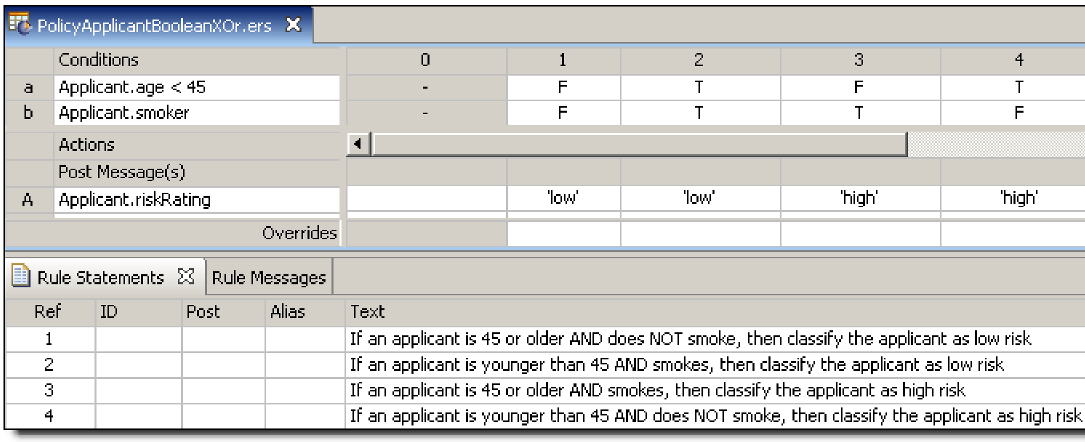
Rule Statements | Rule Messages

Ref	ID	Post	Alias	Text
1				If an applicant is 45 or older and smokes, then classify the applicant as high risk
2				If an applicant is 45 or older and does NOT smoke, then classify the applicant as high risk
3				If an applicant is younger than 45 and smokes, then classify the applicant as high risk
4				If an applicant is younger than 45 and does NOT smoke, then classify the applicant as low risk

XOR

Using "Exclusive Or" logic, `riskRating` is high whenever the age or smoker test, but not both, is satisfied. This construction is shown in the following *`Rulesheet`*:

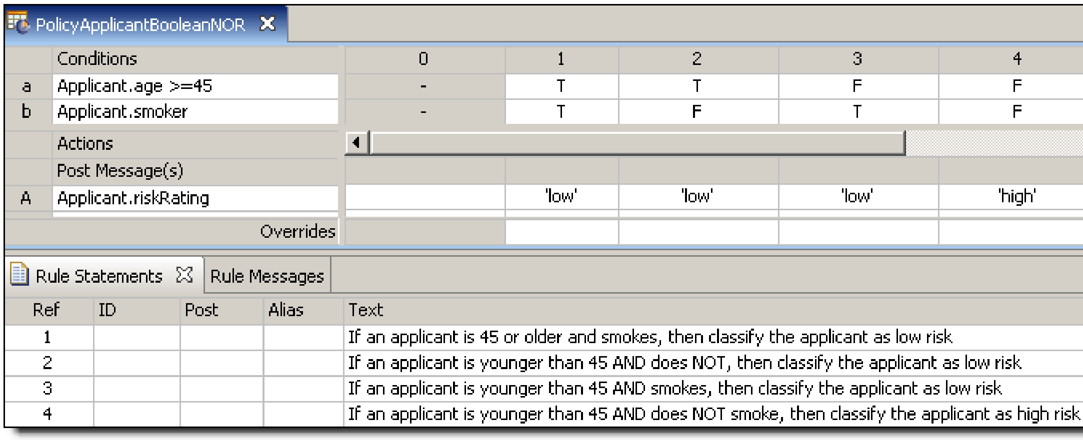
age >= 45	smoker	risk rating
true	true	low
true	false	high
false	true	high
false	false	low



NOR

Also known as "Not Or", this construction is shown in the following *Rulesheet*:

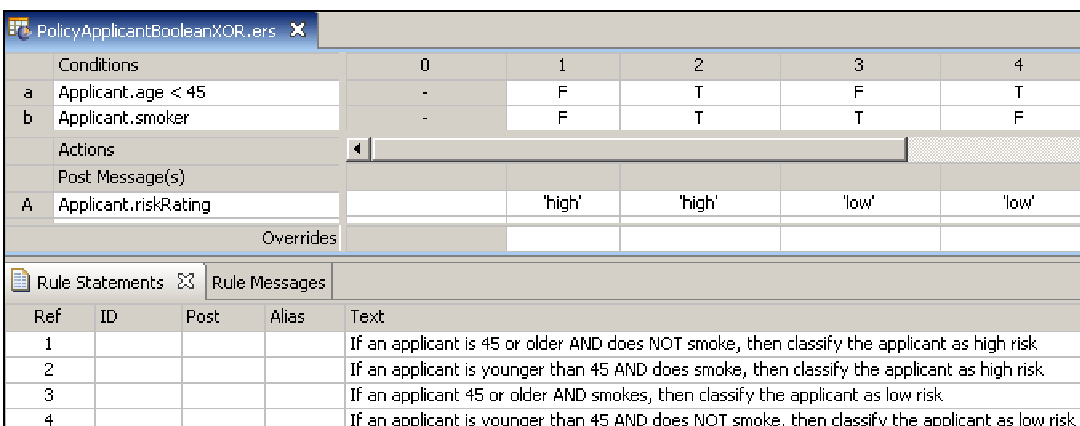
age >= 45	smoker	risk rating
true	true	low
true	false	low
false	true	low
false	false	high



XNOR

Also known as "Exclusive NOR", this construction is shown in the following *Rulesheet*:

age >= 45	smoker	risk rating
true	true	high
true	false	low
false	true	low
false	false	high



B

Building a Vocabulary based on Savvion Dataslots

Studio provides a Vocabulary wizard that helps you create a vocabulary file (.ecore) based on the dataslots defined in the Savvion Process Template (.spt) file. Existing .spt files can be used as source files for generating a vocabulary file, if they have dataslots defined in them.

For details, see the following topics:

- Generating the Vocabulary file
- Dataslot to Vocabulary Mapping

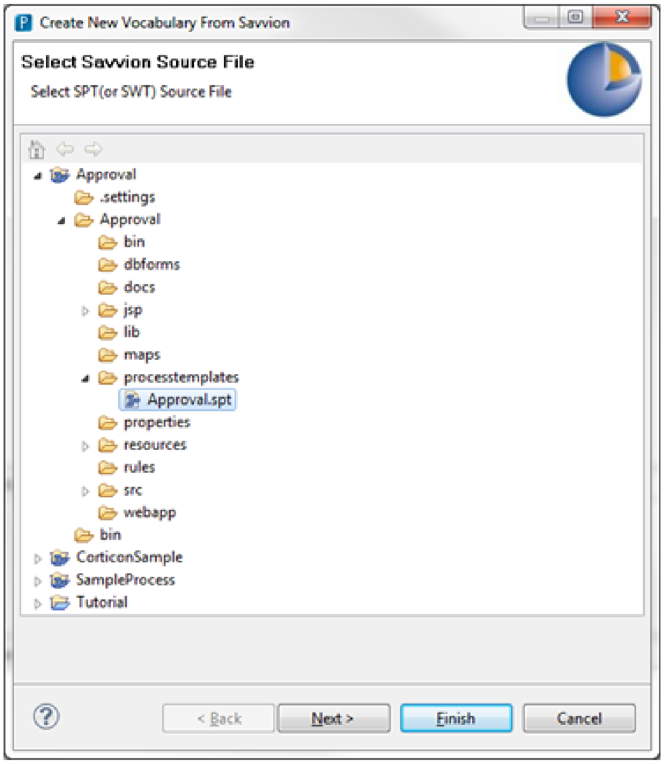
Generating the Vocabulary file

To generate a Vocabulary file:

1. Select **File** > **New > Other** > **Progress Corticon > Rule Vocabulary From Savvion**.

This opens up the **Create New Vocabulary From Savvion** dialog box, as shown in Selecting the Process Template File, which allows you to select the Savvion Process Template file for which you want to create a vocabulary file.

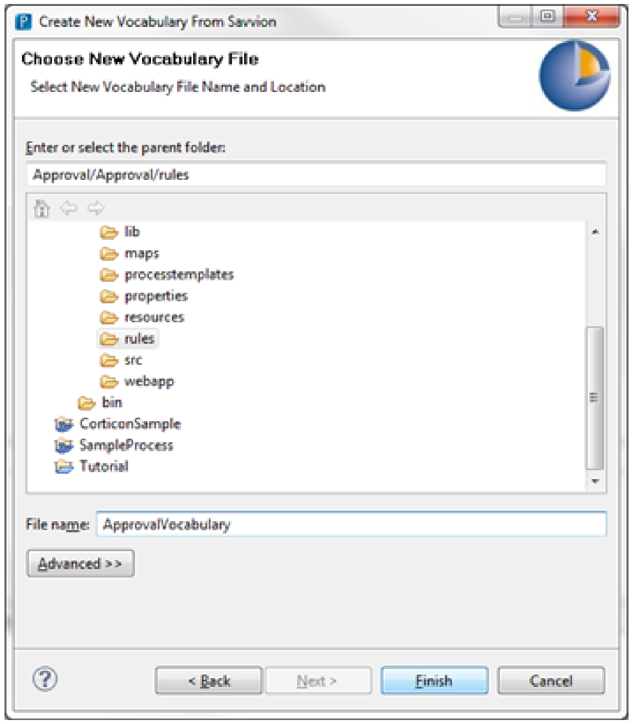
Figure 246: Selecting the Process Template File



2. Click **Next**.

This opens up a new window in the **Create New Vocabulary From Savvion** dialog box, as shown in Location and name for the Vocabulary file, which allows you to select the location for the vocabulary file. It also allows you to enter the file name.

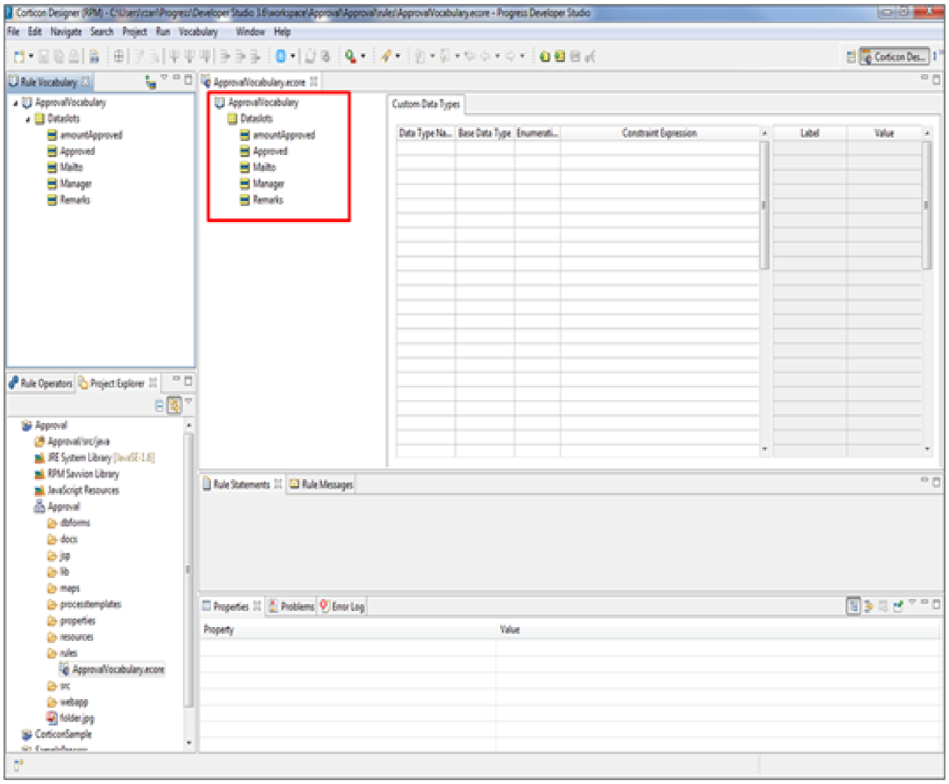
Figure 247: Location and name for the Vocabulary file



3. Click **Finish**.

This automatically generates a vocabulary file, as shown in The Generated Vocabulary File, and the dataslots are available as vocabularies for creating business rules.

Figure 248: The Generated Vocabulary File



Note:

- The 'AllDataSlots' and 'Document' dataslots are ignored while generating the vocabulary file because they are not the supported data types.
- The 'DecisionAuditLog' dataslot is a reserved dataslot and is ignored while generating the vocabulary file.
- A dataslot's Category is not taken into consideration while mapping.

Dataslot to Vocabulary Mapping

The dataslot to vocabulary mapping is done in the following way:

Savvion	Corticon
Process file (.spt, .swt)	The vocabulary file (.ecore)
Process data slots	An entity with a fixed name as "Dataslots", which contains the dataslots of the process.

Savvion	Corticon
Primitive data slot	A process entity attribute having same name with a correct Savvion-to-Corticon type mapping .
Primitive OE data slot	A process entity attribute having same name with a correct Savvion-to-Corticon type mapping .
System Category	A system entity.
System primitive data slot	The attributes of the system entity with a correct Savvion-to-Corticon type mapping .

The following table displays Savvion-to-Corticon type mapping:

Savvion Data Type	Corticon Data Type
String	String
Number:Integer	Integer
Number:Double	Decimal
Number:Decimal	Decimal
Boolean	Boolean
Date	DateTime
OEBPM:Character	String
OEBPM:DateTimeTZ	DateTime
OEBPM:Decimal	Decimal
OEBPM:Int64	Integer
OEBPM:Integer	Integer
OEBPM:Logical	Boolean

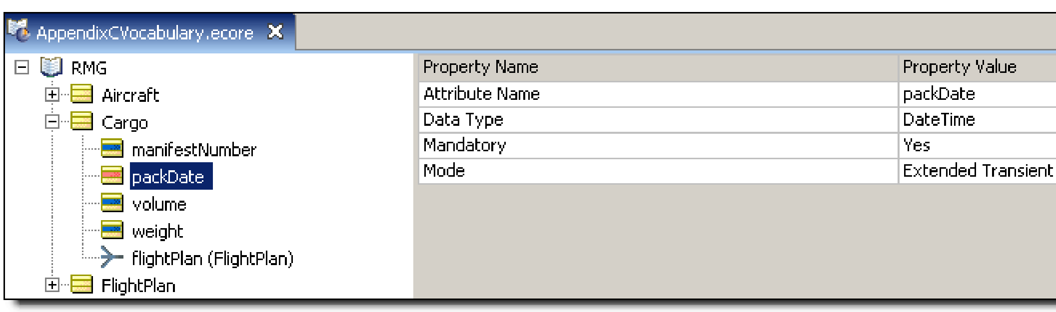
C

Extended transients in the vocabulary

As described in Building the Vocabulary, extended transient terms may be used to carry or hold values while rules are executing within a single *Corticon Rulesheet*. Since XML messages returned by a Decision Service do not contain extended transient attributes, these attributes and their values cannot be used by external components or applications. If an attribute value is used by an external application or component, it must be a base attribute.

To remind the rule modeler which attributes are base and which are extended transient, Corticon Studio color-codes attribute icons as follows:

Figure 249: Vocabulary Editor Showing an Extended Transient Attribute



Notice that the extended transient attribute `packDate` is colored orange in the Vocabulary tree. XML response messages created by Corticon Server will not contain the `packDate` attribute.

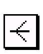
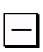
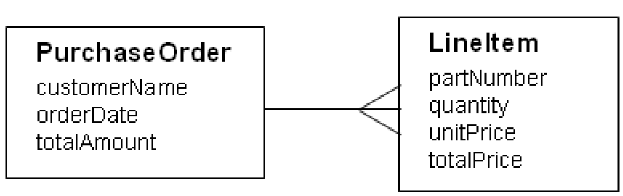
D

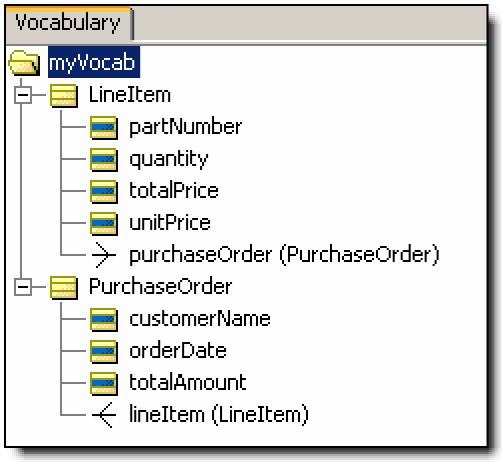
Answers to test yourself questions

For details, see the following topics:

Building the vocabulary

Show me this set of test questions.

1. any three of the following:
   a. provides terms that represent business "things"
   b. provides terms that are used to hold transient (temporary) values within Corticon Studio
   c. provides a federated data model that consolidates entities and attributes from various enterprise data resources
   d. provides a built-in library of literal terms and operators that can be applied to entities and attributes
   e. defines a schema that supplies the contract for sending data to and from a Corticon Decision Service
2. False. The Vocabulary may include transient terms that are used only in rules and that don’t exist in the data model.
3. False. Terms in the data model that are not used by rules do not need to be included in the Vocabulary.
4. False. A Vocabulary may be created before its corresponding object or data model exists.
5. The Vocabulary is an **abstract** model, meaning many of the real complexities of an underlying data model are hidden so that the rule author can focus on only those terms relevant to the rules.
6. The UML model that contains the same types of information as a Vocabulary is called a **Class Diagram**
7. Entities, Attributes, Associations
8. hairColor
9. yellow
10. Attributes
11. Boolean, DateTime, Decimal, Integer, String
12. blue and yellow
13. orange and yellow
14. An Extended Transient Vocabulary term is used when the term is needed to hold a temporary value that is not required to be stored in external data.
15. Associations are **bidirectional** by default
16. cardinality
17. 
18. 
19. Target.source.attribute
20. target
21. 

22. 

23. identify terms, separate terms, assemble and relate terms, diagram vocabulary remove answer from question
24. a
25. operators
26. Rule Language Guide
27. False. Custom Data Types must be based on the 7 base data types. They extend the 7 base data types.
28. b. May match other Custom Data Type Names
29. True
30. value < 10
31. True
32. No
33. 'Airbus'
34. Attribute values are pre-populated in pulldowns based on the enumerated values.
35. Allow you to re-use entities by "bundling" or creating a "sub-set" within the vocabulary. (technically equivalent to packages in Java or namespaces in XML.)
36. True.
37. True.
38. All entities have native attributes, but `Bicyle` = 100% native, the others have 1 native attribute each and 3 inherited. Entities with inherited attributes = `MountainBike`, `RoadBike`, `TandemBike`
39. `cadence`, `gear`, or `speed`
40. True.

Rule scope & context

Show me this set of test questions.

1. 7 root-level entities are present
2. all terms are allowed **except** DVD.actor
3. Movie.supplier
4. **a.** Movie.oscar
   b. Movie.roles
   c. Actor.roles
   d. DVD.supplier
   e. Movie.dVD.extras
   f. Actor.roles.movie.oscar
5. Actor.roles.movie
6. Since the association between Actor and Role is bidirectional, we can use both Actor.roles and **Roles.actor** in our rules.
7. Movie and Award
8. from Movie to Award: goldenGlobe and oscar. From Award to Movie: two unique rolenames exist for this perspective, too, but are not visible in the Vocabulary diagram.
9. The Award entity could be split into two separate entities, or an attribute could be added to Award to identify the *type* of award.
10. Using roles helps to clarify rule context.
11. unique
12. True
13. all examples shown are Boolean expressions
14. can use Movie if it is the root term, or DVD.movie if DVD is the root term The root term can either be Movie or DVD – no conditions in the rule prevent either one from being the root term
15. can use Movie.dVD if Movie is the root term, or DVD if it is the root term The root term can either be Movie or DVD – no conditions in the rule prevent either one from being the root term
16. False. Both Movie and DVD terms in this example are root terms with no relationship to each other.
17. Once for the Movie satisfying the rule conditions and its *associated* DVD
18. Twice: once for each DVD (i.e. the cross product of the DVDs and the Movie satisfying the rule conditions)
19. **a.** High
    b. Low
    c. Low for each DVD
    d. Twice: once for each DVD
    e. Four: each of the 2 rules fired 2 times
    f. cross product
    g. no, each rule should only fire once for the DVD *associated* with the Movie
    h. change the Movie and DVD terms to share the same scope, starting either with Movie as the root term (Movie and Movie.dVD) or DVD as the root term (DVD and DVD.movie)

20. False. Aliases are only *required* to be used in certain circumstances, but they can be used at any time and provide a good way of simplifying rule expressions.
21. Scope is another way of defining a specific **context** or **perspective** in the Vocabulary
22. be updated
23. False. Each alias must be unique and cannot have the same spelling as any term in the Vocabulary.

Rule writing techniques – logical equivalents

Show me this set of test questions.

1. Preconditions act as master rules for all other rules in the same Rulesheet that share the same **scope**
2. An expression that evaluates to a True or False value is called a **Boolean** expression
3. True
4. False. The requirement for complete Values sets only applies to Condition rows.
5. The special term **other** can be used to complete any Condition row values set.
6. not
7. {T, F}
8. all **except** Entity.boolean=F are equivalent, however some expressions are more clear than others!
9. Entity.boolean is probably the best choice since it is the simplest and most straightforward. The other two choices use double negatives which are harder for most people to understand.
10. **a.** OK as is

    b. if the value range is supposed to contain Integer values, then a does not belong. If the range is supposed to contain String values then 1 and a need to be surrounded by single quotes as in {‘1’..’a’, other}

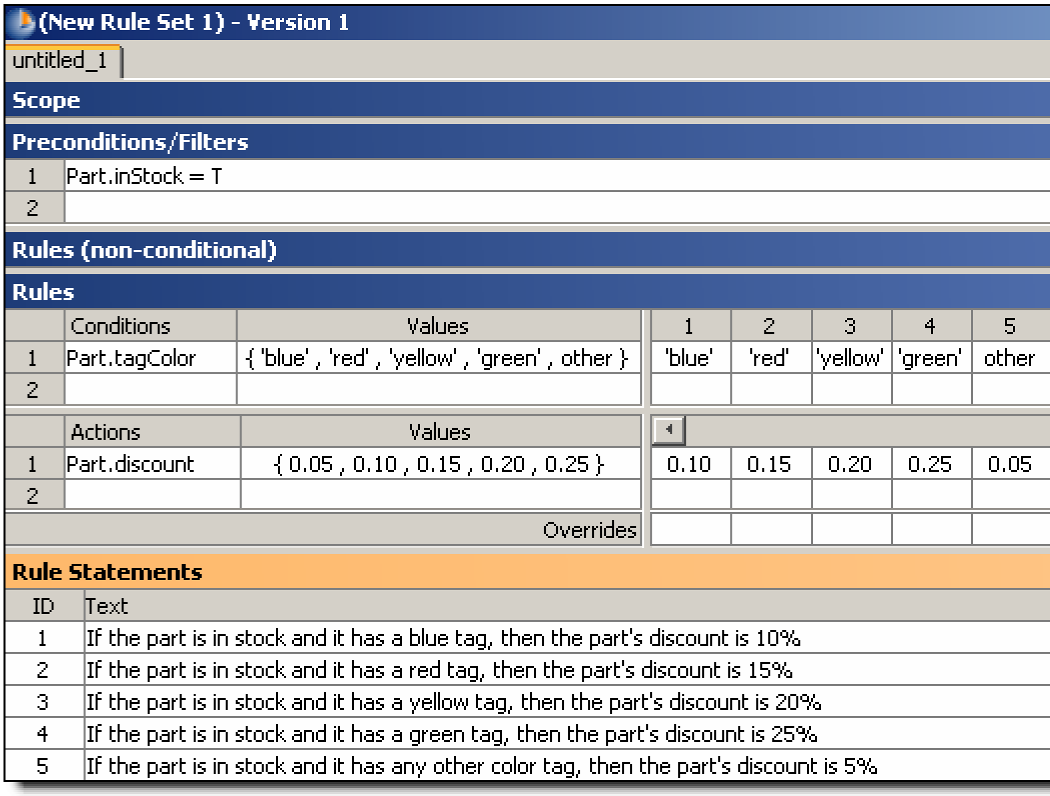
    c. the special word other can’t be used as a range endpoint.

    d. the range contains overlaps between 5 and 10, but this is acceptable in v5.

    e. the range contains an overlap at 10, but this is acceptable in v5.

    f. this is an incomplete set and should be {‘red’, ‘green’, ‘blue’, other}

    g. the range contains overlaps between 3 and 15, but this is acceptable in v5.
11. False. The term other may **not** be used in Action row Values sets since Actions can only assign *specific* values.



12. The Rulesheet would be modeled as shown above.
13. True
14. False. Nonconditional rules are governed by Preconditions on the same Rulesheet only if they share the same scope as the Preconditions.

Collections

Show me this set of test questions.

1. Children of a Parent entity are also known as **elements** of a collection.
2. False. A collection can be comprised of root-level entities.
3. True
4. True
5. Refer to the Rule Language Guide for a full list and description of all collection operators.
6. Rule Language Guide
7. Order total is equal to the sum of the line item prices on the order.
8. items
9. one-to-many (1->*)
10. It is not an acceptable replacement since the use of any collection operator requires that the collection be represented by an alias.

11. set the navigability of the association between Order and LineItem to Order->lineItem. In other words, make the association one-directional from Order to LineItem.
12. Optional, Convenient
13. A collection alias is not required in this case because no collection operator is being applied to the collection.
14. forAll
15. exists
16. **a.** aroles ->size > 3 *where aroles is an alias for Actor.roles*
    b. mdvd ->isEmpty *where mdvd is an alias for Movie.dVD*
    c. mdextras ->exists(deletedScenes=T) *where mdextras is an alias for Movie.dVD.extras*
    d. mgglobes ->exists(win=T) *where mgglobes is an alias for Movie.goldenGlobe*
    e. mroles ->size > 15 *where mroles is an alias for Movie.roles*
    f. mdvd.quantityAvailable ->sum >= 100 *where mdvd is an alias for Movie.dVD*
    g. mdvd.quantityAvailable ->sum < 2 *where mdvd is an alias for Movie.dV*D
    h. mdsuppliers ->size > 1 *where mdsuppliers is an alias for Movie.dVD.supplier*
17. Actor, Distributor, DVDExtras
18. Actor, Movie
19. The ->forAll operator tests whether **all** elements of a collection satisfy a condition. The ->exists operator tests whether **at least one** element of a collection satisfies a condition.
20. The ->notEmpty operator tests whether a collection is not empty, meaning there is at least one element in the collection. The ->isEmpty operator tests whether a collection is empty, meaning there are no elements in the collection.
21. To ensure that the system knows precisely which collection (or copy) you are referring to in your rules, it's necessary to use a unique alias to refer to each collection.

Rules containing calculations & equations

Show me this set of test questions.

1. comparison in Preconditions and Conditions, assignment in Nonconditionals and Actions
2. **a.** 10
   b. 13
   c. 22
   d. 24
   e. 0
3. This assignment is not valid since an Integer attribute cannot contain the digits to the right of the decimal point in a Decimal attribute value.
4. **a.** Integer
   b. String

c. Boolean
d. Decimal
e. Boolean
f. Boolean
g. Boolean

5. a. valid
   b. invalid
   c. valid
   d. valid
   e. valid
   f. invalid
   g. valid

6. The part of Corticon Studio that checks for syntactical problems is called the **Parser**.
7. False. Although the Parser in Corticon Studio is very effective at finding syntactical errors, it is not perfect and can't anticipate all possible combinations of the rule language.
8. This Filter tests if the difference between the current year and the year a movie was released is more than 10 years.
9. This Condition tests if the total quantity of DVDs available divided by the number of DVD versions of a movie is less than or equal to 50,000 or greater than 50,000. This same calculation could be performed by using the ->avg operator by itself.
10. If the average quantity available of a DVD is greater than 50,000 for a movie that is more than 10 years old, then flag the movie with a warning.
11. a. False
    b. False
    c. True
    d. True
    e. True
    f. False
    g. False
    h. True

Rule dependency: dependency & inferencing

Show me this set of test questions.

1. Inferencing involves only a single pass through rules while looping involves multiple passes.
2. A loop that does not end by itself is known as an **infinite** loop.
3. A loop that depends logically on itself is known as a single-rule or **trivial** loop.

4. False. The Rulesheet must have looping enabled in order for the loop detector to notice mutual dependencies.
5. False. The Check for Logical Loops tool can only detect and highlight loops, not fix them.
6. No, looping is neither required nor wanted for these rules. Normal inferencing will ensure the correct sequence of execution of these rules.
7. Yes, having this Rulesheet configured to Process All Logical Loops enables an infinite loop between rule 1 and rule 2 for DVDs meeting the conditions for that rule.
8. Rule 1 would change the DVD's price tier value to Medium, and then rule 2 and rule 1 would execute in an infinite loop, incrementing the DVD's quantity available by 25,000 repeatedly until terminating after the maxloop property setting number of iterations.
9. Process all logical loops
10. Process multi-rule loops only
11. A **dependency network** determines the sequence of rule execution and is generated when a *Rulesheet* is **saved**.

Preconditions & filters

Show me this set of test questions.

1. True
2. False - precondition behavior is optional
3. True - a filter will only "apply" to other rules that share the same scope. This means that other rules acting on data outside the filter's scope will be unaffected.
4. and'ed
5. False. Preconditions/Filters are not stand-alone rules.
6. c
7. a
8. no
9. True
10. full
11. full filter only
12. precondition AND full filter
13. f and d
14. a
15. Oscars:
    a. Movie 1; DVD 1; Oscars 1, 2, 3, 4, 5
    b. Movie 1; DVD 1; Oscars 1, 2, 3, 4, 5
    c. Movie 1; DVD 1; Oscar 2
    d. Movie 1; DVD 1; Oscars 1, 2, 3, 4, 5
    e. Movie 1; DVD 1; Oscars 1, 2

f. none

g. none

h. Movie 1; DVD 1; Oscars 1, 2, 3, 4, 5

i. Movie 1; DVD 1; Oscars 1, 2, 3, 4, 5

j. Movie 1; DVD 1; Oscars 1, 2, 3, 4, 5

k. none

l. Movie 1; DVD 1; Oscars 1, 2, 3, 4, 5

m. none

n. Movie 1; DVD 1

o. Movie 1; DVD 1; Oscars 1, 2, 3, 4, 5

Recognizing & modeling parameterized rules

Show me this set of test questions.

1. When several rules use the same set of Conditions and Actions, but different values for each, we say that these rules share a common **pattern**.
2. Another name for the different values in these expressions is **parameter**.
3. False. It is usually easier to model them as Conditions and Actions that use values sets.
4. You may accidentally introduce ambiguities into your rules.
5. **X** customers buy more than $**Y** of product each year
6. Type of customer: {'Platinum', 'Gold', 'Silver', 'Bronze'} and spend amount: {25000..50000, (50000..75000], (75000..10000], >100000}. Depending on how the rules are modeled, one of these values sets will be part of a Condition and should be completed with the special word `other`.
7. These parameters may be maintained in the values sets of an individual Rulesheet, which is easy to perform, but makes reuse more difficult. They may be maintained as Custom Data Types (Enumerated) in the Vocabulary, which makes reuse easier.

Writing rules to access external data

Show me this set of test questions.

1. Rule scope determines which data is processed during rule execution.
2. So a Database-enabled Rulesheet does not inadvertently retrieve all the corresponding data in a database, which could be a lot of data!
3. It is extended to the database
4. True. Only root-level entities need to be extended – all other entities are extended automatically because their scope is reduced enough to not be as concerned about massive amounts of retrieved data.

5. See the tutorial *Using EDC*, and the *Integration and Deployment Guide*.
6. No. In general, the rule modeler does not need to worry about where data is stored.
7. Yes. The exception is when rules are written using root-level terms. If the Rule Set is Database-enabled, then these root-level terms may need to be extended to the database.

Logical analysis & optimization

Show me this set of test questions.

1. They have the same Conditions but different Actions.
2. All combinations of possible values from the Conditions' values sets are covered in rules on the Rulesheet.
3. No, not all ambiguous rules are wrong or need to be resolved before deployment. Ambiguities may exist in *`Rulesheet`*s where there are rules that are completely unrelated to each other. In those cases, it may be appropriate for both rules to fire if the Conditions for both are met.
4. No, not all incompletenesses are wrong or need to be resolved before deployment. Incomplete *`Rulesheet`*s may be missing combinations of Conditions that cannot or should not occur in real data. In those cases, rules for such combinations may not make sense at all.
5. Conflict Checker – second icon; Compression Tool – fifth icon; Expansion Tool – first icon; Collapse Tool – third icon; Conflict Filter – sixth icon.
6. An ambiguity can be resolved by 1) making the Actions match for both rules, or 2) by setting an override for one of the rules.
7. False. Defining an override does not specify an execution sequence, but rather specifies that the rule with the override will always fire **instead of** the rule being overridden when the Conditions they share are satisfied.
8. False. The Completeness Checker will auto-complete the Condition's value set prior to inserting missing rules. This ensures the *`Rulesheet`*, post-application of the Completeness Check, is truly complete.
9. The Completeness Checker will detect *`Rulesheet`* incompleteness caused by an incomplete values set because it will automatically complete the value set first before inserting missing columns.
10. Yes. One rule can override multiple other rules by holding the **Ctrl** key to multi-select overrides from the drop-down.
11. No, overrides are not transitive and must be specified directly between all applicable rules.
12. No, rules created by the Completeness Checker may be comprised of combinations of Conditions that cannot or should not occur in real data. In those cases, rules for such combinations may not make sense at all.
13. A dash specifies that the Condition should be ignored for this rule.
14. False. The Expansion Tool merely expands a *`Rulesheet`* so that all sub-rules are visible. The results can be reversed by using the Collapse Tool.
15. True. It *may* be reversible using **Undo**, or by manually removing redundant sub-rules after expansion.
16. 64 (4 x 4 x 4)

17. It is not necessary to assign actions for a rule column if that combination of conditions cannot or should not exist in real data. We recommend disabling columns added by the Completeness Check that you determine need no Actions.
18. They may be disabled, deleted, or just left as-is with no Actions (but this last option is not recommended since it will still cause activity which can impact performance).
19. Compression Tool
20. The compression performed by the Completeness Checker is designed to reduce a large set of missing rules into the smallest set of *non-overlapping* columns, while the compression performed by the Compression Tool is designed to reduce the number of rules into the smallest set of general rules (i.e. create columns with the most dashes).
21. Even very large databases may still not contain all possible combinations of data necessary to verify *Rulesheet* completeness. In short, they may be incomplete themselves.
22. Renumber the rules and potentially ask you to consolidate Rule Statements if duplicate row numbers result from the renumbering.
23. Subsumation occurs when the Compression Tool detects that a more general rule expression includes the logic of a more specific rule expression. In this case, the more specific rule can be removed.

Ruleflow versioning & effective dating

Show me this set of test questions.

1. False. *Ruleflow* Effective and Expiration dates may be assigned singly.
2. False. *Ruleflow* Effective and Expiration dates may be assigned singly.
3. False. *Ruleflow* Version numbers are optional.
4. *Ruleflow* Properties, or click on the Properties window in Corticon Studio.
5. False. A *Ruleflow* Version number may only be raised, not lowered.
6. False. *Ruleflow* Effective and Expiration dates are optional.

Troubleshooting rulesheets

Show me this set of test questions.

1. Troubleshooting is based on the principle that *Rulesheet*s behave the same way when tested in Corticon Studio as when executed on **Server**
2. The first step in troubleshooting a suspected rule problem is to reproduce the behavior in a Corticon Studio **Test**.
3. In the integration with *Corticon Server*.
4. All of them!
5. The specific rule where execution behavior begins acting abnormally is called the **breakpoint**.
6. True. In v5, partial rule firing is allowed.
7. Disabling Rulesheets; Filters, Nonconditions, Conditions, Action rows; or rule columns

8. A dark gray-colored Rulesheet tab indicates that Rulesheet has been **disabled**.
9. d
10. loglevel is located in `CcCommon.properties` inside `CcConfig.jar`.
11. loglevel should be set to `INFO`
12. True.
13. Vocabulary (`.ecore`), *`Rulesheet`* (`.ers`), and a *`Ruletest`* (`.ert`) and the *`Ruleflow`* (`.erf`) if any. We also need to know the Corticon Studio version you are using.
14. d and h

E

Third party acknowledgments

One or more products in the Progress Corticon v5.3.3 release includes third party components covered by licenses that require that the following documentation notices be provided:

Progress Corticon v5.3.3 incorporates Apache Commons Discovery v0.2 from The Apache Software Foundation. Such technology is subject to the following terms and conditions: The Apache Software License, Version 1.1 - Copyright (c) 1999-2001 The Apache Software Foundation. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowlegement: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgement may appear in the software itself, if and wherever such third-party acknowledgements normally appear.

4. The names "The Jakarta Project", "Commons", and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache" nor may "Apache" appear in their names without prior written permission of the Apache Group.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see <http://www.apache.org/>.

Progress Corticon v5.3.3 incorporates Apache SOAP v2.3.1 from The Apache Software Foundation. Such technology is subject to the following terms and conditions: The Apache Software License, Version 1.1 Copyright (c) 1999 The Apache Software Foundation. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "SOAP" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org. 5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation. THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see <http://www.apache.org/>.

Progress Corticon v5.3.3 incorporates DOM4J v1.6.1. Such technology is subject to the following terms and conditions: Project License BSD style license Copyright 2001-2005 (C) MetaStuff, Ltd. All Rights Reserved.

Redistribution and use of this software and associated documentation ("Software"), with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain copyright statements and notices. Redistributions must also contain a copy of this document.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Progress Corticon v5.3.3 incorporates JDOM v1.0 GA. Such technology is subject to the following terms and conditions: $Id: LICENSE.txt,v 1.11 2004/02/06 09:32:57 jhunter Exp $ - Copyright (C) 2000-2004 Jason Hunter and Brett McLaughlin. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact <request_AT_jdom_DOT_org>.

4. Products derived from this software may not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management <request_AT_jdom_DOT_org>.

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the JDOM Project (http://www.jdom.org/)."

Alternatively, the acknowledgment may be graphical using the logos available at http://www.jdom.org/images/logos. THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This software consists of voluntary contributions made by many individuals on behalf of the JDOM Project and was originally created by Jason Hunter <jhunter_AT_jdom_DOT_org> and Brett McLaughlin <brett_AT_jdom_DOT_org>. For more information on the JDOM Project, please see <http://www.jdom.org/>.

Progress Corticon v5.3.3 incorporates Saxpath v1.0. Such technology is subject to the following terms and conditions: Copyright (C) 2000-2002 werken digital. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.

3. The name "SAXPath" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@saxpath.org.

4. Products derived from this software may not be called "SAXPath", nor may "SAXPath" appear in their name, without prior written permission from the SAXPath Project Management (pm@saxpath.org).