

Corticon Studio:

Rule Modeling Guide

Copyright

© 2017 Progress Software Corporation and/or one of its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Deliver More Than Expected, Icenium, Kendo UI, Making Software Work Together, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustCode, JustDecompile, JustMock, JustTrace, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Sitefinity, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

Please refer to the Release Notes applicable to the particular Progress product release for any third-party acknowledgements required to be provided in the documentation associated with the Progress product.

Updated: 2017/04/24

Table of Contents

Chapter 1: Building the Vocabulary.....	11
What is a Vocabulary?.....	11
Starting from scratch.....	13
Designing the Vocabulary.....	14
Step 1: Identifying the Terms	14
Step 2: Separating the Generic Terms from the Specific	14
Step 3: Assembling and Relating the Terms	14
Step 4: Diagramming the Vocabulary.....	16
Modeling the Vocabulary in Corticon Studio.....	17
Importing an OpenEdge Business Rules Vocabulary Definition (BRVD) file.....	19
Locking and unlocking Corticon Vocabularies.....	21
Applying an updated BRVD to a Vocabulary.....	23
Accessing the Vocabulary metadata of a Decision Service.....	25
Custom Data Types.....	28
Constraint Expressions.....	28
Enumerations.....	30
Relaxing enforcement of Custom Data Types.....	42
Domains.....	44
Domains in a Rulesheet.....	45
Domains in a Ruletest.....	46
Support for inheritance.....	47
Inherited Attributes	48
Inherited Associations	48
Controlling the tree view.....	49
Using aliases with inheritance.....	49
Inheritance's effects on rule execution.....	50
Inheritance and Java object messaging.....	53
Test yourself questions: Building the vocabulary.....	54
 Chapter 2: Rule scope and context.....	 59
Rule scope.....	65
Aliases.....	68
Scope and perspectives in the vocabulary tree.....	69
Roles.....	71
Technical aside.....	76
Test yourself questions: Rule scope and context.....	77
 Chapter 3: Rule writing techniques and logical equivalents.....	 83

Working with rules and filters in natural language.....	84
Filters vs. conditions.....	87
Qualifying rules with ranges and lists.....	89
Using ranges and lists in conditions and filters.....	89
Using ranges and value sets in condition cells.....	91
Creating rules by importing an Excel worksheet.....	104
Using standard boolean constructions.....	110
Finding references to an entity, attribute, or association in a project.....	110
Embedding attributes in posted rule statements.....	112
Including apostrophes in strings.....	114
Test yourself questions: Rule writing techniques and logical equivalents.....	114
 Chapter 4: Collections.....	117
Understanding how Corticon Studio handles collections.....	118
Visualizing collections.....	118
A basic collection operator.....	119
Filtering collections.....	120
Using aliases to represent collections.....	120
Sorted aliases.....	129
Advanced collection sorting syntax.....	132
Statement blocks.....	134
Using sorts to find the first or last in grandchild collections.....	136
Singletons.....	136
Special collection operators.....	139
Universal quantifier.....	139
Existential quantifier.....	141
Another example using the existential quantifier.....	144
Aggregations that optimize database access.....	150
Test yourself questions: Collections.....	150
 Chapter 5: Rules containing calculations and equations.....	155
Terminology.....	155
Operator precedence and order of evaluation.....	156
Datatype compatibility and casting.....	158
Datatype of an expression.....	161
Defeating the parser.....	162
Manipulating datatypes with casting operators.....	162
Supported uses of calculation expressions.....	164
Calculation as a comparison in a precondition.....	165
Calculation as an assignment in a noncondition.....	166
Calculation as a comparison in a condition.....	166
Calculation as an assignment in an action.....	168
Unsupported uses of calculation expressions.....	168

Calculations in value sets and column cells.....	168
Calculations in rule statements.....	169
Test yourself questions: Rules containing calculations and equations.....	169
Chapter 6: Rule dependency: Chaining and looping.....	173
What is rule dependency?.....	173
Forward chaining.....	174
Rulesheet processing: modes of looping.....	175
Types of loops.....	176
Looping controls in Corticon Studio.....	180
Identifying loops.....	180
Looping examples.....	182
Given a date, determine the next working day.....	183
Removing duplicated children in an association	186
Using conditions as a processing threshold.....	190
Test yourself questions: Rule dependency: chaining and looping.....	192
Chapter 7: Filters and preconditions	193
What is a filter?.....	193
Full filters.....	195
Limiting filters.....	197
Database filters.....	203
What is a precondition?.....	205
Summary of filter and preconditions behaviors.....	210
Performance implications of the precondition behavior.....	210
Using collection operators in a filter.....	212
Location matters.....	213
Multiple filters on collections.....	216
Filters that use OR.....	218
Test yourself questions: Filters and preconditions.....	218
Chapter 8: Recognizing and modeling parameterized rules.....	221
Parameterized rule where a specific attribute is a variable (or parameter) within a general business rule.....	221
Parameterized rule where a specific business rule is a parameter within a generic business rule....	223
Populating an AccountRestriction table from a sample user interface	225
Test yourself questions: Recognizing and modeling parameterized rules.....	227
Chapter 9: Writing Rules to access external data.....	229
A Scope refresher.....	231
Showing vocabulary details in a Rulesheet.....	231
Clearing database mappings in the vocabulary.....	233

Validation of database properties.....	235
Enabling Database Access for Rules using root-level Entities.....	235
Testing the Rulesheet with Database Access disabled	237
Testing the Rulesheet with Database Access enabled.....	238
Optimizing Aggregations that Extend to Database.....	243
Precondition and Filters as query filters.....	244
Filter Query qualification criteria.....	245
Operators supported in Query Filters.....	245
Using multiple filters in Filter Queries.....	246
Defining database caching in Studio.....	247
Test yourself questions: Writing rules to access external data.....	251
 Chapter 10: Logical analysis and optimization.....	253
Testing, analysis, and optimization.....	253
Scenario testing.....	254
Rulesheet analysis and optimization.....	254
Traditional means of analyzing logic.....	255
Flowcharts.....	256
Test databases.....	258
Validating and testing Rulesheets in Corticon Studio.....	261
Expanding rules.....	261
The conflict checker.....	263
The completeness checker.....	268
Logical loop detection.....	275
Testing rule scenarios in the Ruletest Expected panel.....	275
Navigating in Ruletest Expected comparison results.....	275
Reviewing test results when using the Expected panel.....	276
Techniques that refine rule testing.....	278
Optimizing Rulesheets.....	283
The compress tool.....	283
Producing characteristic Rulesheet patterns.....	285
Compression creates sub-rule redundancy.....	288
Effect of compression on Corticon Server performance.....	288
Precise location of problem markers in editors.....	289
Test yourself questions: Logical analysis and optimization.....	289
 Chapter 11: Advanced Ruleflow techniques and tools.....	291
Using a Ruleflow in another Ruleflow.....	291
Conditional Branching in Ruleflows.....	294
Example of Branching based on a Boolean.....	297
Example of Branching based on an Enumeration.....	302
Logical analysis of a branch container.....	306
How branches in a Ruleflow are processed.....	309

Generating Ruleflow dependency graphs.....	310
Ruleflow versioning and effective dating	316
Setting a Ruleflow version.....	316
Major and minor versions.....	316
Setting effective and expiration dates.....	317
Setting a specific target date for a Ruletest	317
Test yourself questions: Ruleflow versioning and effective dating.....	318
Chapter 12: Localizing Corticon Studio	319
Localizing the Vocabulary.....	319
Localizing the Rulesheet.....	321
Chapter 13: The Corticon Studio reporting framework.....	323
Chapter 14: Applying logging and override properties to Corticon Studio and its built-in Server.....	327
Chapter 15: Troubleshooting Rulesheets and Ruleflows.....	329
Where did the problem occur?.....	330
Using Corticon Studio to reproduce the behavior.....	330
Running a Ruletest in Corticon Studio	330
Analyzing Ruletest results.....	333
Tracing rule execution.....	334
Identifying the breakpoint.....	334
At the breakpoint.....	335
Partial rule firing.....	336
Initializing null attributes.....	337
Handling nulls in compare operations.....	337
Test yourself questions: Troubleshooting rulesheets and ruleflows.....	339
Appendix A: Standard Boolean constructions.....	341
Boolean AND.....	341
Boolean NAND.....	343
Boolean OR.....	344
Boolean XOR.....	344
Boolean NOR.....	345
Boolean XNOR.....	346
Appendix B: Answers to test-yourself questions.....	347
Test yourself answers: Building the vocabulary.....	347

Test yourself answers: Rule scope and context.....	349
Test yourself answers: Rule writing techniques and logical equivalents.....	351
Test yourself answers: Collections.....	352
Test yourself answers: Rules containing calculations and equations.....	353
Test yourself answers: Rule dependency: dependency and inferencing.....	354
Test yourself answers: Filters and preconditions.....	355
Test yourself answers: Recognizing and modeling parameterized rules.....	356
Test yourself answers: Writing rules to access external data.....	356
Test yourself answers: Logical analysis and optimization.....	357
Test yourself answers: Ruleflow versioning and effective dating.....	358
Test yourself answers: Troubleshooting rulesheets.....	358
 Appendix C: Access to Corticon knowledge resources.....	 361

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?](#)
- [Designing the Vocabulary](#)
- [Modeling the Vocabulary in Corticon Studio](#)
- [Custom Data Types](#)
- [Domains](#)
- [Support for inheritance](#)
- [Test yourself questions: Building the vocabulary](#)

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:

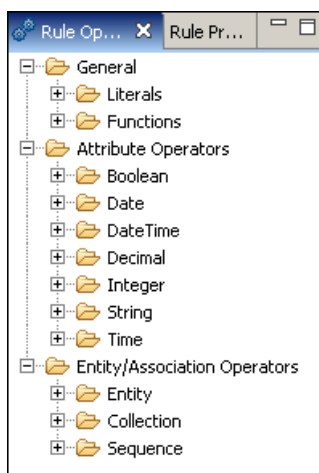
- Provides terms that represent business “things”. Throughout the product documentation, we will refer to these things as *entities*, and properties or characteristics of these things as *attributes*. Entities and their attributes in underlying data sources (such as tables in a relational database or 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:

- `itemSubTotal` is equal to the product of `itemCount` and `itemPrice`
- `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 the note on [Transients](#).

- 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 (Ruleflows 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 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 -- two Boeing 747s and one 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: Identifying 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

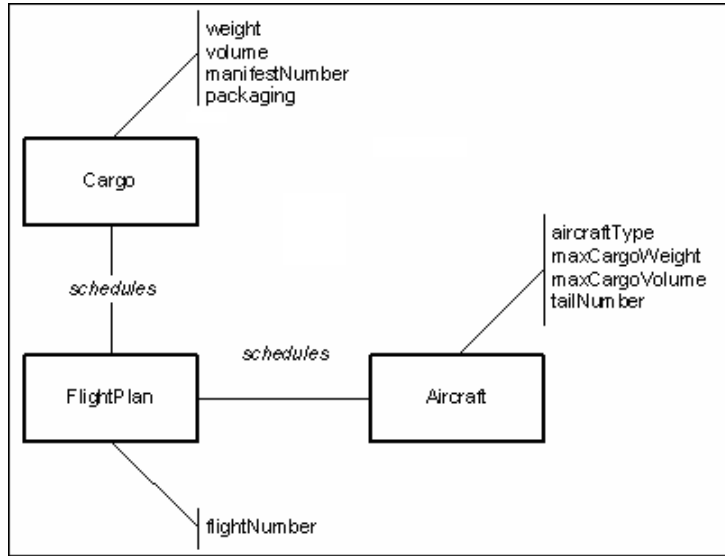
Entity: flight plan

Attributes: flight number

Step 4: Diagramming the Vocabulary

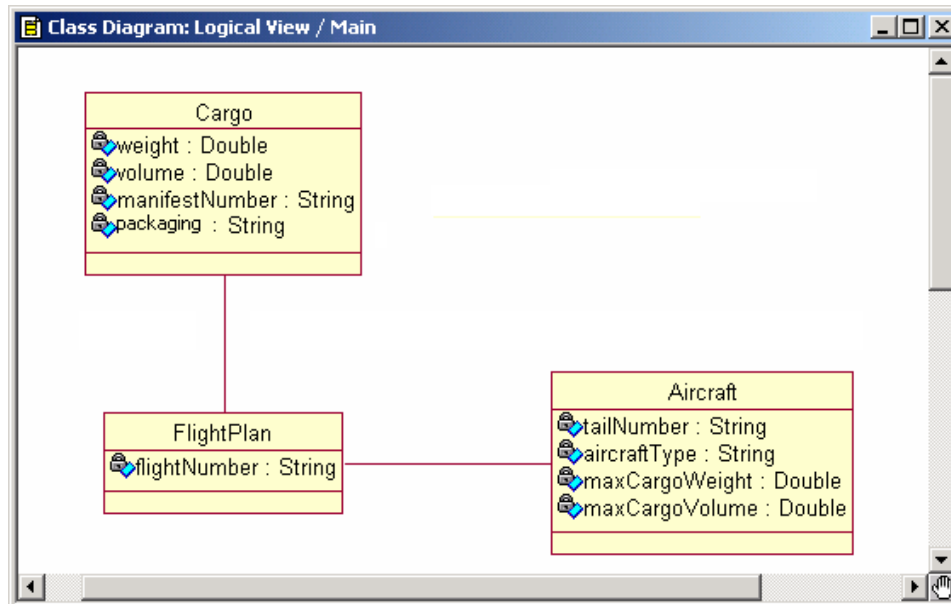
Using this breakdown, we can sketch a simple Fact Model that illustrates the entities and their relationships, or *associations*. In our Fact Model, we will represent entities as rectangular boxes, associations between entities as straight lines connecting the entity boxes, and entity-to-attribute relationships 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 "Vocabulary" chapter of the *Quick Reference Guide* for complete details on building a Vocabulary inside Studio.

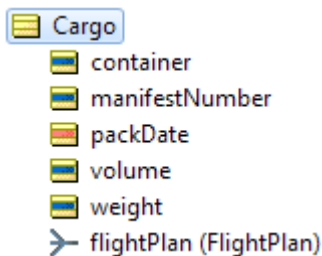
The following considerations apply to this transformation process:

- 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 are classified according to the method by which their values are assigned. They are either:
 - **Base** -- Values are obtained directly from input data or request message, or
 - **Transient** -- Created, derived, or assigned by rules in Studio.

Note:

Transient attributes carry or hold values while rules are executing within a single Rulesheet. Since XML messages returned by a Decision Service do not contain 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 show the rule modeler which attributes are base and which are transient, Corticon Studio adds an orange bar to transient attributes, as shown here for `packDate`:



XML response messages created by Corticon Server will not contain the `packDate` attribute.

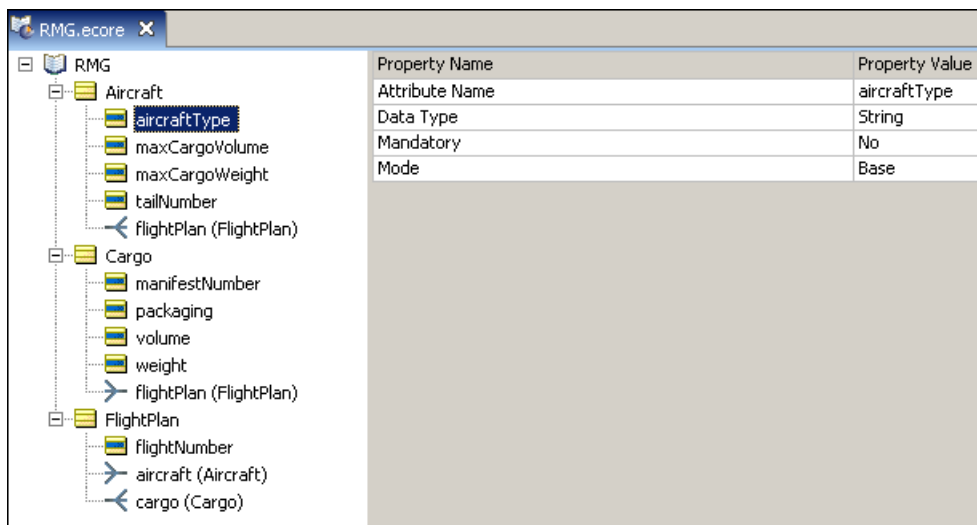
It is a good idea to use a naming convention that distinguishes transient attributes from base attributes. For example, you could start a transient attribute's name with `t_` such as `t_packDate`. We caution against modifying the names of terms so that they are cryptic. The intent is to express them in a language accessible to business users, as well as 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 may use the Vocabulary term `flightPlan.aircraft.tailNumber` but may not use `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



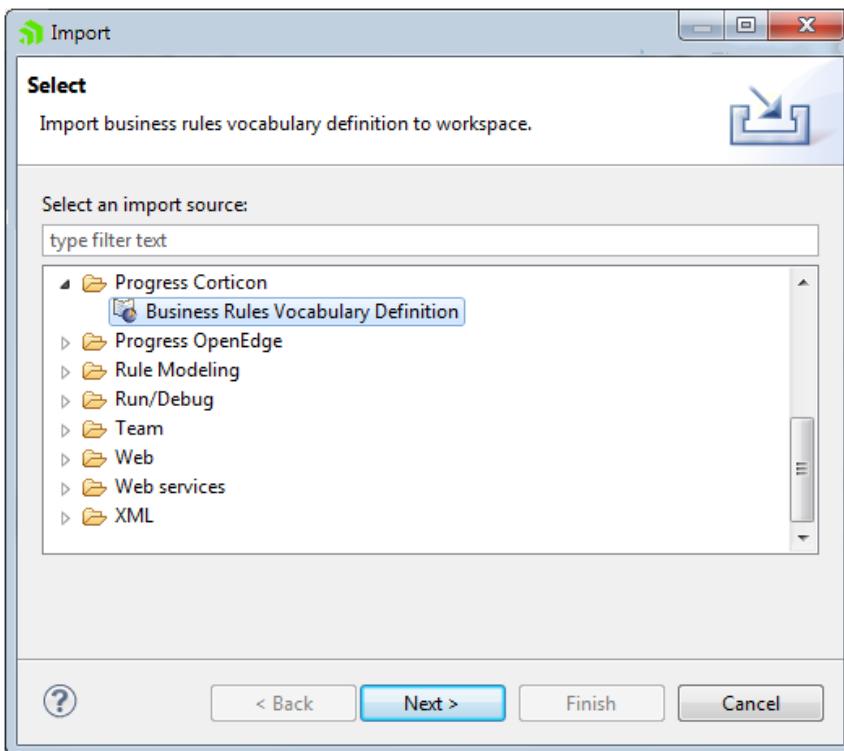
Note: In this figure, Corticon Studio Vocabulary is hiding its details. When changed to show Vocabulary details, the list exposes properties related to mapping.

Importing an OpenEdge Business Rules Vocabulary Definition (BRVD) file

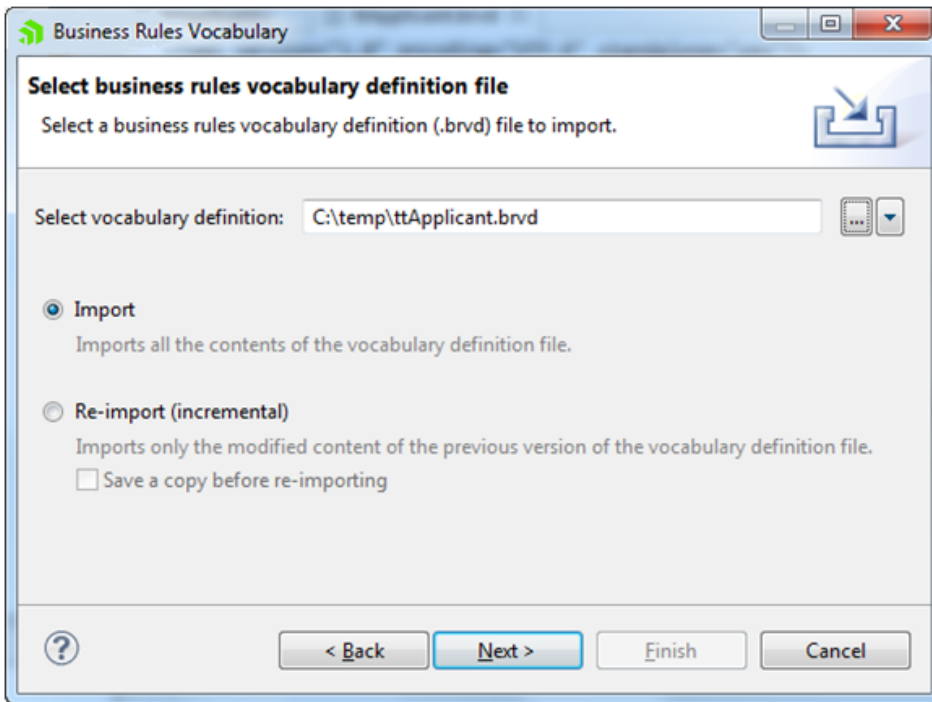
OpenEdge developers can use Corticon for their business rules, using Progress Developer Studio to integrate their ABL projects with Corticon Decision Services. A schema exported from Progress OpenEdge can be imported and used as the basis for Vocabulary entities and attributes in Corticon Studio.

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:

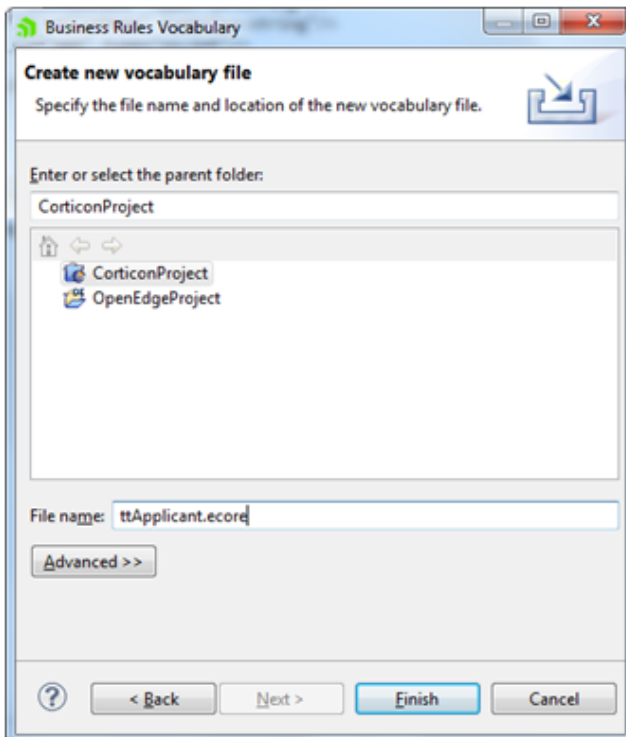


9. In the **Business Rules Vocabulary** dialog, locate the `.brvd` file -- in this example, `ttApplicant.brvd`, that was staged in the `c:\temp` folder -- that was created in OpenEdge.
10. Select **Import**, as shown:



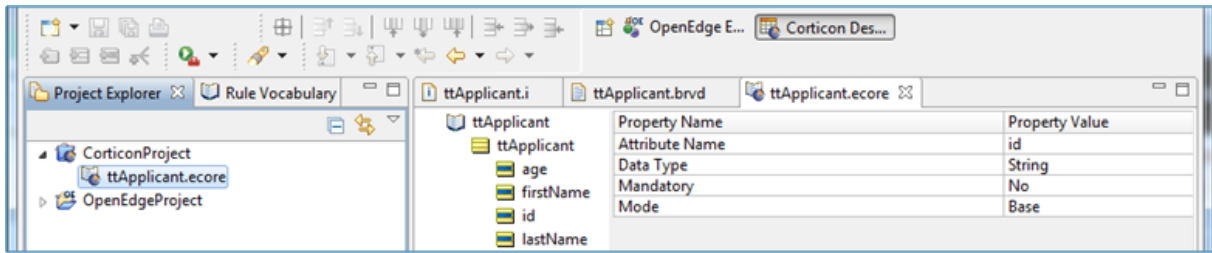
11. Click **Next**.

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.

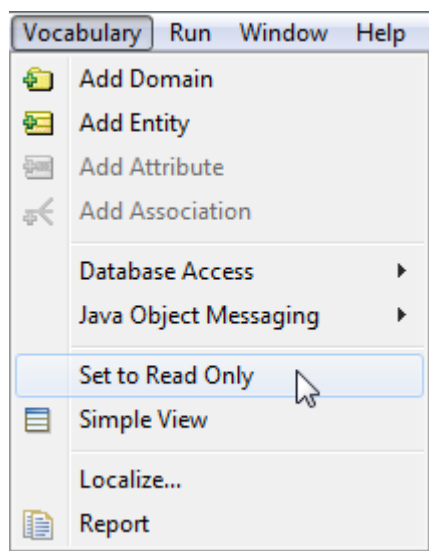
You can now create and test Rulesheets and Ruleflows, and then publish a Decision Service to Corticon Server (such as the one that runs in an OpenEdge Web Server).

Locking and unlocking Corticon Vocabularies

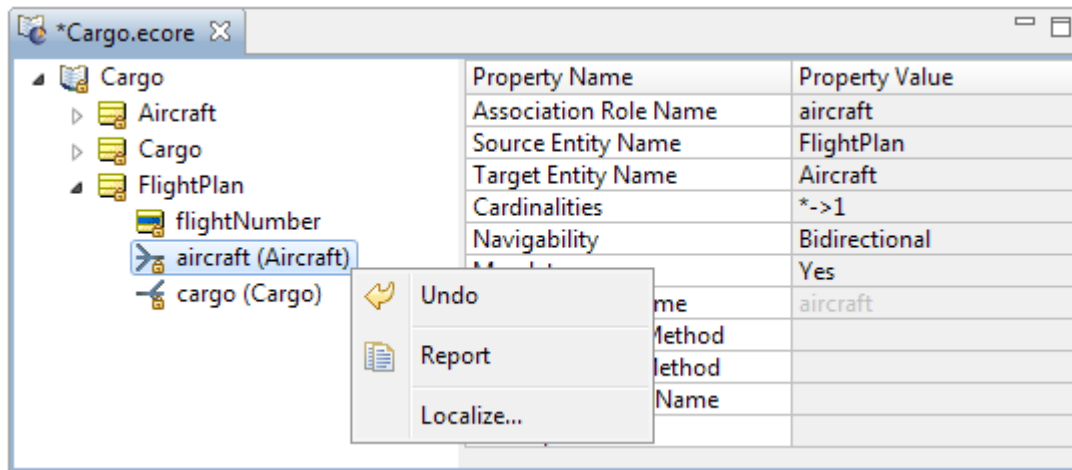
There are advantages to locked and unlocked Vocabularies:

- When your rule development has many Rulesheets and Ruletests, you might want to add a measure of control over Vocabulary changes. Locking the Vocabulary prevents you from accidentally changing an entity or attribute, thereby invalidating rules and calls from databases.
- When OpenEdge developers use Corticon for their business rules, they use 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, attributes, and associations as read-only. This protection keeps you from accidentally modifying the Vocabulary such that it no longer matches its source in OpenEdge. If you want to define transient attributes, constraint expressions, or make other modifications to the vocabulary in Corticon you will need to unlock it. When doing so, you need to be sure that your modifications do not make the vocabulary incompatible with its source in OpenEdge.

You can choose to lock a Vocabulary in Studio by selecting the **Set to Read Only** option, as shown:

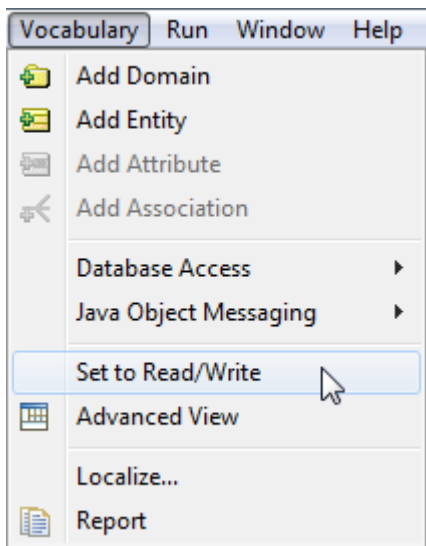


Once locked, all the icons in the Vocabulary display a padlock symbol.

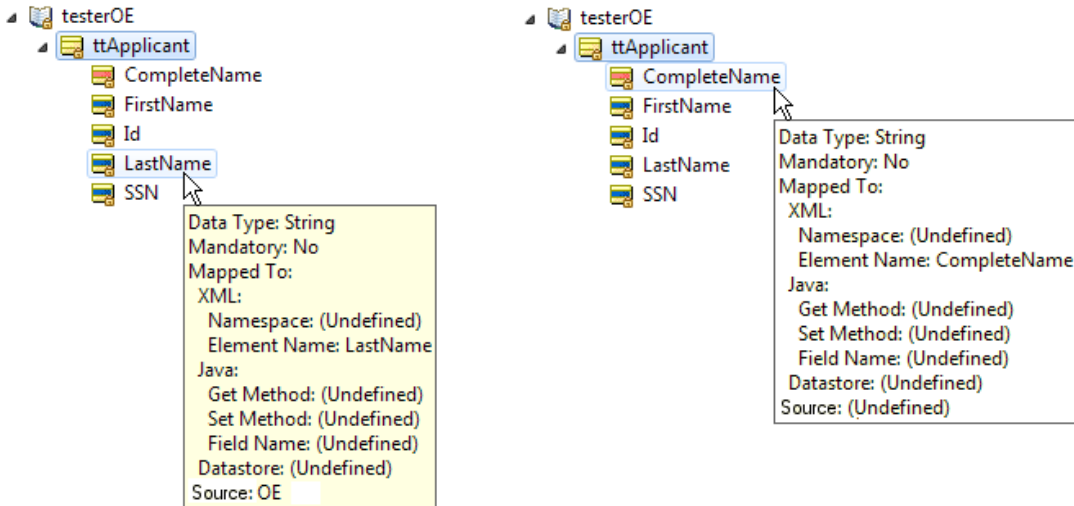


In a locked Vocabulary, all functions in the Vocabulary editor are display-only, including Custom Data Types and Database Access. **Undo** (and **Redo**) options will toggle the mode until the Vocabulary file is saved.

A locked Vocabulary can be unlocked by selecting the **Set to Read/Write** option.



Note: In Vocabularies created through BRVD import, changes that you make when unlocked and then locked again, continue to reflect the OE origin of BRVD imported attributes and entities.



You should never rename attributes or entities that are bound to OpenEdge or EDC resources as they would likely cause data synchronization problems.

Applying an updated BRVD to a Vocabulary

If you make changes to your OpenEdge application, you might need to regenerate the BRVD file, and then re-import it into Corticon to update your Vocabulary.

A re-imported BRVD file impacts any changes you might have made if you set the imported Vocabulary into Read/Write mode. While you cannot change the Vocabulary when it is in Read Only mode, the re-import actions will proceed on a locked or unlocked Vocabulary.

Note: A re-import of a BRVD file changes the current Vocabulary. It is a good practice to choose **Save a copy before re-importing** in the import dialog to backup the existing Vocabulary before applying changes to it, especially if you entered Read/Write mode and made changes.

Before the re-import action is applied, a dialog displays the entities, attributes, and associations in the Vocabulary and how they will be impacted.

The status and intended actions on Vocabulary entities, attributes, and associations during a re-import can be the following:

- **Match-** No action if the imported element and the existing element are identical in the following ways:
 - Entity: Same name and same origination (BRVD import or user-defined action)
 - Attribute: Same name, same datatype, and same origination (BRVD import or user-defined action)
 - Association: Same name, same target entity, and same cardinality
- **UserDefined** - No action if the existing element was created in Corticon.
- **Add** - The imported element is entered into the Vocabulary and marked as originating from the BRVD as follows:
 - Entity: The entity name in the BRVD does not match an existing entity in the Vocabulary

- **Attribute:** The attribute name in the BRVD does not match an existing attribute in the corresponding entity in the Vocabulary.
- **Association:** The association name in the BRVD does not match an existing association in the Vocabulary
- **Remove** - Deletes the existing element if it was previously created from a BRVD element, but is not in the import file.
- **Remove/Re-add** - Removes the existing element and then recreates it from its definition in the import file, under the following conditions:
 - **Attribute:** Same name but the datatype is different. (When the datatype is an enumerated list or constraint expression, if that custom data type has the same base data type as the imported attribute in the BRVD, it is a Match.)
 - **Association:** Same name but with a different Target Entity.

Note: The Remove/Re-Add action does not apply to an Entity that originated from a BRVD. A re-import can either Add or Remove a BRVD originated entity.

- **Merge** - Revises an existing element marked as created locally that is now a BRVD element in the import file. The existing element will transform to be defined as an imported element (originated from BRVD) and marked as Merge, as follows:
 - **Entity:** Same name, but created in the Vocabulary editor.
 - **Attribute:** Same name, same datatype, but created in the Vocabulary editor. If the existing element was defined as transient mode, it is changed to base mode.
 - **Association:** Same name, same target entity, but different cardinality or created in the Vocabulary editor.

The dialog lets you choose to **Copy to Clipboard** to retain the action list as this information will not be available once you proceed to apply the re-import actions.

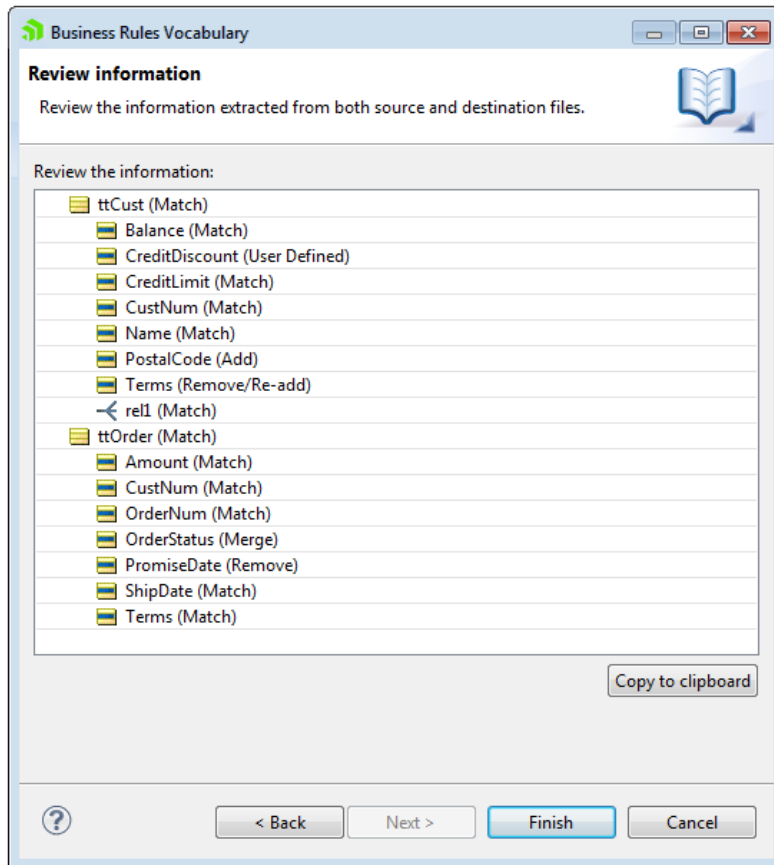
Note: If you choose to **Cancel** at this point, nothing was changed.

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

Example

The following figure shows how some BRVD changes to attributes are displayed for review:

Figure 5: Review information before re-import is applied.



The revised BRVD file is being re-imported into a Vocabulary previously created from a BRVD. The following attribute changes are shown in the Review Information dialog above:

- **Match** – Unchanged attributes that originated in OpenEdge.
- **User Defined** – Added `CreditDiscount` only in Corticon.
- **Add** – `Postal code` added in OpenEdge and then added into the latest BRVD file.
- **Remove/Re-add** – Changed `Terms` to an integer only in Corticon, but it is still a decimal in OpenEdge and the BRVD. After this action, `Terms` is a decimal.
- **Merge** – Added `OrderStatus` to Corticon while awaiting BRVD change from OpenEdge. The new BRVD has `OrderStatus` so it is a merge
- **Remove** - Dropped the `Promise date` in OpenEdge from the latest BRVD file.

Accessing the Vocabulary metadata of a Decision Service

There is a REST API for retrieving vocabulary metadata from a deployed Decision Service. This is useful for integrating Corticon with other applications that need to format REST or SOAP calls to a Decision Service.

Structure of a request

To retrieve vocabulary metadata, make an HTTP GET request to the `getVocabularyMetadata` endpoint specifying the Decision Service name and version as the following URL parameters:

- `name=Decision Service name`
- `majorVersion= Major version of the Decision Service`
- `minorVersion= Minor version of the Decision Service`

For example:

```
http://localhost:8850/axis/corticon/decisionService/getVocabularyMetadata
?name=ProcessOrder&majorVersion=1&minorVersion=1
```

The metadata API is available for Corticon Decision Services deployed for either a Java Server or a .NET Server.

The following JSON-formatted document is an example of a response:

```
{
  "majorVersion": 1,
  "name": "MetadataTest",
  "minorVersion": 0,
  "entities": [
    {
      "name": "Entity_1",
      "associations": [
        {
          "name": "associationObject1",
          "targetEntity": "AssociationObject1",
          "inContext": true,

          "mandatory": false
          "cardinality": "1"
        },
        {
          "name": "associationObjectOverride",
          "targetEntity": "AssociationObject2",
          "inContext": false,
          "mandatory": true
          "cardinality": "*"
        }
      ],
      "attributes": [
        {
          "dataType": "Boolean",
          "name": "boolean1",
          "inContext": true,
          "type": "Base",
          "mandatory": true
        },
        {
          "dataType": "Date",
          "name": "date1",
          "inContext": false,
          "type": "Transient",
          "mandatory": false
        },
        {
          "dataType": "DateTime",
          "name": "datetime1",
          "inContext": true,
          "type": "Base",
          "mandatory": true
        },
        {
          "cdtConstraintExpr": "value < 100.0",
          (Constraint expression associated with this Attribute.)
        }
      ]
    }
  ]
}
```

```

        "dataType": "Decimal",
        "name": "decimal1",
        "inContext": false,
        "type": "Transient",
        "mandatory": false
    },
    {
        "dataType": "Integer",
        "name": "int1",
        "cdtEnumeration": [    (A CDT that is values only, no labels.)
            { "value": "1" },
            { "value": "2" },
            { "value": "3" },
            { "value": "4" }
        ],
        "inContext": true
        "type": "Base",
        "mandatory": true
    },
    {
        "dataType": "String",
        "name": "string1",
        "cdtEnumeration": [    (A CDT that has labels and values.)
            {
                "value": "s"
                "label": "Small"
            },
            {
                "value": "m"
                "label": "Medium"
            },
            {
                "value": "l"
                "label": "Large"
            }
        ],
        "inContext": false,
        "type": "Transient",
        "mandatory": false
    },
    {
        "dataType": "Time",
        "name": "time1",
        "inContext": true,
        "type": "Base",
        "mandatory": true
    },
]
},
{
    "name": "AssociationObject1",
    "associations": [],
    "attributes": [
        {
            "dataType": "String",
            "name": "string1",
            "inContext": true
            "mandatory": true
        },
    ],
}
},
{
    "name": "AssociationObject2",
    "associations": [],
    "attributes": [
        {
            "dataType": "String",
            "name": "string1",

```

```

        "inContext": false
        "mandatory": false
    },
    ]
}
}

```

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

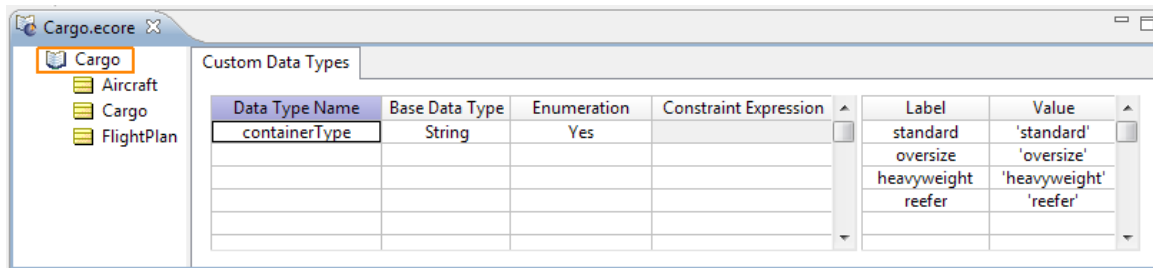
When defining a custom data type, you must give it a name with no blanks. The name must comply with standard entity naming conventions (see the *Quick Reference Guide* for details) and must not overlap (match) any of the base data types, any other custom data type names, or the names of any Vocabulary 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 6: 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
<code>value > 5</code>	Integer values greater than 5
<code>value >= 10.2</code>	Decimal values greater than or equal to 10.2
<code>value in (1.1..9.9]</code>	Decimal values between 1.1 (exclusive) and 9.9 (inclusive)
<code>value in ['1/1/2014 12:30:00 PM'..'1/2/2019 11:00:00 AM')</code>	DateTime values between '1/1/2014 12:30:00 PM' (inclusive) and '1/2/2019 11:00:00 AM' (exclusive)
<code>value in ['1:00:00 PM'..'2:00:00 PM']</code>	Time values between '1:00:00 PM' (inclusive) and '2:00:00 PM' (inclusive)
<code>value.size >= 6 and (value.indexOf(1) > 0 or value.indexOf(2) > 0)</code>	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

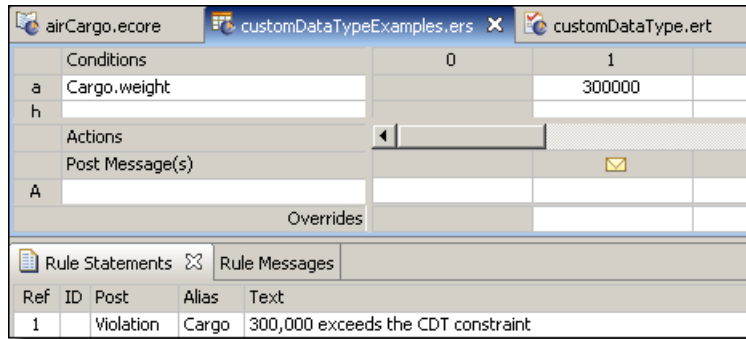
Non-enumerated custom data types use **Constraint Expressions** and do not cause Rulesheet drop-downs to become populated with custom sets. Also, manually entering a cell value that violates the custom data type's **Constraint Expression** is not prohibited in the Rulesheet. For example, in the example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

Figure 7: Non-enumerated Custom Data Types

Custom Data Types			
Data Type Name	Base Data Type	Enumeration	Constraint Expression
weightRange	Decimal	No	value < 200000

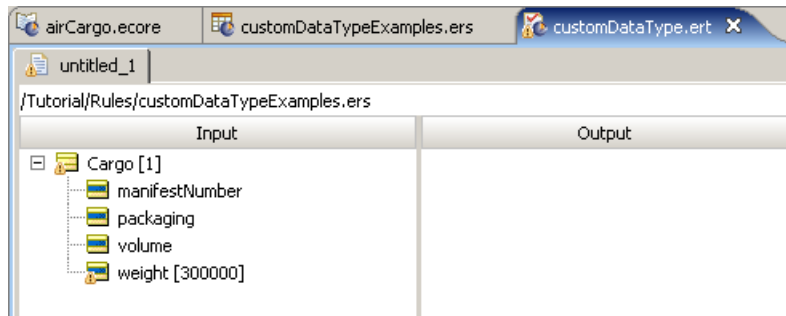
Then, after assigning it to the Vocabulary attribute `Cargo.weight`, it is used in a Rulesheet Condition row as shown below:

Figure 8: Using Custom Data Types in a Rulesheet



Notice in *Using Custom Data Types in a Rulesheet* that the 300000 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 9: Violating a Custom Data Type's Constraint Expression




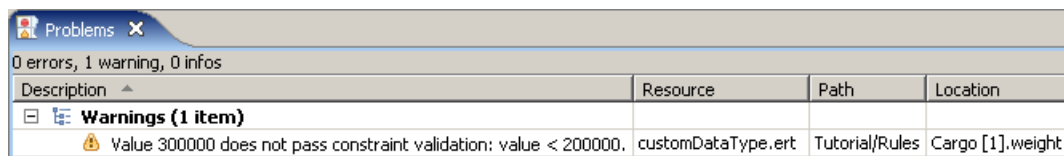

Notice that the small yellow warning icon  indicates a problem in the attribute, entity, and both Ruletest tabs. Such an error is hard to miss! Also, a Warning message will appear in the **Problems** tab (if open and visible) as shown below. If the Problems tab is closed, you can display it by selecting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

Figure 10: 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

Enumerations are lists of strictly typed unique values that are the valid values for each attribute that is assigned the custom data type name as its data type. These lists also prompt Rulesheet and Ruletest designers to use a specific list of values. Enumerated lists can be maintained directly in the Vocabulary, or retrieved and updated from a database.

Each item list can be partnered with a unique *label* that you select in Rulesheets and Ruletests.

How enumeration labels and values behave

Before you start setting up and using enumerations, you should get acquainted with labels and values.

Note: It is important that you determine whether you want to use labels, as changing a set of enumerations later to add or remove the labels data will impact any Rulesheets and Ruletests that use that custom data type's enumerations as you can observe in this topic.

At the Vocabulary root, we created a String enumeration with only values. The base data type can be any Corticon data type except boolean. Every line requires a unique entry of its type, and the list must have no blank lines from the top down to the last line.

The following examples are String values. They can contain spaces and pretty much any other character. It needs to be set off in plain single quote marks. If you enter or paste text with the delimiters, they are added for you. Like this:

Custom Data Types			Database Access	
Data Type Name	Base Data...	Enum...	Label	Value
colorLabeled	String	Yes		'red'
colorUnlabeled	String	Yes		'blue'

If you want to use labels, the label is always a String of any alphanumeric characters but cannot contain spaces. Each must be unique and must have a corresponding value. Even when you use labels, the values must be unique.

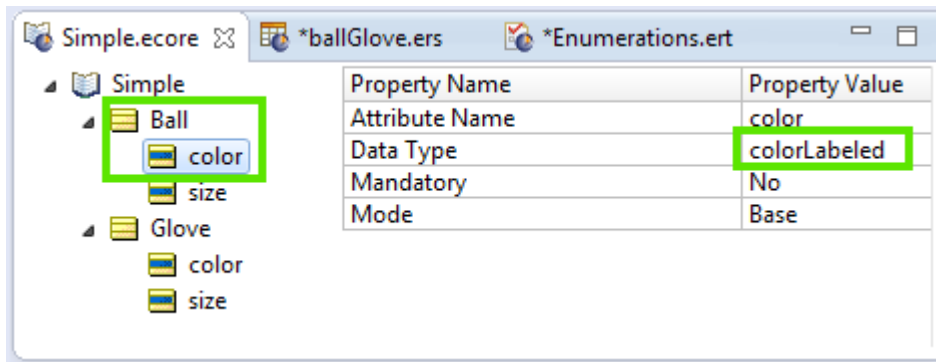
Custom Data Types			Database Access	
Data Type Name	Base Data...	Enum...	Label	Value
colorLabeled	String	Yes	red	'Crimson'
colorUnlabeled	String	Yes	blue	'Cerulean'

We set the `Glove.color` to use the `colorUnlabeled` data type:

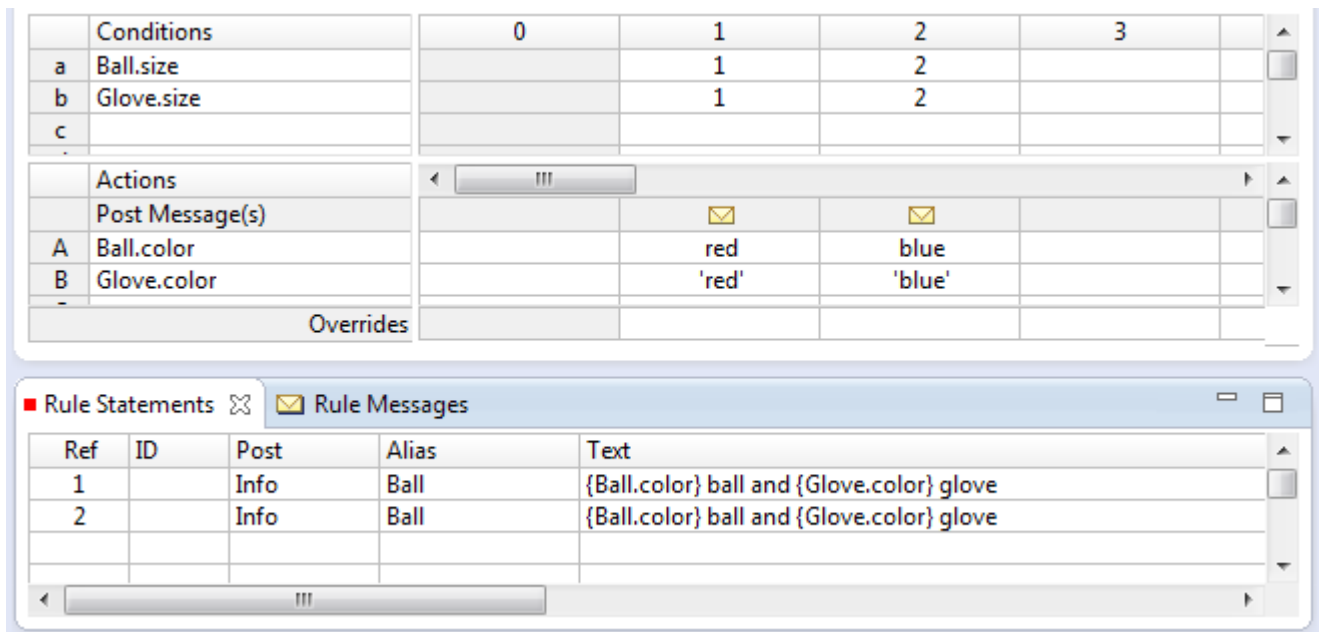
The screenshot shows the Corticon IDE with three open files: `Simple.ecore`, `*ballGlove.ers`, and `*Enumerations.ert`. On the left, a tree view shows the project structure: `Simple` (containing `Ball` and `Glove`), `Ball` (containing `color` and `size`), and `Glove` (containing `color` and `size`). The `Glove.color` property is highlighted with a green box. On the right, the 'Property Value' table for `Glove.color` is shown:

Property Name	Property Value
Attribute Name	color
Data Type	colorUnlabeled
Mandatory	No
Mode	Base

We set the `Ball.color` to use the `colorLabeled` data type:



When we create a Rulesheet, the list offered at A1 contains the label (`Ball.color = red`), while the list offered at B1 contains the value in quotes (`Glove.color='red'`).



We added Rule Statements so that we can see how the labeled and unlabeled items are handled.

In a simple Ruletest, we add some size tests to see what results. As shown, the labels and values in the result Output are both unquoted. The Rule Messages display the value when the label was in use and the value of the value-only enumeration.

untitled_1

/simple/ballGlove.ers Differences: 0

Input	Output
<ul style="list-style-type: none"> Ball [1] <ul style="list-style-type: none"> size [1] Glove [1] <ul style="list-style-type: none"> size [1] Ball [2] <ul style="list-style-type: none"> size [2] Glove [2] <ul style="list-style-type: none"> size [2] 	<ul style="list-style-type: none"> Ball [1] <ul style="list-style-type: none"> color [red] size [1] Glove [1] <ul style="list-style-type: none"> color [red] size [1] Ball [2] <ul style="list-style-type: none"> color [blue] size [2] Glove [2] <ul style="list-style-type: none"> color [blue] size [2]

Rule Messages

Severity	Message	Entity
Info	Crimson ball and red glove	Ball[1]
Info	Cerulean ball and blue glove	Ball[2]

Entry of test values in the Ruletest list the label+value's label...

Input

- Ball [1]
 - color [red]
 - size [1]
- Glove [1]
 - color [blue]
 - size [1]
- Ball [2]
 - size [2]
- Glove [2]
 - size [2]

... while the value-only list has quoted values...

Input

- Ball [1]
 - color [red]
 - size [1]
- Glove [1]
 - color [red]
 - size [1]
- Ball [2]
 - color [blue]
 - size [2]
- Glove [2]
 - color [blue]
 - size [2]

... but both are reconciled to unquoted values in the displayed Input and Output.

Input

- Ball [1]
 - color [red]
 - size [1]
- Glove [1]
 - color [red]
 - size [1]
- Ball [2]
 - size [2]
- Glove [2]
 - size [2]

Output

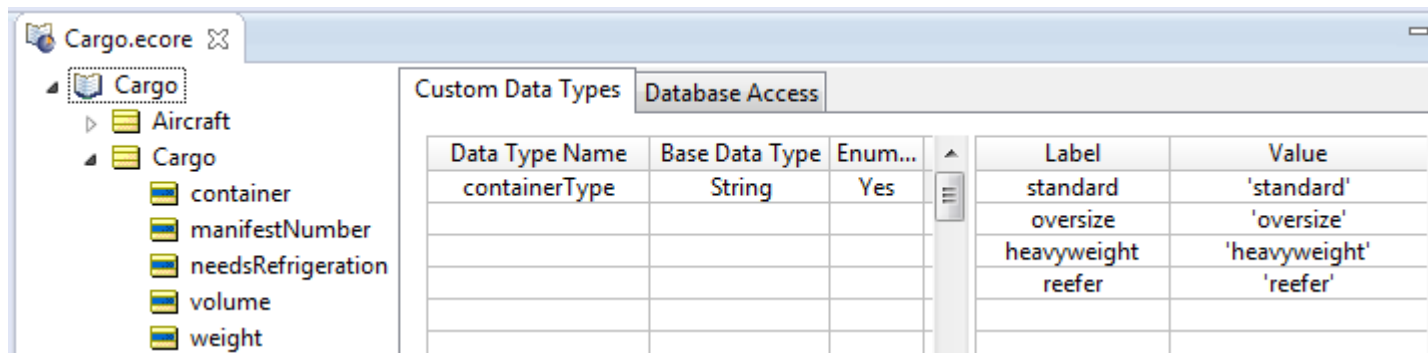
- Ball [1]
 - color [red]
 - size [1]
- Glove [1]
 - color [red]
 - size [1]
- Ball [2]
 - color [blue]
 - size [2]
- Glove [2]
 - color [blue]
 - size [2]

Severity	Message	Entity
Info	Crimson ball and red glove	Ball[1]
Info	Cerulean ball and blue glove	Ball[2]

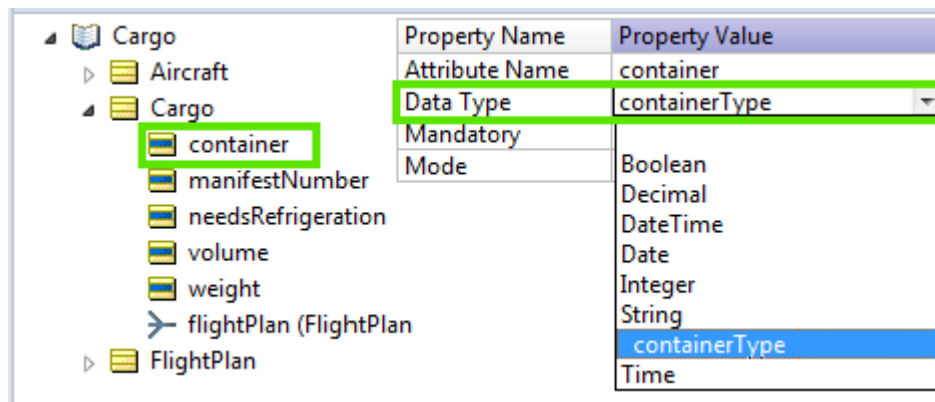
Note: It is important that you determine in each custom data type whether you want to use labels. Some enumerations can have labels while others do not. Changing a set of enumerations later to add or remove the labels data will impact any Rulesheets and Ruletests that use that custom data type's enumerations as you can observe in this topic.

Enumerations defined in the Vocabulary

To set up an Enumeration, open the project's Vocabulary, and then click on its root, Cargo in this tutorial example. Then enter a preferred unique name without spaces, and then click on the Base Data type cell of the row to choose the data type (the values are all red until you have added a successful value or label/value pair). Click on the Enumeration cell to choose Yes. Now enter a value on the first row, and a label if you want one. All the cells are validated and the red markers are cleared. Then you can add other value or label/value pairs on the next lines.



When you have completed a valid Custom Data Type, choose attributes in the Vocabulary that will be constrained to the enumeration.



If your custom data type is a local enumeration, then you enter the enumerated values of the base data type into the **Value** column, and, if you intend to use labels, enter label text into the **Labels** column.

Note: Pasting in labels and values - If you have the source data in a spreadsheet or text file, you can copy from the source and paste into the Vocabulary after you have defined the name, base data type, and chosen yes to enumeration. When you paste two columns of data, click on the first label row. If you have one column of data you want to use for both the label and the value, paste it in turn into each column. If the data type is String, Date, Time, or DateTime, the paste action will add the required single quote marks.

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 11: Custom Data Type, example 1

Custom Data Types					
Data Type Name	Base Data Type	Enumerati...	Constraint Expression	Label	Value
containerType	String	Yes			2
PrimeNumbers	Integer	Yes			3
USHolidays2015	Date	Yes			5
ShirtSize	Integer	Yes			7
RiskProfile	Integer	Yes			11
DevTeam	String	Yes			13

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

Figure 12: Custom Data Type, example 2

Custom Data Types					
Data Type Name	Base Data Type	Enumerati...	Constraint Expression	Label	Value
containerType	String	Yes		standard	'standard'
PrimeNumbers	Integer	Yes		oversize	'oversize'
USHolidays2015	Date	Yes		heavyweight	'heavyweight'
ShirtSize	Integer	Yes		reefer	'reefer'
RiskProfile	Integer	Yes			
DevTeam	String	Yes			

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

Figure 13: Custom Data Type, example 3

Custom Data Types					
Data Type Name	Base Data Type	Enumerati...	Constraint Expression	Label	Value
containerType	String	Yes		New_year	'1/1/2015'
PrimeNumbers	Integer	Yes		Independen...	'7/4/2015'
USHolidays2015	Date	Yes		Labor_Day	'9/7/2015'
ShirtSize	Integer	Yes		Thanksgiving	'11/26/2015'
RiskProfile	Integer	Yes		Christmas	'12/25/2015'
DevTeam	String	Yes			

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

Figure 14: Custom Data Type, example 4

Custom Data Types					
Data Type Name	Base Data Type	Enumerati...	Constraint Expression	Label	Value
containerType	String	Yes		S	1
PrimeNumbers	Integer	Yes		M	2
USHolidays2015	Date	Yes		L	3
ShirtSize	Integer	Yes		XL	4
RiskProfile	Integer	Yes		XXL	5
DevTeam	String	Yes			

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

Figure 15: Custom Data Type, example 5



Custom Data Types					
Data Type Name	Base Data Type	Enumerati...	Constraint Expression	Label	Value
containerType	String	Yes		Low	1
PrimeNumbers	Integer	Yes		Medium	2
USHolidays2015	Date	Yes		High	3
ShirtSize	Integer	Yes		VeryHigh	4
RiskProfile	Integer	Yes			
DevTeam	String	Yes			

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

Figure 16: Custom Data Type, example 6

Custom Data Types						
Data Type Name	Base Data Type	Enumerati...	Constraint Expression		Label	Value
containerType	String	Yes				'Dave'
PrimeNumbers	Integer	Yes				'John'
USHolidays2015	Date	Yes				'Jim'
ShirtSize	Integer	Yes				'Prashant'
RiskProfile	Integer	Yes				'Mahesh'
DevTeam	String	Yes				'Kendall'
						'George'
						'Cheryl'
						'Amish'
						'Eric'
						'Marian'

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 get 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 your Vocabulary has a verified connection to a supported database, the **Custom Data Types** tab presents three additional columns, as shown:

Figure 17: Custom Data Type columns for defining database retrieval

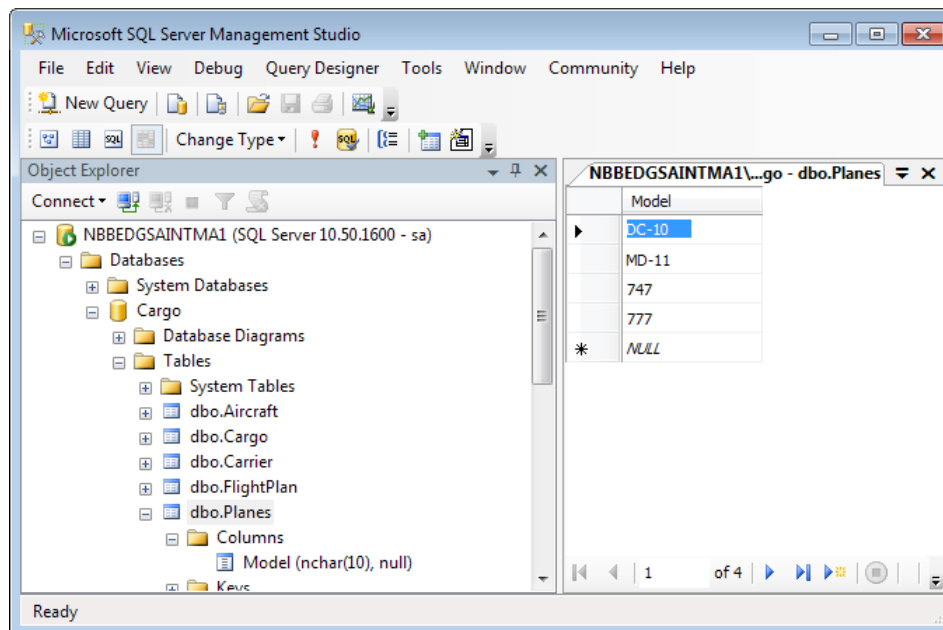
Custom Data Types						
Database Access						
Data Type Name	Base Data Type	Enumeration	Constraint Expression	Lookup Table Name	Labels Column	Values Column
containerType	String	Yes				

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.

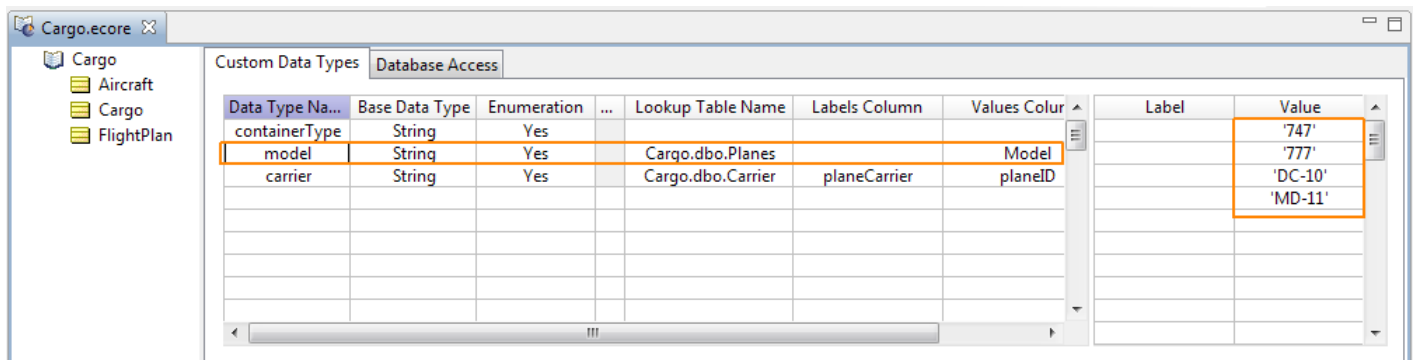
The following examples show two options:

Figure 18: SQL Server table with values to use in the Vocabulary



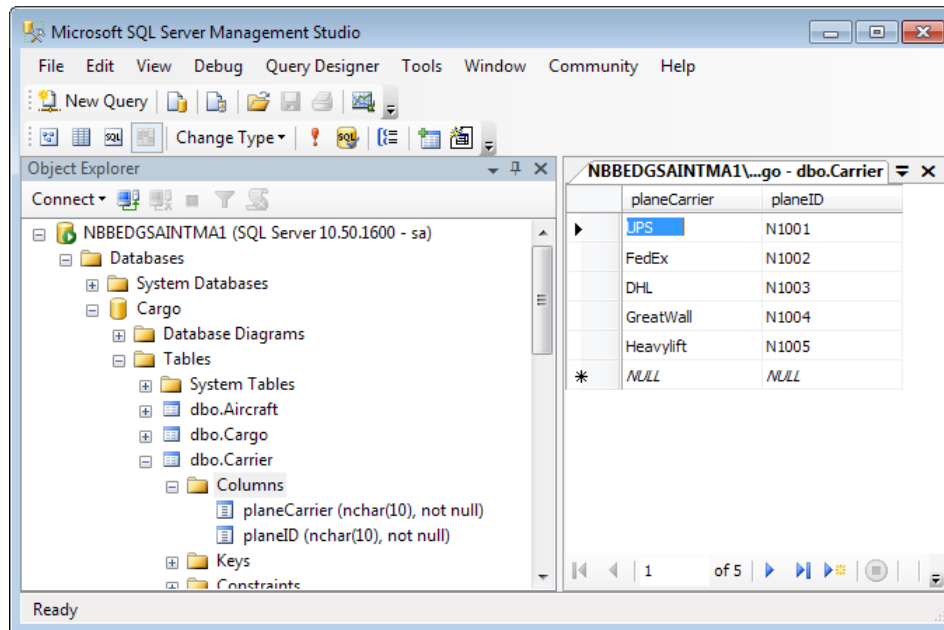
The value data is retrieved into the Vocabulary as highlighted:

Figure 19: Definition and retrieved values in the Corticon Studio



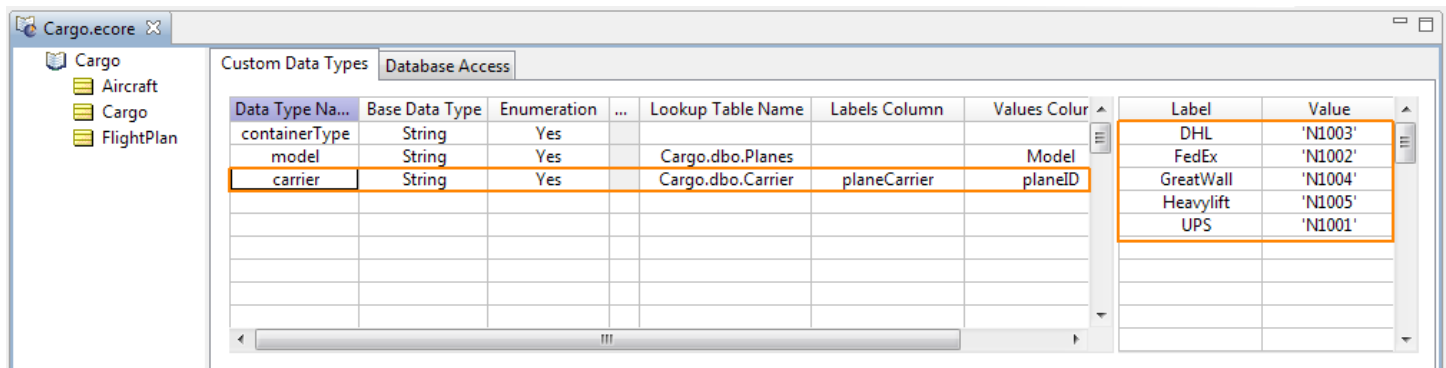
Another example retrieves name-value pairs.

Figure 20: SQL Server table with labels and values to use in the Vocabulary



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

Figure 21: Definition and retrieved label-values in the Corticon Studio



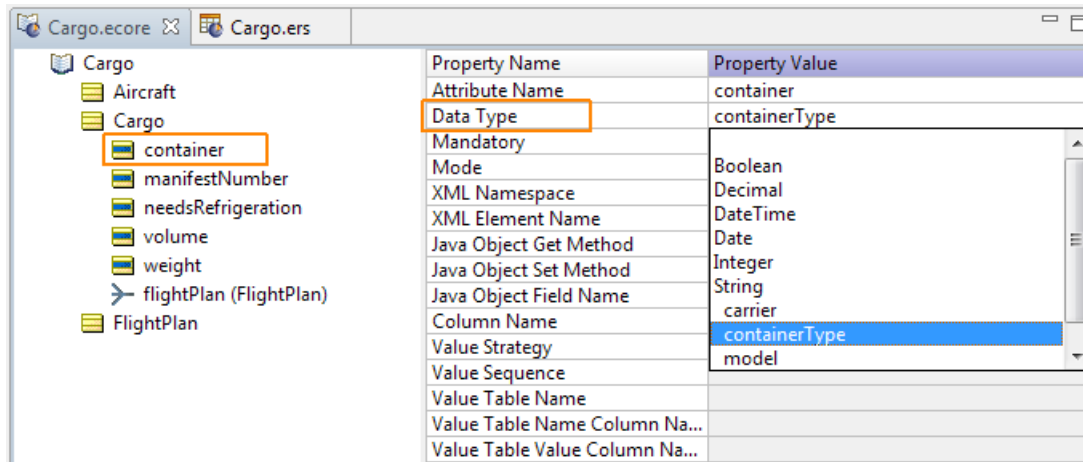
Note: This functionality uses Corticon's Enterprise Data Connector. A section of the EDC tutorial covers this topic in detail, *"Importing an attribute's possible values from database tables" in the "Implementing EDC" section of the Integration and Deployment Guide*

Using Custom Data Types

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 22: Using Custom Data Types in the Vocabulary

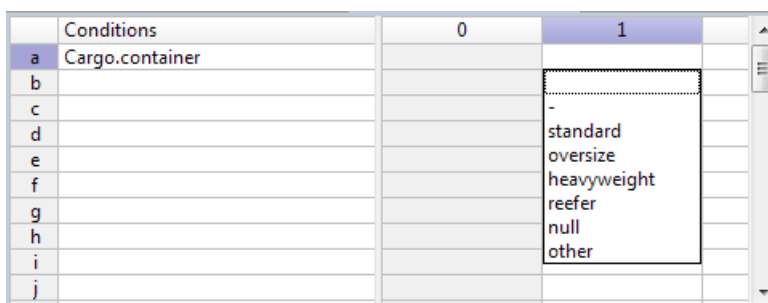


Notice in this figure that multiple attributes can use the same custom data type; the custom data type `containerType` is shown in the drop-down as a sub-category of the String-based data type. The other custom data types will be grouped with their base data types as well.

Using enumerated Custom Data Types in Rulesheets

Once an enumeration 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 23: 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 label using the # tag, as shown:

Figure 24: Using # tag to test a custom data type

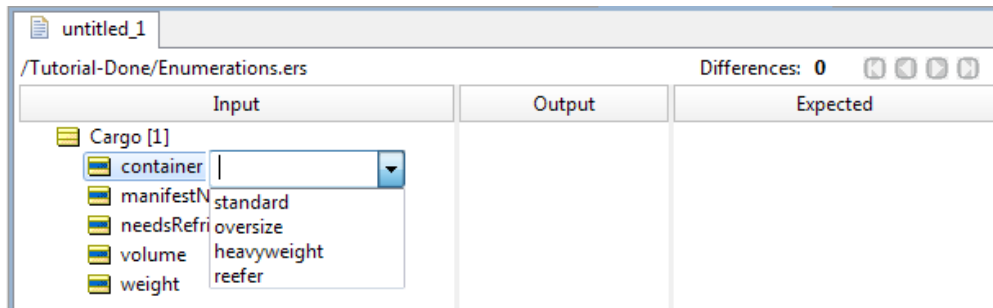
Conditions	0	1	2
a Cargo.container = containerType#reefer			
b			
c			
d			
e			
f			

Note: Using a dot instead of a # tag works but, if there is custom data type with the same name as an entity, the expression will be invalid.

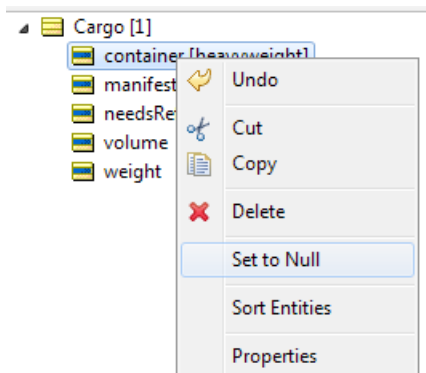
Using enumerated Custom Data Types in Ruletests

An enumeration's Values and Labels are available as selectable inputs in a Ruletest, as shown:

Figure 25: 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:



Using IN operator with an enumerated list

When your rule condition or filter are not defined by a range of values, you might have tried use a series of test and logical OR operations to describe the test. For example, `entity1.attribute1='This' or entity1.attribute1='That' or entity1.attribute1='TheOther'` is long, and could evolve into a very long expression. You can eliminate the use of the long form of enumeration literals by using the `in` operator's list format to reduce that filter or condition expression to `entity1.attribute1 in {'This', 'That', 'TheOther'}`.

You can go a step further by defining enumerated lists to define even more brisk expressions, where the labels that you choose are abbreviations for the full names. For example, `Regions.state in {MA, NH, VT, CT, RI, ME}` to qualify only US New England states.

For more information about these features, see the topics in [Qualifying rules with ranges and lists](#) on page 89

Relaxing enforcement of Custom Data Types

Using Custom Data Types lets you define general limitations of an attribute's values that are enforced on all Rulesheets and Ruletests in the project and its Decision Services. While they are valuable in focusing on what is valid in rule designs, violations of the constraints or lists cause rule processing -- Ruletests in Studio; Decision Services on Servers -- to halt at the first violation. Such exceptions indicate that values in attributes are not within numeric constraint ranges or not included in enumerated lists that have been set in the Vocabulary's Custom Data Types.

Note: It is recommended that you use relaxed enforcement of CDTs only in test environments. In production, you should enforce data constraints and lists to ensure valid processing by rules.

For Ruleflows, a rule that throws an exception in earlier Rulesheets disables processing in subsequent Rulesheets. In the following example, the Advanced Tutorial testsheet outputs the following statements:

Severity	Message	Entity
Info	[Checks,2] The customer is a Preferred Cardholder	Customer[1]
Info	[coupons,2] \$2 off next purchase when 3 or more Soda/Juice items are purchased in a single visit.	ShoppingCart[1]
Info	[coupons,3] 10% off next gas purchase when total is over \$75.	ShoppingCart[1]
Info	[coupons,B0] \$1.649800 cashBack bonus earned today, new cashBack balance is \$10.889800.	ShoppingCart[1]
Info	[use__cashBack,1] cashback.bonus has been deducted from the total. New total = \$71.600200. Today's savings = \$10.889800.	ShoppingCart[1]

Note: The rule tracing feature reveals which Rulesheets fired which rules.

By defining a Custom Data Type that specifies the `Item` attribute `price` must be greater than zero, and then entering the input value `-1.00` for an item on the testsheet, the first constraint error stops all the subsequent rules from firing:

Severity	Message
Violation	An unexpected error occurred in Input Data: com.corticon.cdo.ConstraintViolationException: constraint violation setting Item.price to value [-1]

Relaxing the enforcement of Custom Data Type constraints produces warnings instead of violations, so that development teams and pre-production testing teams can expedite their debugging of rules and error handling, as shown:

Severity	Message	Entity
Warning	constraint violation setting Item.price to value [-1]	Item[3]
Info	The customer is a Preferred Cardholder	Customer[1]
Info	\$2 off next purchase when 3 or more Soda/Juice items are purchased in a single visit.	ShoppingCart[1]
Info	\$1.379800 cashBack bonus earned today, new cashBack balance is \$10.619800.	ShoppingCart[1]
Info	cashback.bonus has been deducted from the total. New total = \$58.370200. Today's savings = \$10.619800.	ShoppingCart[1]

This example might indicate that the applications that format requests should handle the data constraint before forwarding a request into the rules engine.

Detailed Example

The following example uses the `Cargo` Vocabulary. It has two Custom Data Types, one numeric constraint (assigned to `Cargo.weight` and `Cargo.volume`) and an enumeration list (assigned to `Cargo.container`.)

Custom Data Types					
Data Type Name	Base Data Type	Enumeration	Constraint Expression	Label	Value
containerType	String	Yes		standard	'standard'
positiveInteger	Integer	No	value >=1	oversize	'oversize'
				heavyweight	'heavyweight'
				reefer	'reefer'

A value that is outside the constraints (`Cargo [1] volume = -1`) is noted as violating the attribute's data type constraint on each input attribute and its entity, as well as noted on the **Problems** tab. But when the Ruletest runs, it halts on the first Violation, as shown:

The screenshot shows the Corticon Studio interface with the following components:

- untitled_1** window showing the rule test setup for `/Tutorial/Cargo.ers`.
- Input** and **Output** sections showing the data being tested.
- Rule Messages** tab showing a **Violation** message: "An unexpected error occurred in Input Data: com.corticon.cdo.ConstraintViolationException: constraint violation setting Cargo.container to value [secure]".
- Problems** tab showing 0 errors, 2 warnings, and 0 others.
- Error Log** tab showing the details of the violations.

Description	Resource	Location	Path	Type
Warnings (2 items)				
Value -1 does not pass constraint validation: value >=1.	Cargo.ert	Cargo [1].volume	/Tutorial	Validation Me...
Value secure is not in defined Values List of the CustomDataType of standard, oversize, heavyweight, reefer	Cargo.ert	Cargo [1].container	/Tutorial	Validation Me...

The details of that first exception are entered in the log (when the `loglevel` is `INFO` or higher, and the `logInfoFilter` does not include `VIOLATION` -- thereby accepting that type of info into the log.) No further processing occurs.

Note: See the topic *"Changing logging configuration" in the Using Corticon Server logs section of Integration and Deployment Guide* for more information.

By setting the `CcServer` property that relaxes enforcement of Custom Data Types, `com.corticon.vocabulary.cdt.relaxEnforcement=true`, and then restarting the Studio, the errors are still flagged in the data and the **Problem** information is unchanged. However, the **Rule Messages** section flags each of the constraint breaches as a **Warning**, lets them proceed, and then fires all the other rules.

The screenshot displays the Corticon Studio interface for a rule model named 'Cargo'. The main workspace is divided into three panels: Input, Output, and Expected. The Input panel shows a 'Cargo [1]' entity with properties 'container [secure]', 'volume [-1]', and 'weight [100]'. The Output panel shows a 'Cargo [1]' entity with properties 'container [standard]', 'volume [-1]', and 'weight [100]'. The Expected panel is empty.

Below the workspace is the 'Rule Messages' panel, which shows a list of messages with columns for Severity, Message, and Entity. The messages are:

Severity	Message	Entity
Warning	constraint violation setting Cargo.container to value [secure]	Cargo[1]
Warning	constraint violation setting Cargo.volume to value [-1]	Cargo[1]
Info	Cargo weighing <= 20,000 kilos must be packaged in a standard container.	Cargo[1]

At the bottom is the 'Problems' panel, which shows a list of errors and warnings. The messages are:

Description	Resource	Location	Path	Type
Value -1 does not pass constraint validation: value >=1.	Cargo.ert	Cargo [1].volume	/Tutorial	Validation
Value secure is not in defined Values List of the CustomDataType of standard, oversize, heavyweight, reefer	Cargo.ert	Cargo [1].container	/Tutorial	Validation

Note: It is recommended that you create or update the standard last-loaded properties file `brms.properties` to list override properties such as this for Corticon Studios and Servers. See the introductory topics in *"Configuring Corticon properties and settings"* in the *Server Integration and Deployment Guide* for information on where to locate this properties file.

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.


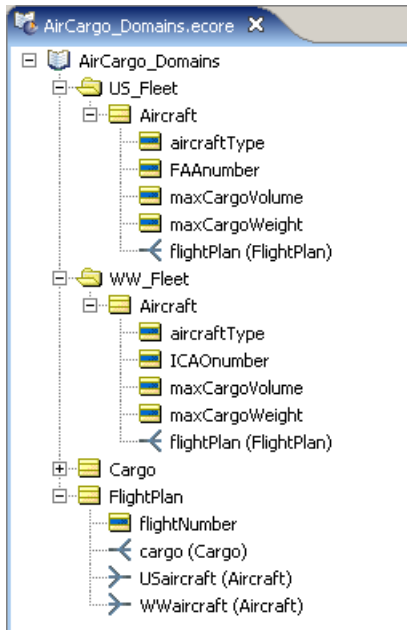
A new folder  is listed in the Vocabulary tree. Assign it a name. The example in the following figure shows a Vocabulary with two Domains, US_Fleet and WW_Fleet:

Figure 26: Using domains in the Vocabulary>

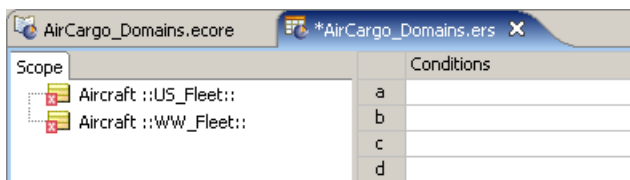



Notice that the entity `Aircraft` appears in each Domain, using the same spelling and containing slightly different attributes (`FAANumber` vs. `ICANumber`). Notice too that the association role names from `FlightPlan` to `Aircraft` have been named manually to ensure uniqueness: one is now `USaircraft`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

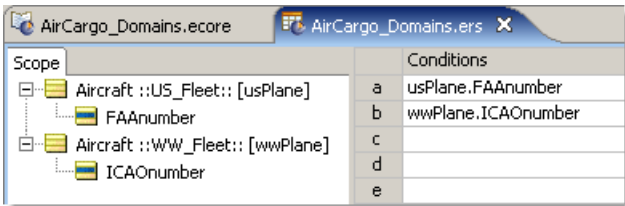
Figure 27: Non-unique Entity names prior to defining Aliases



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*.

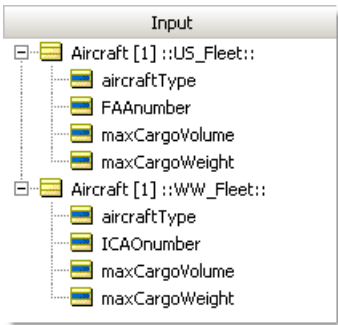
Figure 28: Non-unique Entity names after defining Aliases



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**.

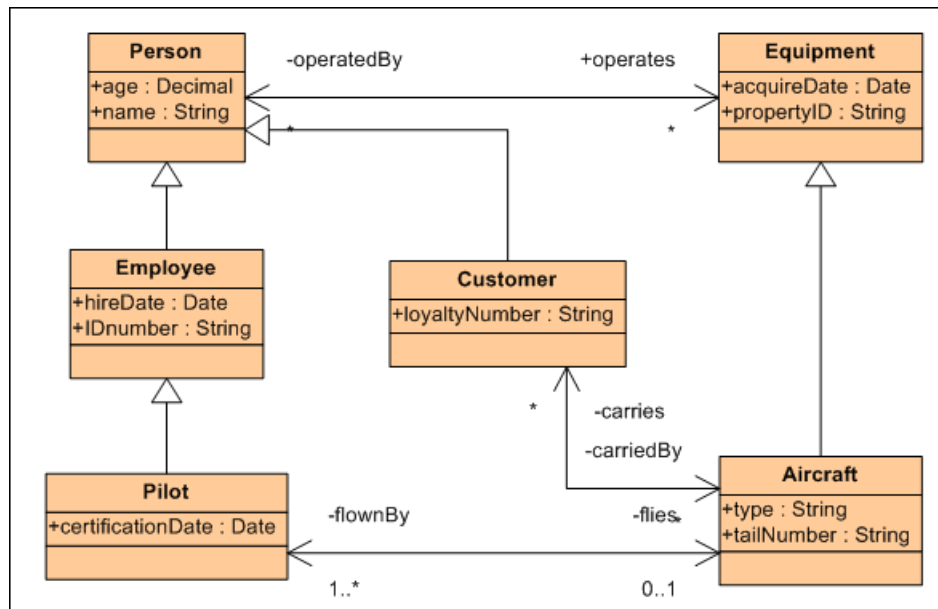
Figure 29: Domains in a Ruletest



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:

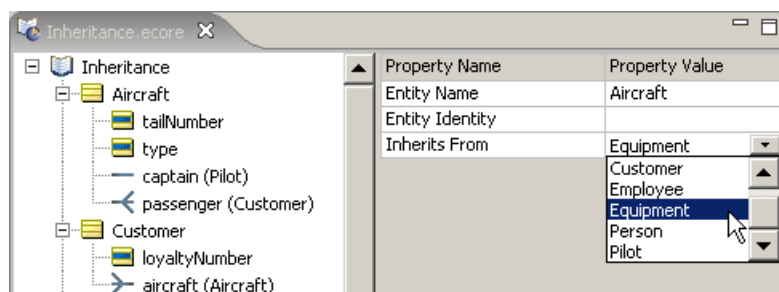
Figure 30: Rose UML Model Showing Inheritance



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`.

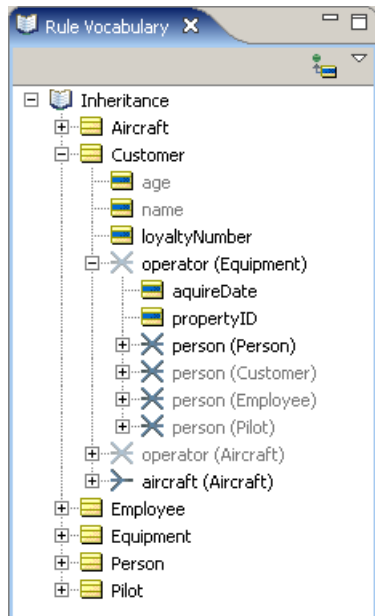
Modeling this UML Class Diagram as a Corticon Vocabulary is straightforward. All Entities, Attributes and Associations are created as per normal practice. To incorporate the elements of inheritance, we only need add one additional setting for each of the descendant entities, as shown:

Figure 31: 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 32: 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


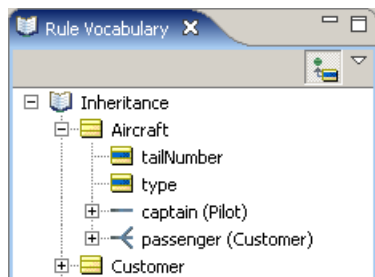
In cases where a Vocabulary contains inheritance (and includes the various icons and color schemes described above) but the modelers who use it do not intend to use inheritance in their rules, the inherited associations and filtered rolenames can be hidden from view by clicking the  icon in the upper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

Figure 33: 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.

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 34: Non-Unique Nodes used in the Scope Window

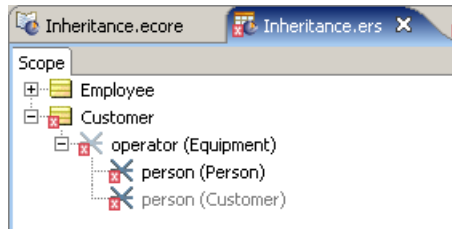
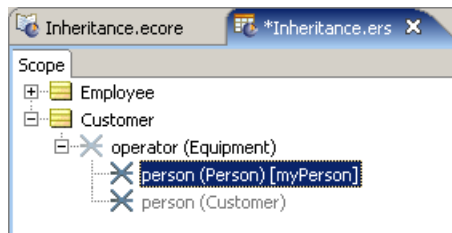


Figure 35: Uniqueness Established using an Alias





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 two rules that test the `age` value of the `Employee` entity. There are no explicit Actions taken by these rules, only the posting of messages.

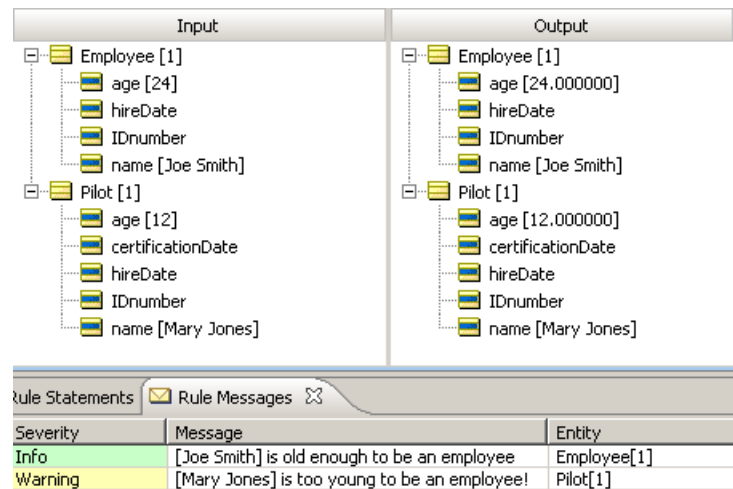
Figure 36: Rules written on Employee

Conditions		0	1	2
a	Employee.age < 18	-	T	F
b				
c				
Actions				
Post Message(s)				
A				
R				
Overrides				

Rule Statements		Rule Messages		
Ref	ID	Post	Alias	Text
1		Warning	Employee	[{Employee.name}] is too young to be an employee!
2		Info	Employee	[{Employee.name}] is old enough to be an 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 37: Inheritance in action

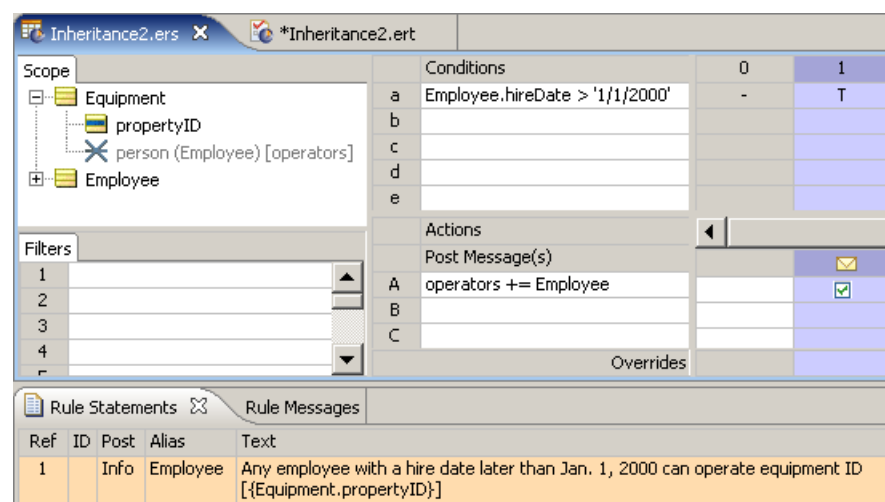


Using inheritance can be an efficient and powerful way to write rules on many different types of employees (such as pilots, gate agents, baggage handlers, and mechanics) without needing to write individual rules for each.

Inherited Association

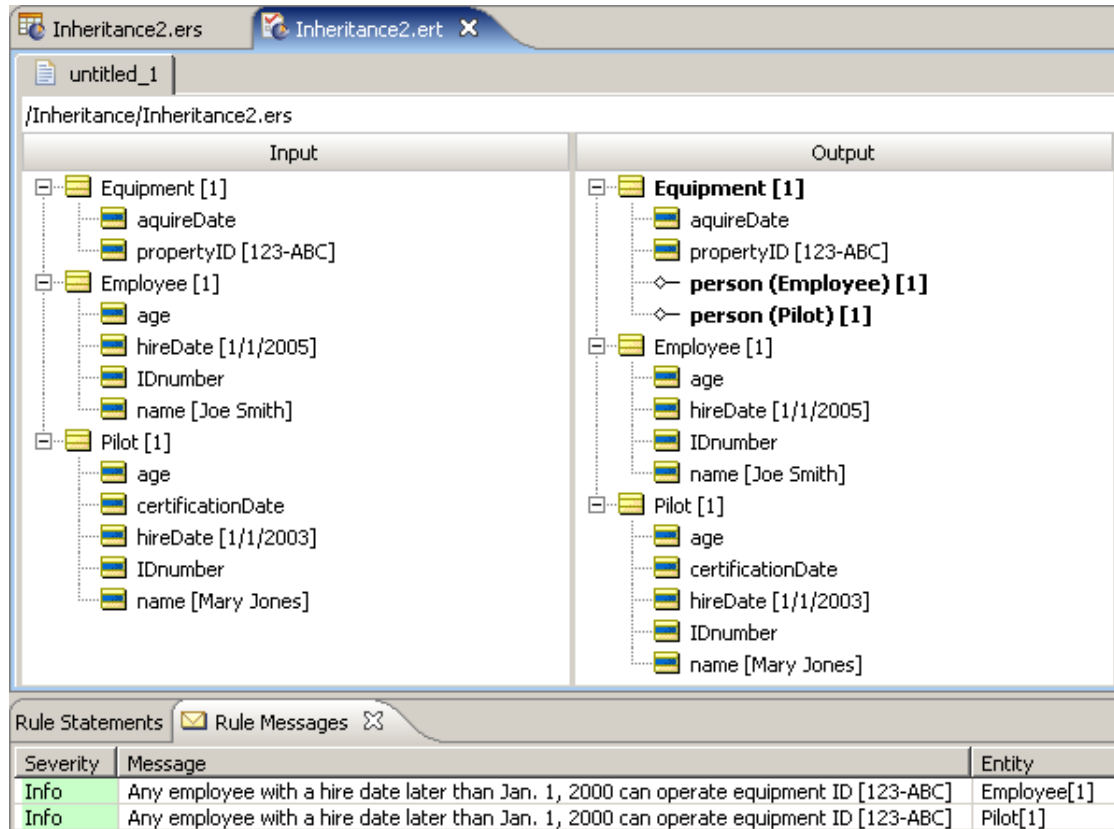
A similar test demonstrates how associations are inherited during rule execution. In this case, we test `Employee.hireDate` to see who's "qualified" to operate a piece of Equipment. The `+=` syntax used by the Action row is explained in more detail in the *Rule Language Guide*.

Figure 38: 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`!

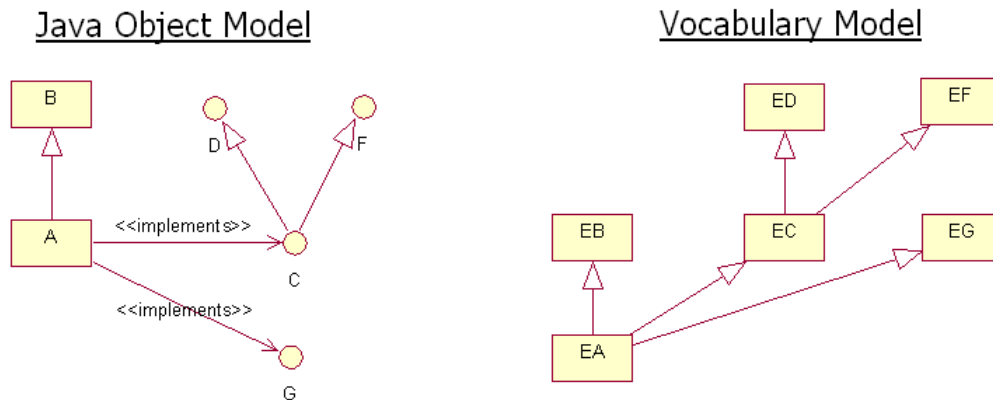
Figure 39: Inheriting an Association



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 40: 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 descendants found in DS context
- AE = An Ancestor Entity (one with descendants) 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 [Test yourself answers: Building the vocabulary](#) on page 347 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 Transient attribute icon?
14. What is the purpose of a 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:

- The number of possible associated entities.
- The number of attributes for each entity.
- The number of associations possible within an entity.
- 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?

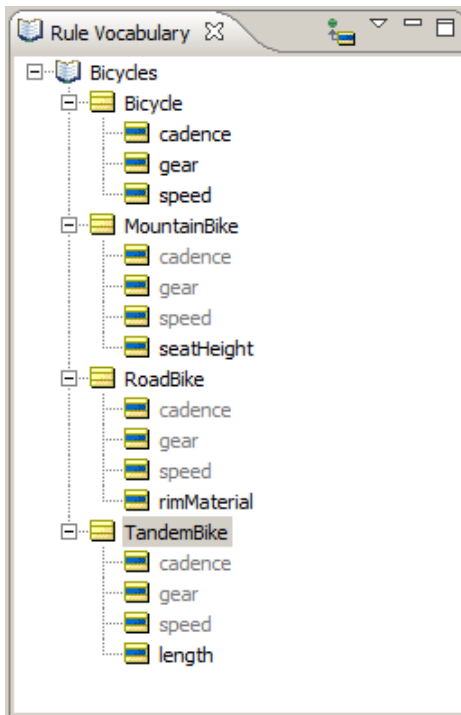
- Cannot match any other vocabulary entity names
- May match other Custom Data Type Names

- 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 exposes 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 Constraint Expression. Give an example of a Constraint Expression:

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

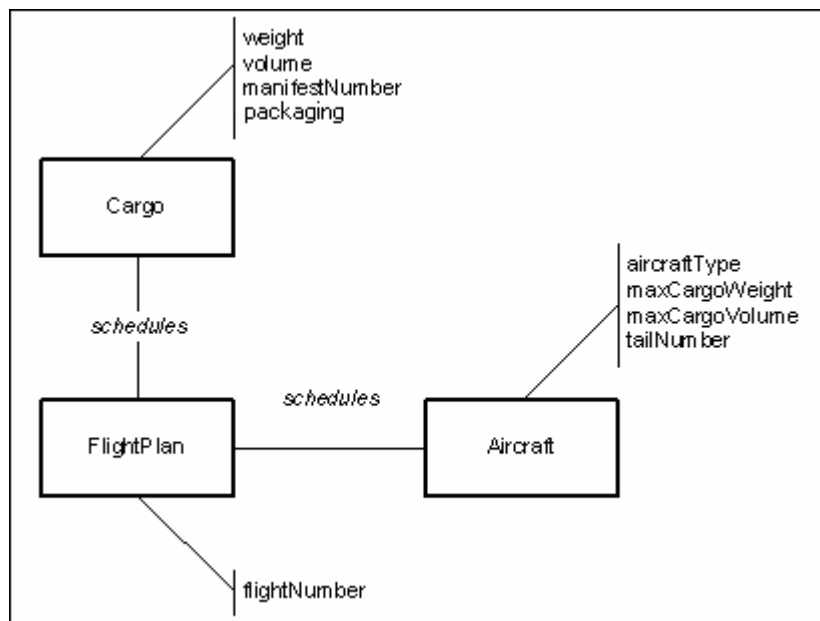
- _____
40. True or False. Using Inheritance can be a way to write efficient and powerful rules. For example, one rule could be used to modify the cadence attribute for all the entities in the Vocabulary example above.

Rule scope and 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 41: 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.

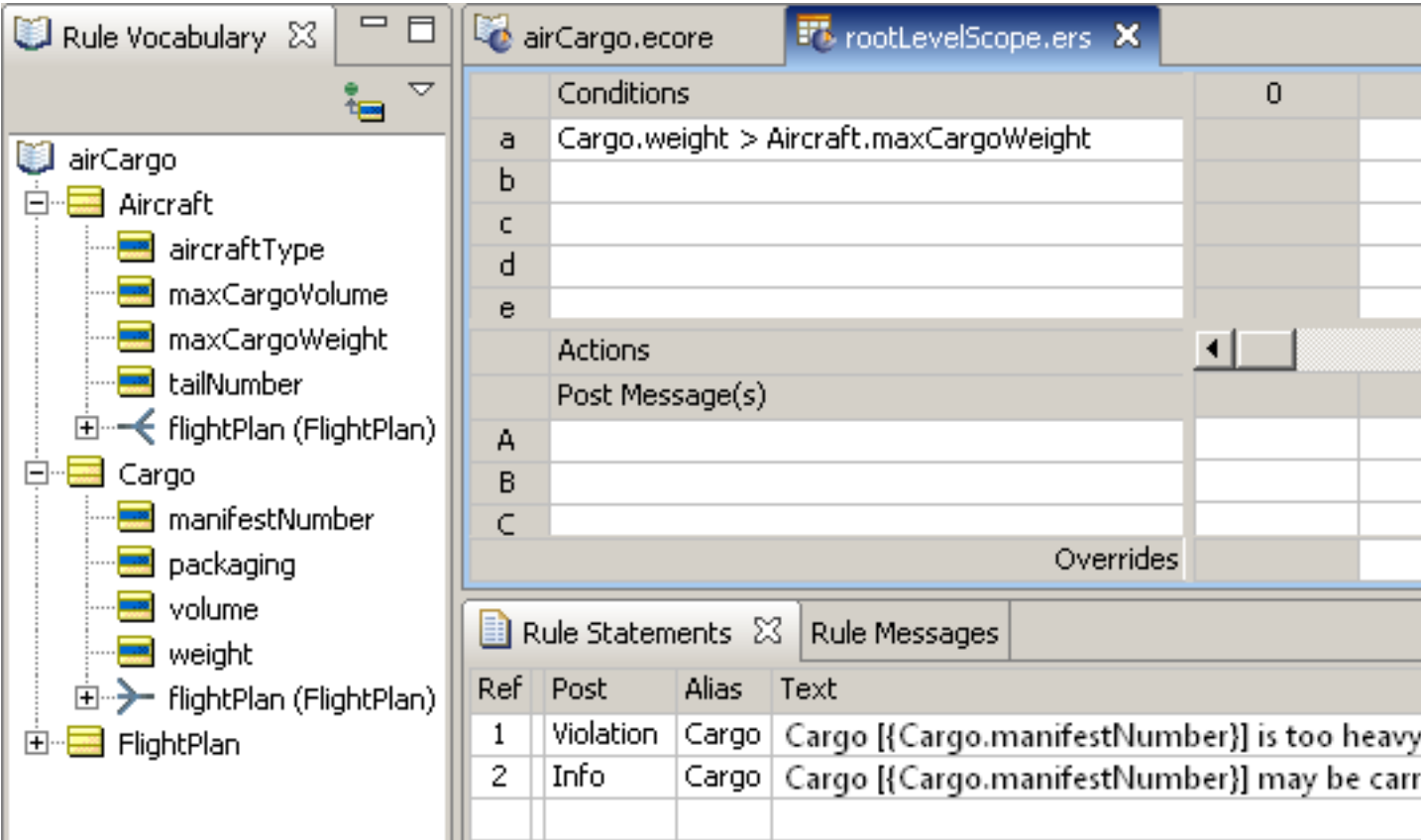
Also recall that the company wants to build a system that automatically checks flight plans to ensure no scheduling rules or guidelines are violated. One of the many business rules that need to be checked by this system is:

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 42: Expressing the Rule Using “Root-Level” Vocabulary Terms

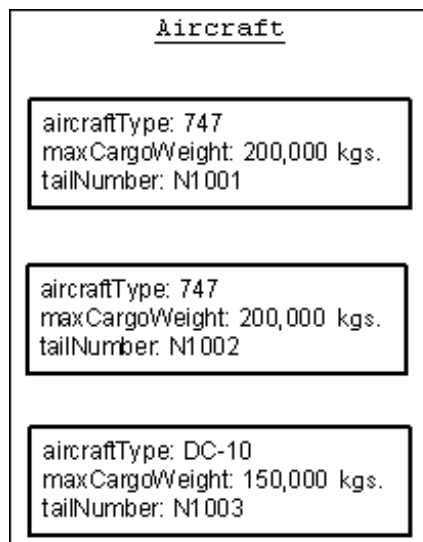


Refer to [Embedding attributes in posted rule statements](#) on page 112 for additional details regarding the syntax introduced in the **Rule Statements** portion of the following figure, example 5, in the [Custom data types](#) topic.

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

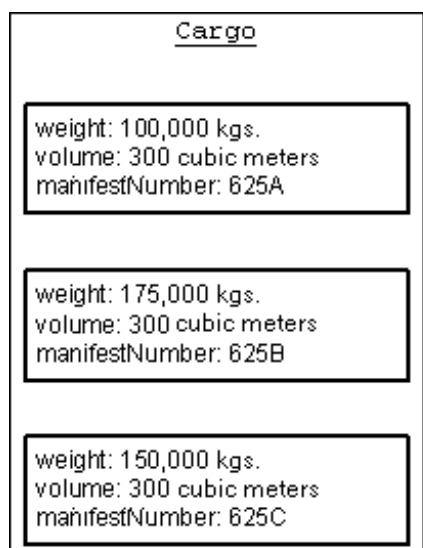
The company owns 3 Aircraft, 2 747s and a DC-10, each with different tail numbers. The 3 Aircraft are shown in the following figure, example 6 in the [Custom data types](#) topic. Each box represents a real-life example (or *instance*) of the `Aircraft` term from our Vocabulary.

Figure 43: The Cargo Company's 3 Aircraft



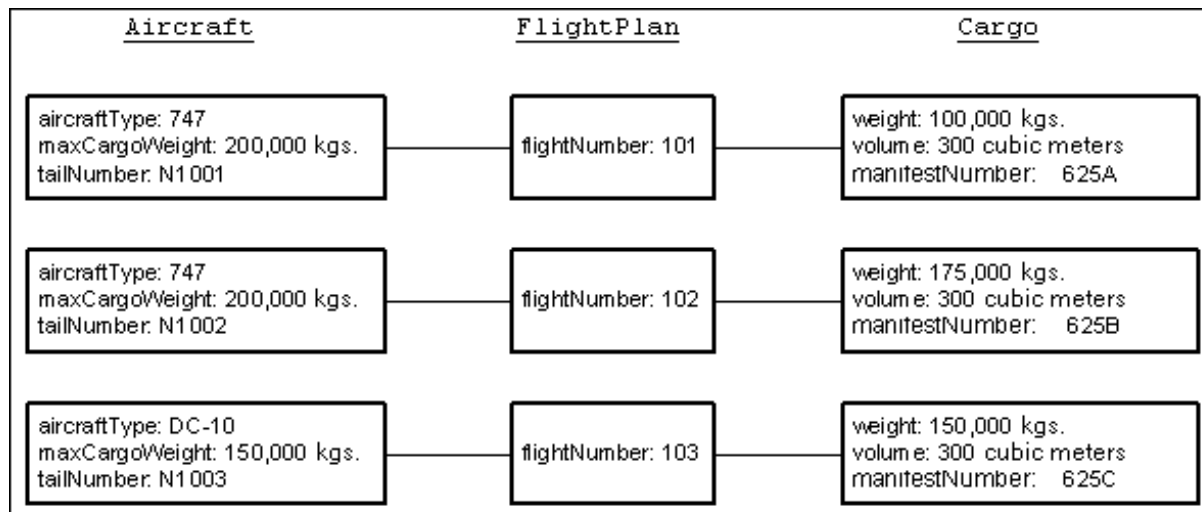
These `Aircraft` give the company the ability to schedule 3 `Cargo` shipments each night {there is another business rule implied here – “an Aircraft must not be scheduled for more than one flight per night”, but we won’t address this now because it is not relevant to the discussion}. On a given night, 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 44: The 3 Cargo Shipments for the Night of June 25th



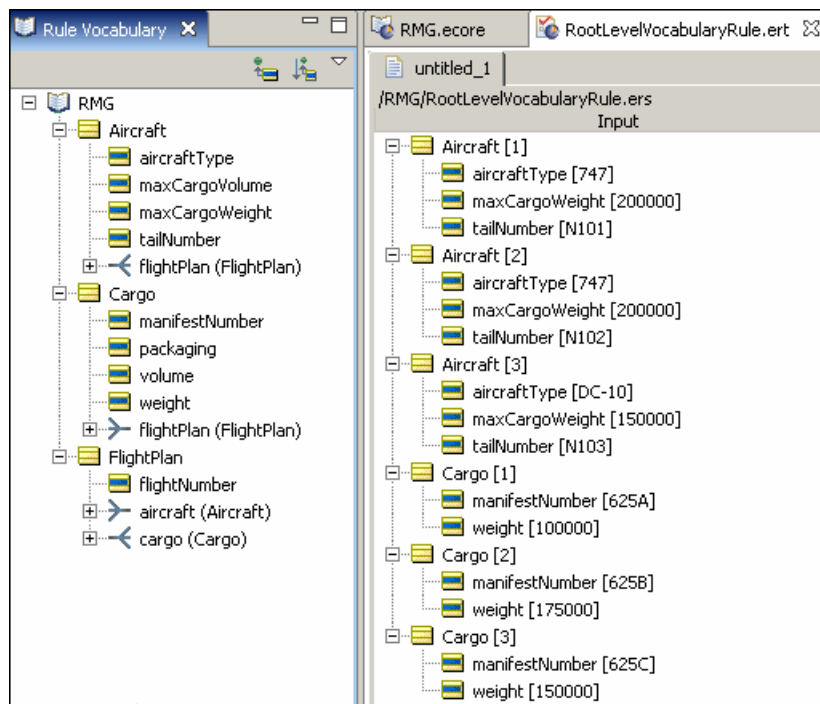
Finally, our sample business process manually matches specific aircraft and cargo shipments together as three flightplans, shown below. This organization of data is consistent with the structure and constraints implicit in our Vocabulary.

Figure 45: 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 46: Test the Rule Using “Root-Level” Vocabulary Terms



Running the Ruletest :

Figure 47: Results of the Ruletest

The screenshot shows the Ruletest interface with the following components:

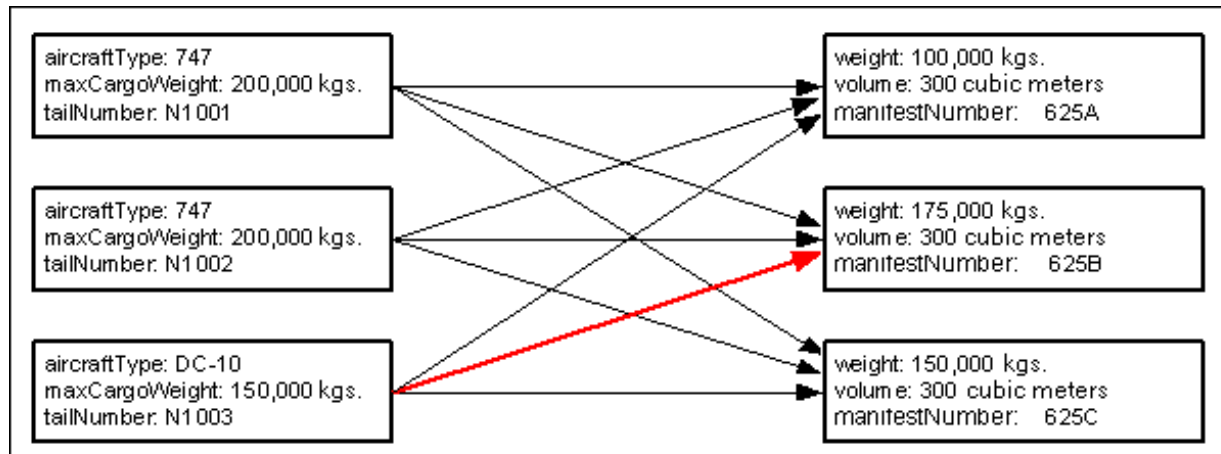
- Input Data:**
 - Aircraft [1]:** aircraftType [747], maxCargoWeight [200000], tailNumber [N101]
 - Aircraft [2]:** aircraftType [747], maxCargoWeight [200000], tailNumber [N102]
 - Aircraft [3]:** aircraftType [DC-10], maxCargoWeight [150000], tailNumber [N103]
 - Cargo [1]:** manifestNumber [625A], weight [100000]
 - Cargo [2]:** manifestNumber [625B], weight [175000]
 - Cargo [3]:** manifestNumber [625C], weight [150000]
- Output Data:**
 - Aircraft [1]:** aircraftType [747], maxCargoWeight [200000.000000], tailNumber [N101]
 - Aircraft [2]:** aircraftType [747], maxCargoWeight [200000.000000], tailNumber [N102]
 - Aircraft [3]:** aircraftType [DC-10], maxCargoWeight [150000.000000], tailNumber [N103]
 - Cargo [1]:** manifestNumber [625A], weight [100000.000000]
 - Cargo [2]:** manifestNumber [625B], weight [175000.000000]
 - Cargo [3]:** manifestNumber [625C], weight [150000.000000]
- Rule Messages Table:**

Severity	Message	Entity
Violation	Cargo [625B] is too heavy for Aircraft [N103]	Cargo[2]
Info	Cargo [625C] may be carried by Aircraft [N101]	Cargo[3]
Info	Cargo [625B] may be carried by Aircraft [N101]	Cargo[2]
Info	Cargo [625A] may be carried by Aircraft [N101]	Cargo[1]
Info	Cargo [625C] may be carried by Aircraft [N103]	Cargo[3]
Info	Cargo [625A] may be carried by Aircraft [N103]	Cargo[1]
Info	Cargo [625C] may be carried by Aircraft [N102]	Cargo[3]
Info	Cargo [625B] may be carried by Aircraft [N102]	Cargo[2]
Info	Cargo [625A] may be carried by Aircraft [N102]	Cargo[1]

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 Corticon 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 48: 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 kilograms, while the cargo weighs 175,000 kilograms. 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 42: Expressing the Rule Using Root-Level Vocabulary Terms](#) on page 60, 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 49: Rule Expressed Using `FlightPlan` as the Rule Scope

Ref	Post	Alias	Text
1	Violation	FlightPlan	Cargo [{FlightPlan.cargo.manifestNumber}] is too heavy for Aircraft [{FlightPlan.aircraft.tailNumber}]
2	Info	FlightPlan	Cargo [{FlightPlan.cargo.manifestNumber}] may be carried by Aircraft [{FlightPlan.aircraft.tailNumber}]

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 questions: Rule scope and context](#)

Rule scope

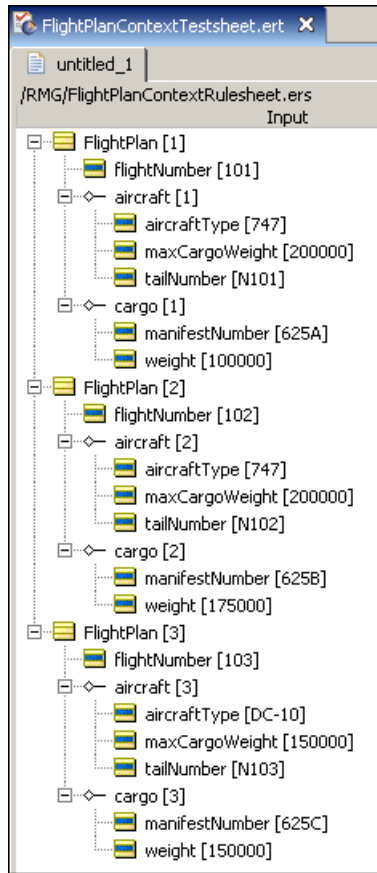
Because the rule is evaluating both `Cargo` and `Aircraft` in the context of a `FlightPlan`, we say that the rule has *scope*, which means that *the rule evaluates only that data which matches the rule's scope*. This has an interesting effect on the way the rule is evaluated. When the rule is executed, its scope ensures that the Corticon Server evaluates only those pairings that *match 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 Ruletests, 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 [Figure 45: The 3 FlightPlans with their related Aircraft and Cargo instances](#) on page 62. The rule will evaluate these combinations and identify any violations.

Figure 50: 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 51: Ruletest Results Using Scope – Note no Violations

The screenshot displays the Ruletest Results Using Scope interface for the rule *FlightPlanScope.ers*. The interface is divided into two main sections: **Input** and **Output**, each showing a hierarchical tree of entities and their attributes. The **Input** tree shows three *FlightPlan* entities, each with associated *Aircraft* and *Cargo* entities. The **Output** tree shows the same structure but with additional *maxCargoWeight* values. The **Rule Statements** section at the bottom shows three *Info* messages indicating that specific cargo items can be carried by specific aircraft.


Severity	Message	Entity
Info	Cargo [625A] may be carried by Aircraft [N101]	FlightPlan[1]
Info	Cargo [625B] may be carried by Aircraft [N102]	FlightPlan[2]
Info	Cargo [625C] may be carried by Aircraft [N103]	FlightPlan[3]

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

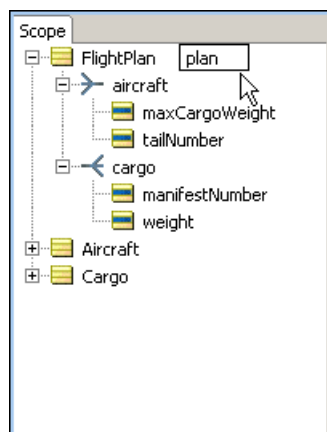
To clean up and simplify rule expression, Corticon Studio allows you to declare *aliases* in a Rulesheet Using an alias to express scope results in a less cluttered Rulesheet.

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:

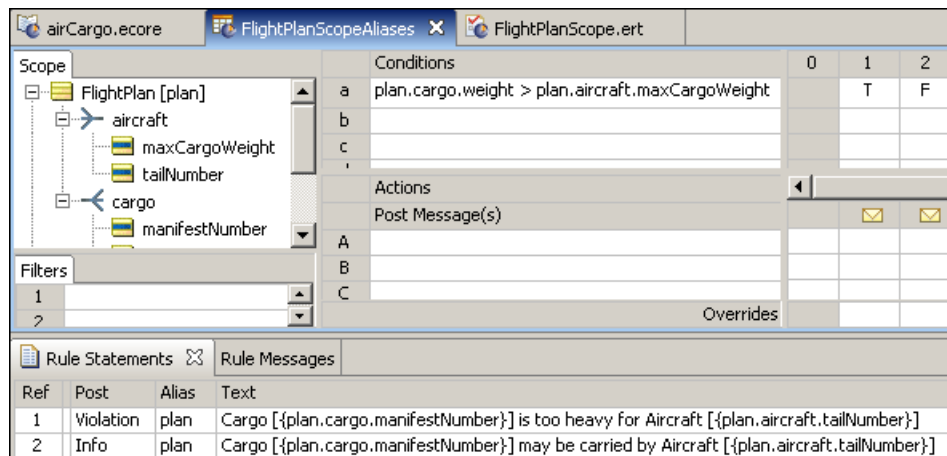
Figure 52: Defining an Alias in the Scope window



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 53: 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`.

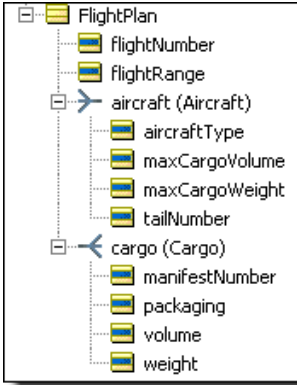
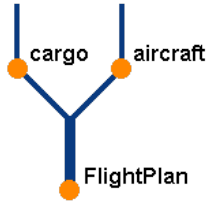
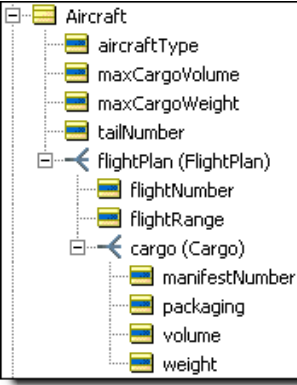

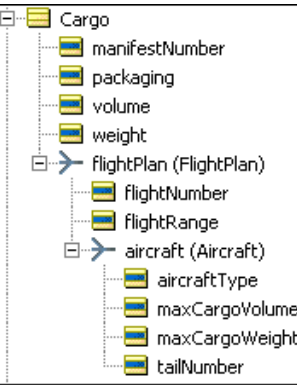

Aliases work in all sections of the Rulesheet, including the **Rule Statement** section. Modifying an alias name defined in the Scope section causes the name to update everywhere it is used in the Rulesheet.

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

Vocabulary Tree	Description	Branch Diagram
 <p>A screenshot of a vocabulary tree with 'FlightPlan' as the root. It branches into 'FlightNumber', 'FlightRange', 'aircraft (Aircraft)', and 'cargo (Cargo)'. 'aircraft (Aircraft)' further branches into 'aircraftType', 'maxCargoVolume', 'maxCargoWeight', and 'tailNumber'. 'cargo (Cargo)' branches into 'manifestNumber', 'packaging', 'volume', and 'weight'.</p>	<p>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 <code>FlightPlan</code> root or trunk. The <code>FlightPlan</code>'s associated <code>cargo</code> and <code>aircraft</code> terms are branches from the trunk.</p> <p>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.</p>	 <p>A branch diagram showing a vertical trunk labeled 'FlightPlan' at the bottom. Two branches, labeled 'cargo' and 'aircraft', originate from the top of the trunk.</p>
 <p>A screenshot of a vocabulary tree with 'Aircraft' as the root. It branches into 'aircraftType', 'maxCargoVolume', 'maxCargoWeight', 'tailNumber', 'flightPlan (FlightPlan)', and 'cargo (Cargo)'. 'flightPlan (FlightPlan)' branches into 'flightNumber' and 'flightRange'. 'cargo (Cargo)' branches into 'manifestNumber', 'packaging', 'volume', and 'weight'.</p>	<p>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>.</p> <p>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.</p>	 <p>A branch diagram showing a vertical trunk labeled 'Aircraft' at the bottom. A branch labeled 'flightPlan' originates from the top of the trunk. From the top of the 'flightPlan' branch, another branch labeled 'cargo' originates.</p>
 <p>A screenshot of a vocabulary tree with 'Cargo' as the root. It branches into 'manifestNumber', 'packaging', 'volume', 'weight', 'flightPlan (FlightPlan)', and 'aircraft (Aircraft)'. 'flightPlan (FlightPlan)' branches into 'flightNumber' and 'flightRange'. 'aircraft (Aircraft)' branches into 'aircraftType', 'maxCargoVolume', 'maxCargoWeight', and 'tailNumber'.</p>	<p>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>.</p> <p>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.</p>	 <p>A branch diagram showing a vertical trunk labeled 'Cargo' at the bottom. A branch labeled 'flightPlan' originates from the top of the trunk. From the top of the 'flightPlan' branch, another branch labeled 'aircraft' originates.</p>

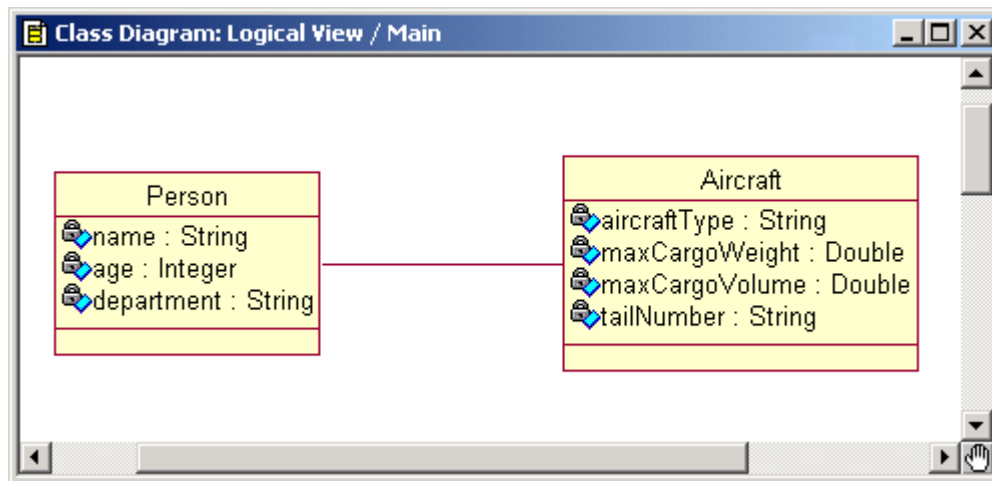
Scope can also be thought of as hierarchical, meaning that a rule written with scope of `Aircraft` applies to all root-level `Aircraft` data. And other rules using some piece (or branch) of the tree beginning with root term `Aircraft`, including `Aircraft.flightPlan` and `Aircraft.flightPlan.cargo`, 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 Rulesheets 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 54: UML Class Diagram without Roles



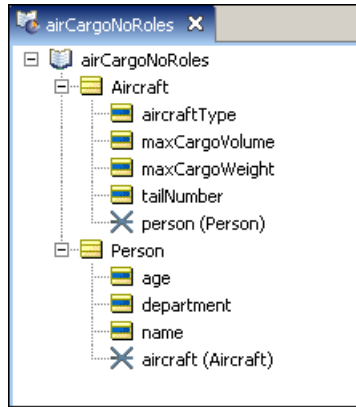
As shown in this class diagram, the entities `Person` and `Aircraft` are joined by an association. However, can this single association sufficiently represent multiple relationships between these entities? For example, a prior Fact Model might state that “a pilot flies an aircraft” and “a passenger rides 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:

1. By FAA regulations, 747 aircraft must be flown by at least 2 pilots
2. A DC-10 may not carry more than 200 passengers

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 55: 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`).

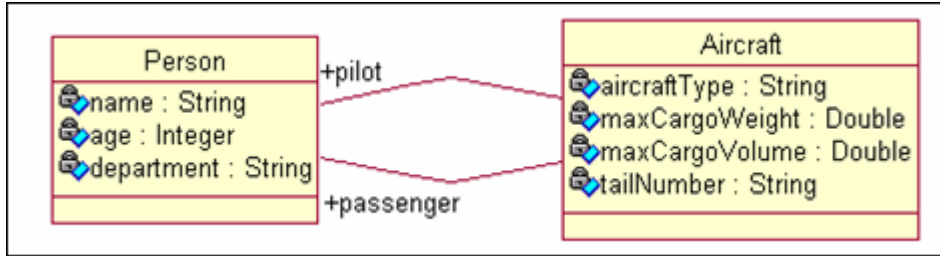
Add an Attribute to Person

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

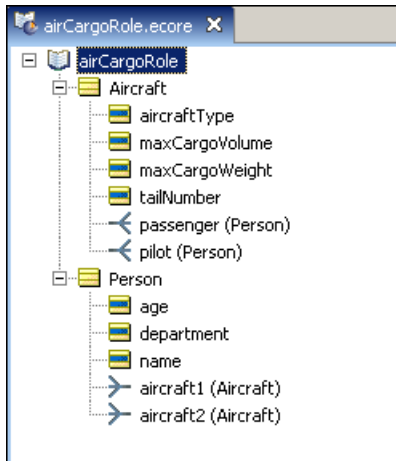
A role is a noun that labels one end of an association between two entities. For example, in our `Person-Aircraft` Vocabulary, the `Person` may have more than one role, or more than one kind of relationship, with `Aircraft`. An instance of `Person` may be a `pilot` or a `passenger`; each is a different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

Figure 56: UML Class Diagram with Roles



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

Figure 57: 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 58: Rule #1 Implemented using Roles

The screenshot shows the Corticon Studio interface for Rule #1. The **Scope** tree on the left shows the hierarchy: `Aircraft [plane]` containing `aircraftType` and `pilot [pilotOfPlane]`. The **Conditions** table has four rows (a, b, c, d) and four columns (0, 1, 2, 3). Row a contains `plane.aircraftType` with values `'747'`. Row b contains `pilotOfPlane -> size` with values `{ 0, 1 }`. The **Actions** table shows a `Post Message(s)` action for the `plane` alias. The **Rule Statements** table at the bottom lists three statements:

Ref	ID	Post	Alias	Text
1		Violation	plane	Exactly 2 pilots are required to fly a 747 - fewer than 2 violates FAA regulations
2		Info	plane	Exactly 2 pilots are required to fly a 747 - 2 are assigned to this flight
3		Warning	plane	Exactly 2 pilots are required to fly a 747 - more than 2 is unnecessary but not unsafe

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 59: Ruletest with no `Person` entities in `Pilot` role

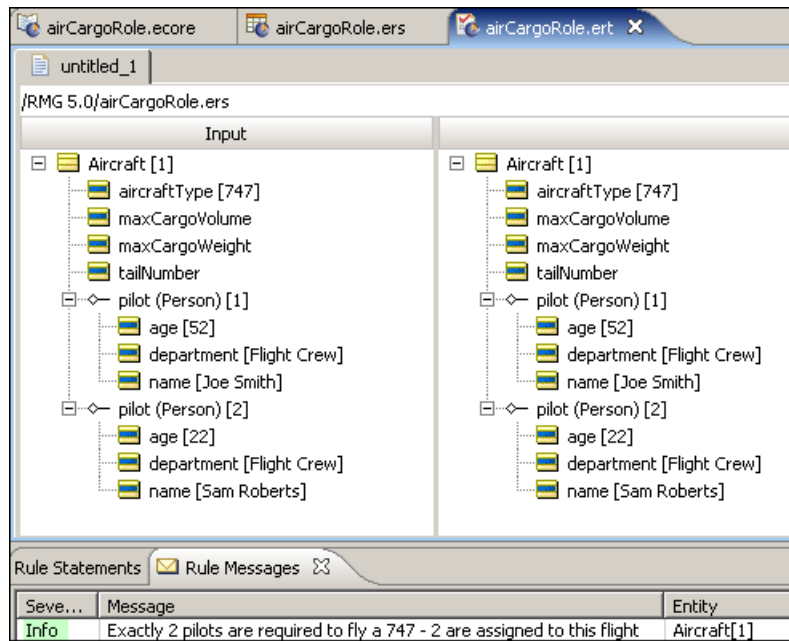
The screenshot shows the Corticon Studio interface for a Ruletest. The **Input** tree on the left shows the hierarchy: `Aircraft [1]` containing `aircraftType [747]`, `maxCargoVolume`, `maxCargoWeight`, and `tailNumber`. It also contains two `Person` entities: `Person [1]` (age [25], department [Flight Crew], name [Joe Smith]) and `Person [2]` (age [32], department [Flight Crew], name [Bob Roberts]). The **Rule Messages** table at the bottom shows a single message:

Severity	Message	Entity
Violation	Exactly 2 pilots are required to fly a 747 - fewer than 2 violates FAA regulations	Aircraft[1]

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.

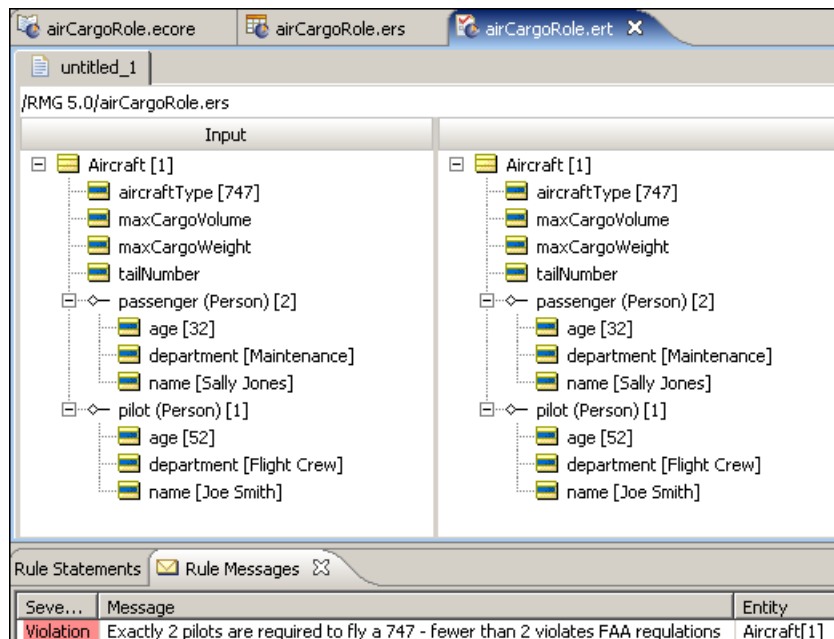
If we create a new Input Ruletest, this time with both persons in the role of pilot, we see a different result, as shown:

Figure 60: Ruletest with both Person entities in role of Pilot



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

Figure 61: 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 62: 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 *Corticon 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 kilograms. 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”.

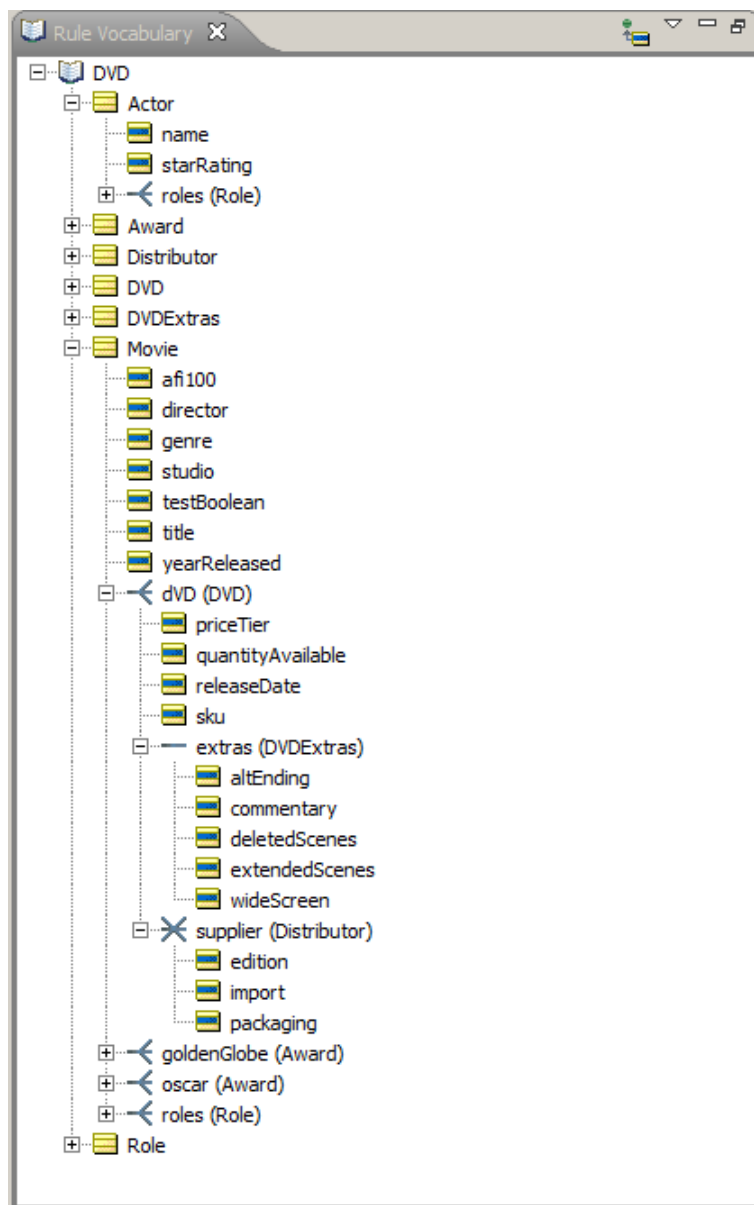
When `FlightPlan` is used as the scope for our rule, Corticon Studio automatically ensures that the collection of data contains matching foreign keys. That's why, when we rewrote the rule using proper scope, the rule only fired 3 times – there are only 3 examples of `Aircraft-Cargo` combinations where 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 questions: Rule scope and context

Note: Try this test, and then go to [Test yourself answers: Rule scope and context](#) on page 349 to correct yourself.

Use the following Vocabulary to answer the next questions.



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
----------	--------	----------	------------

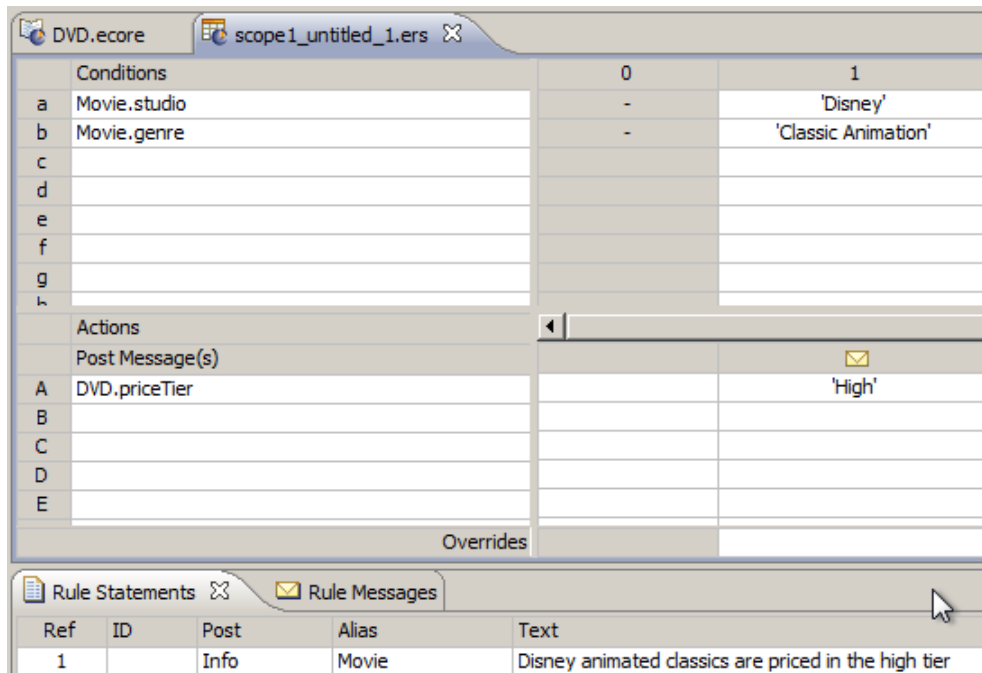
12. True or False. Rules evaluate only data that shares the same scope
13. Write a conditional expression in a Rulesheet for each of the following phrases:

- 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:

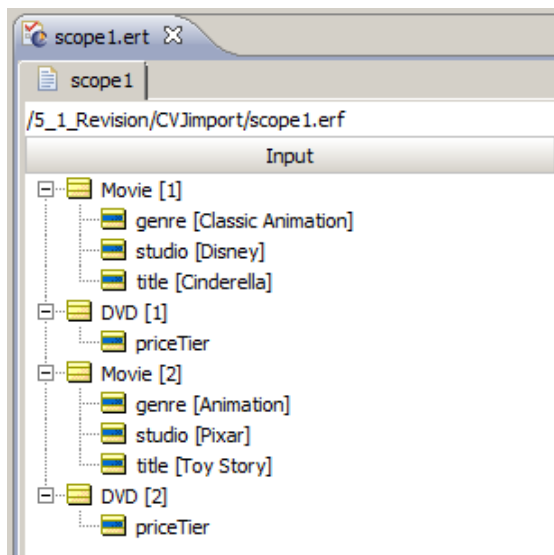
- Which term should be used to represent Movie?
- Which term should be used to represent DVD?
- True or False. The following Rulesheet correctly relates the Movie and DVD entities?



Conditions		0	1
a	Movie.studio	-	'Disney'
b	Movie.genre	-	'Classic Animation'
c			
d			
e			
f			
g			
h			
Actions			
Post Message(s)			
A	DVD.priceTier		High
B			
C			
D			
E			
Overrides			

Ref	ID	Post	Alias	Text
1		Info	Movie	Disney animated classics are priced in the high tier

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



- 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:

scope1Untitled_1.ers				
Conditions		0	1	2
a	Movie.studio	-	'Disney'	not {'Disney', 'MGM', 'BBC', 'PBS', 'Pixar'}
b	Movie.genre	-	'Classic Animation'	'Animation'
c				
d				
e				
f				
g				
h				
Actions				
Post Message(s)				
A	DVD.priceTier		High	Low
B				
C				
D				
E				
Overrides				
Rule Statements				
Ref	ID	Post	Alias	Text
1		Info	Movie	Disney animated classics are priced in the high tier
2		Warning	Movie	Other animated movies are priced in the low tier

- 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?

Rule writing techniques and 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:

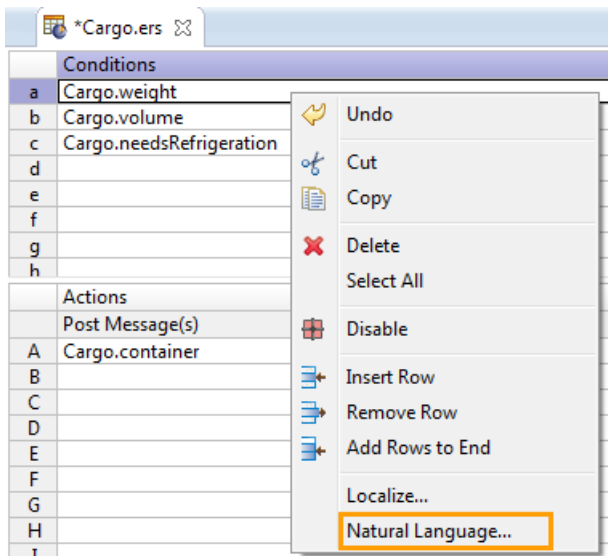
- [Working with rules and filters in natural language](#)
- [Filters vs. conditions](#)
- [Qualifying rules with ranges and lists](#)
- [Creating rules by importing an Excel worksheet](#)
- [Using standard boolean constructions](#)
- [Finding references to an entity, attribute, or association in a project](#)
- [Embedding attributes in posted rule statements](#)
- [Including apostrophes in strings](#)
- [Test yourself questions: Rule writing techniques and logical equivalents](#)

Working with rules and filters in natural language

Progress Corticon lets you use Natural Language (NL) words, phrases, and sentences as substitute terms in Rulesheet conditions and actions, making it easier to discuss the rules with stakeholders and analysts.

To use natural language on a Rulesheet:

1. Right-click within a Rulesheet, and then choose the dropdown menu command **Natural Language**, as shown:



Note: If the **Natural Language** window does not open, choose the menu command **Window>Show View>Natural Language**.

The Natural Language view typically places itself above the Rulesheet, as shown:

The screenshot shows two windows. The top window, titled 'Natural Language', has a table with 'Condition Expression' and 'Action Expression' columns, both set to 'English (United States)'. The conditions listed are 'a Cargo.weight', 'b Cargo.volume', and 'c Cargo.needsRefrigeration'. The action listed is 'A Cargo.container'. The bottom window, titled '*Cargo.ers', is a Rulesheet. It has a table with columns 0 through 4. The 'Conditions' section shows logical expressions for each condition. The 'Actions' section shows the container types for each condition.

Conditions	0	1	2	3	4
a Cargo.weight		<= 20000	-	> 20000	-
b Cargo.volume		-	> 30	<= 30	-
c Cargo.needsRefrigeration		-	-	-	T

Actions	0	1	2	3	4
Post Message(s)		✉	✉	✉	✉
A Cargo.container		standard	oversize	heavyweight	reefer


2. Enter plain language descriptive text for each condition and action, as shown:

The screenshot shows the 'Natural Language' window with descriptive text entered for each condition and action.

Condition Expression	English (United States)
a Cargo.weight	If the cargo weight is...
b Cargo.volume	If the cargo volume is...
c Cargo.needsRefrigeration	Does the cargo need refrigeration?


Action Expression	English (United States)
A Cargo.container	Then the type of container required by this cargo is...

While your use of natural language might vary, it is good practice to use a consistent, clear style. Here are some tips:

- Use **If** in the text for Conditions and **Then** in the text for Actions.
 - Conditions that are **True/False** often read better as questions.
 - Adding ... helps a reader continue the expression with the values in its column cells.
 - If you enter no natural language text, the existing expression is shown.
3. Expose your natural language expressions in the Rulesheet by either clicking the **Show Natural Language** toolbar button , or the menu command **Rulesheet > Show Natural Language**. The natural language is displayed as shown:

Cargo.ers		0	1	2	3
Conditions					
a	If the Cargo's weight is...	-	<= 20000	-	> 20000
b	If the Cargo's volume is...	-	-	> 30	<= 30
c	If the Cargo needs refrigeration...	-	-	-	-
d					
Actions					
Post Message(s)					
A	Then the type of container required by this Cargo is...		standard	oversize	heavyweight
B					
C					
D					
Overrides				{1, 4}	

In Natural Language mode, the values in rule columns can be edited but the Condition and Action expressions are locked and cannot be edited.

- Save the Rulesheet to store its expressions as well as its natural language data.
- You can revert to the actual, editable expressions by clicking the **Hide Natural Language** toolbar button , or the menu command **Rulesheet > Hide Natural Language**.

- Close the **Natural Language** view by clicking its close button.

Using natural language as an aid to Rulesheet design

You can create Natural Language phrases for the conditions, actions, and filters *before* defining those expressions.

Natural Language

Filter Expression	English (United States)
1 Cargo.weight < Aircraft.maxCargoWeight	Reject any package that exceeds the assigned aircraft weight capacity
2	Reject any package that exceeds the assigned aircraft volume capacity
3	
Condition Expression	English (United States)
a Cargo.weight	What is the weight (in kilograms) of the package?
b Cargo.volume	What is the volume (LxWxH in cubic meters) of the package?
c	
Action Expression	English (United States)
A Cargo.container	Then use this type of container...
B	

*Cargo.ers

Scope

- Aircraft
- Cargo

Filters

- 1 Reject any package that exceeds the assigned aircraft weight capacity
- 2 Reject any package that exceeds the assigned aircraft volume capacity
- 3

Conditions		0	1	2	3
a	What is the weight (in kilograms) of the package?		<= 20000	-	> 20000
b	What is the volume (LxWxH in cubic meters) of the package?		-	> 30	<= 30
c					
Actions					
Post Message(s)					
A	Then use this type of container...		standard	oversize	heavyweight
B					
C					
Overrides				1	

Adding the natural language phrase makes the next line available for additional entries. Then, in the Rulesheet, define the expression that satisfies the natural language phrase, as shown:

The screenshot shows two windows from the Corticon Studio interface. The top window, titled 'Natural Language', displays three sections: 'Filter Expression', 'Condition Expression', and 'Action Expression', all set to 'English (United States)'. The 'Filter Expression' section lists two rules: '1 Cargo.weight < Aircraft.maxCargoWeight' with the description 'Reject any package that exceeds the assigned aircraft weight capacity', and '2 Cargo.volume < Aircraft.maxCargoVolume' with the description 'Reject any package that exceeds the assigned aircraft volume capacity'. The 'Condition Expression' section lists three rules: 'a Cargo.weight' with the description 'What is the weight (in kilograms) of the package?', 'b Cargo.volume' with the description 'What is the volume (LxWxH in cubic meters) of the package?', and 'c' (empty). The 'Action Expression' section lists two rules: 'A Cargo.container' with the description 'Then use this type of container...', and 'B' (empty).

The bottom window, titled '*Cargo.ers', shows a Rulesheet. The 'Scope' section lists 'Aircraft' and 'Cargo'. The 'Filters' section lists two rules: '1 Cargo.weight < Aircraft.maxCargoWeight' and '2 Cargo.volume < Aircraft.maxCargoVolume'. The 'Conditions' section lists three rules: 'a Cargo.weight', 'b Cargo.volume', and 'c'. The 'Actions' section lists two rules: 'A Cargo.container' and 'B'. The 'Overrides' section shows a table with columns 0, 1, 2, and 3, and rows for 'Post Message(s)', 'standard', 'oversize', and 'heavyweight'. The 'Overrides' table shows that for rule 'A', the 'standard' override is selected, and for rule 'B', the 'oversize' override is selected.

Localization with natural language

When your stakeholders are comfortable in different natural languages, you can accommodate them easily with the natural language feature.

When you enable locales, the Natural Language window adds columns for the other locales. You can then define Natural Language text for each of those locales, as illustrated:

The screenshot shows the 'Natural Language' window with columns for 'English (United States)', 'French', 'Portuguese (Brazil)', and 'Spanish'. The 'Condition Expression' section lists three rules: 'a Cargo.weight' with the description 'If the cargo weight is...', 'b Cargo.volume' with the description 'If the cargo volume is...', and 'c Cargo.needsRefrigeration' with the description 'If the cargo must be refrigerated...'. The 'Action Expression' section lists two rules: 'A Cargo.container' with the description 'Then the type of container required by this cargo is...', and 'B' (empty).

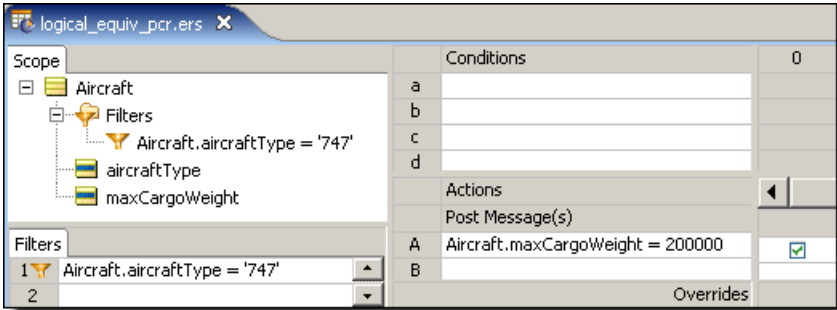
See the topic [Localizing Corticon Studio](#) on page 319 for more information.

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:

1. A 747 has a maximum cargo weight of 200,000 kilograms.

Figure 63: Rulesheet Using a Filter and Nonconditional Rule

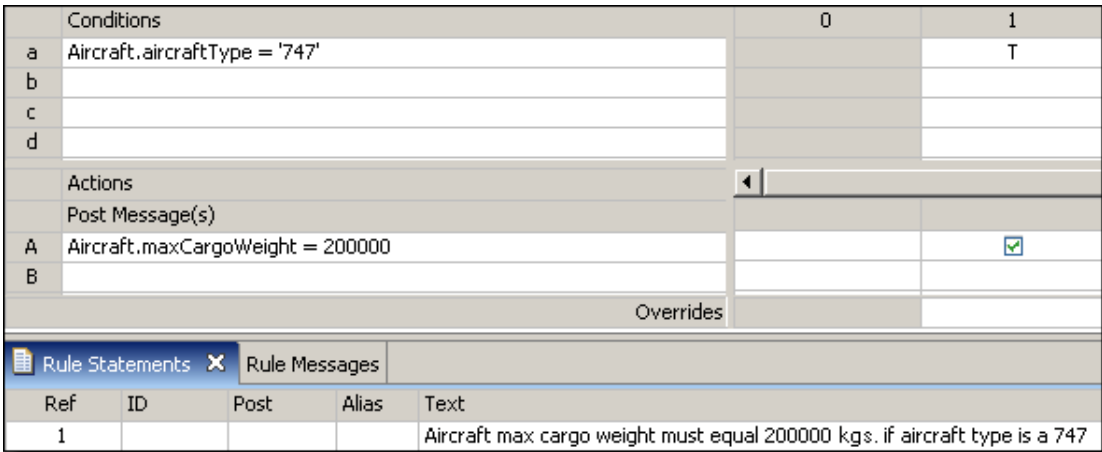


Here, the value of an Aircraft's `maxCargoWeight` attribute is assigned by column 0 in the Conditions/Actions pane (what we sometimes call a “Nonconditional” rule because it has no Conditions). The Filter acts as a master conditional expression because only Aircraft that satisfy the Filter - in other 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 kilograms. Using this method, additional Rulesheets 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 kilograms”.

Figure 64: 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 Rulesheets. 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 Rulesheets to express these same rules using the Filters section. This leads us to the next topic of discussion.

Qualifying rules with ranges and lists

You can use values -- Integers, Decimals, Strings, or Date/Time/DateTime data types -- in Conditions, Condition cells, and Filters.

These values can be imprecise -- they can be in the form of a *range* expressed in the format: `x . . y`, where `x` and `y` are the starting and ending values for the range.

The values can also be very specific -- they can be in the form of a *list* expressed in the format `{x, z, y}`, where the values are in any order but must adhere to the data type or the defined labels when the data type is bound to an enumerated list with labels.

Using ranges and lists in conditions and filters

Conditions and filters can qualify data by testing for inclusion in a *from-to* range of values or in a comma-delimited list. The result returned is `true` or `false`. All attribute data types except Boolean can use ranges and lists in conditions and filters.

Value ranges in condition and filter expressions

You can use value range expressions in conditions or filters.

Syntax of value ranges in conditions and filter rows

When you use the `in` operator to specify a range of values, you can specify the range in a several ways. The following illustration shows how you can encapsulate a range:

Figure 65: Rulesheet Filters showing ways to encapsulate a range

Filters	
7	
8	Entity_1.integer1 in 100..300
9	Entity_1.integer1 in {100..300}
10	Entity_1.integer1 in (100..300)
11	Entity_1.integer1 in [100..300]
12	Entity_1.integer1 in (100..300]
13	Entity_1.integer1 in [100..300]

Filter 8 does no encapsulation, Filter 9 uses braces, and Filter 10 uses parentheses. All these are valid but cannot be mixed. However, the use of parentheses and square brackets can be mixed to provide additional capability. The bracket on either side expresses the value on that side passes the test. In Filters 10 through 13, the brackets indicate the permutations of this capability. It is a good practice to routinely use parentheses for exclusionary ranges.

Examples of value ranges in filter rows

The following value ranges show how the Corticon data types can be used as Filter expressions.

Figure 66: Rulesheet filters showing the syntax of ranges for each data type

Filters	
1	Entity_1.dateOnly1 in ['1/1/15'..'12/31/17']
2	Entity_1.dateTime1 in ('12/25/15 00:00:00'..'12/25/15 9:59:59')
3	Entity_1.decimal1 in [-.01..99.99)
4	Entity_1.integer1 in (-128.6..136.4)
5	Entity_1.string1 in ['a'..'z'] or Entity_1.string1 in ['A'..'Z']
6	Entity_1.timeOnly1 in ('9:00 AM'..'5:00 PM')
7	

Notice that ranges are always *from..to*. The examples show that negative decimal and integer values can be used, and that uppercase and lowercase characters are filtered separately.

Value lists in condition and filter expressions

You can use value list expressions in conditions or filters.

Syntax of value list in conditions and filter rows

When you use the `in` operator to specify a list of values, you can encapsulate the range in only one way:

Figure 67: Rulesheet Filters showing encapsulation of a list

Filters	
1	E1.a1 in {RED,BLUE,YELLOW}
2	
3	

The value list is always enclosed in braces. The order of the items in the comma-delimited list is arbitrary.

Using ranges and value sets 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 68: Rulesheet Using Value Ranges in the Column Cells of a Condition Row

The screenshot shows the 'ValueRanges.ers' rulesheet. The 'Conditions' section has a table with 5 columns (0-4) and 3 rows (a-c). Row 'a' contains the condition 'FlightPlan.FlightNumber' with values '<=100', '101..200', '201..300', and '>300'. The 'Actions' section has a table with 5 columns (0-4) and 2 rows (A). Row 'A' contains the action 'FlightPlan.aircraft.maxCargoWeight' with values '50000', '100000', '150000', and '200000'. The 'Rule Statements' section shows 4 rules with their respective text descriptions.

Ref	ID	Post	Alias	Text
1				Aircraft max cargo weight must be 50000 when flight number is less than or equal to 100
2				Aircraft max cargo weight must be 100000 when flight number is between 101 and 200, inclusive
3				Aircraft max cargo weight must be 150000 when flight number is between 201 and 300, inclusive
4				Aircraft max cargo weight must be 200000 when flight number is greater than 300

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 69: Rulesheet Using Open-Ended Value Ranges in Condition Cells

The screenshot shows the 'ValueRangesExclusiveInclusive' rulesheet. The 'Conditions' section has a table with 5 columns (0-4) and 3 rows (a-c). Row 'a' contains the condition 'FlightPlan.FlightNumber' with values '<=100', '101..200', '(200..300]', and '>300'. The 'Actions' section has a table with 5 columns (0-4) and 2 rows (A). Row 'A' contains the action 'FlightPlan.aircraft.maxCargoWeight' with values '50000', '100000', '150000', and '200000'. The 'Rule Statements' section shows 4 rules with their respective text descriptions.

Ref	ID	Post	Alias	Text
1				Aircraft max cargo weight must be 50000 when flight number is less than or equal to 100
2				Aircraft max cargo weight must be 100000 when flight number is between 101 and 200, inclusive
3				Aircraft max cargo weight must be 150000 when flight number is between 201 and 300, inclusive
4				Aircraft max cargo weight must be 200000 when flight number is greater than 300

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

Figure 70: 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 [Using ranges and lists in conditions and filters](#) on page 89 for details on usage.

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

ValueSetsInConditionalRules.ers					
Conditions		0	1	2	3
a	Aircraft.aircraftType		'DC-10'	'A340'	'747'
b					
c					
d					
Actions					
Post Message(s)					
A	Aircraft.maxCargoWeight		100000	150000	200000
B					
Overrides					

Rule Statements		Rule Messages		Problems	
Ref	ID	Post	Alias	Text	
A1				Aircraft max cargo weight must be 100000 when aircraft type is a DC-10	
A2				Aircraft max cargo weight must be 150000 when aircraft type is an A340	
A3				Aircraft max cargo weight must be 200000 when aircraft type is a 747	

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 72: Exclusionary Logic Using Boolean Condition, Pt. 1

airCargo.ecore		ExclusionarySyntax.ers	
Conditions		0	1
a	Aircraft.aircraftType <> '747'		T
b			
-			
Actions		◀	
Post Message(s)			✉
A	Aircraft.maxCargoWeight = 100000		✓
B			
-			
Overrides			

Rule Statements		Rule Messages		
Ref	ID	Post	Alias	Text
1	Info	Aircraft		Aircraft max cargo weight must be 100,000 when aircraft type is NOT a 747

Figure 73: Exclusionary Logic Using Boolean Condition, Pt. 2

airCargo.ecore		ExclusionarySyntax.ers	
Conditions		0	1
a	Aircraft.aircraftType = '747'		F
b			
-			
Actions		◀	
Post Message(s)			✉
A	Aircraft.maxCargoWeight = 100000		☑
B			
-			
Overrides			

Rule Statements		Rule Messages		
Ref	ID	Post	Alias	Text
1	Info	Aircraft		Aircraft max cargo weight must be 100,000 when aircraft type is NOT a 747

Figure 74: Exclusionary Logic Using Negated Value

airCargo.ecore		ExclusionarySyntax.ers	
Conditions		0	1
a	Aircraft.aircraftType		not '747'
b			
-			
Actions			
Post Message(s)			
A	Aircraft.maxCargoWeight = 100000		<input checked="" type="checkbox"/>
B			
-			
Overrides			

Rule Statements					Rule Messages
Ref	ID	Post	Alias	Text	
1		Info	Aircraft	Aircraft max cargo weight must be 100,000 when aircraft type is NOT a 747	

Notice 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.

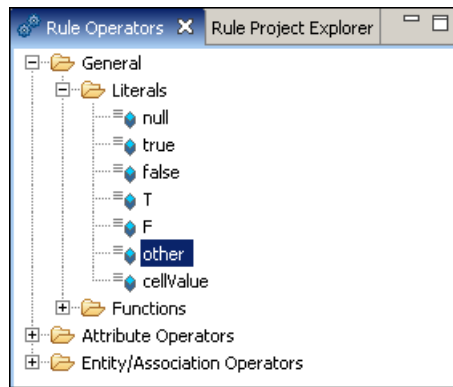
Once again we see that the same rule can be expressed in different ways on the Rulesheet, with identical results. It is left to the rule modeler to decide which way of expressing the rule is preferable in a given situation. We recommend, however, avoiding double negatives. Most people find it easier 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 75: 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 76: Rulesheet Using `other` in a Condition Cell

The screenshot shows the 'OtherOperator.ers' rulesheet editor. It contains a decision table with 5 columns (Conditions, 0, 1, 2, 3, 4) and 3 rows (a, b, c) for conditions, and 3 rows (A, B) for actions. The 'Conditions' row 'a' has values 'Aircraft.aircraftType', 'DC-10', 'A340', '747', and 'other'. The 'Actions' row 'A' has values 'Aircraft.maxCargoWeight', 100000, 150000, 200000, and 50000. Below the table is an 'Overrides' section. At the bottom, the 'Rule Statements' tab is active, showing a list of rule statements with their references, IDs, post-conditions, aliases, and text.

Conditions	0	1	2	3	4
a Aircraft.aircraftType		'DC-10'	'A340'	'747'	other
b					
c					

Actions	0	1	2	3	4
A Aircraft.maxCargoWeight		100000	150000	200000	50000
B					

Ref	ID	Post	Alias	Text
1				Aircraft max cargo weight equals 100,000 kgs when the aircraft is a DC-10
2				Aircraft max cargo weight equals 150,000 kgs when the aircraft is an A340
3				Aircraft max cargo weight equals 200,000 kgs when the aircraft is a 747
4				Aircraft max cargo weight equals 50,000 kgs for all other aircraft types

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.

Numeric value ranges in conditions

Figure 77: Rulesheet using Numeric Value Ranges in Condition Values Set

Conditions		0	1	2	3	4
a	Entity1.integer1		< 100	101..200	201..300	> 300
b						
c						
d						
e						
Actions						
Post Message(s)						
A	Entity1.integer2		50000	100000	150000	200000
B						
C						
Overrides						

Ref	ID	Post	Alias	Text
1				If integer1 is less than 100, then assign a value of 50000 to integer2
2				If integer1 is between 101 and 200, inclusive, then assign a value of 100000 to integer2
3				If integer1 is between 201 and 300, inclusive, then assign a value of 150000 to integer2
4				If integer1 is greater than 300, then assign a value of 200000 to integer2

In this example, we are assigning an `integer2` value to `Entity1` depending on its `integer1` value. The value range `101..200` represents all values (integers in this case) between 101 and 200, including 101 and 200. This is an inclusive range because both the starting and ending values are included in the range.

String value ranges in condition cells

When using value range syntax with String types, be sure to enclose literal values inside single quotes, as shown in the following figure. Corticon Studio will usually perform this for you, but always check to make sure it has interpreted your entries correctly.

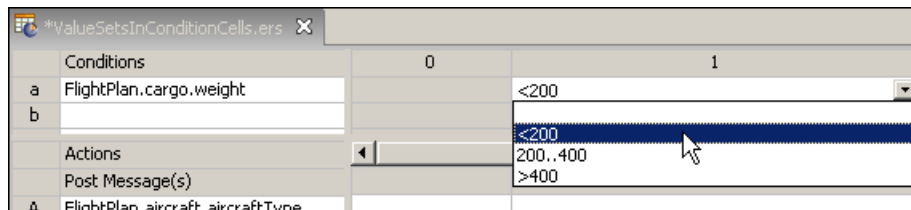
Figure 78: Rulesheet using String Value Ranges in Condition Values Set

Conditions		0	1	2	3
a	Entity1.string1		'a'..'z'	'A'..'Z'	other
b					
c					
d					
e					
Actions					
Post Message(s)					
A	Entity1.string2		'lower case'	'upper case'	'other char'
B					

Using value sets in condition cells

Most Conditions implemented in the Rules section of the Rulesheet use a single value in a Cell, as shown in the following figure:

Figure 79: Rulesheet with One Value Selected in Condition Cell



Sometimes, however, it is useful to combine more than one value in the same Cell. This is accomplished by holding **CTRL** while clicking to select multiple values from the Condition Cell drop-down box. When multiple values are selected in this manner, pressing **ENTER** will automatically enclose the resulting set in curly brackets { . . } in the Cell as shown in the sequence of [Rulesheet with Two Values Selected in Condition Cell](#) and [Rulesheet with Value Set in Condition Cell](#). Additional values may also be typed into Cells. Be sure the comma separators and curly brackets remain correct during hand-editing.

Figure 80: Rulesheet with Two Values Selected in Condition Cell

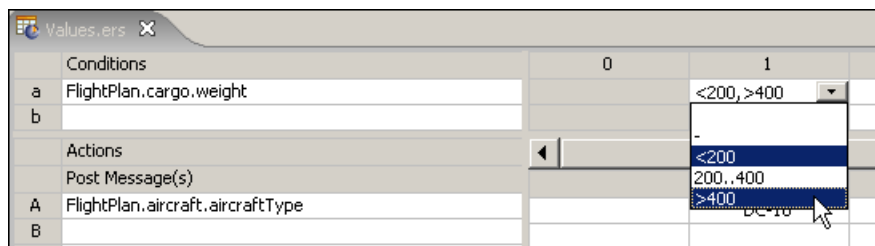
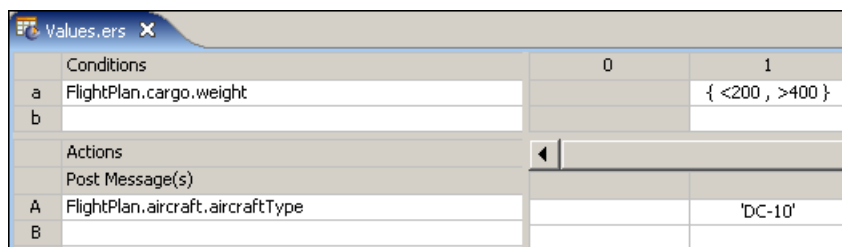


Figure 81: Rulesheet with Value Set in Condition Cell




The rule implemented in Column 1 of [Rulesheet with Value Set in Condition Cell](#) is logically equivalent to the Rulesheet shown in [Rulesheet with Two Rules in Lieu of Value Set](#). Both are implementations of the following rule statement:

1. If a flightplan's cargo weight is less than 200 **OR** greater than 400, then the flightplan's aircraft type must be a DC-10

Figure 82: Rulesheet with Two Rules in Lieu of Value Set

Conditions		0	1	2
a	FlightPlan.cargo.weight	-	< 200	> 400
b				
Actions				
Post Message(s)				
A	FlightPlan.aircraft.aircraftType		'DC-10'	'DC-10'
B				

If you write rules that are logically **OR**'ed in separate Columns, performing a Compression  will reduce the Rulesheet to the fewest number of Columns possible by creating value sets in Cells wherever possible. Fewer Columns results in faster Rulesheet execution, even when those Columns contain value sets. Compressing the Rulesheet in [Rulesheet with Two Rules in Lieu of Value Set](#) will result in the Rulesheet in [Rulesheet with Value Set in Condition Cell](#).

Condition Cell value sets can also be negated using the **not** operator. To negate a value, simply type **not** in front of the leading curly bracket **{** as shown in [Negating a Value Set in a Condition Cell](#). This is an implementation of the following rule statement:

1. If a flightplan's cargo weight is **NOT** less than 200 **OR NOT** greater than 400, then the flightplan's aircraft type must be a DC-10

which, given the Condition Cell's value set, is equivalent to:

1. If a flightplan's cargo weight is between 200 and 400 (inclusive), then the flightplan's aircraft type must be a DC-10

Figure 83: Negating a Value Set in a Condition Cell

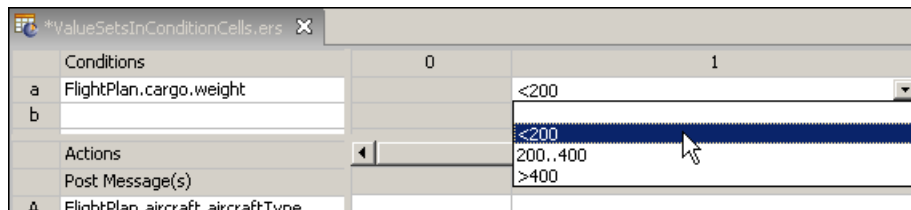
Conditions		0	1
a	FlightPlan.cargo.weight		not { <200 , >400 }
b			
Actions			
Post Message(s)			
A	FlightPlan.aircraft.aircraftType		'DC-10'

Value sets can also be created in the Overrides Cells at the foot of each Column. This allows one rule to override multiple rules in the same Rulesheet.

Using value lists in condition cells

Most Conditions implemented in the Rules section of the Rulesheet use a single value in a Cell, as shown in the following figure:

Figure 84: Rulesheet with One Value Selected in Condition Cell



Sometimes, however, it is useful to combine more than one value in the same Cell. This is accomplished by holding **CTRL** while clicking to select multiple values from the Condition Cell drop-down box. When multiple values are selected in this manner, pressing **ENTER** will automatically enclose the resulting set in curly brackets { . . } in the Cell as shown in the sequence of [Rulesheet with Two Values Selected in Condition Cell](#) and [Rulesheet with Value Set in Condition Cell](#). Additional values may also be typed into Cells. Be sure the comma separators and curly brackets remain correct during hand-editing.

Figure 85: Rulesheet with Two Values Selected in Condition Cell

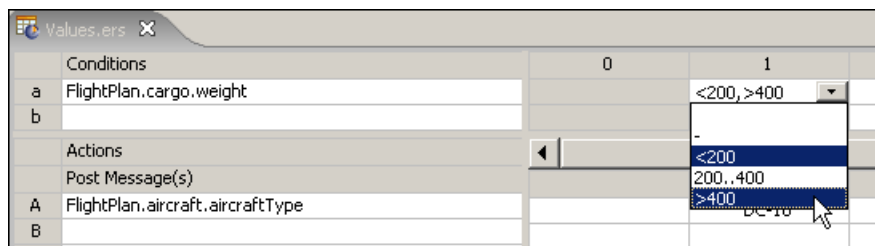
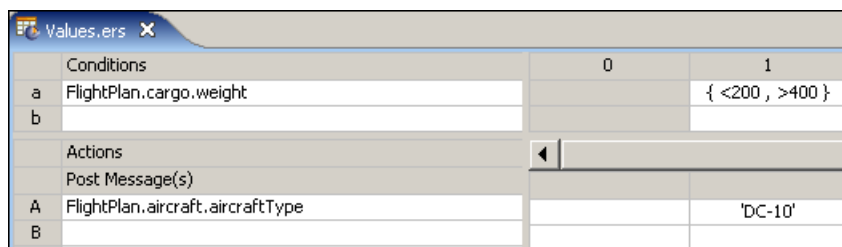


Figure 86: Rulesheet with Value Set in Condition Cell




The rule implemented in Column 1 of [Rulesheet with Value Set in Condition Cell](#) is logically equivalent to the Rulesheet shown in [Rulesheet with Two Rules in Lieu of Value Set](#). Both are implementations of the following rule statement:

1. If a flightplan's cargo weight is less than 200 **OR** greater than 400, then the flightplan's aircraft type must be a DC-10

Figure 87: Rulesheet with Two Rules in Lieu of Value Set

Conditions		0	1	2
a	FlightPlan.cargo.weight	-	< 200	> 400
b				
Actions				
Post Message(s)				
A	FlightPlan.aircraft.aircraftType		'DC-10'	'DC-10'
B				

If you write rules that are logically **OR**'ed in separate Columns, performing a Compression  will reduce the Rulesheet to the fewest number of Columns possible by creating value sets in Cells wherever possible. Fewer Columns results in faster Rulesheet execution, even when those Columns contain value sets. Compressing the Rulesheet in [Rulesheet with Two Rules in Lieu of Value Set](#) will result in the Rulesheet in [Rulesheet with Value Set in Condition Cell](#).

Condition Cell value sets can also be negated using the **not** operator. To negate a value, simply type **not** in front of the leading curly bracket **{** as shown in [Negating a Value Set in a Condition Cell](#). This is an implementation of the following rule statement:

1. If a flightplan's cargo weight is **NOT** less than 200 **OR NOT** greater than 400, then the flightplan's aircraft type must be a DC-10

which, given the Condition Cell's value set, is equivalent to:

1. If a flightplan's cargo weight is between 200 and 400 (inclusive), then the flightplan's aircraft type must be a DC-10

Figure 88: Negating a Value Set in a Condition Cell

Conditions		0	1
a	FlightPlan.cargo.weight		not { <200 , >400 }
b			
Actions			
Post Message(s)			
A	FlightPlan.aircraft.aircraftType		'DC-10'

Value sets can also be created in the Overrides Cells at the foot of each Column. This allows one rule to override multiple rules in the same Rulesheet.

Using variables as condition cell values

You can use a variable as a condition's cell value. However, there are constraints:

- Either **all** of the rule cell values for a condition row contain references to the *same* variable (with the exception of dashes), or **none** of the rule cell values for a condition row reference *any* variable.
- Only one variable can be referenced by various rules for the same condition row.
- Logical expressions in the various rules for the same condition row should be logically non-overlapping.

- A condition value that uses a colon, such as A:B, is not valid.

Derived value sets are created by accounting for all logical ranges possible around the variable.

The following Rulesheet uses the `Cargo` Vocabulary to illustrate the valid and invalid use of variables. Note that the Vocabulary editor marks invalid values in red.

	Conditions	0	1	2	3
a	Aircraft.maxCargoVolume		< Cargo.volume	> Cargo.volume	Cargo.volume
b	Aircraft.maxCargoVolume		<= Cargo.volume	> Cargo.volume	-
c	Aircraft.maxCargoVolume		< Cargo.volume	> Cargo.volume	-
d	Aircraft.maxCargoVolume		< Cargo.volume	-	-
e					
f	Aircraft.maxCargoVolume		< Cargo.volume	FlightPlan.cargo.volume	Cargo.volume
g	Aircraft.maxCargoVolume		< Cargo.volume	5	10..15
h	Aircraft.maxCargoVolume		< Cargo.volume	<= Cargo.volume	Cargo.volume
i	Aircraft.maxCargoVolume		A1:B2		

Derived values when using variables

The following tables abbreviate the attribute references shown in the illustration.

Table 2: Rulesheet columns

Conditions	1	2	3	Derived Value Set
A.maxCV	< C.v	> C.v	C.v	{< C.v, > C.v, C.v}
A.maxCV	<= C.v	> C.v		{<= C.v, > C.v }
A.maxCV	< C.v	> C.v		{< C.v, > C.v, C.v }
A.maxCV	< C.v			{< C.v, >= C.v}

Improper use of variables

Table 3: Rulesheet condition f: Attempt to use multiple variables

Conditions	1	2	3
A.maxCV	< C.v	> FP.c.v	C.v

Table 4: Rulesheet condition g: Attempt to mix variables and literals

Conditions	1	2	3
A.maxCV	< C.v	5	10..15

Table 5: Rulesheet condition h: Attempt to use logically overlapping expressions

Conditions	1	2	3
A.maxCV	< C.v	<= C.v	C.v

DateTime, date, and time value ranges in condition cells

When using value range syntax with date types, be sure to enclose literal date values inside single quotes, as shown:

Figure 89: Rulesheet using a Date Value Range in Condition Cells

The screenshot shows a Rulesheet titled 'DateandSubtypesinConditions.ers'. It has a table with 5 columns (Conditions, 0, 1, 2, 3) and 5 rows (a, b, c, d, e). Row 'a' contains the condition 'Entity1.dateTime1' and the ranges '<'1/1/2006'', ''1/1/2006'..'12/31/2006'', and '>'1/1/2007''. Below the table is an 'Actions' section with a 'Post Message(s)' field. At the bottom, there is a 'Rule Statements' tab with a table of 5 columns (Ref, ID, Post, Alias, Text) and 3 rows of rule statements.

Conditions	0	1	2	3
a Entity1.dateTime1		<'1/1/2006'	'1/1/2006'..'12/31/2006'	>'1/1/2007'
b				
c				
d				
e				

Ref	ID	Post	Alias	Text
1				If dateTime1 is before Jan. 1 2006, then string1 is assigned a value of 'earlier'
2				If dateTime1 is between Jan. 1 2006 and Dec. 31 2006, then string1 is assigned a value of 'current'
3				If dateTime1 is on or after Jan. 1 2007, then string1 is assigned a value of 'later'

Inclusive and exclusive ranges

Corticon Studio also gives you the option of defining value ranges where one or both of the starting and ending values are “exclusive”, meaning that the starting/ending value is **not** included in the range of values. [Rulesheet using an Integer Value Range in Condition Values Set](#) shows the same Rulesheet as in [Rulesheet using Numeric Value Ranges in Condition Values Set](#), 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 excluded – it is **not** included in the range of possible values. The ending bracket] indicates that the ending value is inclusive. Since integer1 is an Integer value, and therefore no fractional values are allowed, 201..300 and (200..300] are equivalent and our Values set in [Rulesheet using an Integer Value Range in Condition Values Set](#) is still complete as it was in [Rulesheet using Numeric Value Ranges in Condition Values Set](#).

Figure 90: Rulesheet using an Integer Value Range in Condition Values Set

The screenshot shows a Rulesheet titled 'RulesheetUsingAnIntegerValueRange.ers'. It has a table with 6 columns (Conditions, 0, 1, 2, 3, 4) and 5 rows (a, b, c, d, e). Row 'a' contains the condition 'Entity1.integer1' and the ranges '< 100', '101..200', '(200..300]', and '> 300'. Below the table is an 'Actions' section with a 'Post Message(s)' field. At the bottom, there is a 'Rule Statements' tab with a table of 5 columns (Ref, ID, Post, Alias, Text) and 4 rows of rule statements.

Conditions	0	1	2	3	4
a Entity1.integer1		< 100	101..200	(200..300]	> 300
b					
c					
d					
e					

Ref	ID	Post	Alias	Text
1				If integer1 is less than 100, then assign a value of 50000 to integer2
2				If integer1 is between 101 and 200, inclusive, then assign a value of 100000 to integer2
3				If integer1 is between 201 and 300, inclusive, then assign a value of 150000 to integer2
4				If integer1 is greater than 300, then assign a value of 200000 to integer2

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

`(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 [Rulesheet using Numeric Value Ranges in Condition Values Set](#) and [Rulesheet using an Integer Value Range in Condition Values Set](#), if a value range has no enclosing parentheses or brackets, it is assumed to be closed. It is therefore not necessary to use the `[..]` notation for a closed range in Corticon Studio; in fact, if you try to create a closed value range by entering `[..]`, 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)`.

When using range notation, always ensure `x` is less than `y`, i.e., an ascending range. A range where `x` is greater than `y` (a descending range) may result in errors during rule execution.

Overlapping value ranges

One final note about value ranges: they **might overlap**. In other words, Condition Cells may contain the two ranges `0..10` and `5..15`. It is important to understand that when overlapping ranges exists in rules, the rules containing the overlap are frequently ambiguous and more than one rule may fire for a given set of input Ruletest data.

Rulesheet with Value Range Overlap shows an example of value range overlap.

Figure 91: Rulesheet with Value Range Overlap

Conditions		1	2	3
a	Entity1.integer1	< 100	100..150	150..300
b				
Actions				
Post Message(s)				
A	Entity1.integer2	50000	100000	150000
R				
Overrides				

Ref	ID	Post	Alias	Text
1		Info	Entity1	integer1 is less than 100
2		Warning	Entity1	integer1 is between 100 and 200
3		Violation	Entity1	integer1 is between 150 and 300

Figure 92: Rulesheet expanded with Ambiguity Check applied

Conditions		1	2.1	2.2	3.1	3.2
a	Entity1.integer1	< 100	[100..150)	150	150	(150..300]
b						
Actions						
Post Message(s)						
A	Entity1.integer2	50000	100000	100000	150000	150000
R						
Overrides						

Ref	ID	Post	Alias	Text
1		Info	Entity1	integer1 is less than 100
2		Warning	Entity1	integer1 is between 100 and 200
3		Violation	Entity1	integer1 is between 150 and 300

Figure 93: Ruletest showing multiple rules firing for given test data

Input		Output	
Entity1 [1] integer1 [175]		Entity1 [1] integer1 [175] integer2 [150000]	

Severity	Message	Entity
Warning	integer1 is between 100 and 200	Entity1[1]
Violation	integer1 is between 150 and 300	Entity1[1]

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 94: Rulesheet Using Boolean Conditions to Express Value Ranges

Conditions		0	1	2	3	4
a	FlightPlan.flightNumber > 100		F	T	T	T
b	FlightPlan.flightNumber > 200		F	F	T	T
c	FlightPlan.flightNumber > 300		F	F	F	T
Actions						
Post Message(s)						
A	FlightPlan.aircraft.maxCargoWeight					
Overrides						

Ref	ID	Post	Alias	Text
A1				Aircraft max cargo weight must be 50000 kgs when flight number is less than or equal to 100
A2				Aircraft max cargo weight must be 100000 kgs when flight number is between 101 and 200, inclusive
A3				Aircraft max cargo weight must be 150000 kgs when flight number is between 201 and 300, inclusive
A4				Aircraft max cargo weight must be 200000 kgs when flight number is greater than 300

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 50000 kgs.

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

An Aircraft's max cargo weight must be 50000 kgs 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 Rulesheets 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.

Creating rules by importing an Excel worksheet

Often decisions are based on statistical data in tables that are actuarial or real-world data that cannot be derived through a formula. Insurance underwriting, manufacturing tooling, life sciences, and census data have examples of data that is key to decision making yet revised only at regular time intervals.

When used in Corticon, a Rulesheet that provides such data is very useful but could require hours of data entry to create what might be thousands of rules. This feature makes that task fast, easy, and accurate.

Consider the following excerpt of the sample Excel worksheet with data as an example for life insurance underwriting. (In a Studio installation, the file is located at

[CORTICON_WORK_DIR]\Samples\ImportExcel\SampleDimensionalData.xlsx.)

Column A contains headers that identify the applicant's age, and row 1 contains headers that identify the beneficiary's age. The cells are the policy factor that adjusts the policy premium:

	A	B	C	D
1	Dimensions	50	51	52
2	40	0.9680	0.9698	0.9715
3	41	0.9652	0.9670	0.9689
4	42	0.9618	0.9640	0.9660

When this two-dimensional information is imported into a Corticon Rulesheet, each of the two dimensions and the data area are assigned to Vocabulary attributes within an entity. Then, each of the two dimensions defines a Condition and the data point specifies an Action, as shown:

	Conditions	0	1	2	3
a	Policy.applicantAge		50	51	52
b	Policy.beneficiaryAge		40	40	40
	Actions				
	Post Message(s)				
A	Policy.factor		0.968	0.9698	0.9715

When the data range expands, there could be thousands of rules in the Rulesheet that were all created by one import action, and later updated by performing the same import from the revised worksheet.

Preparing Excel sheets

In order to achieve successful import of a dimensional Excel worksheet into Corticon, the sheet must have a specific layout. The sample sheet **Two Dimensions** is modified and offset here to present a few concepts:

	A	B	C	D	E	F	G	H	I	J	K	L
1	C3:L12		Customer age is in the rows and beneficiary age is in the columns									
2		Dimensions	50	51	52	53	54	55	56	57	58	59
3	This is the	40	0.9680	0.9698	0.9715	0.9732	0.9747	0.9762	0.9776	0.9789	0.9802	0.9813
4	customer age	41	0.9652	0.9670	0.9689	0.9707	0.9724	0.9741	0.9756	0.9771	0.9785	0.9798
5	x bene age	42	0.9618	0.9640	0.9660	0.9680	0.9699	0.9717	0.9734	0.9750	0.9766	0.9780
6	factor table	43	0.9582	0.9605	0.9628	0.9650	0.9670	0.9690	0.9709	0.9727	0.9744	0.9760
7		44	0.9541	0.9567	0.9592	0.9616	0.9639	0.9660	0.9681	0.9700	0.9719	0.9737
8		45	0.9496	0.9524	0.9551	0.9577	0.9603	0.9627	0.9650	0.9671	0.9692	0.9711
9	2015	46	0.9447	0.9477	0.9506	0.9535	0.9562	0.9589	0.9614	0.9638	0.9660	0.9682
10		47	0.9392	0.9425	0.9456	0.9487	0.9517	0.9546	0.9574	0.9601	0.9626	0.9651
11		48	0.9332	0.9367	0.9402	0.9436	0.9467	0.9499	0.9529	0.9559	0.9587	0.9613
12		49	0.9266	0.9304	0.9341	0.9377	0.9413	0.9447	0.9480	0.9512	0.9542	0.9572
13												

- The data range, C3:L12, has no empty cells, and each cell has the same data type.
- Column B, to the left of the start point, is assumed to have the values for each row in the vertical dimension.
- Row 2, above the start point, is assumed to have the values for each column in the horizontal dimension.
- The dimension values have no empty cells, no duplicate values, and each cell has the same data type.
- The dimension and data cells, B2:L12, are the only cells read by the import feature. Data in any other cells have no impact on the import. Conversely, setting a range that includes blank cells, such as C3:M13, will not import successfully.

A one dimensional table follows similar rules. The only consideration is to specify only the appropriate dimension,, as in the sample sheet **One Dimension** that uses the vertical attribute:

	A	B
1	Dimension	
2	40	0.9934
3	41	0.9933
4	42	0.9932
5	43	0.9931
6	44	0.9929
7	45	0.9927
8	46	0.9925
9	47	0.9923
10	48	0.9920
11	49	0.9916

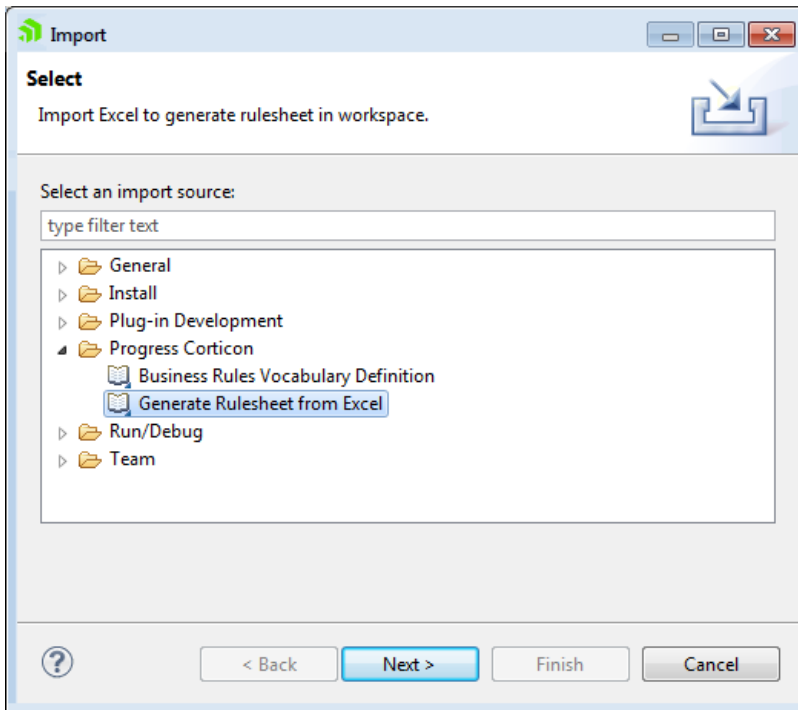
Note: Decimal precision - Some decimal values might seem imprecise between the Excel workbook value and their corresponding Rulesheet action value. A good practice is to set the Excel workbook precision before initiating an import into Corticon. With the Workbook open and the data formatted to the preferred decimal precision, click the **Microsoft Office** Button, click **Excel Options**, and then click **Advanced** category. In the **When calculating this workbook** section, select the workbook that you want, click to select the **Set precision as displayed** check box, and then click **OK**. You are alerted that you are losing precision while in fact you are ensuring that the value displayed is the value that is imported into Corticon. Save and close the Excel workbook, and then start the import process.

Importing the Excel sheet into a Rulesheet format

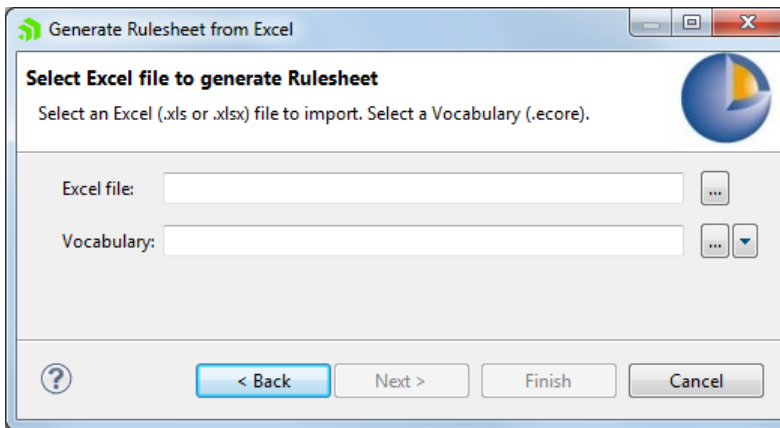
Once your Excel sheet has been prepared, you can import it into a Rulesheet.

To import an Excel sheet into Corticon Studio:

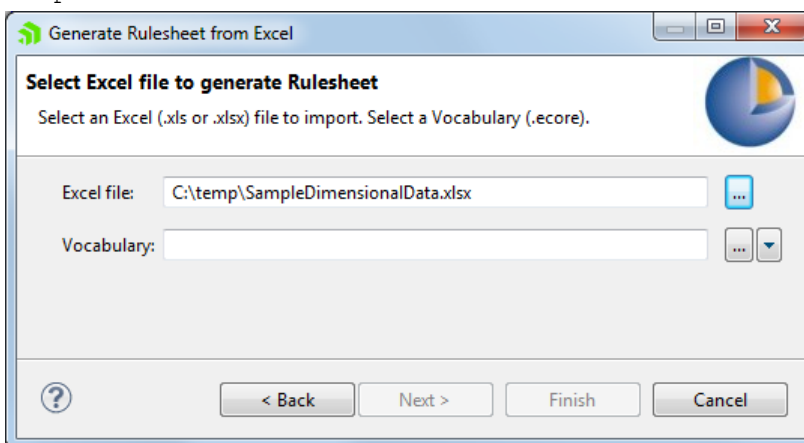
1. Select the menu command **File > Import**. You could instead navigate to the Vocabulary of the project where the Rulesheet will be created to right-click on it, and then select **Import**.



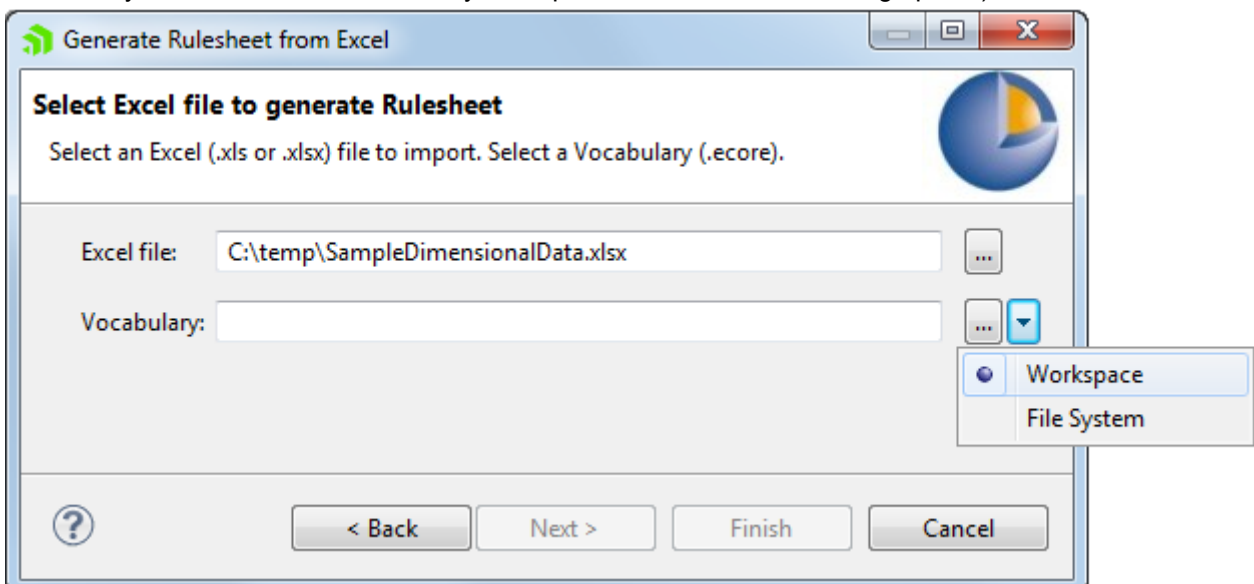
Click **Next**. The **Generate Rulesheet from Excel** dialog box opens:



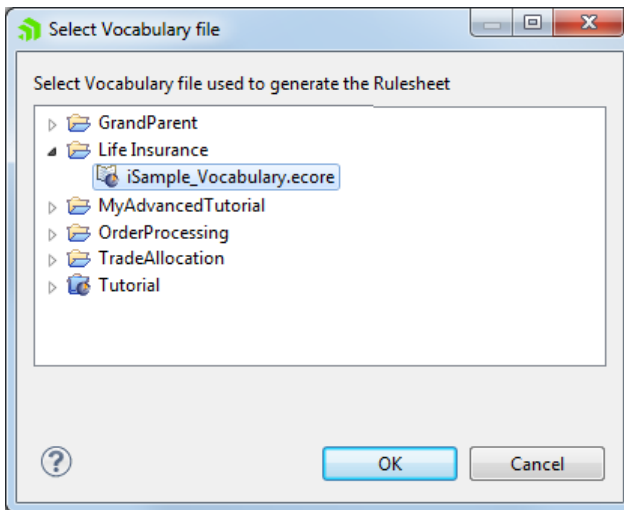
2. Click the **Excel file** search button, and then locate and open the Excel file. For this example, select `SampleDimensionalData.xlsx`.



3. Click the **Vocabulary** search button, and then locate and open the Vocabulary file. As illustrated, you can choose to find it in the workspace or in the file system. (If you launched the **Import** function with the Vocabulary file selected, the Vocabulary file is pre-selected when the dialog opens.)



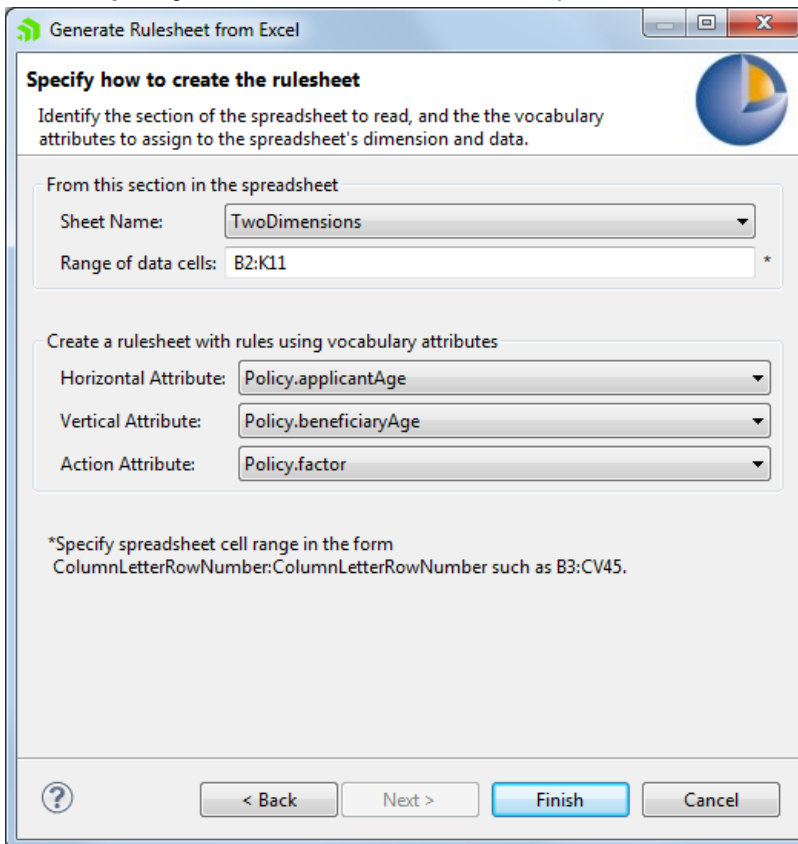
4. To follow the example, choose the Life Insurance sample's Vocabulary:



Click **OK** to close the selection dialog, and then click **Finish** to enter the import parameters.

Note: If you are following these steps hands on in Studio, you need to add three attributes to the Life Insurance sample's Vocabulary **Policy** entity, one named `applicantAge` with the datatype `human_age`, one named `beneficiaryAge` with the datatype `human_age`, and another named `factor` with the datatype `decimal`.

5. On the **Specify how to create the Rulesheet** panel, choose or enter the parameters as shown:



Click **Finish**. The import process creates a Rulesheet in the project with the name of the Excel sheet.

6. Review the results in the Studio:

iSample_Vocabulary.ecore		TwoDimensions.ers											
	Conditions	0	1	2	3	4	5	6	7	8	9	10	11
a	Policy.applicantAge		50	51	52	53	54	55	56	57	58	59	50
b	Policy.beneficiaryAge		40	40	40	40	40	40	40	40	40	40	41
Actions													
	Post Message(s)												
A	Policy.factor		0.968	0.9698	0.9715	0.9732	0.9747	0.9762	0.9776	0.9789	0.9802	0.9813	0.9652
B													

The Rulesheet has 100 columns, one for each rule from the 10x10 matrix that was imported.

- To follow up with an import of the one dimensional sheet, repeat steps 1 through 4, and then choose or enter the parameters as shown:

Generate Rulesheet from Excel

Specify how to create the rulesheet

Identify the section of the spreadsheet to read, and the the vocabulary attributes to assign to the spreadsheet's dimension and data.

From this section in the spreadsheet

Sheet Name: OneDimension

Range of data cells: B2:B11

Create a rulesheet with rules using vocabulary attributes

Horizontal Attribute:

Vertical Attribute: Policy.applicantAge

Action Attribute: Policy.factor

*Specify spreadsheet cell range in the form ColumnLetterRowNumber:ColumnLetterRowNumber such as B3:CV45.

Back

Next >

Finish

Cancel

Notice that the Horizontal attribute was left blank and the range specifies only one column.

- Review the results in the Studio:

iSample_Vocabulary.ecore		TwoDimensions.ers		OneDimension.ers									
	Conditions	0	1	2	3	4	5	6	7	8	9	10	11
a	Policy.applicantAge		40	41	42	43	44	45	46	47	48	49	-
b													
Actions													
	Post Message(s)												
A	Policy.factor		0.9934	0.9933	0.9932	0.9931	0.9929	0.9927	0.9925	0.9923	0.992	0.9916	
B													

The Rulesheet has 10 columns, as shown, one for each rule from the 10x1 matrix that was imported.

Updating an imported sheet

When you run these import tasks again, the existing Rulesheet of the same name in the project is replaced.

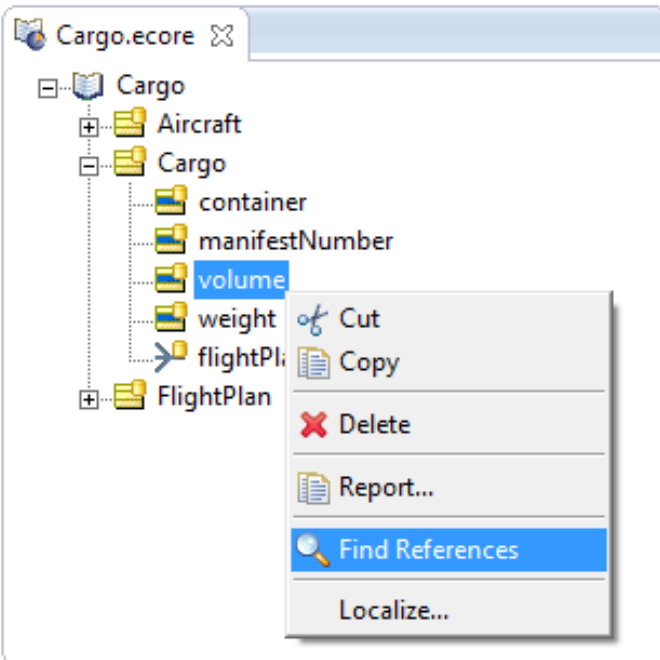
Using standard boolean constructions

A decision table is a graphical method of organizing and formalizing logic. 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*.

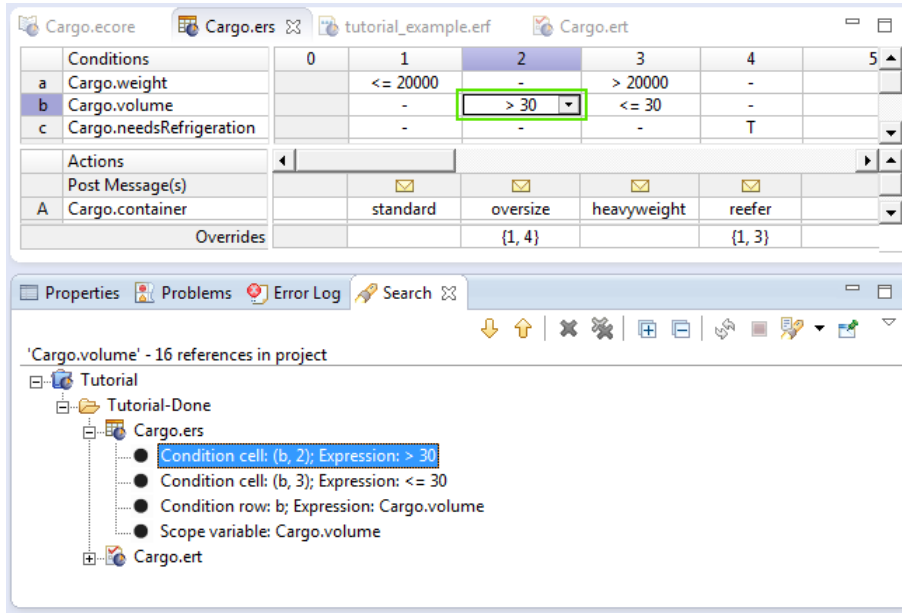
See the section [Standard Boolean constructions](#) on page 341 where several standard truth tables are presented with equivalent Rulesheets.

Finding references to an entity, attribute, or association in a project

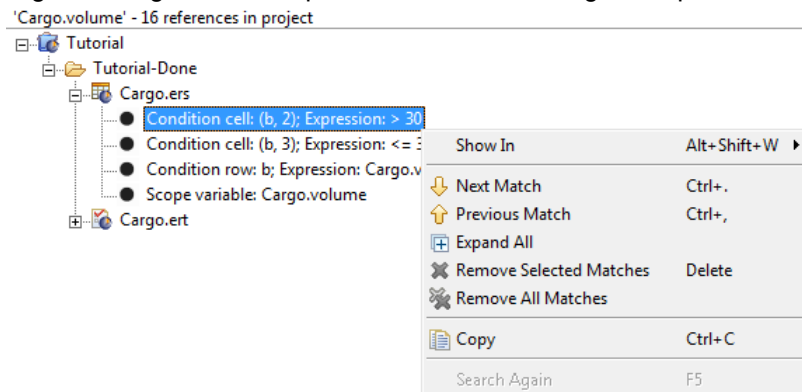
When working on large projects with many assets, it is a real convenience to find everywhere an entity, attribute, or association is referenced in the project. This feature reveals the vocabulary item's usage and impact, and lets you easily navigate to each instance. Just right-click on an entity, attribute, or association name in the Vocabulary editor, and then choose **Find References** to initiate a search, as shown:



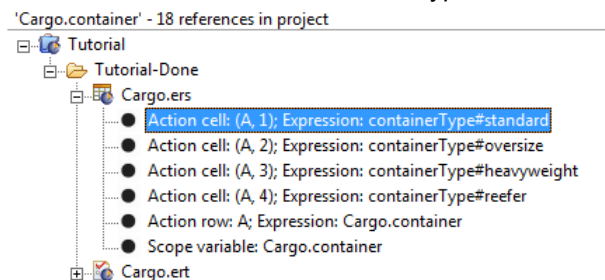
The search examines all Rulesheets, Ruleflows, and Ruletests in the project, then lists each match within each asset. Double-clicking on a match opens the asset and puts the focus at that location, as shown:



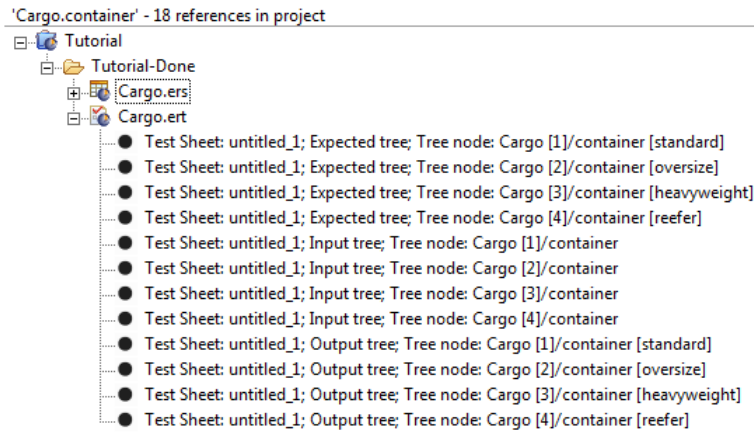
Right-clicking on a match presents a menu of navigation options as well as their keyboard shortcuts, as shown:



When the attribute is a custom data type, the search results indicate the Data Type Name and value, as shown:



When results are in Ruletests, the results detail the testsheet, tree, and node, as shown:



Embedding attributes in posted rule statements

It is frequently useful to “embed” attribute values within a Rule Statement, so that posted messages contain actual data. Special syntax must be used to differentiate the static text of the rule statement from the dynamic value of the attribute. As shown in [Sample Rulesheet with Rule Statements Containing Embedded Attributes](#), an embedded attribute must be enclosed by curly brackets { . . } to distinguish it from the static Rule Statement text.

It may also be helpful to indicate which parts of the posted message are dynamic, so a user seeing a message knows which part is based on “live” data and which part is the standard rule statement. As shown in [Sample Rulesheet with Rule Statements Containing Embedded Attributes](#), square brackets are used immediately outside the curly brackets so that the dynamic values inserted into the message at rule execution will be “bracketed”. The use of these square brackets is optional – other characters may be used to achieve the intended visual distinction.

Remember, Action Rows execute in numbered order (from top to bottom in the Actions pane), so a Rule Statement that contains an embedded attribute value must not be posted before the attribute has a value. Doing so will result in a `null` value inserted in the posted message.

Figure 95: Sample Rulesheet with Rule Statements Containing Embedded Attributes

Conditions		0	1	2	3
a	Entity1.integer1		< 18	18..25	> 25
b					
c					
Actions					
Post Message(s)			✉	✉	✉
A					
Overrides					

Ref	ID	Post	Alias	Text
1		Info	Entity1	This person is [{Entity1.integer1}] which is less than 18, so this person can't drink or vote
2		Info	Entity1	This person is [{Entity1.integer1}] which is between 18 and 25, so this person can drink, vote and be drafted, but not rent a car
3		Info	Entity1	This person is [{Entity1.integer1}] which is greater than 25, so this person can drink, vote, be drafted and rent a car

Figure 96: Rule Messages Window Showing Bracketed Embedded Attributes (Orange Box)

Input		Output	
Entity1 [1]	integer1 [15]	Entity1 [1]	integer1 [15]
Entity1 [2]	integer1 [23]	Entity1 [2]	integer1 [23]
Entity1 [3]	integer1 [33]	Entity1 [3]	integer1 [33]

Save...	Message	Entity
Info	This person is [15] which is less than 18, so this person can't drink or vote	Entity1[1]
Info	This person is [23] which is between 18 and 25, so this person can drink, vote and be drafted, but not rent a car	Entity1[2]
Info	This person is [33] which is greater than 25, so this person can drink, vote, be drafted and rent a car	Entity1[3]

When an attribute uses an Enumerated Custom Data Type, the dynamic value embedded in the posted Rule Message will be the Value, not the Label. See the *Rule Modeling Guide*, “Building the Vocabulary” chapter for more information about Custom Data Types.

No expressions in Rule Statements

A reminder about the table in [Summary Table of Vocabulary Usage Restriction](#), which specifies that the only parts of the Vocabulary that may be embedded in Rule Statements are attributes and functions (`today` and `now`). No operators or expressions are permitted inside Rule Statements. Often, operators will cause error messages when you try to save a Rule Set. Sometimes the Rule Statement itself will turn red. Sometimes an embedded equation will even execute as you intended. But sometimes no obvious error will occur, but the rule does not executed as intended. Just remember that operators and expressions are not supported in Rule Statements.

Including apostrophes in strings

String values in Corticon Studio are always enclosed in single quotes. But occasionally, you may want the String value to include single quote or apostrophe characters. If you enter the following text in Corticon Studio:

```
entity1.string1='Jane's dog Spot'
```

The text will turn red, because Corticon Studio thinks that the `string1` value is `'Jane'` and the remaining text `s dog Spot'` is invalid. To properly express a String value that includes single quotes or apostrophes, you must use the special character backslash (`\`) that tells Corticon Studio to ignore the apostrophe(s) as follows:

```
entity1.string1='Jane\'s dog Spot'
```

When preceded by the backslash, the second apostrophe will be ignored and assumed to be just another character within the String. This notation works in all sections of the Rulesheet, including Values sets. It also works in the Possible Values section of the Vocabulary Editor.

Test yourself questions: Rule writing techniques and logical equivalents

Note: Try this test, and then go to [Test yourself answers: Rule writing techniques and logical equivalents](#) on page 351 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)

<code>Entity.boolean1</code>	<code>Entity.boolean1<>F</code>	<code>Entity.boolean1=F</code>	<code>not</code> <code>(Entity.boolean1=F)</code>
------------------------------	---------------------------------------	--------------------------------	--

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.

Collections

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

For details, see the following topics:

- [Understanding how Corticon Studio handles collections](#)
- [Visualizing collections](#)
- [A basic collection operator](#)
- [Filtering collections](#)
- [Using aliases to represent collections](#)
- [Sorted aliases](#)
- [Advanced collection sorting syntax](#)
- [Statement blocks](#)
- [Using sorts to find the first or last in grandchild collections](#)
- [Singletons](#)
- [Special collection operators](#)
- [Aggregations that optimize database access](#)
- [Test yourself questions: Collections](#)

Understanding how Corticon Studio handles collections

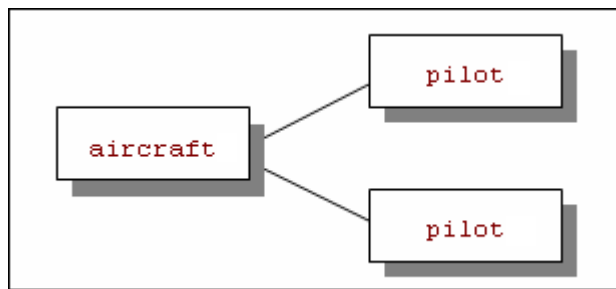
Support for using collections is extensive in Corticon Studio – in fact, the integration of collection support in the Rules Language is so seamless and complete, the rule modeler will often discover that rules are performing multiple evaluations on collections of data beyond what he/she anticipated! 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:

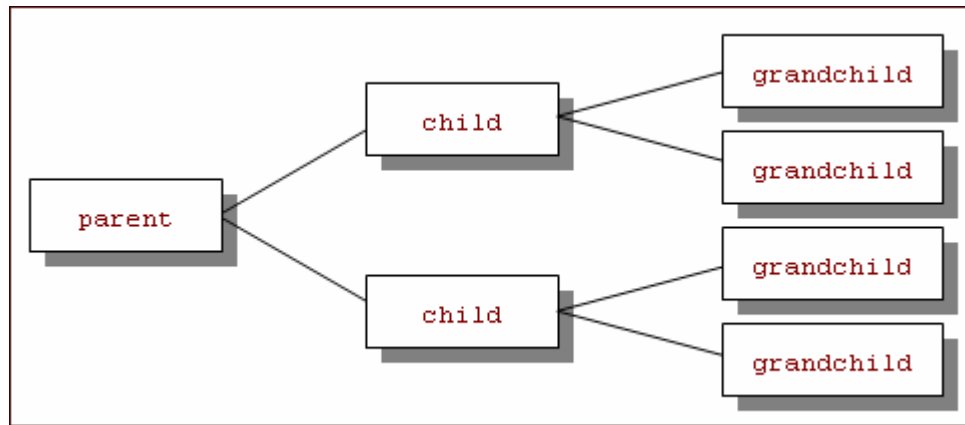
Figure 97: Visualizing a Collection of Pilots



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 the Corticon Rule Language. 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 include more than one type or “level” of association:

Figure 98: 3-Level Collection



This collection is expressed as `parent.child.grandchild` in the Corticon Rule Language. 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](#) topic for more details.

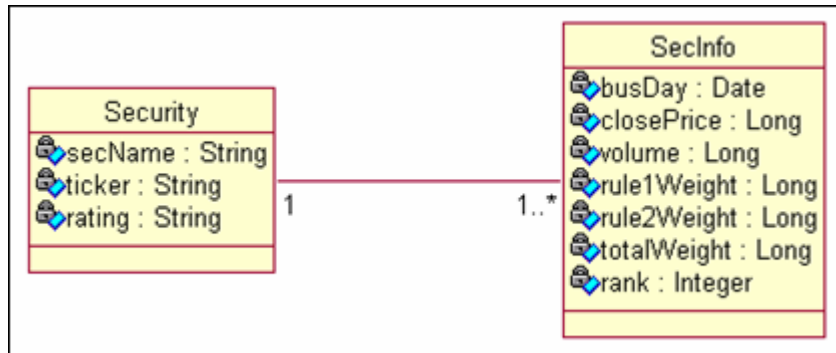
Using aliases to represent collections

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 99: 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 day must have a value of 0.5 assigned to its rule 1 weight; otherwise, a value of 0 must be assigned to its rule 1 weight.
2. A security whose trading volume today is greater than its trading volume on the previous business day must have a value of 0.25 assigned to its rule 2 weight; otherwise, a value of 0 must be assigned 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. Rulesheets that accomplish these tasks are shown in the next two figures.

Figure 100: Rulesheet with Ranking Model Rules 1 and 2

	0	1	2	3	4
a		T	F	-	-
b		-	-	T	F
c					
A		0.5	0		
B				0.25	0
Overrides					

Ref	ID	Post	Alias	Text
1		Info	sec	If today's closing price > last business day's closing price, then rule 1 weight = 0.5
2		Info	sec	If today's closing price <= last business day's closing price, then rule 1 weight = 0
3		Info	sec	If today's closing volume > last business day's closing volume then rule 2 weight = 0.25
4		Info	sec	If today's closing volume <= last business day's closing volume then rule 2 weight = 0

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

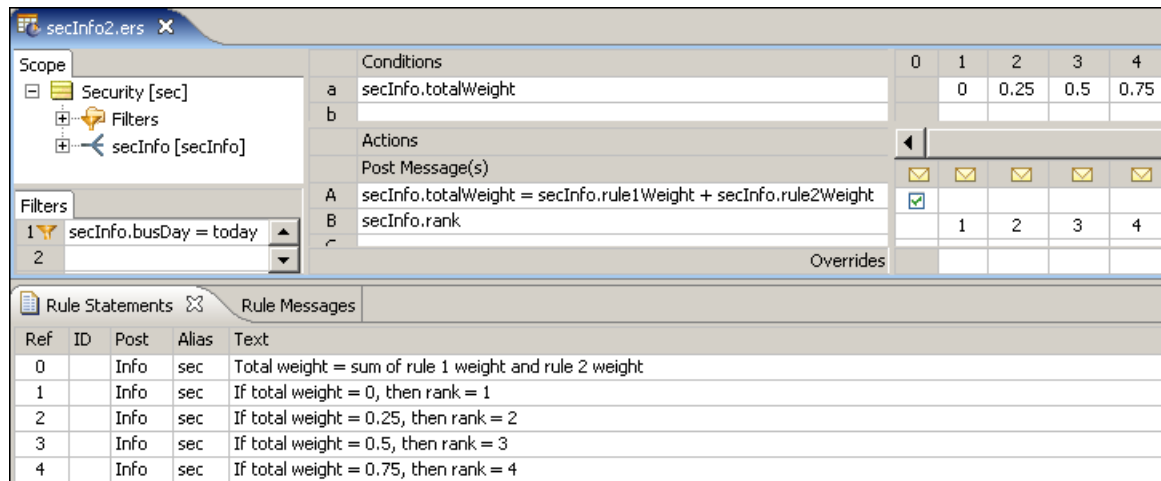
`Security` is the scope for our Rulesheet, which is not a new concept. But then we see that there are two aliases for the `SecInfo` entities associated with `Security`: `secinfo1` and `secinfo2`. Each of these aliases represents a separate but identical collection of the `SecInfo` entities associated 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 102: Rulesheet with Total Weight Calculation and Rank Determination

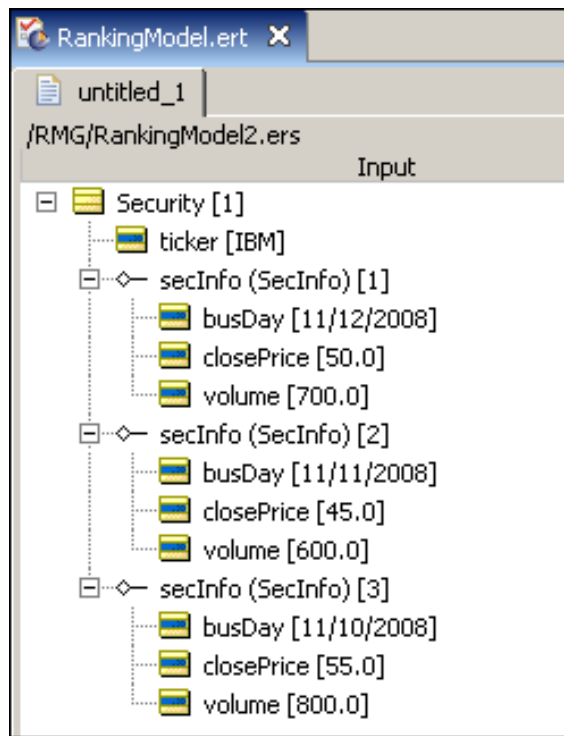


We can test our new rules using a Ruleflow to combine the two Rulesheets. In a Ruletest which executes the Ruleflow, we would expect to see the following results:

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.

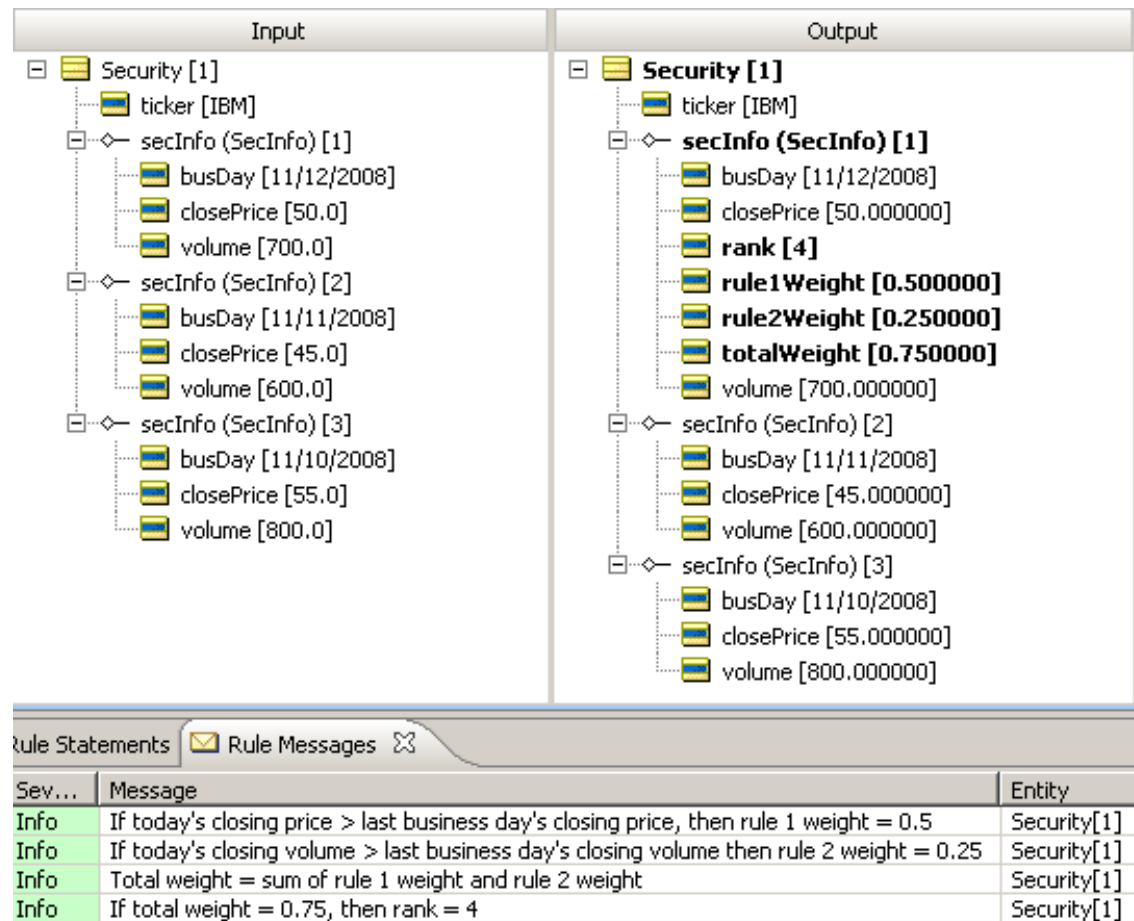
A Ruleflow constructed for testing the ranking model rules is as shown:

Figure 103: 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 11/12/2008, 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 104: 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 100: Rulesheet with Ranking Model Rules 1 and 2](#) on page 121. 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 105: Rulesheet with Alternate Rank Determination Rules

The screenshot shows the Progress Corticon Rulesheet Editor for the file `*RankingModel.ert` with the rulesheet `AlternateRank.ers` open. The interface is divided into several sections:

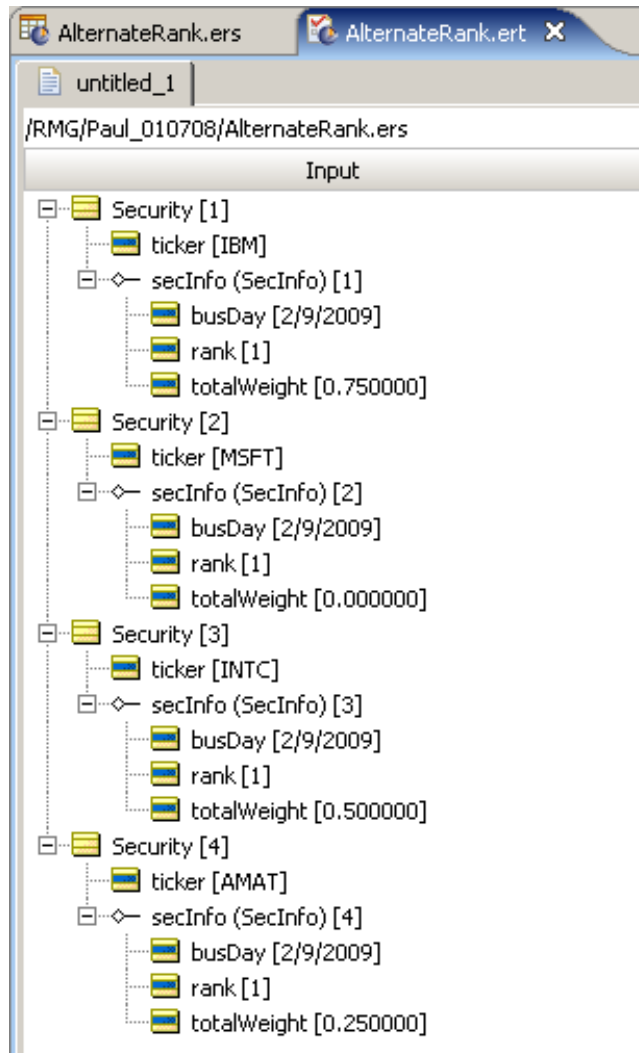
- Scope:** A tree view on the left showing the hierarchy of objects: `Security [sec1]` (containing `Filters` and `ticker`), `secInfo [secinfo1]`, `Security [sec2]` (containing `Filters` and `ticker`), and `secInfo [secinfo2]`.
- Conditions:** A table with columns for conditions (a-q) and three columns for evaluation results (0, 1, 2). Condition 'a' is `secinfo1.totalWeight > secinfo2.totalWeight`, with results `-`, `T`, and `F` respectively.
- Actions:** A table with columns for actions (A-E) and three columns for evaluation results (0, 1, 2). Action 'A' is `secinfo1.rank += 1`, with results `-`, `✓`, and `✗` respectively.
- Filters:** A list of filters on the left:
 - `sec1 <> sec2`
 - `secinfo1.busDay = today`
 - `secinfo2.busDay = today`
 -
 -
- Rule Statements:** A table at the bottom with columns for Ref, ID, Post, Alias, and Text.

Ref	ID	Post	Alias	Text
1		Info		If Security 1 [{sec1.ticker}] total weight > Security 2 [{sec2.ticker}] total weight, then increment [{sec1.ticker}] rank by 1
2		Info		If Security 1 [{sec1.ticker}] total weight <= Security 2 [{sec2.ticker}] total weight, then take no action

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 Corticon 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 106: 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 107: Rulesheet for Alternate Security Ranking Model Rules

The screenshot displays the 'AlternateRank.ers' rulesheet interface. The top section shows the 'Input' data for four securities, organized into a tree structure. The bottom section shows the 'Rule Statements' and 'Rule Messages'.

Input Data:

Security	ticker	secInfo (SecInfo)	busDay	rank	totalWeight
Security [1]	IBM	[1]	2/9/2009	1	0.750000
Security [2]	MSFT	[2]	2/9/2009	1	0.000000
Security [3]	INTC	[3]	2/9/2009	1	0.500000
Security [4]	AMAT	[4]	2/9/2009	1	0.250000

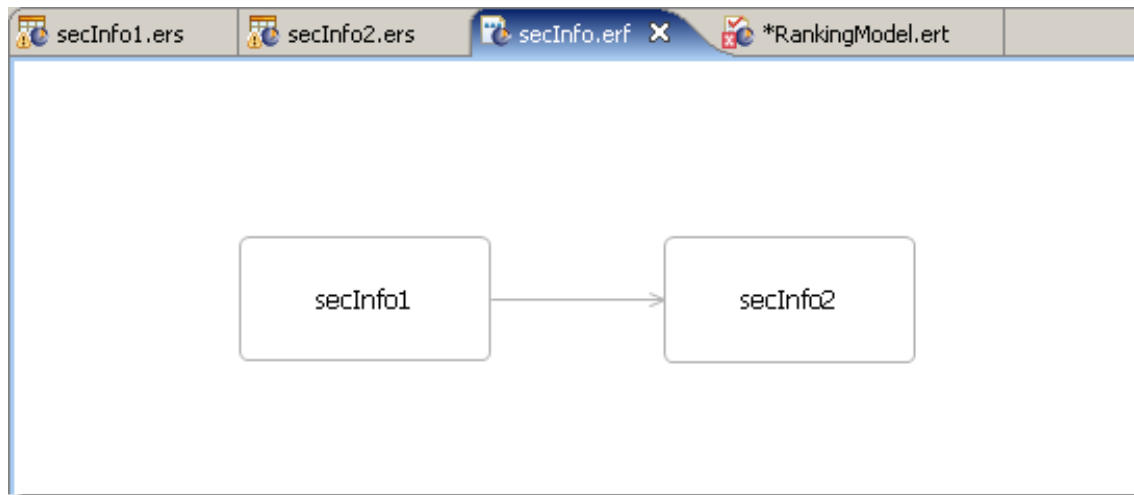
Rule Statements:

Severity	Message	Entity
Info	If Security 1 [IBM] total weight > Security 2 [AMAT] total weight, then increment [IBM] rank by 1	Security[1]
Info	If Security 1 [AMAT] total weight > Security 2 [MSFT] total weight, then increment [AMAT] rank by 1	Security[4]
Info	If Security 1 [INTC] total weight > Security 2 [MSFT] total weight, then increment [INTC] rank by 1	Security[3]
Info	If Security 1 [IBM] total weight > Security 2 [INTC] total weight, then increment [IBM] rank by 1	Security[1]
Info	If Security 1 [INTC] total weight > Security 2 [AMAT] total weight, then increment [INTC] rank by 1	Security[3]
Info	If Security 1 [IBM] total weight > Security 2 [MSFT] total weight, then increment [IBM] rank by 1	Security[1]
Info	If Security 1 [AMAT] total weight <= Security 2 [INTC] total weight, then take no action	Security[4]
Info	If Security 1 [MSFT] total weight <= Security 2 [AMAT] total weight, then take no action	Security[2]
Info	If Security 1 [AMAT] total weight <= Security 2 [IBM] total weight, then take no action	Security[4]
Info	If Security 1 [INTC] total weight <= Security 2 [IBM] total weight, then take no action	Security[3]
Info	If Security 1 [MSFT] total weight <= Security 2 [IBM] total weight, then take no action	Security[2]
Info	If Security 1 [MSFT] total weight <= Security 2 [INTC] total weight, then take no action	Security[2]

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.

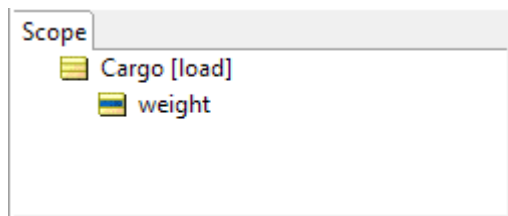
Figure 108: Ruleflow to test two Rulesheets in succession



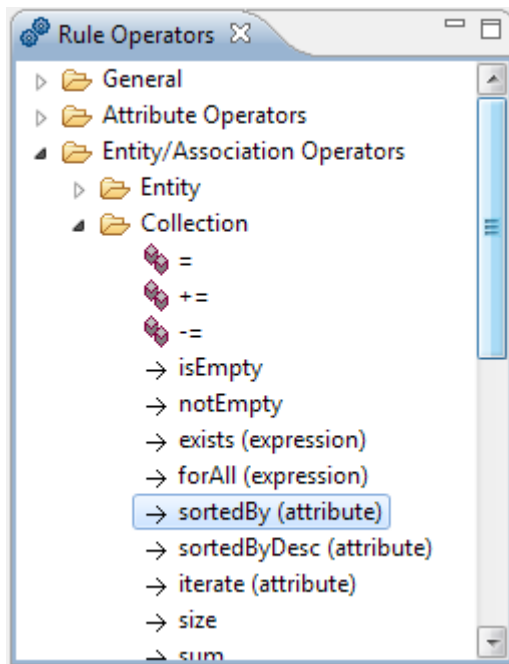
Sorted aliases

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.

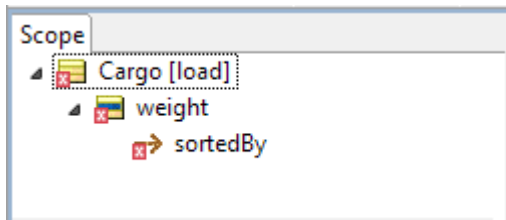
The following example shows a Rulesheet based on the `Cargo` Vocabulary. We brought the `Cargo` entity and its `weight` attribute into the scope:



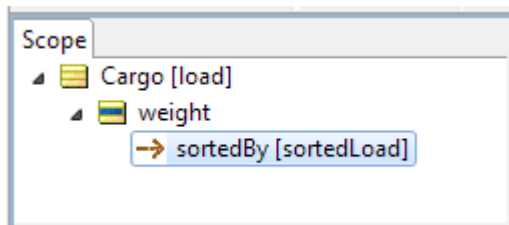
The operators `sortedBy` and `sortedByDesc` enable sorting ascending or descending order of the numeric or alphabetic values of the attribute in the set of data. Note that an attribute with a Boolean data type is not valid for this operation.



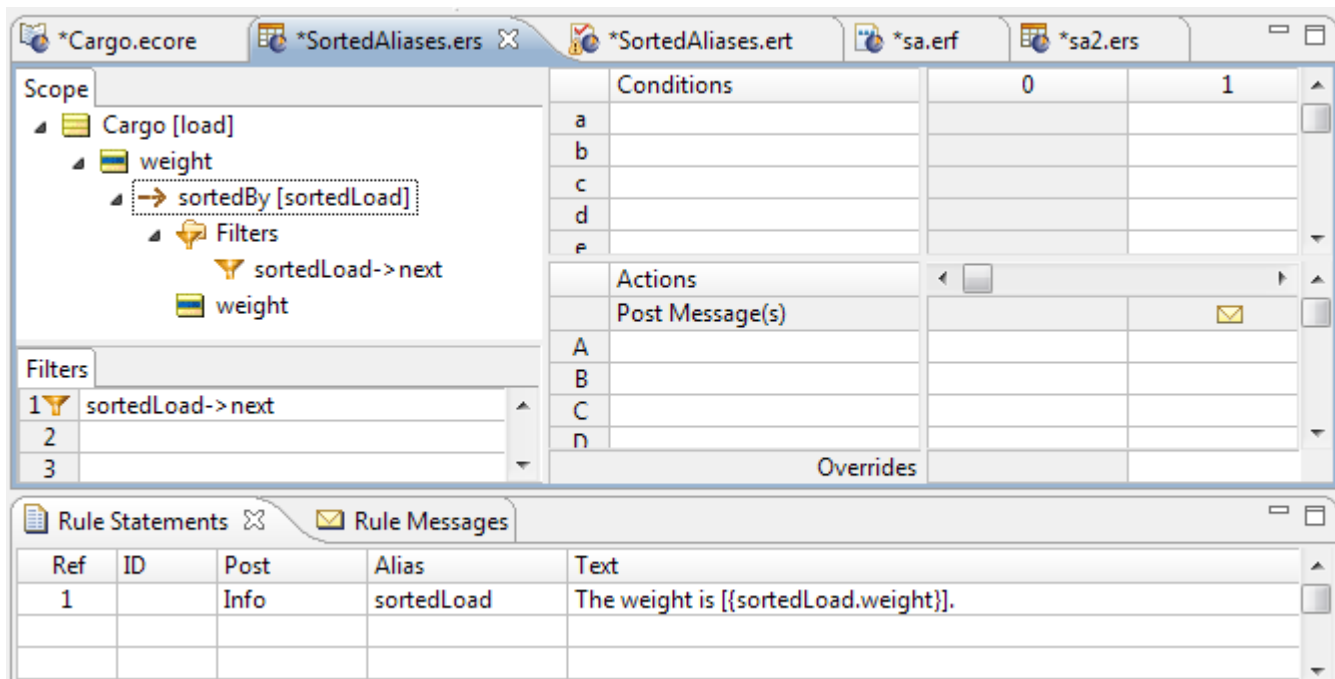
Dragging the `sortedBy` operator and dropping it (you cannot type it in) on the attribute `weight` places it in the scope yet shows an error:



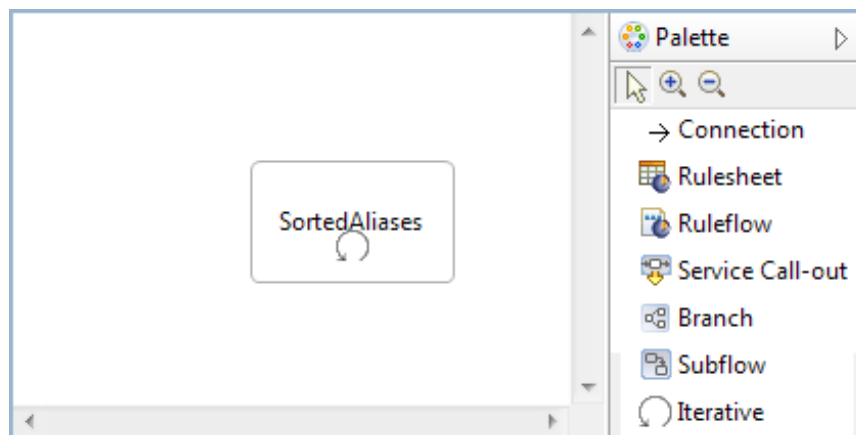
The error message notes that a sorted alias node requires an alias name. When we enter an alias name, the scope is complete.



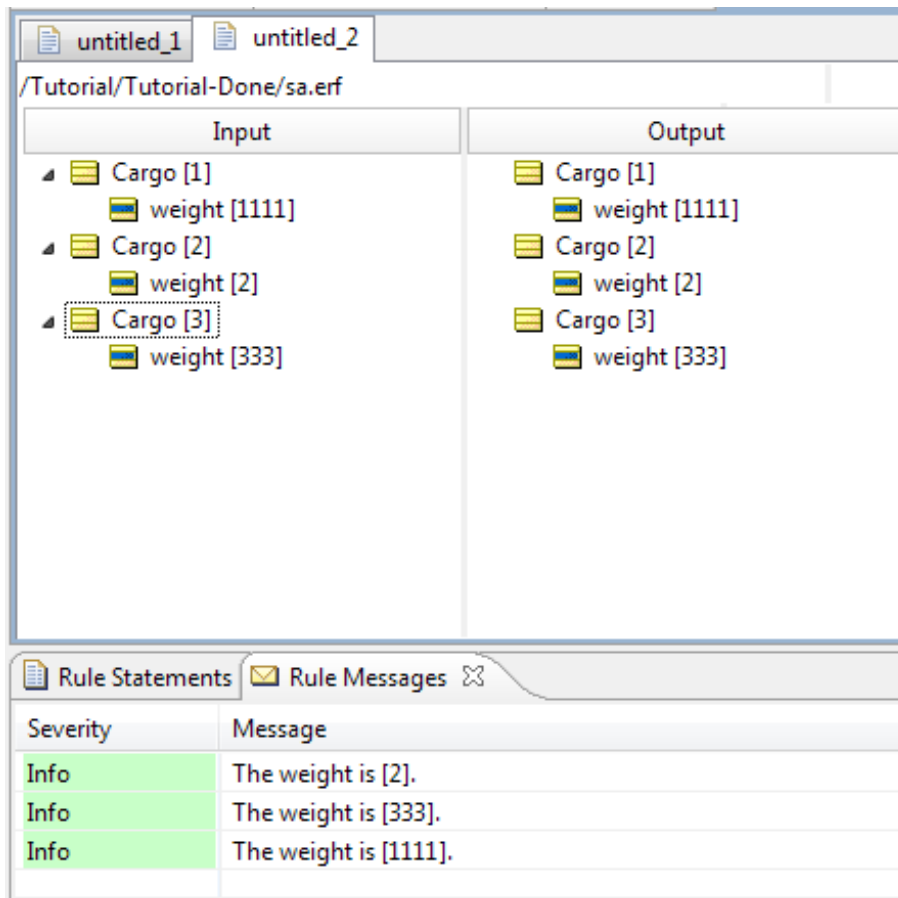
We add a filter expression to establish that, when we iterate through the list, each pass will present the next sequential item in the sorted set. We defined this by dragged `sortedBy` from the scope to filter line 1, and then appended the `->next` operator. We added a rule statement based on sorted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

Advanced collection sorting syntax

Collection syntax contains some subtleties worth learning. It is helpful when writing collection expressions to step through them, left to right, as though you were reading a sentence. This helps you better understand how the pieces combine to create the full expression. It also helps you to know what else you can safely add to the expression to increase its utility. Let's try this approach in order to dissect the following expression:

```
Collection1 -> sortBy(attribute1) -> last.attribute2
```

1. `Collection1`

This expression returns the collection $\{e_1, e_2, e_3, e_4, e_5, \dots, e_n\}$ where e_x is an element (an entity) in `Collection1`. We already know that alias `Collection1` represents the entire collection.

2. `Collection1 -> sortBy(attribute1)`

This expression returns the collection $\{e_1, e_2, e_3, e_4, e_5, \dots, e_n\}$ arranged in ascending order based on the values of `attribute1` (which we call the "index").

3. `Collection1 -> sortBy(attribute1) -> last`

returns $\{e_n\}$ where e_n is the last element in `Collection1` when sorted by `attribute1`

This expression returns a **specific entity** (element) from `Collection1`. It does not return a specific value, but once we have identified a specific entity, we can easily reference the value of any attribute it contains, as in the following:

```
4. Collection1 -> sortBy(attribute1) -> last.attribute2
```

which returns `{en.attribute2}`

Entity Context

The complete expression not only returns a specific value, but just as importantly, it also returns the entity to which the value belongs. This “entity context” is important because it allows us to do things to the entity itself, like assign a value to one of its attributes. For example:

```
Collection1 -> sortBy(attribute1) -> last.attribute2='xyz'
```

The above expression will assign the value of `xyz` to `attribute2` of the entity whose `attribute1` is highest in `Collection1`. Contrast this with the following:

```
Collection1.attribute1 -> sortBy(attribute1) -> last
```

which returns a single integer value, like 14.

Notice that all we have now is a number, a *value*. We have lost the entity context, so we can't do anything to the entity that owns the attribute with value of 14. In many cases, this is just fine. Take for example:

```
Collection1.attribute1 -> sortBy(attribute1) -> last > 10
```

In this expression, it is not important that we know which element has the highest value of `attribute1`, all we want to know is if the highest value (whomever it “belongs” to) is greater than 10.

Understanding the subtleties of collection syntax and the concept of entity context is important because it helps us to use the returned entities or values correctly. For example:

Return the lower of the following two values:

- 12
- The age of the oldest child in the family

What is really being compared here? Do we care *which* child is oldest? Do we need to know his or her name? No. We simply need to compare the age of that child (whichever one is oldest) with the value of 12. So this is the expression that models this logic:

```
family.age -> sortByDesc(age) -> first.min(12)
```

The `.min` operator, as we know, is an operator that *acts upon* numeric data types (Integer or Decimal). And since we also know that `family.age -> sortByDesc(age) -> first` returns a number, then it is legal and valid to use `.min` at the end of this expression.

What about this scenario: Name the youngest child Junior.

```
family -> sortByDesc(age) -> last.name='Junior'
```

Now we want to return a **specific entity** – that of the youngest child – and assign to its name a value of `Junior`. We need to keep the entity context in order to make this assignment, and the expression above accomplishes this.

Statement blocks

Sequence operators can easily extract an attribute value from the first, last or other specific element in a sorted collection (see `->first`, `->last`, or `->at(n)` for examples). This is especially useful when the attribute's value is involved in a comparison in a Conditional or Preconditional rule. Sometimes, however, you want to identify a particular element in a sequence and “flag” or “tag” it for use in subsequent rules. This can be accomplished using special syntax called Statement Blocks.

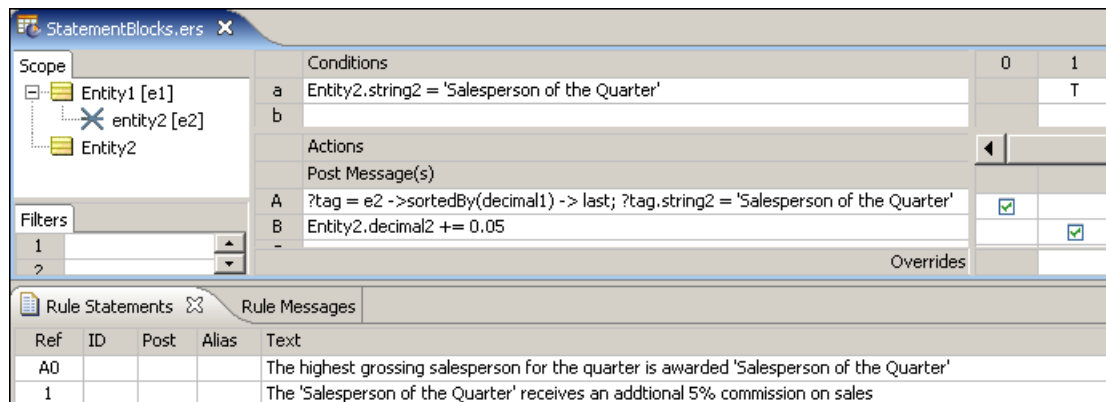
Statement Blocks, permitted only in the Action rows of the Rulesheet, use special variables, prefixed by a `?` character, to “hold” or “pin” an element so that further action may be taken on it, including tagging it by assigning a value to one of its attributes. These special holder variables may be declared “on the fly”, meaning they do not need to be defined anywhere prior to use.

Here's an example. In a sales management system, the performance of sales reps is analyzed every quarter, and the highest grossing sales rep is awarded “Salesperson of the Quarter”. This special status is then used to automatically increase the rep's commission percentage on sales made in the following quarter. We will use the same generic Vocabulary as in all previous examples, but make these assumptions:

Vocabulary Term	Meaning
Entity2	a salesperson
Entity1.entity2	collection of salespeople
Entity2.string1	a salesperson's name
Entity2.decimal1	a salesperson's quarterly sales
Entity2.string2	a salesperson's award
Entity2.decimal2	a salesperson's commission percentage

Using this Vocabulary, we construct the Rulesheet shown in

Figure 109: Rulesheet using Statement Block to Identify and Reward Winner



Important Notes about Statement Blocks

As expressed in Action row A in the figure above, a statement block consists of two separate expressions:

1. The first part assigns an element of a sequence to a special holder variable, prefixed by the ? character. This variable is unusual because it represents an *element*, not a *value*. Here, the highest grossing salesperson is expressed as the last element of the collection of salespeople (e2), sorted in ascending order according to quarterly sales (decimal1). Once identified by the sequencing operator ->last, this salesperson is momentarily “held” by the ?tag variable, which we declared “on-the-fly”.
2. The second part of the statement – the part following the semicolon – assigns a value to an attribute of the element held by the ?tag. In our example, we are assigning a value of 'Salesperson of the Quarter' to the string2 attribute of the salesperson held by ?tag. In effect, we have “tagged” the highest grossing salesperson with this award.

These two parts must be included on the same Action Row, separated by a semicolon. If the two parts are separated in different sections or in different Rows of the same section, the element represented by the ? variable is “lost”, in other words, ?tag loses its “grip” on the element identified by the sequencing operator.

Note: Using semicolons - The semicolon is an action statement end character that creates a compound action statement -- each action statement is executed sequentially. Its use, however, can make it harder to read action statements in Rulesheets and reports. It is recommended that you use semicolons only when there is no alternative, as in this example.

Now that we have tagged our winner, we can use the tagged element (awardee) to take additional actions. In the Conditional rule, we increase the commission percentage of the winner by 5% using the increment operator.

The next figure shows a Ruletest Input and Output pane. As expected, our highest grossing salesperson has been awarded “Salesperson of the Year” honors, and has had her commission raised by an additional 5%.

Figure 110: Output Panel with Winner and Adjusted Commission in Bold Black Text

Input	Output
<div>Entity1 [1]</div> <div> <div>entity2 (Entity2) [1]</div> <div> <div>decimal1 [100000]</div> <div>decimal2 [0.10]</div> <div>string1 [Joe Smith]</div> </div> </div> <div> <div>entity2 (Entity2) [2]</div> <div> <div>decimal1 [120000]</div> <div>decimal2 [0.10]</div> <div>string1 [Mary Jones]</div> </div> </div> <div> <div>entity2 (Entity2) [3]</div> <div> <div>decimal1 [85000]</div> <div>decimal2 [0.10]</div> <div>string1 [Ellen Barker]</div> </div> </div> <div> <div>entity2 (Entity2) [4]</div> <div> <div>decimal1 [115000]</div> <div>decimal2 [0.1]</div> <div>string1 [Josie Wales]</div> </div> </div> <div> <div>entity2 (Entity2) [5]</div> <div> <div>decimal1 [98000]</div> <div>decimal2 [0.10]</div> <div>string1 [Peter Calhoun]</div> </div> </div>	<div>Entity1 [1]</div> <div> <div>entity2 (Entity2) [1]</div> <div> <div>decimal1 [100000.000000]</div> <div>decimal2 [0.100000]</div> <div>string1 [Joe Smith]</div> </div> </div> <div> <div>entity2 (Entity2) [2]</div> <div> <div>decimal1 [120000.000000]</div> <div>decimal2 [0.150000]</div> <div>string2 [Salesperson of the Quarter]</div> </div> </div> <div> <div>entity2 (Entity2) [3]</div> <div> <div>decimal1 [85000.000000]</div> <div>decimal2 [0.100000]</div> <div>string1 [Ellen Barker]</div> </div> </div> <div> <div>entity2 (Entity2) [4]</div> <div> <div>decimal1 [115000.000000]</div> <div>decimal2 [0.100000]</div> <div>string1 [Josie Wales]</div> </div> </div> <div> <div>entity2 (Entity2) [5]</div> <div> <div>decimal1 [98000.000000]</div> <div>decimal2 [0.100000]</div> <div>string1 [Peter Calhoun]</div> </div> </div>

Using sorts to find the first or last in grandchild collections

The `SortedBy->first` and `SortedBy->last` constructs work as expected for any first-level collection regardless of datatype, determining the value of the first or last element in a sequence that was derived from a collection.

When associations are involved, you have to take care that the collection operator is not working at a grandchild level. You could construct a single collection of multiple children (rather than multiple collections of a single child) by “bubbling up” the relevant value into the child level, and then sort at that level. Another technique is to change the scope to treat the root level entity as the collection, and then apply filters so that only the ones matching the common attribute values across the associations are considered, and when you apply `SortedBy->first` or `SortedBy->last`, the intended value is the result.

Singletons

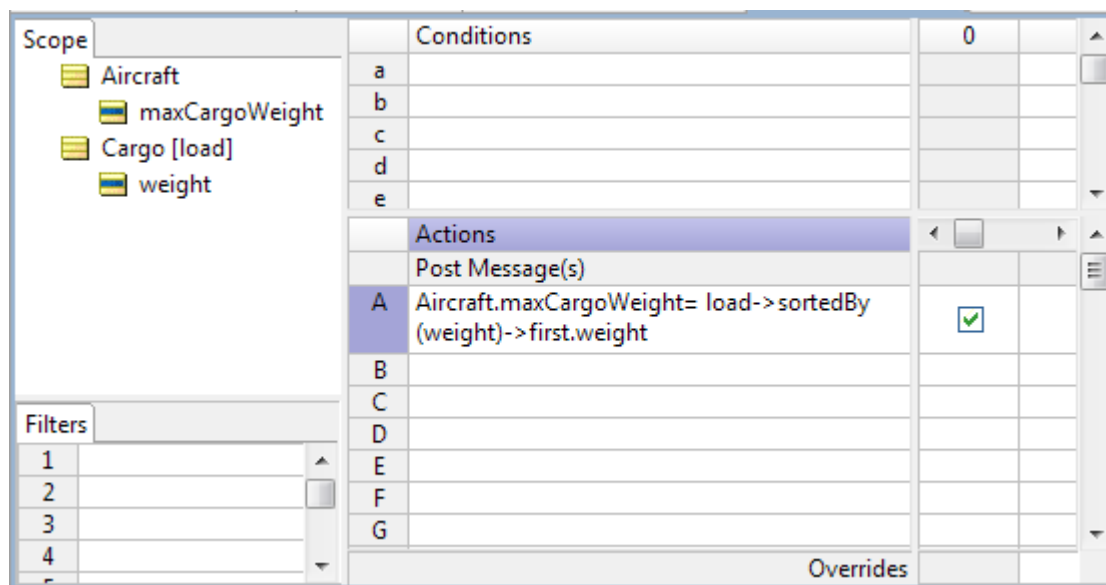
Singletons are collection operations that iterate over a set to extract one arithmetic value - the first, the last, the trend, the average, or the element at a specified position. We saw this behavior when the `sortedAlias` found the first and last element in an iterative list (as well as the elements 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 111:

Input	Output
<ul style="list-style-type: none"> Cargo [1] <ul style="list-style-type: none"> weight [1111] Cargo [2] <ul style="list-style-type: none"> weight [2] Cargo [3] <ul style="list-style-type: none"> weight [333] Aircraft [1] <ul style="list-style-type: none"> aircraftType maxCargoVolume maxCargoWeight tailNumber 	<ul style="list-style-type: none"> Cargo [1] <ul style="list-style-type: none"> weight [1111] Cargo [2] <ul style="list-style-type: none"> weight [2] Cargo [3] <ul style="list-style-type: none"> weight [333] Aircraft [1] <ul style="list-style-type: none"> aircraftType maxCargoVolume maxCargoWeight [2.000000] tailNumber

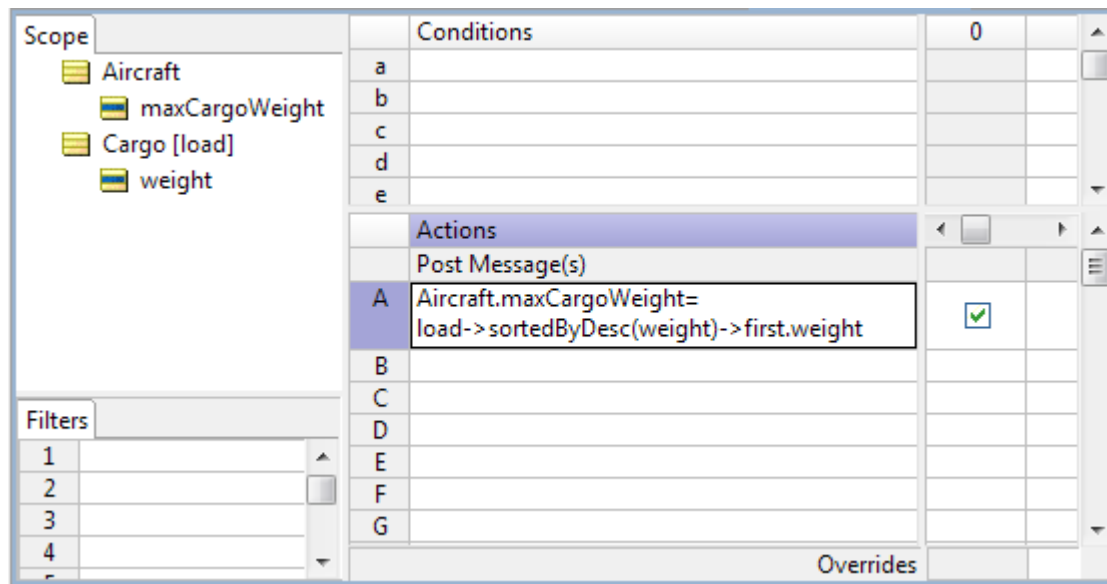
The same result is output when we modify the rule to select last item when we sort the items by descending weight.

Figure 112:

Scope	Conditions	Actions	Filters
<ul style="list-style-type: none"> Aircraft <ul style="list-style-type: none"> maxCargoWeight Cargo [load] <ul style="list-style-type: none"> weight 	a b c	A Aircraft.maxCargoWeight = load -> sortByDesc(weight) -> last.weight B C D	1 2 3

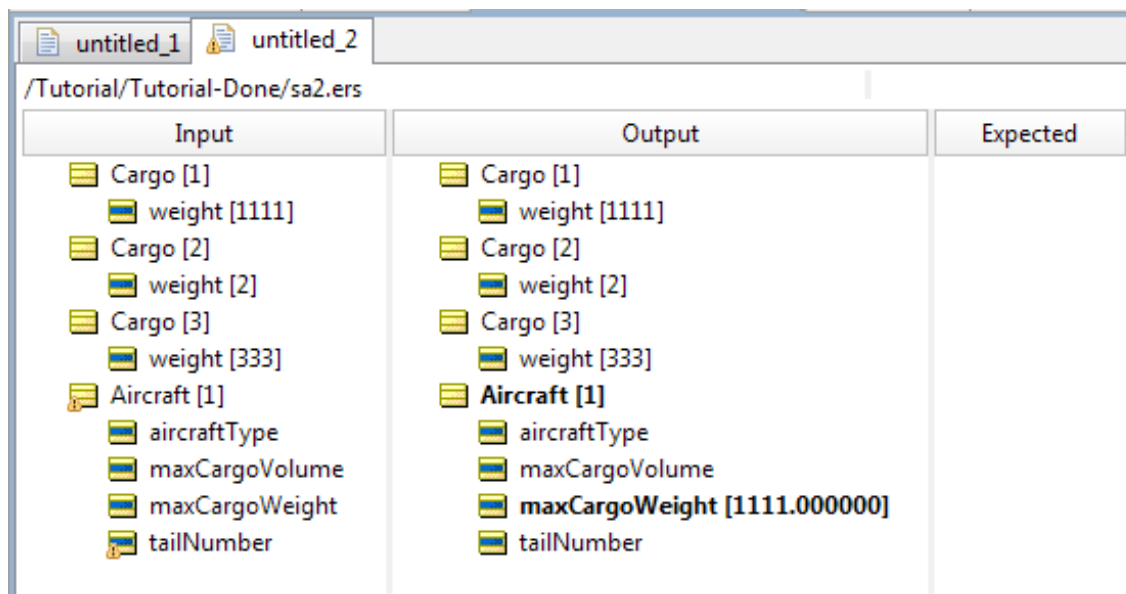
Now we reverse the test to select the first item when we sort the items by descending weight...

Figure 113:



...and the heaviest item is output.

Figure 114:



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 Corticon 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 115: Rulesheet with Universal Quantifier (“for all”) Condition

The screenshot shows the Corticon Studio Rulesheet editor. On the left is a tree view of the Operator Vocabulary, with 'forAll (expression)' selected under 'Entity/Association Operators' > 'Collection'. The main editor area is titled 'UniversalQuantifier.ers'. It contains a 'Scope' section with a tree showing 'Security [secty]' containing 'rating' and 'secInfo (SecInfo) [secinfo]'. Below this is a 'Filters' section with five empty rows. To the right is a 'Conditions' table with three columns (0, 1, 2) and five rows (a, b, c, d, e). Row 'a' contains the condition 'secinfo -> forAll(secinfo.rank >=3)' with values '-', 'T', and 'F' in columns 0, 1, and 2 respectively. Below the conditions is an 'Actions' section with a 'Post Message(s)' table. Row 'A' contains 'secty.rating' with values ''High'' and ''Low'' in columns 1 and 2. At the bottom is a 'Rule Statements' table with columns 'Ref', 'ID', 'Post', 'Alias', 'Text', and 'Rule Name'. It contains two statements: Ref 1, ID Info, Post secty, Text 'A security for which all rank values are greater than or equal to 3 should be assigned a rating of high', and Rule Name empty; and Ref 2, ID Info, Post secty, Text 'A security for which not all rank values are greater than or equal to 3 should be assigned a rating of low', and Rule Name empty.

Ref	ID	Post	Alias	Text	Rule Name
1	Info	secty		A security for which all rank values are greater than or equal to 3 should be assigned a rating of high	
2	Info	secty		A security for which not all rank values are greater than or equal to 3 should be assigned a rating of low	

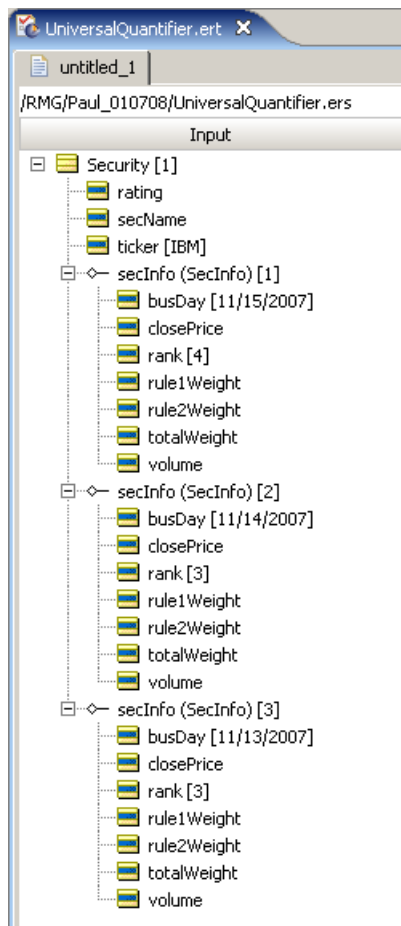
In this figure, we see the Condition

```
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.

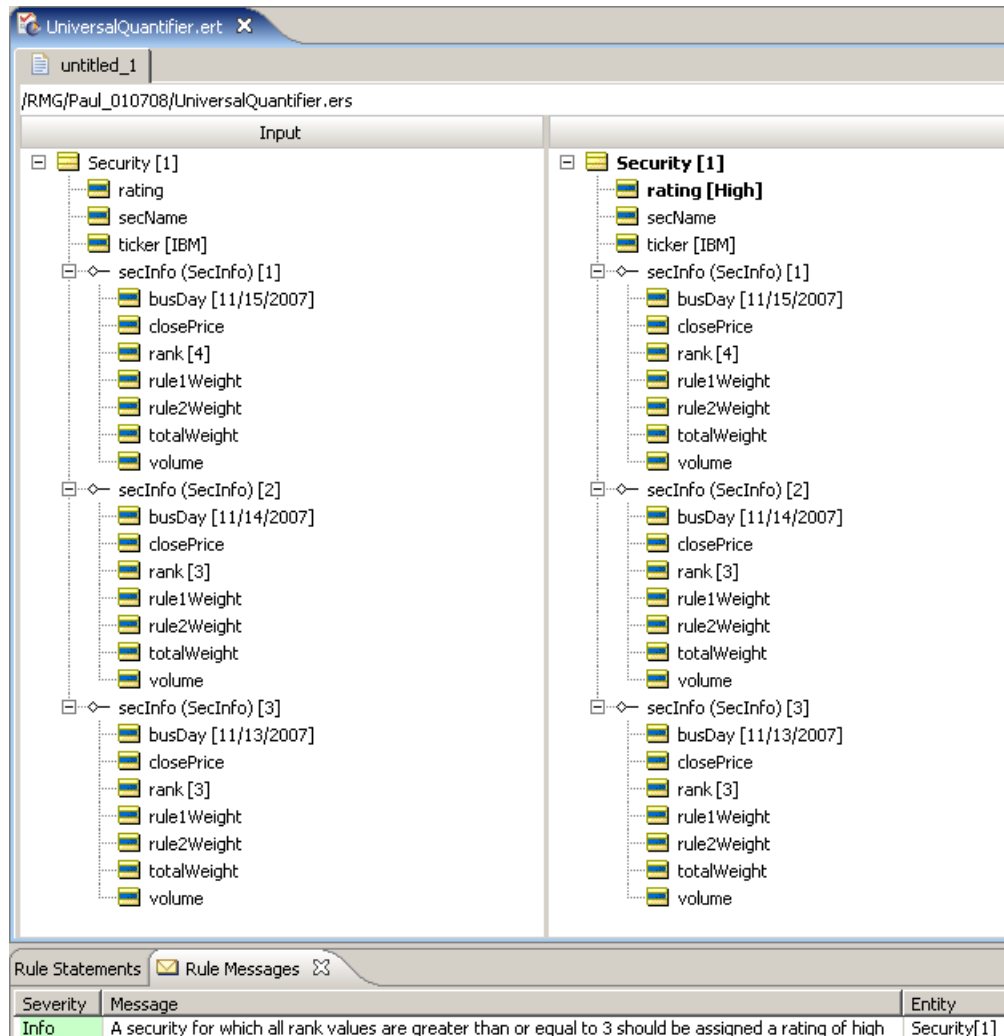
The following figure shows a Ruletest constructed to test our “for all” Condition rules.

Figure 116: 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 117: Ruletest for “for all” Condition Rules

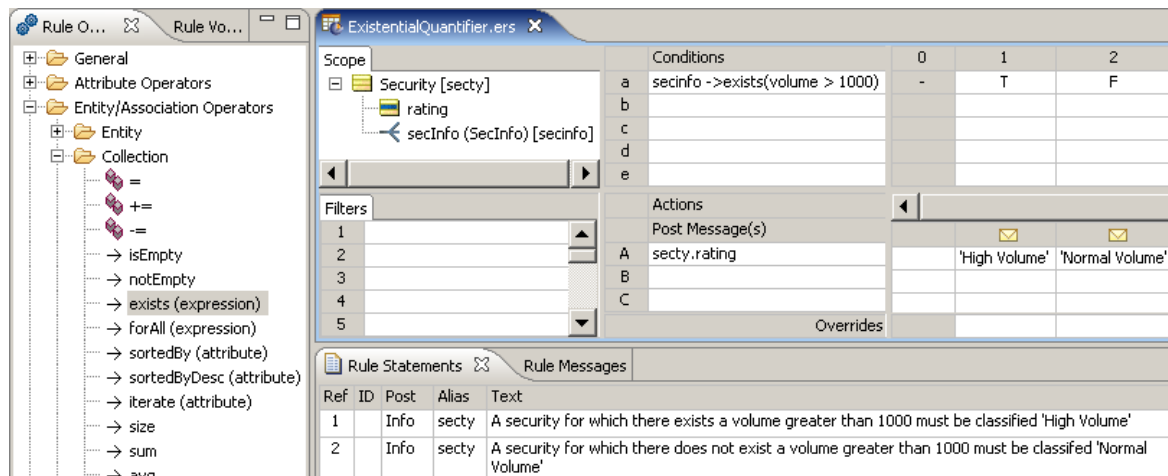


Existential quantifier

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:

Figure 118: Rulesheet with Existential Quantifier (“exists”) Condition



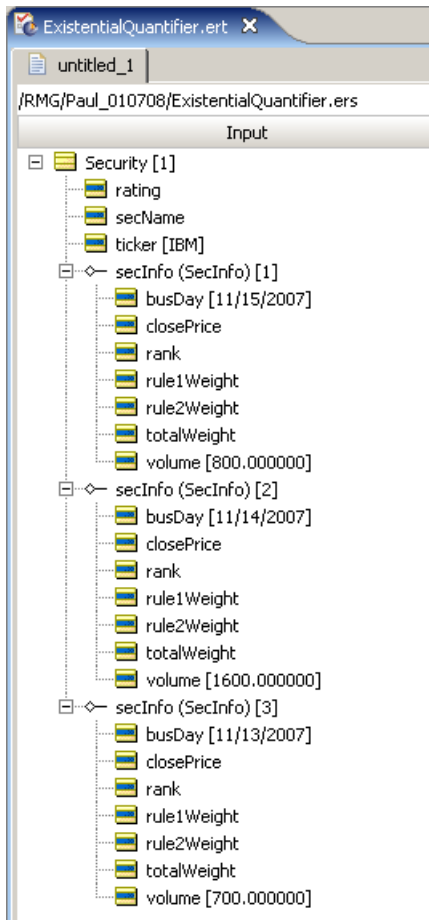
In this Rulesheet, we see the Condition

```
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 119: 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 120: Ruletest for `exists` Condition Rules

The screenshot displays the 'ExistentialQuantifier.ert' ruletest interface. It is divided into two main panels: 'Input' and 'Output'. Both panels show a tree structure representing a 'Security' entity with three associated 'SecInfo' objects.

Input Tree:

- Security [1]
 - rating
 - secName
 - ticker [IBM]
 - secInfo (SecInfo) [1]
 - busDay [11/15/2007]
 - closePrice
 - rank
 - rule1Weight
 - rule2Weight
 - totalWeight
 - volume [800.000000]
 - secInfo (SecInfo) [2]
 - busDay [11/14/2007]
 - closePrice
 - rank
 - rule1Weight
 - rule2Weight
 - totalWeight
 - volume [1600.000000]
 - secInfo (SecInfo) [3]
 - busDay [11/13/2007]
 - closePrice
 - rank
 - rule1Weight
 - rule2Weight
 - totalWeight
 - volume [700.000000]

Output Tree:

- Security [1]
 - rating [High Volume]
 - secName
 - ticker [IBM]
 - secInfo (SecInfo) [1]
 - busDay [11/15/2007]
 - closePrice
 - rank
 - rule1Weight
 - rule2Weight
 - totalWeight
 - volume [800.000000]
 - secInfo (SecInfo) [2]
 - busDay [11/14/2007]
 - closePrice
 - rank
 - rule1Weight
 - rule2Weight
 - totalWeight
 - volume [1600.000000]
 - secInfo (SecInfo) [3]
 - busDay [11/13/2007]
 - closePrice
 - rank
 - rule1Weight
 - rule2Weight
 - totalWeight
 - volume [700.000000]

At the bottom, the 'Rule Messages' tab is active, showing a message:

Severity	Message	Entity
Info	A security for which there exists a volume greater than 1000 must be classified 'High Volume'	Security[1]

Another example using the existential quantifier

Collection operators are powerful parts of the Corticon Rule Language – 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:

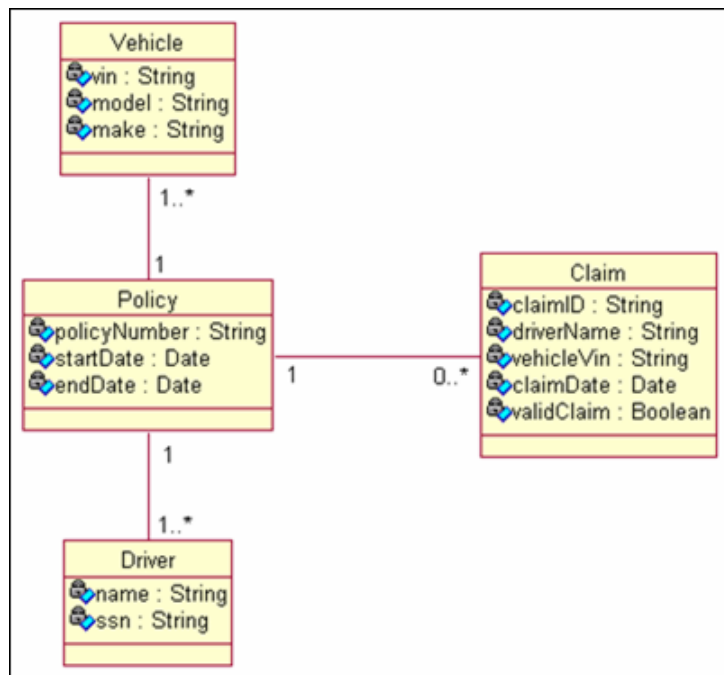
1. A claim is valid if the driver and vehicle involved in a claim are both listed on the policy against which the claim is submitted.

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`.

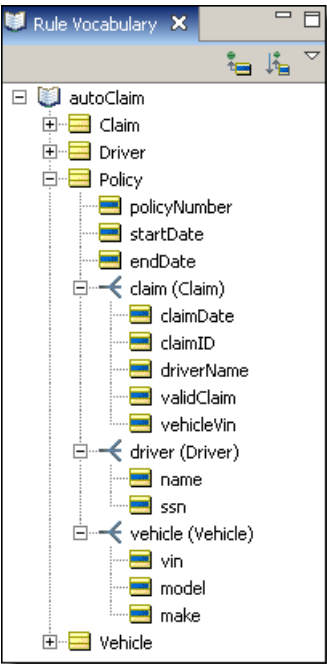
Note: Alternatively, the Claim entity could have referenced `Driver.name` and `Vehicle.vin` (by adding associations between Claim and both Driver and Vehicle), respectively, but the denormalized structure is probably more representative of a real-world scenario.

Figure 121: UML Class Diagram



This model imports into Corticon Studio as the Vocabulary shown in [Figure 122: Vocabulary](#) on page 146

Figure 122: Vocabulary



Model the following rules in Corticon Studio as shown in [Figure 123: Rules Modeled in Corticon Studio](#) on page 146

- 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 123: Rules Modeled in Corticon Studio

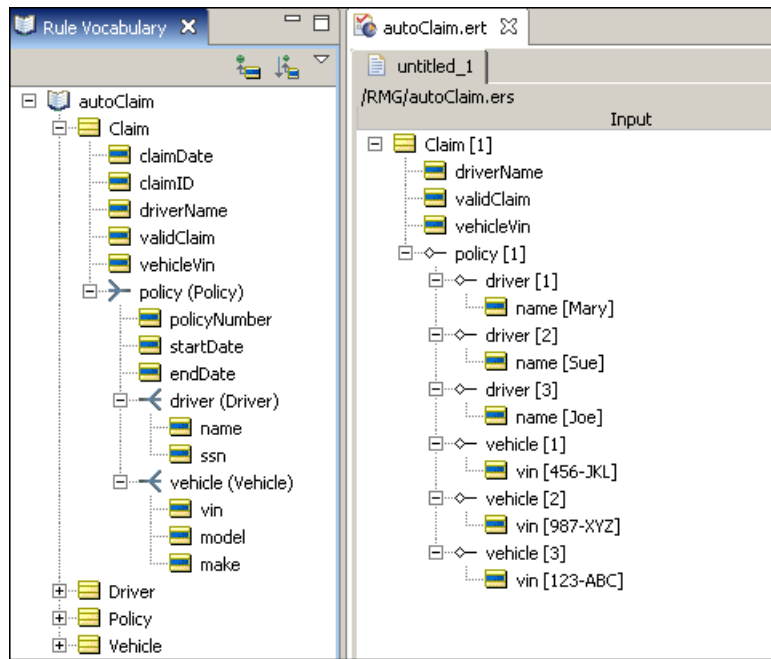
	Conditions	0	1	2	3
a	aClaim.driverName = aClaim.policy.driver.name	-	T	F	-
b	aClaim.vehicleVin = aClaim.policy.vehicle.vin	-	T	-	F
c					
d					

	Actions			
Post Message(s)				
A	aClaim.validClaim			
B				

Ref	ID	Post	Alias	Text
1		Info	aClaim	A claim is valid if its driver [{aClaim.driverName}] AND Vehicle match the policy against it was submitted [{aClaim.policy.driver.name}] and [{aClaim.policy.vehicle.vin}]
2		Warning	aClaim	A claim is not valid if its driver [{aClaim.driverName}] is not on the policy against which it was submitted [{aClaim.policy.driver.name}]
3		Warning	aClaim	A claim is not valid if its vehicle [{aClaim.vehicleVin}] is not on the policy against which it was submitted [{aClaim.policy.vehicle.vin}]

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 124: Ruletest](#) on page 147

Figure 124: Ruletest



Notice in [Figure 125: Ruletest](#) on page 148 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 125: Ruletest

The screenshot shows the Corticon Studio Ruletest interface. The top pane displays the 'Input' and 'Output' trees. The 'Input' tree shows a 'Claim [1]' object with attributes 'driverName [Joe]', 'validClaim', and 'vehicleVin [123-ABC]'. It also contains a 'policy (Policy) [1]' object with three 'driver (Driver)' objects (names: Mary, Sue, Joe) and three 'vehicle (Vehicle)' objects (vin: 456-JKL, 987-XYZ, 123-ABC). The 'Output' tree shows the same structure, but 'validClaim' is set to '[false]'. The bottom pane shows the 'Rule Messages' table.

Severity	Message	Entity
Info	A claim is valid if its driver [Joe] AND Vehicle match the policy against it was submitted [Joe] and [123-ABC]	Claim[1]
Warning	A claim is not valid if its driver [Joe] is not on the policy against which it was submitted [Sue]	Claim[1]
Warning	A claim is not valid if its driver [Joe] is not on the policy against which it was submitted [Mary]	Claim[1]
Warning	A claim is not valid if its vehicle [123-ABC] is not on the policy against which it was submitted [987-XYZ]	Claim[1]
Warning	A claim is not valid if its vehicle [123-ABC] is not on the policy against which it was submitted [456-JKL]	Claim[1]

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 126: Rules Rewritten Using Existential Quantifier.

The screenshot shows the ExistentialAutoClaim.ers rule editor. The Scope tree on the left includes Claim [c], driverName, validClaim, vehicleVin, policy, driver [cpd], and vehicle [cpv]. The Conditions table has four columns (0, 1, 2, 3) and eight rows (a-h). The Actions table has four columns (0, 1, 2, 3) and six rows (A-F). The Rule Messages table at the bottom lists three rules (A1, A2, A3) with their respective IDs, post messages, aliases, and text.

Scope		Conditions	0	1	2	3
a	cpd -> exists(name = c.driverName)			F	-	T
b	cpv -> exists(vin = c.vehicleVin)			-	F	T
c						
d						
e						
f						
g						
h						

Actions		0	1	2	3
A	c.validClaim		F	F	T
B					
C					
D					
E					
F					

Ref	ID	Post	Alias	Text
A1		Warning	c	A claim is not valid if its driver [{c.driverName}] is not on the policy against which it is submitted
A2		Warning	c	A claim is not valid if its vehicle [{c.vehicleVin}] is not on the policy against which it is submitted
A3		Info	c	A claim is valid if its driver [{c.driverName}] AND vehicle [{c.vehicleVin}] match those on the policy against which it is submitted

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:

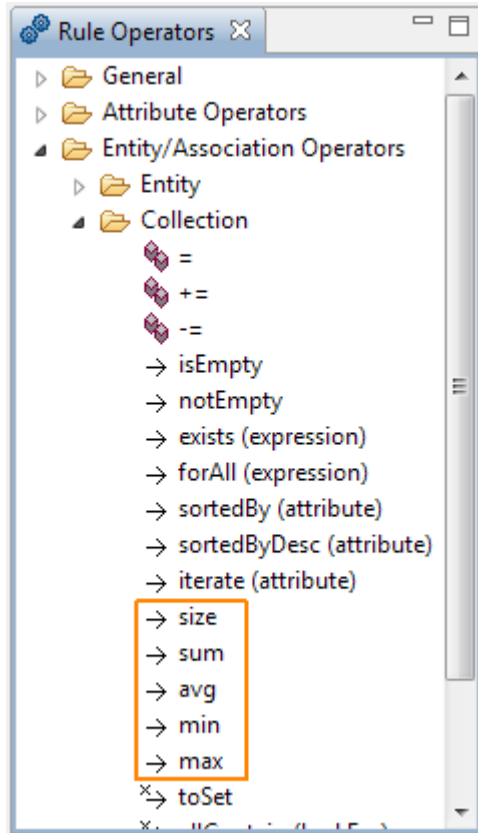
The screenshot shows the autoClaim.ert rule test results. The Input tree on the left shows a Claim [1] with driverName [Joe], validClaim, vehicleVin [123-ABC], and a policy (Policy) [1] containing three drivers (Mary, Sue, Joe) and three vehicles (456-JKL, 987-XYZ, 123-ABC). The Output tree on the right shows the same Claim [1] but with validClaim [true]. The Rule Messages table at the bottom shows a single message: 'A claim is valid if its driver [Joe] AND vehicle [123-ABC] match those on the policy against which it is submitted' with severity Info and entity Claim[1].

Severity	Message	Entity
Info	A claim is valid if its driver [Joe] AND vehicle [123-ABC] match those on the policy against which it is submitted	Claim[1]

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 243

Test yourself questions: Collections

Note: Try this test, and then go to [Test yourself answers: Collections](#) on page 352 to correct yourself.

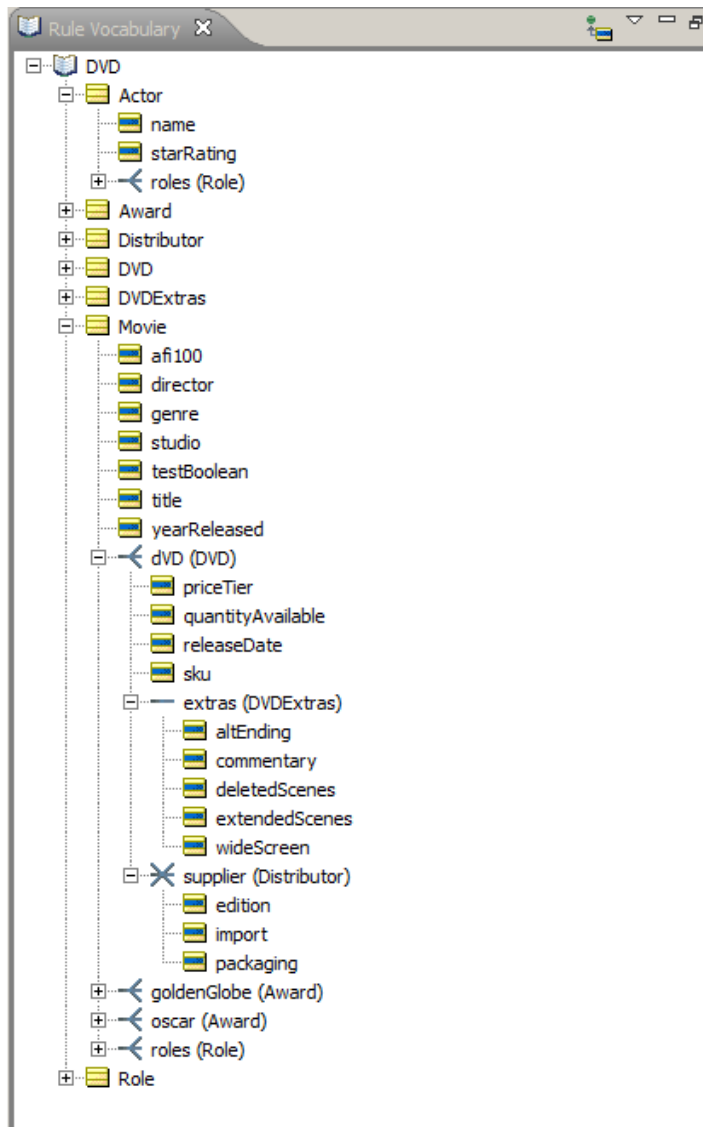
1. Children of a Parent entity are also known as _____ of a collection.
2. True or False. All collections must have a parent entity
3. True or False. Root-level entities may form a collection
4. True or False. A collection operator must operate on a collection alias.
5. List three Collection operators 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 entities 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?

Rules containing calculations and 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](#)
- [Operator precedence and order of evaluation](#)
- [Datatype compatibility and casting](#)
- [Supported uses of calculation expressions](#)
- [Unsupported uses of calculation expressions](#)
- [Test yourself questions: Rules containing calculations and equations](#)

Terminology

First we will introduce some terminology that will be used throughout this chapter. In the simple 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 such a simple 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, where a common solution is to use two different operators to distinguish between the two meanings -- the symbol `==` might signify a comparison operation, whereas `:=` might signify an assignment.

In Corticon Studio, 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.

Note: A Boolean attribute does not reset when non-boolean input is provided for a non-conditional rule

While this is the expected behavior in the Corticon language, it can cause unexpected results. On input of a Boolean attribute, if the value of the XML element is `true` or `1`, Corticon interprets that as a `true` Boolean value, otherwise it defaults to a `false` Boolean value. Attributes in the input document are not modified unless the value is actually changed in the rule; that is, setting a `true` Boolean attribute to the value of `true` does not modify the XML.

You can have reliable behavior when you use following workaround. To guarantee a modification in the XML, you need to guarantee that the rules actually change the value of the attribute. For example, instead of action...

```
Entity_1.booleanAttr1 = T
```

...first set the value of the attribute to null, and then set it to true:

```
Entity_1.booleanAttr1 = null
Entity_1.booleanAttr1 = T
```

Operator precedence and order of evaluation

Operator precedence -- the order in which Corticon Studio evaluates multiple operators in an equation -- is described in the following table (also in 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`.

The precedence of operators affects the grouping and evaluation of expressions. Expressions with higher-precedence operators are evaluated first. Where several operators have equal precedence, they are evaluated from left to right. The following table summarizes Corticon's Rule Operator precedence and their order of evaluation .

Operator precedence	Operator	Operator Name	Example
1	()	Parenthetic expression	(5.5 / 10)
2	-	Unary negative	-10
	not	Boolean test	not 10
3	*	Arithmetic: Multiplication	5.5 * 10
	/	Arithmetic: Division	5.5 / 10
	**	Arithmetic: Exponentiation (Powers and Roots)	5 ** 2 25 ** 0.5 125 ** (1.0/3.0)
4	+	Arithmetic: Addition	5.5 + 10
	-	Arithmetic: Subtraction	10.0 – 5.5
5	<	Relational: Less Than	5.5 < 10
	<=	Relational: Less Than Or Equal To	5.5 <= 5.5
	>	Relational: Greater Than	10 > 5.5
	>=	Relational: Greater Than Or Equal To	10 >= 10
	=	Relational: Equal	5.5=5.5
	<>	Relational: Not Equal	5.5 <> 10
6	(<i>expression</i> , <i>expression</i>)	Logical: AND	(>5.5,<10)
	(<i>expression</i> or <i>expression</i>)	Logical: OR	(<5.5 or >10)

Note: While expressions within parentheses that are separated by logical AND / OR operators are valid, the component expressions are not evaluated individually when testing for completeness, and might cause unintended side effects during rule execution. Best practice within a Corticon Rulesheet is to represent AND conditions as separate condition rows and OR conditions as separate rules -- doing so allows you to get the full benefit of Corticon's logical analysis.

Note: It is recommended that you place arithmetic exponentiation expressions in parentheses.

Datatype compatibility and casting

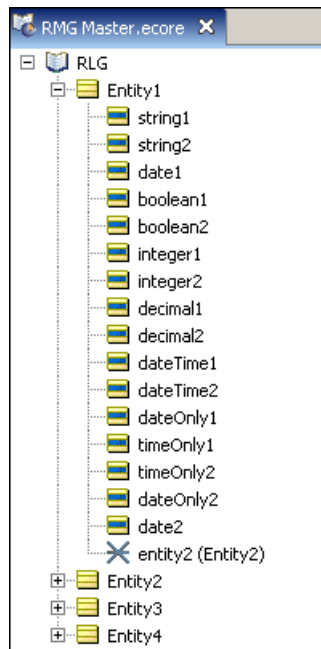
An important prerequisite of any comparison or assignment operation is data type compatibility. In other words, the data type of the equation's LHS (the data type of *A*) must be compatible with whatever data type results from the evaluation of the equation's RHS (the data type of *B*). For example, 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 *orange* (a String) to *July 4, 2014* (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.

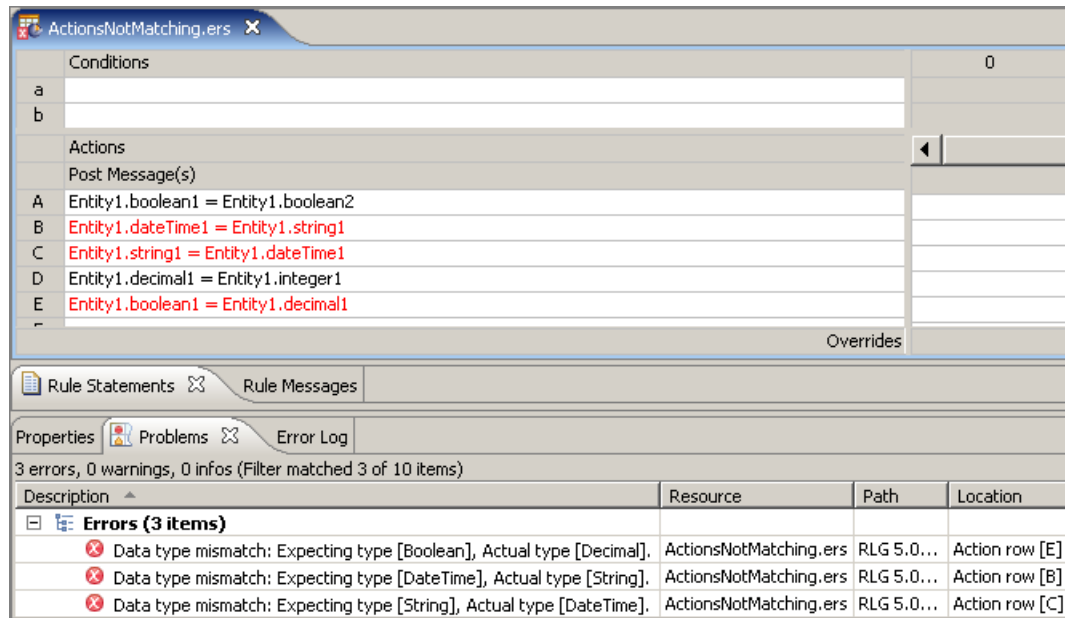
In the examples that follow, we will use the generic Vocabulary from the *Rule Language Guide*, since the generic attribute names indicate their data types:

Figure 127: Generic Vocabulary used in 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 128: 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).

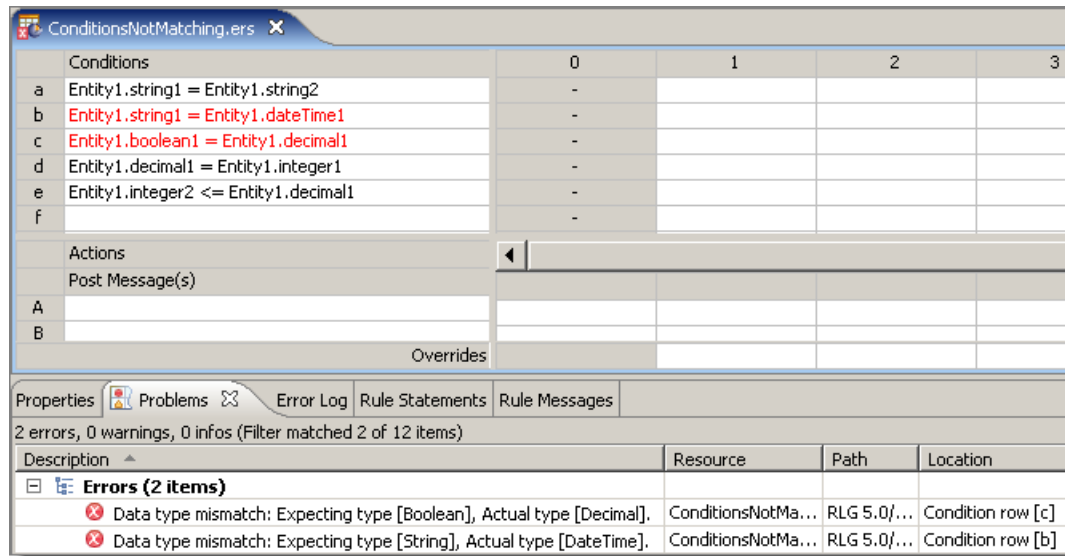
D – this expression is valid because the data types of the LHS and RHS sides of the equation are compatible *even though they are different!* This is an example of the one exception to our general rule regarding data type compatibility: Decimals can safely hold Integer values.

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 129: Datatype Mismatches in Comparison 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).

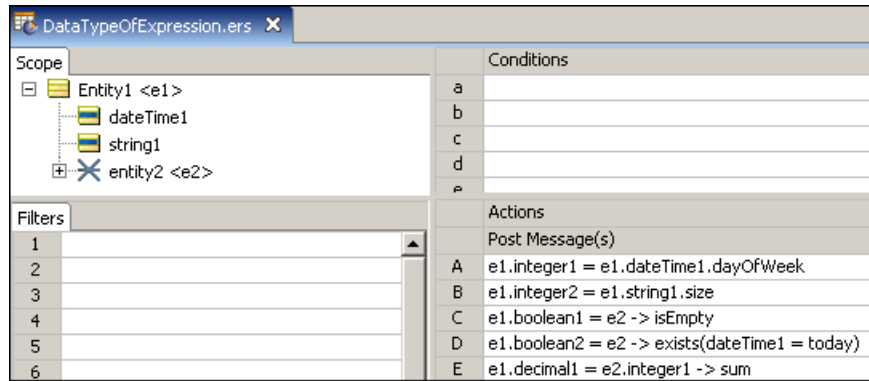
d – This comparison expression is valid because the data types of the LHS and RHS sides of the equation are compatible. This is another example of the one exception to our general rule regarding data type compatibility: Decimals may be safely compared to Integer values.

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 130: 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 `dateTime1`) 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.

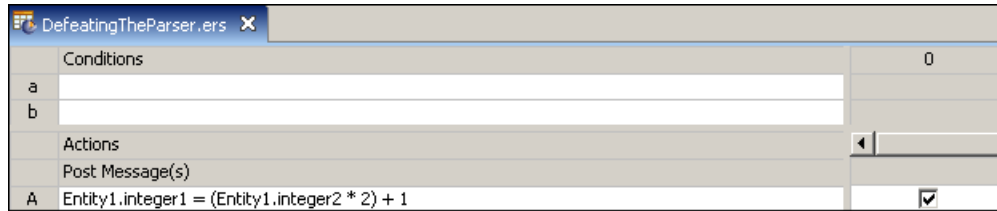
E – the RHS of this equation resolves to an Integer because the `->sum` collection operator adds up the values of all occurrences of an attribute (in this case, `integer2`) in a collection (in this case, `e2` again). Since the LHS has a Decimal data type, the assignment operation is valid. This is the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

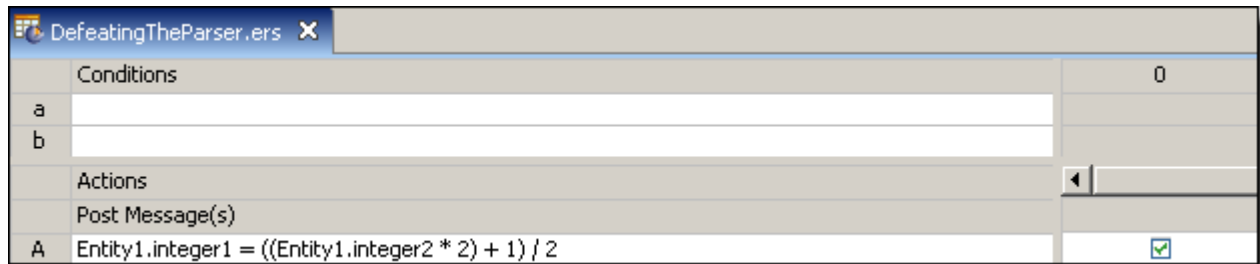
The part of Corticon Studio that checks for data type mismatches (along with all other syntactical problems) is the Parser. The Parser exists to ensure that whatever is expressed in a Rulesheet can be correctly translated and compiled into code executable by Corticon Studio's Ruletest as well as by the 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 131: 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 132: 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 Corticon 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

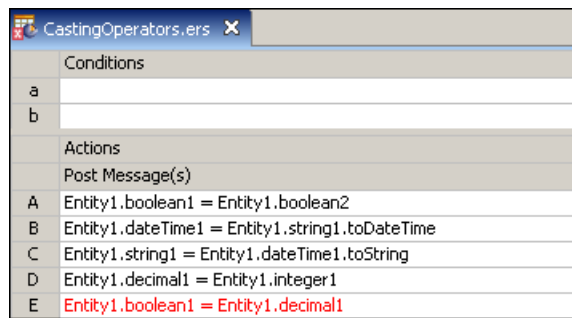
A special set of operators is provided in the Corticon Studio's Operator Vocabulary that allows the rule modeler to control the data types of attributes and expressions. These casting operators are described below:

Table 6: 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 Comparison Expressions](#), we use these casting operators to correct some of the previous problems:

Figure 133: Using Casting Operators

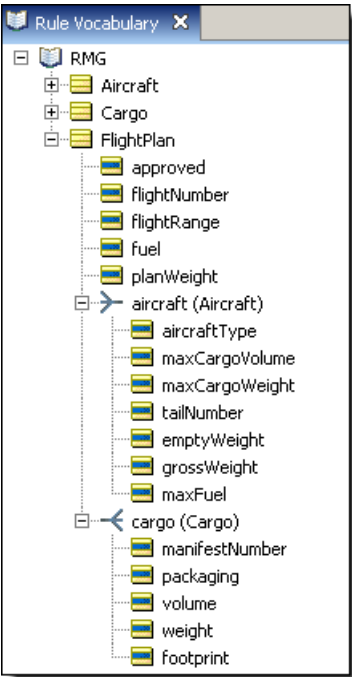


Casting operators have been used in Nonconditional rules N.2 and N.3 to make the data types of the LHS and RHS match. Notice however, that no casting operator exists to cast a Decimal into a Boolean data type.

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 134: Basic Tutorial Vocabulary Extended

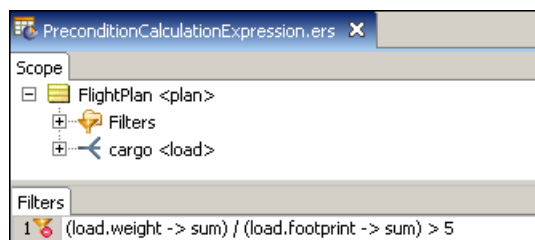


The new attributes are described in the table below:

Table 7: Table: Table of New Attributes Added to the Basic Tutorial Vocabulary

Attribute	Data type	Description
<code>Aircraft.emptyWeight</code>	Decimal	The weight of an Aircraft with no fuel or cargo onboard.(kilograms)
<code>Aircraft.grossWeight</code>	Decimal	The maximum amount of weight an Aircraft can safely lift, equal to the sum of cargo and fuel weights. (kilograms)
<code>Aircraft.maxfuel</code>	Decimal	The maximum amount of fuel an Aircraft can carry. (liters)
<code>Cargo.footprint</code>	Decimal	The floor space required for this Cargo. (square meters)
<code>FlightPlan.approved</code>	Boolean	Indicates whether the FlightPlan has been approved or “cleared” for operation.
<code>FlightPlan.planWeight</code>	Decimal	The total amount of all Aircraft and Cargo weights for this FlightPlan. (kilograms)
<code>FlightPlan.flightRange</code>	Decimal	The distance the Aircraft is expected to fly. (kilometers)
<code>FlightPlan.fuel</code>	Decimal	The amount of fuel actually loaded on the Aircraft assigned to this FlightPlan. (liters)

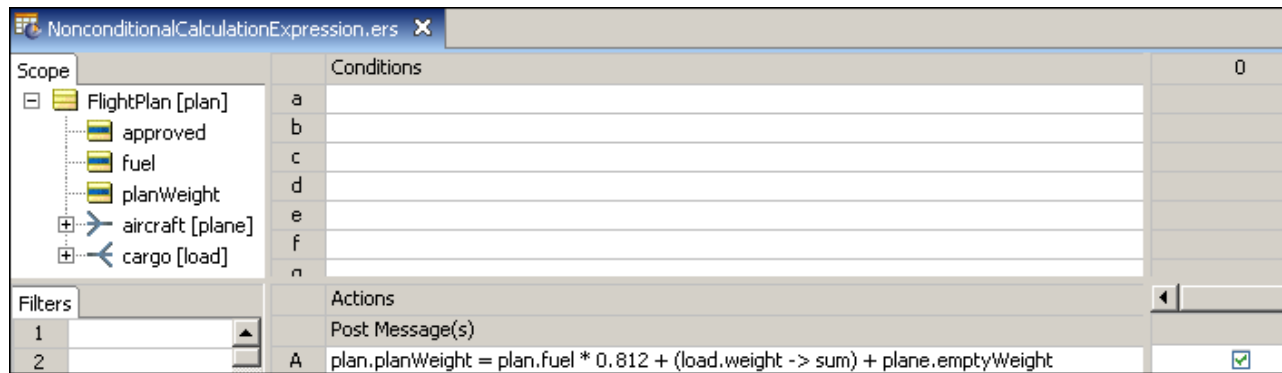
Calculation as a comparison in a precondition

Figure 135: 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 136: 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 * 0.812
```

converts a fuel load measured in liters -- the unit of measure that airlines purchase and load fuel -- into a weight measured in kilograms -- unit of measure used for the weight of the cargo as well as the aircraft and crew. Note that this conversion is a bit conservative as Jet A1 fuel expands as it warms up so this figure considers it to be at the cool end of its range. This portion is then added to:

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

which 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 Rulesheets.

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 137: planWeight Derived and Used in Same Rulesheet

Scope		Conditions	0	1
FlightPlan [plan]	approved	a		T
	fuel	b		
	planWeight	c		
	aircraft [plane]	d		
	cargo [load]	e		
		f		
		a		
Filters		Actions		
1		Post Message(s)		
2		A		<input checked="" type="checkbox"/>
3		B		F

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.

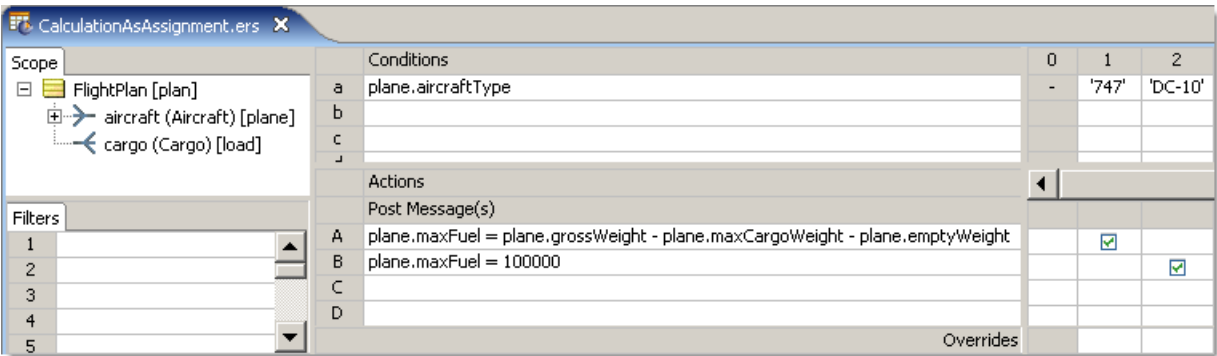
Figure 138: A Calculation in a Conditional Expression

Scope		Conditions	0	1
FlightPlan [plan]	approved	a		T
	fuel	b		
	planWeight	c		
	aircraft [plane]	d		
	cargo [load]	e		
		f		
		a		
Filters		Actions		
1		Post Message(s)		
2		A		<input checked="" type="checkbox"/>
				F

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 “transient” attributes in the Vocabulary, which causes their icons to change from blue/yellow to orange/yellow. More details on transient attributes are included in [Modeling the Vocabulary in Corticon Studio](#) on page 17 of this guide.

Calculation as an assignment in an action

Figure 139: A Calculation in an Action Expression



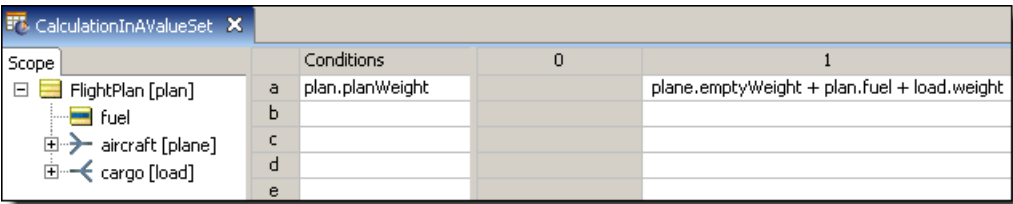
This figure shows two rules that each make an assignment to `maxFuel`, depending on the type of aircraft. In rule 1, the `maxFuel` load for 747s is derived by subtracting `maxCargoWeight` and `emptyWeight` from `grossWeight`. In rule 2, `maxFuel` for DC-10s is simply assigned the literal value 100,000.

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 140: 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 141: Calculation in a Rule Statement

Ref	ID	Post	Alias	Text
1				2 * 3 + 4

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 142: Embedding a Calculation in a Rule Statement

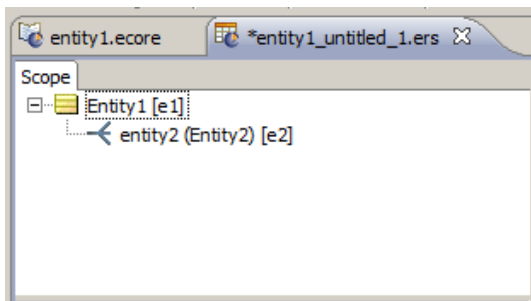
Ref	ID	Post	Alias	Text
1				The value of maxFuel squared is {plane.maxFuel ** 2}
				problem parsing:The value of maxFuel squared is {plane.maxFuel ** 2}:Invalid Token: **

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 [Test yourself answers: Rules containing calculations and equations](#) on page 353 to correct yourself.

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

Conditions	0	1	2
a aDVD.quantityAvailable -> sum / aDVD->size	-	<= 50000	> 50000
b			
c			
d			
e			
f			
g			
h			

Actions	0	1	2
Post Message(s)		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
A			
B			
C			
D			
E			

Filters
1 today.year - film.yearReleased > 10
2
3
4
5
6
7

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 ____

Rule dependency: Chaining and looping

For details, see the following topics:

- [What is rule dependency?](#)
- [Forward chaining](#)
- [Rulesheet processing: modes of looping](#)
- [Looping controls in Corticon Studio](#)
- [Looping examples](#)
- [Using conditions as a processing threshold](#)
- [Test yourself questions: Rule dependency: chaining and looping](#)

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 Ruleflow 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 use 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 Rulesheets.

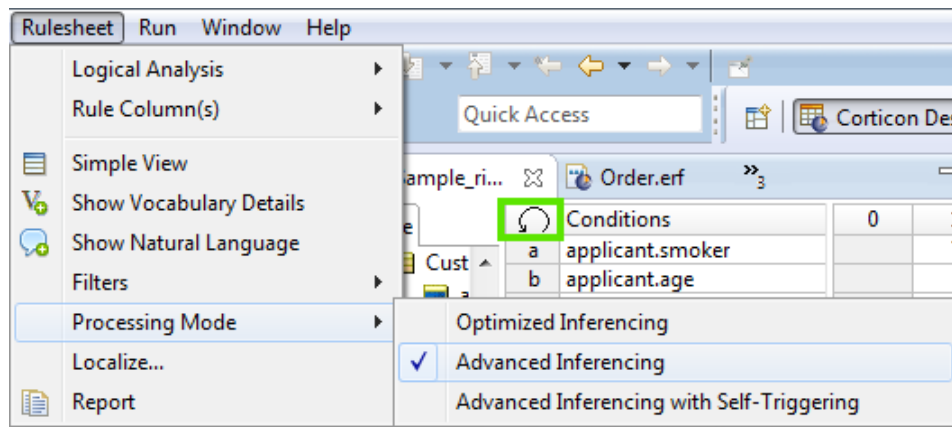
Optimized Inferencing processing is a powerful capability that enables the rule modeler to “break up” complex logic into a series of smaller, less complex rules. Once broken up into smaller or simpler rules, the logic will be executed in the proper sequence automatically, based on the dependencies determined 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 143: 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 144: 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

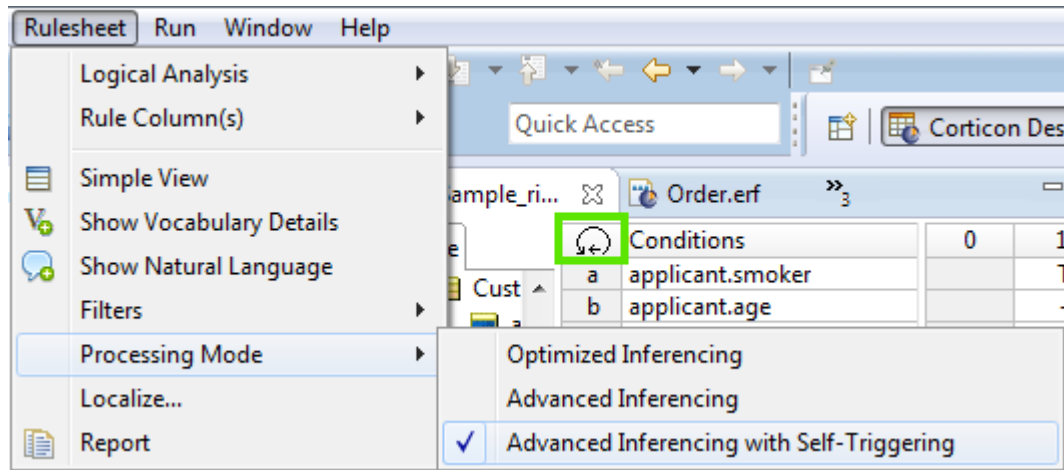
In the example above, looping between rules 2, 3 and 4 continues indefinitely because there is nothing to stop the cycle. Some loops, especially those inadvertently introduced, are not self-terminating. Because these loops will not end by themselves, they are called infinite loops. Infinite loops can 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 the Corticon Studio menubar, as shown in

Figure 145: Selecting Advanced Inferencing with Self-Triggering Processing Mode for a Rulesheet



Notice 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 146: Example of an Infinite Single-Rule Loop

The screenshot shows a rulesheet titled 'InfiniteSingleLoop.ers'. It has a 'Conditions' section with three rows (a, b, c) and an 'Actions' section with one row (A). The 'Conditions' section has a self-triggering icon (a circle with a curved arrow) next to the header. The 'Actions' section has a checkmark in the 'Post Message(s)' column for action A.

Conditions	0	1
a	Cargo.weight >= 0	-
b		T
c		

Actions	Post Message(s)
A	Cargo.weight += 1

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 attributes 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 147: Example of an Infinite Loop created by a Self-Triggering Rule

The screenshot shows a rulesheet titled 'SingleLoopAsNonconditional.ers'. It has a 'Conditions' section with three rows (a, b, c) and an 'Actions' section with one row (A). The 'Conditions' section has a self-triggering icon (a circle with a curved arrow) next to the header. The 'Actions' section has a checkmark in the 'Post Message(s)' column for action A.

Conditions	0
a	
b	
c	

Actions	Post Message(s)
A	Cargo.weight += 1

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 148: Example of a Finite Single-Rule Loop

FiniteSingleRuleLoop.ers		0	1
Conditions			
a	Cargo.weight	-	0..20
b			
c			
Actions			
Post Message(s)			
A	Cargo.weight +=1		<input checked="" type="checkbox"/>
B			

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:

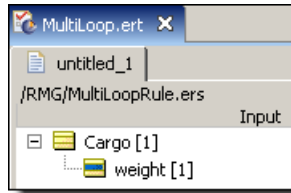
Figure 149: Example of a Finite Multi-Rule Loop

MultiLoopRule.ers		0	1	2	3
Conditions					
a	Cargo.weight	-	1..10	-	> 10
b	Cargo.volume > 0	-	-	T	-
-					
Actions					
Post Message(s)					
A	Cargo.volume = Cargo.weight * 2		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
B	Cargo.weight += 1			<input checked="" type="checkbox"/>	
C					
Overrides					

Ref	ID	Post	Alias	Text
1		Info	Cargo	If weight is between 0 and 10 [{Cargo.weight}] then volume is twice weight [{Cargo.volume}]
2		Warning	Cargo	As long as volume is greater than 0, increment weight by 1 [now{Cargo.weight}]
3		Violation	Cargo	weight has exceeded the threshold, so the loop terminates

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 150: 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 151: Ruletest for the Multi-rule Rulesheet

Input:

- Cargo [1]
 - weight [1]

Output:

- Cargo [1]
 - volume [20.000000]
 - weight [11.000000]

Rule Messages:

Severity	Message	Entity
Info	If weight is between 0 and 10 [1] then volume is twice weight [=2.000000]	Cargo[1]
Warning	As long is volume is greater than 0, increment weight by 1 [now=2.000000]	Cargo[1]
Info	If weight is between 0 and 10 [2.000000] then volume is twice weight [=4.000000]	Cargo[1]
Warning	As long is volume is greater than 0, increment weight by 1 [now=3.000000]	Cargo[1]
Info	If weight is between 0 and 10 [3.000000] then volume is twice weight [=6.000000]	Cargo[1]
Warning	As long is volume is greater than 0, increment weight by 1 [now=4.000000]	Cargo[1]
Info	If weight is between 0 and 10 [4.000000] then volume is twice weight [=8.000000]	Cargo[1]
Warning	As long is volume is greater than 0, increment weight by 1 [now=5.000000]	Cargo[1]
Info	If weight is between 0 and 10 [5.000000] then volume is twice weight [=10.000000]	Cargo[1]
Warning	As long is volume is greater than 0, increment weight by 1 [now=6.000000]	Cargo[1]
Info	If weight is between 0 and 10 [6.000000] then volume is twice weight [=12.000000]	Cargo[1]
Warning	As long is volume is greater than 0, increment weight by 1 [now=7.000000]	Cargo[1]
Info	If weight is between 0 and 10 [7.000000] then volume is twice weight [=14.000000]	Cargo[1]
Warning	As long is volume is greater than 0, increment weight by 1 [now=8.000000]	Cargo[1]
Info	If weight is between 0 and 10 [8.000000] then volume is twice weight [=16.000000]	Cargo[1]
Warning	As long is volume is greater than 0, increment weight by 1 [now=9.000000]	Cargo[1]
Info	If weight is between 0 and 10 [9.000000] then volume is twice weight [=18.000000]	Cargo[1]
Warning	As long is volume is greater than 0, increment weight by 1 [now=10.000000]	Cargo[1]
Info	If weight is between 0 and 10 [10.000000] then volume is twice weight [=20.000000]	Cargo[1]
Warning	As long is volume is greater than 0, increment weight by 1 [now=11.000000]	Cargo[1]
Violation	weight has exceeded the threshold, so the loop terminates	Cargo[1]

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 149: Example of a Finite Multi-Rule Loop](#) on page 178) 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

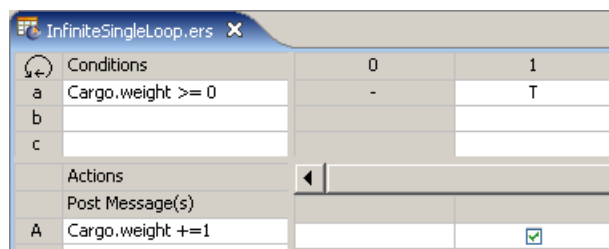
Assuming we haven't intentionally incorporated looping logic in our Rulesheet, then we need a way to discover if unintentional loops occur in our rules.

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 152: Example of an Infinite Single-Rule Loop



Conditions		0	1
a	Cargo.weight >= 0	-	T
b			
c			

Actions		
Post Message(s)		
A	Cargo.weight += 1	<input checked="" type="checkbox"/>

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 153: Checking for Logical Loops in a Rulesheet

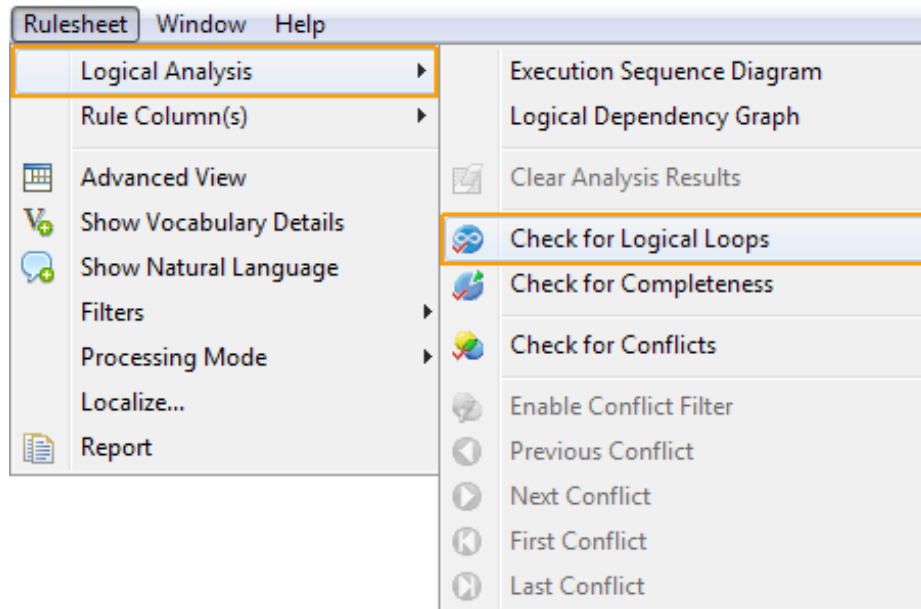
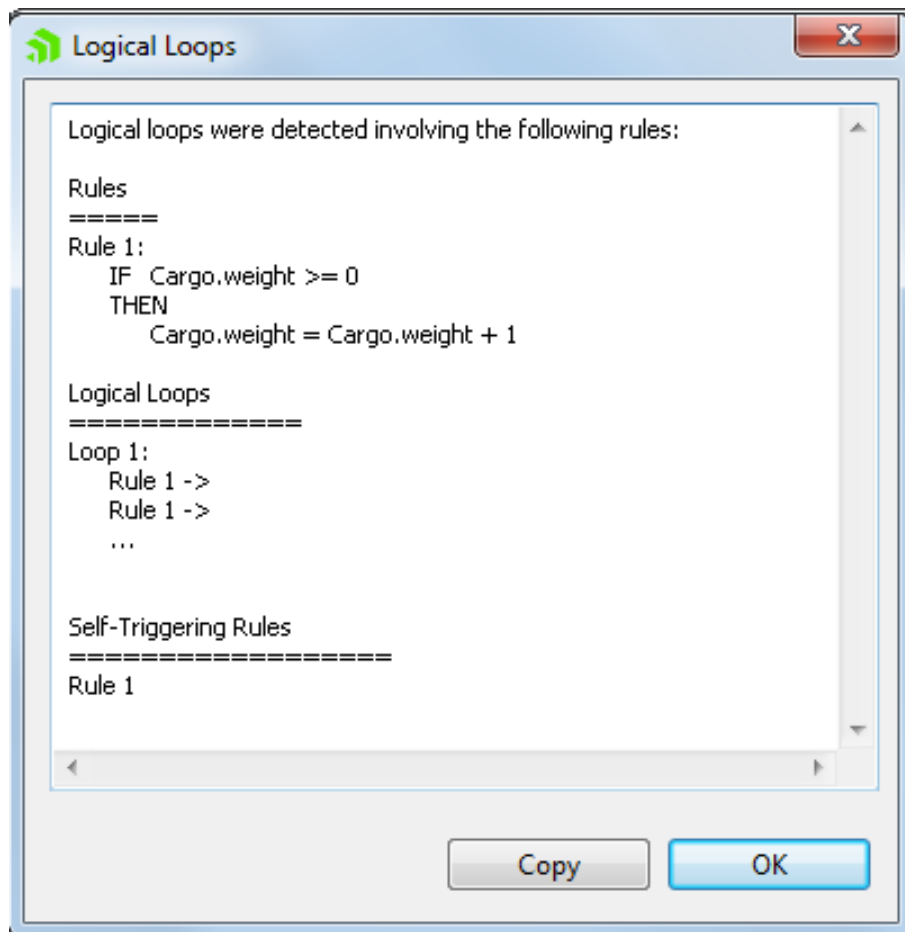


Figure 154: A Single-Rule Loop Detected by the Check for Logical Loops Tool



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

Removing loops

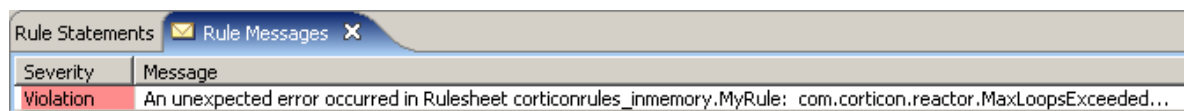
If the Check for Logical Loops tool detects loops, we can take one of several corrective actions:

- 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 155: 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 Rulesheets 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 examples

The following examples show how looping can be useful in your models.

Given a date, determine the next working day

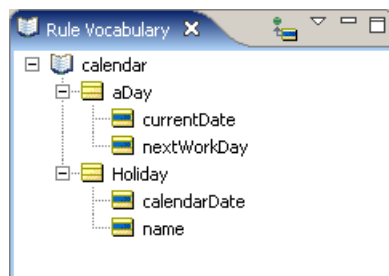
Problem

For any given date, 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 156: Sample Vocabulary for Holiday Rules



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

Figure 157: Sample Rulesheet for Determining Next Workday

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)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
B	aDay.nextWorkDay = aDay.nextWorkDay.addDays(1)		<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>		
C	aDay.nextWorkDay = aDay.nextWorkDay.addDays(2)					<input checked="" type="checkbox"/>	
D							
Overrides							
Ref	Post	Alias	Rule Messages				
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				

Let's step through this Rulesheet.

1. First, notice that the Scope section is not used. We are using a very simple Vocabulary with short entity names and no associations, so aliases are not necessary. Furthermore, none of our rules use collection operations, so aliases representing collections are not 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 one 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 two 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:

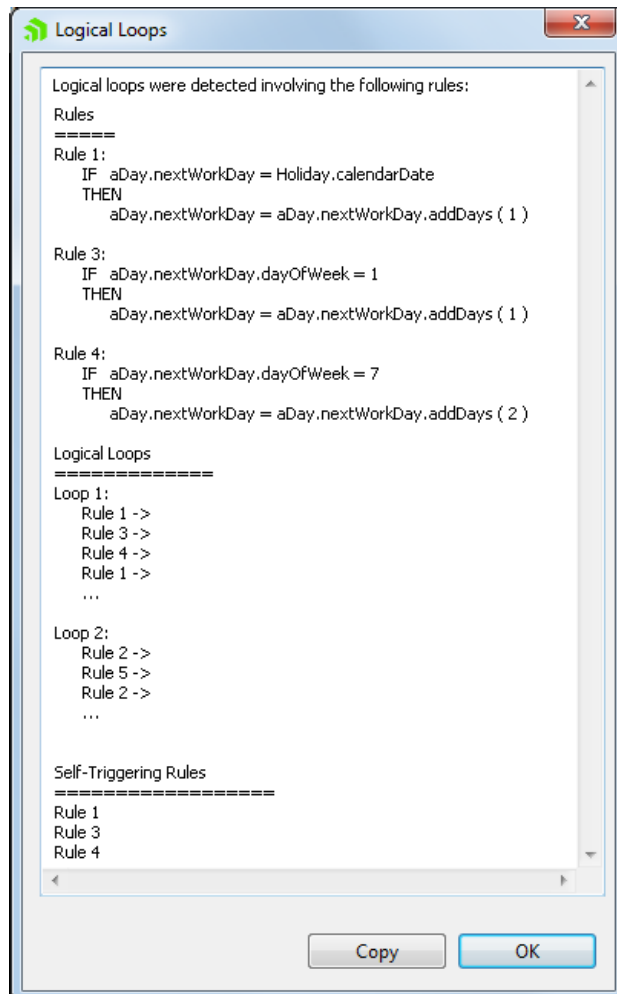
Figure 158: Holiday Rules with Ambiguities Resolved by Overrides

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)	<input checked="" type="checkbox"/>					
C	aDay.nextWorkDay = aDay.nextWorkDay.addDays(2)		<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>		
D						<input checked="" type="checkbox"/>	
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:

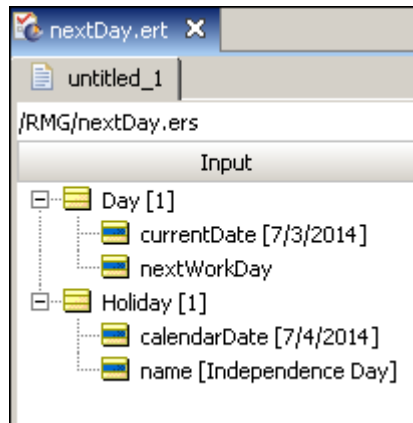
Figure 159: Results of Logical Loop Check



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.

Figure 160: Ruletest for Holiday Rules



Given that July 4th, 2014 falls on a Friday, we expect `nextWorkDay` to contain a final value of July 7th, 2014 – a Monday – when the loops terminate. When we run the Ruletest, we see the following:

Figure 161: Ruletest for Holiday Rules

Severity	Message	Entity
Info	Today is [7/3/2014] and the next day is [7/4/2014]	aDay[1]
Warning	The next day falls on a holiday, so increment to the next day [7/5/2014]	aDay[1]
Warning	The next day falls on a Saturday, so increment two days to [7/7/2014]	aDay[1]
Info	The next day does not fall on a holiday, so do not increment	aDay[1]
Info	The next day does not fall on a Saturday or Sunday, so do not increment	aDay[1]

As you can see, we got the result we wanted...a three-day weekend!

Removing duplicated children in an association

Problem

For a `Customer->Address` association (one-to-many), each address must be unique.

Solution

Compare every address associated with a customer with every other address associated with that customer, and -- when a match is found -- remove (or mark) one of the addresses.

The following example compares ALL pairs of addresses that meet a filter condition. That process occurs in no specific order so you might notice that one run starts with address 4 and address 2 ($id=1 < id=4$), yet the next time it runs, it might start with address 3 and address 1 ($id=2 < id=3$). So the results might seem different. However, all that is required is that only **one** of each unique address survive.

To assure that we can control the filtering process, we need unique identifier attribute values to distinguish the instances. If the address already has an attribute that is a unique identifier, then you could use that in the filter; otherwise we need to create a transient, integer attribute, `id`, in the `Address` entity in the Vocabulary:

The screenshot shows the Vocabulary editor for a project named 'testRemove'. The 'Address' entity is selected, and its properties are listed in a table:

Property Name	Property Value
Attribute Name	id
Data Type	Integer
Mandatory	No
Mode	Transient

On the left, the entity hierarchy shows 'testRemove' containing 'Address' (with sub-entities 'city', 'comment', 'id', 'street') and 'Customer' (with an association 'address (Address)').

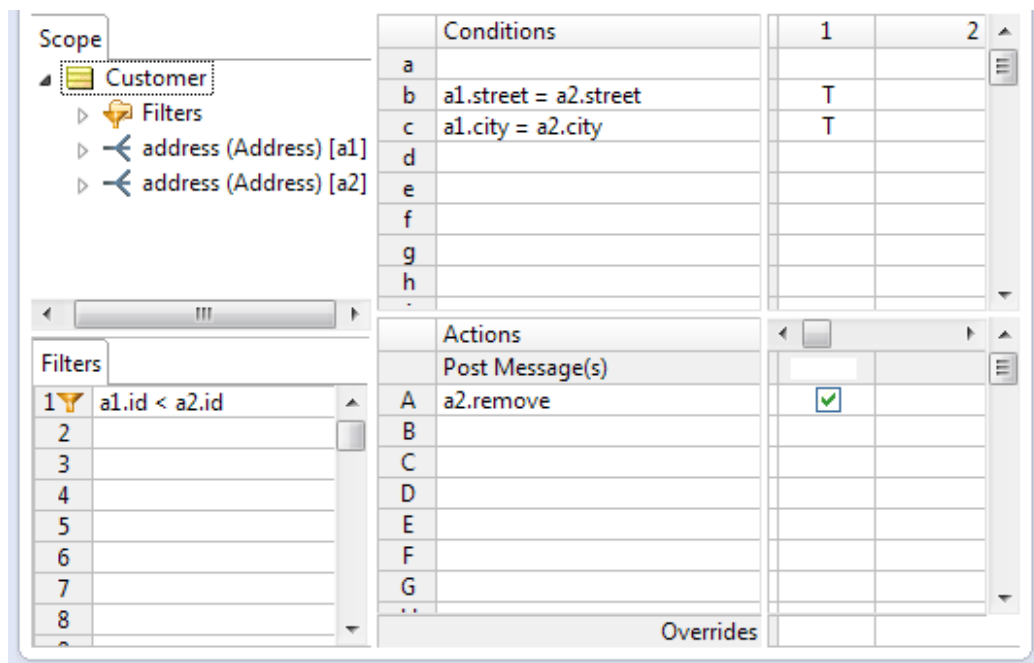
Using our created identifier attribute, we create a Rulesheet to identify each unique address. It uses two aliases to run through the addresses associated with a given customer. The actions initialize the `id`, and then add an incremented `id` value to each associated `Address` in memory:

The screenshot shows the Rulesheet editor. The 'Scope' is set to 'Customer' with two aliases: 'address (Address) [all]' and 'address (Address) [any]'. The 'Filters' section is empty. The 'Actions' section contains two actions:

Post Message(s)	Value
A any.id	0
B any.id = all.id -> max+1	<input checked="" type="checkbox"/>

The 'Overrides' section is empty.

Once each address has a unique identity, the second Rulesheet will do the removal action. It iterates through the associations to identify whether an association has a match, and -- if it does -- to remove the matching association from memory, as shown:



A Ruleflow puts the two Rulesheets into sequence, as shown:



A Ruletest that uses this Ruleflow as the test subject shows the "survivors" in its output:

Input	Output
<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> address (Address) [1] <ul style="list-style-type: none"> city [city1] street [street1] address (Address) [2] <ul style="list-style-type: none"> city [city1] street [street1] address (Address) [3] <ul style="list-style-type: none"> city [city2] street [street2] address (Address) [4] <ul style="list-style-type: none"> city [city1] street [street1] 	<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> address (Address) [3] <ul style="list-style-type: none"> city [city2] id [4] street [street2] address (Address) [4] <ul style="list-style-type: none"> city [city1] id [1] street [street1]

After this processing is done, subsequent Rulesheets in the Ruleflow will see only unduplicated addresses for each customer.

Note: Rule Statements were not requested for this process. As we are actually removing the duplicates during the execution of the rule, each removed address has been dropped from memory, and no longer has a meaningful reference when statement message is generated.

Flagging duplicate children

You might want to identify the duplicated records rather than delete them. To do so, just uncheck (or delete) the `.remove` action, and add an appropriate `.comment` value to the address. This examples uses, 'Duplicate', as shown:

The screenshot shows the Ruletest configuration interface. On the left, the **Scope** pane shows a hierarchy: **Customer** (expanded) containing **Filters**, **address (Address) [a1]**, and **address (Address) [a2]**. Below this, the **Filters** pane shows a list of filters, with the first filter being **1 a1.id < a2.id**. In the center, the **Conditions** pane shows a list of conditions: **a**, **b a1.street = a2.street**, **c a1.city = a2.city**, **d**, **e**, **f**, **g**, and **h**. On the right, the **Actions** pane shows a list of actions: **A a2.remove**, **B a2.comment 'Duplicate'**, **C**, **D**, **E**, **F**, and **G**. The **Post Message(s)** pane is also visible, showing a list of messages. The **Overrides** pane is at the bottom right.

When the same Ruletest runs, this time shows all the input records, with duplicated records displaying their comment value:

The screenshot shows the Ruletest output. On the left, the **Input** tree shows a hierarchy: **Customer [1]** (expanded) containing **address (Address) [1]**, **address (Address) [2]**, **address (Address) [3]**, and **address (Address) [4]**. Each address node contains **city** and **street** nodes. On the right, the **Output** tree shows a hierarchy: **Customer [1]** (expanded) containing **address (Address) [1]**, **address (Address) [2]**, **address (Address) [3]**, and **address (Address) [4]**. Each address node contains **city**, **comment**, **id**, and **street** nodes. The **comment** nodes for the duplicated records (address [1] and [2]) are labeled **Duplicate**.

Note: Again, Rule Statements were not used. There are in fact three duplicates – address 4 and address 1, address 4 and address 2, address 1 and address 2 – so three messages (referencing 1, 4, and 4) would be generated since all of the addresses are still in memory. Two get marked as duplicates, and one survives. In a subsequent Rulesheet, you could delete all addresses that have been flagged as 'Duplicate'.

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 162: Rulesheet and Ruletest, no threshold condition, CaPT disabled

Conditions		0	1
a	Thing.aSize	-	'small'
b			
c			
d			
e			
f			
Actions			
Post Message(s)			
A	Thing.selected		T

Input	Output
Thing [1] aSize [huge]	Thing [1] aSize [huge]
Thing [2] aSize [small]	Thing [2] aSize [small] selected [true]
Thing [3] aSize [small]	Thing [3] aSize [small] selected [true]

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 163: Rulesheet and Ruletest, threshold condition added, CaPT disabled

Conditions		0	1
a	Thing.selected	-	F
b	Thing.aSize	-	'small'
c			
d			
e			
f			
Actions			
Post Message(s)			
A	Thing.selected = false	<input checked="" type="checkbox"/>	<input type="checkbox"/>
B	Thing.selected		T

Input	Output
Thing [1] aSize [huge]	Thing [1] aSize [huge]
Thing [2] aSize [small]	selected [false] Thing [2] aSize [small]
Thing [3] aSize [small]	selected [true] Thing [3] aSize [small]
	selected [true]

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 tempted 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 happened 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 164: Rulesheet and Ruletest, threshold condition added, CaPT enabled

Conditions		0	1
a	Thing.selected	-	F
b	Thing.aSize	-	'small'
c			
d			
e			
f			
Actions			
Post Message(s)			
A	Thing.selected = false	<input checked="" type="checkbox"/>	<input type="checkbox"/>
B	Thing.selected		T

Input	Output
Thing [1] aSize [huge]	Thing [1] aSize [huge]
Thing [2] aSize [small]	selected [false] Thing [2] aSize [small]
Thing [3] aSize [small]	selected [true] Thing [3] aSize [small]
	selected [false]

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

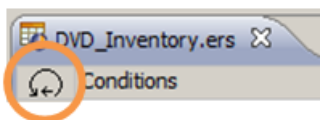
Note: Try this test, and then go to [Test yourself answers: Rule dependency: dependency and inferencing](#) on page 354 to correct yourself.

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.

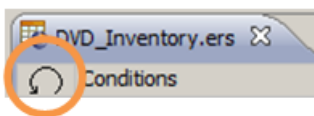
Referring to the following illustration, answer questions 6 through 8.

Conditions		0	1	2
a	DVD.priceTier		'High'	'Medium'
b	DVD.quantityAvailable		> 100000	-
c	DVD.releaseDate > today.addMonths(-6)		-	T
d				
Actions				
Post Message(s)				
A	DVD.priceTier		'Medium'	
B	DVD.quantityAvailable += 25000			<input checked="" type="checkbox"/>
Overrides				
Rule Statements		Rule Messages		
Ref	ID	Post	Alias	Text
1		Warning	DVD	If DVD price tier is High and > 100,000 copies are available, change price tier to medium to decrease inventory
2		Info	DVD	If DVD price tier is Medium and DVD < 6 months old, then increase inventory by 25,000 copies to meet expected demand

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 _____.

Filters and 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.

Filters can be set to be [Database filters](#) on page 203 when its entity is defined to persist in a datastore and the entity is set to extend to database.

Going forward, we 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.

For details, see the following topics:

- [What is a filter?](#)
- [What is a precondition?](#)
- [Using collection operators in a filter](#)
- [Filters that use OR](#)
- [Test yourself questions: Filters and preconditions](#)

What is a filter?

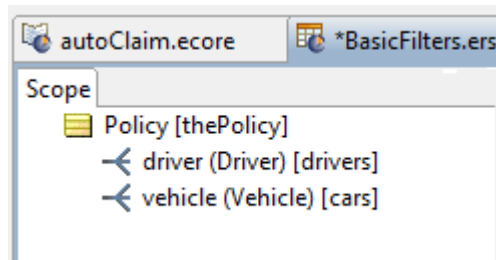
A Filter expression acts to limit or reduce the data in working memory to only that subset whose members satisfy the expression. A Filter *does not* permanently remove or delete any data; it simply *excludes* data from evaluation by other rules in the same Rulesheet.

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, regardless of its full behavior, is unaffected by Filter expressions in other Rulesheets.

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

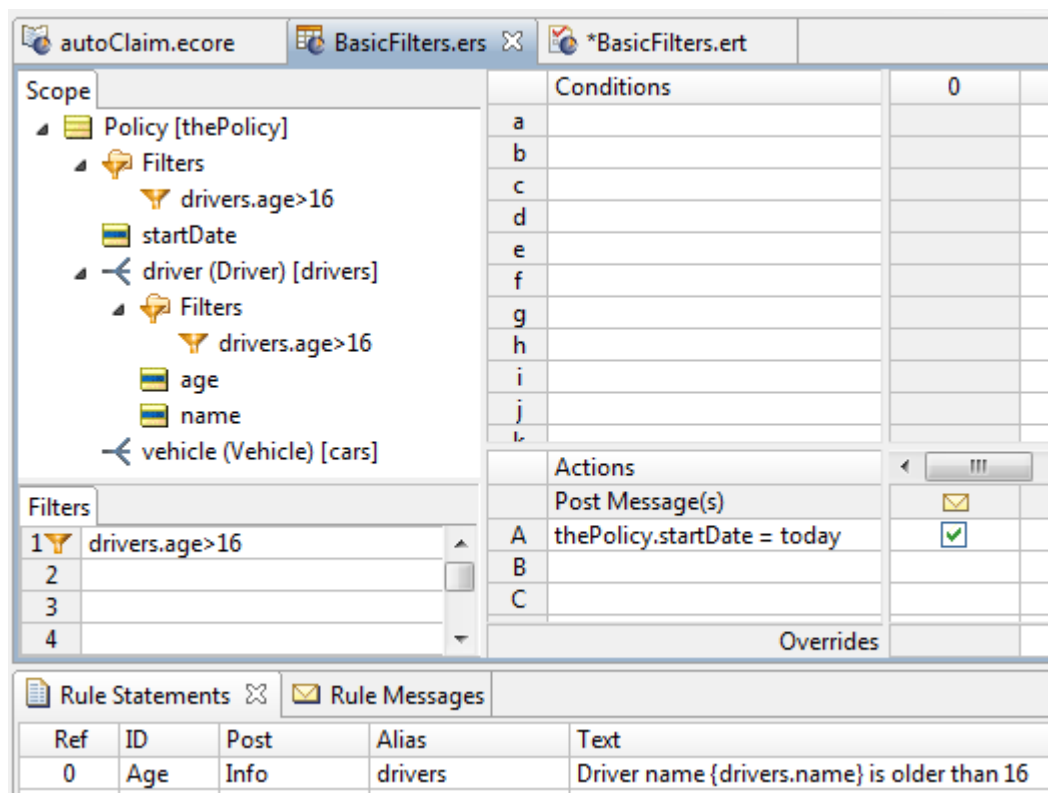
Figure 165: Aliases Declared



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 166: 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 167: Ruletest to test Filter Behavior

The screenshot shows the Ruletest interface with the following components:

- Input Panel:** Displays a tree structure for `Policy [1]` containing:
 - `startDate`
 - `driver (Driver) [1]` with `age [18]` and `name [Jacob]`
 - `driver (Driver) [2]` with `age [14]` and `name [John]`
 - `driver (Driver) [3]` with `age [21]` and `name [Lisa]`
- Output Panel:** Displays a tree structure for `Policy [1]` containing:
 - `startDate [03/25/13]`
 - `driver (Driver) [1]` with `age [18]` and `name [Jacob]`
 - `driver (Driver) [2]` with `age [14]` and `name [John]`
 - `driver (Driver) [3]` with `age [21]` and `name [Lisa]`
- Rule Messages Table:**

Severity	Message
Info	Driver name Lisa is older than 16
Info	Driver name Jacob is older than 16

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 is 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 168: Ruletest for Full Filter

Input	Output
<ul style="list-style-type: none"> Policy [1] <ul style="list-style-type: none"> startDate driver (Driver) [1] <ul style="list-style-type: none"> age [13] name [Jacob] driver (Driver) [2] <ul style="list-style-type: none"> age [14] name [John] driver (Driver) [3] <ul style="list-style-type: none"> age [10] name [Lisa] 	<ul style="list-style-type: none"> Policy [1] <ul style="list-style-type: none"> startDate driver (Driver) [1] driver (Driver) [2] driver (Driver) [3]

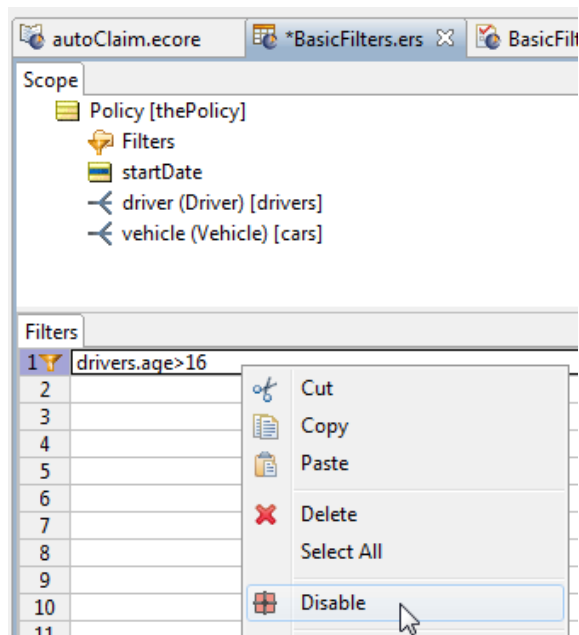
Notice two important things about this Ruletest's results: first, none of the `Driver` entities in the Input are older than 16, which means none of them survives the Filter. Second, because the parent `Policy` entity does not contain at least one `Driver` which satisfies the Filter, then the parent `Policy` 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.

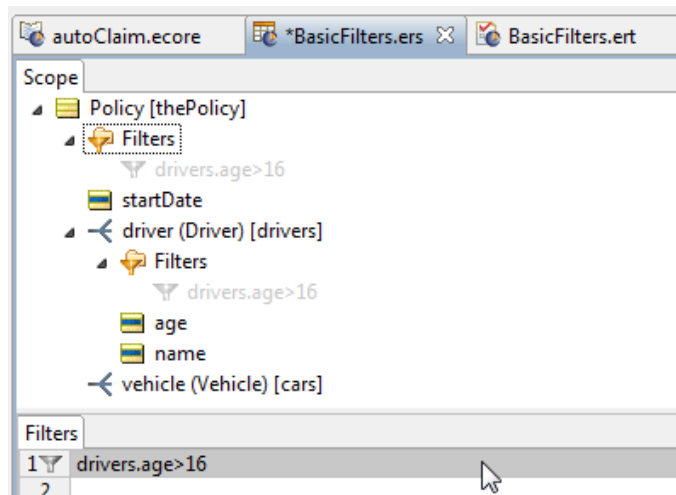
Note: Full filtering, or *maximum* filtering, is also the original behavior of Filters (and of their Precondition/Filter counterparts in prior releases of Corticon Studio), so for backward compatibility purposes with older models written with these expectations, we have kept it this way as new versions 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:



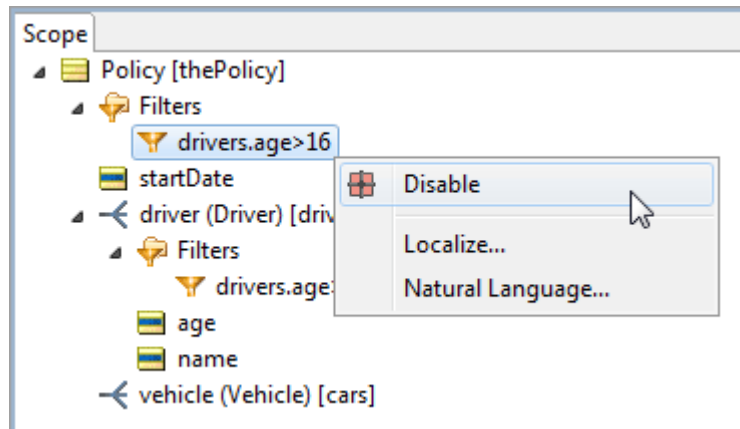
A disabled full filter is really no filter at all. You can perform the corresponding action to again **Enable** the filter.

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.

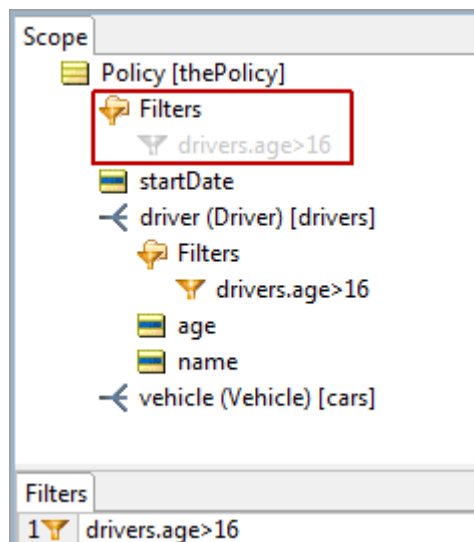
To turn a Filter expression into a limiting Filter, right-click on a Filter in the scope section and select **Disable** from the menu, as shown:

Figure 169: Selecting to limit a filter



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

Figure 170:



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 filter 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.

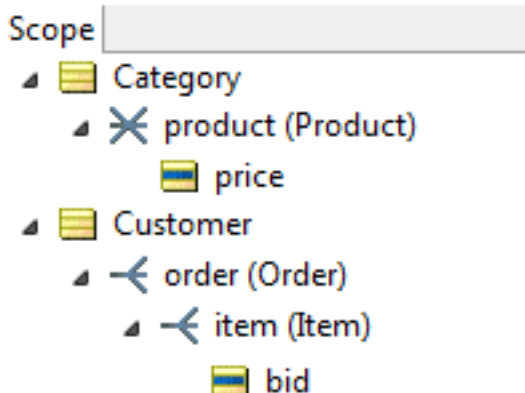
You see how one filter can apply limits to the full filter to achieve the preferred profile of what survives the filter and what gets filtered out.

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

Example of limiting filters

Consider the following Rulesheet Scope of a Vocabulary:

Figure 171: Scope in a Rulesheet that will be filtered

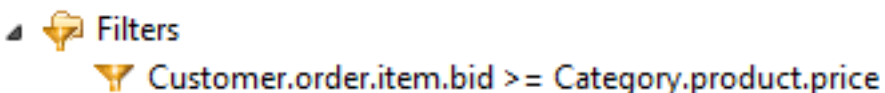


Consider the filter to be applied to data:

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

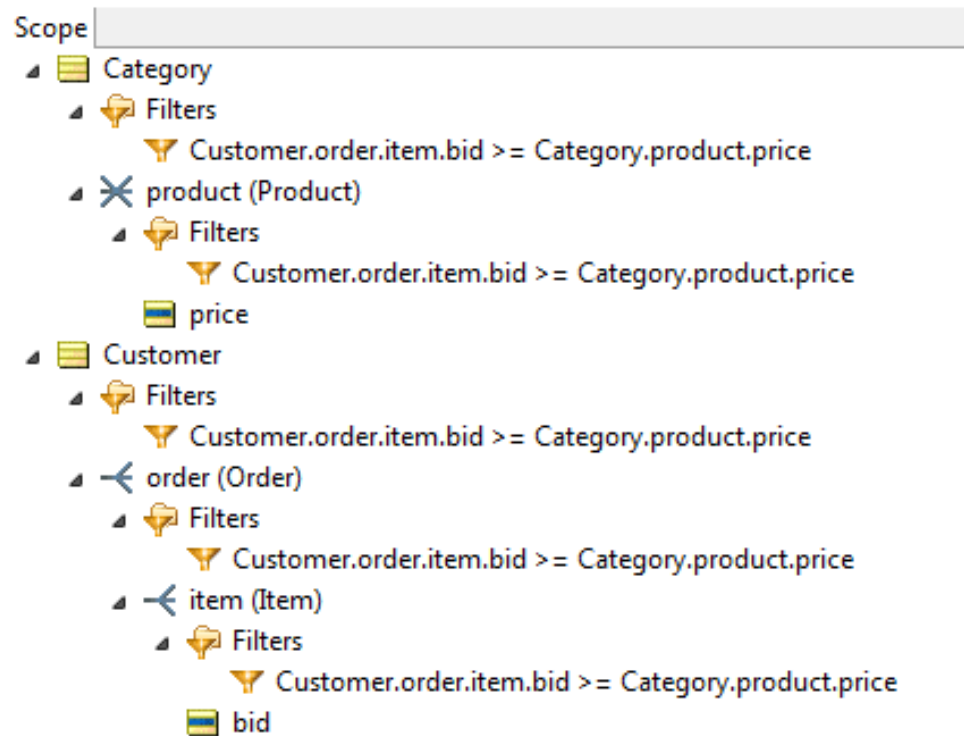
This is shown in the Rulesheet's **Filters** section as:

Figure 172: Definition of a filter

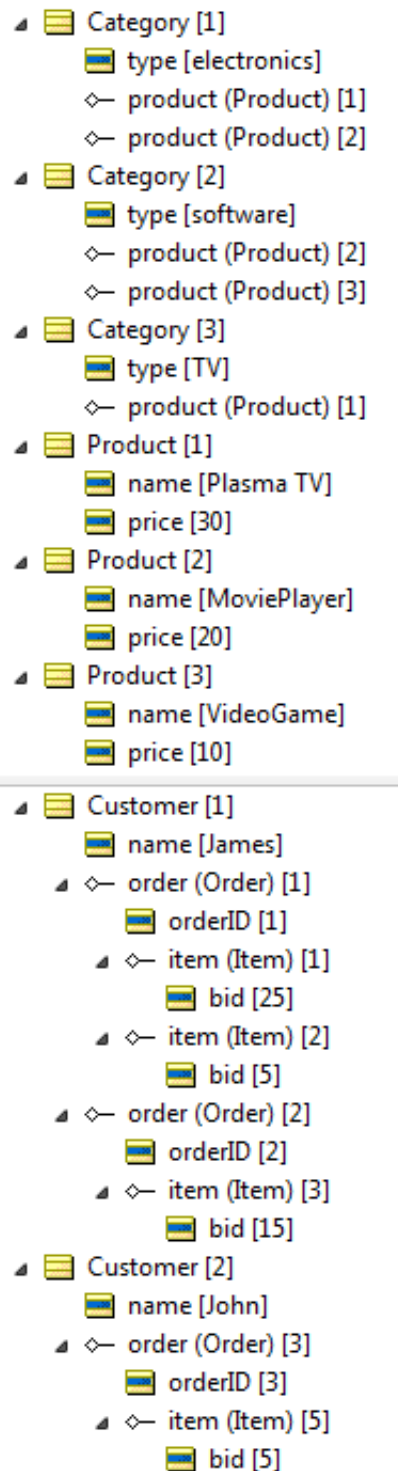


The resulting filter application applies at several levels, as shown:

Figure 173: Application of the filter to the Scope's tree structure



A Ruletest Testsheet might be created as follows:



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:

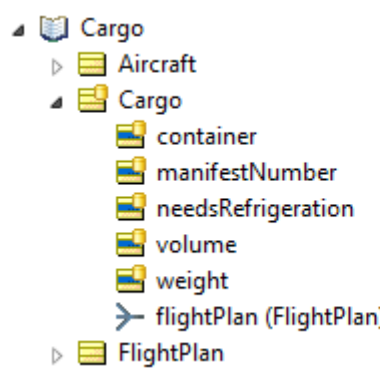
Notice how the filter is applied towards each discrete entity referenced in the expression:

- 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.

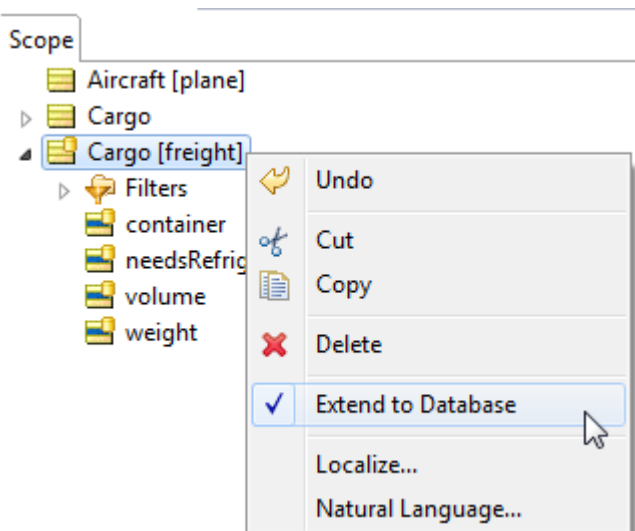
Database filters

When an entity has its **Datastore Persistent** property set to **Yes** in the Vocabulary, it places a database cylinder decoration on the entity and its attributes, as shown:

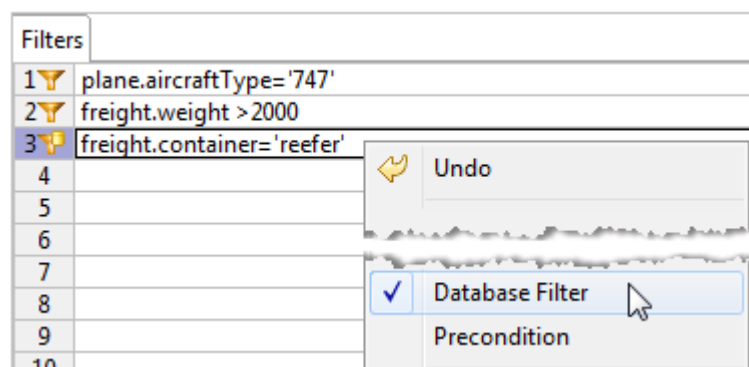


Property Name	Property Value
Entity Name	Cargo
Entity Identity	
Inherits From	
XML Namespace	
XML Element Name	Cargo
Java Package	
Java Class Name	
Datastore Persistent	Yes
Table Name	
Datastore Caching	No
Identity Strategy	

Once that is set, right-clicking on a Entity's alias in a Rulesheet's Scope section presents the menu command to **Extend to Database**, as shown:



Then you can define filters and set them each as a **Database Filter**, as shown:



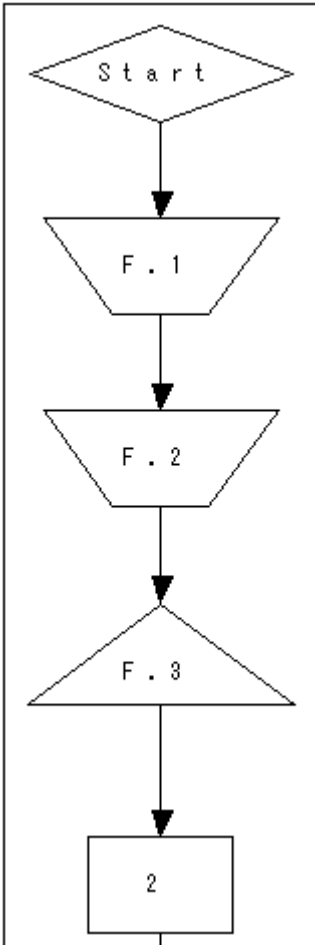
When checked, the filter becomes a *database query* that will retrieve data from the connected database, and then add the retrieved data to working memory.

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

Note: See [Precondition and Filters as query filters](#) on page 244 for qualifications and supported operators.

Database filters in an execution sequence diagram

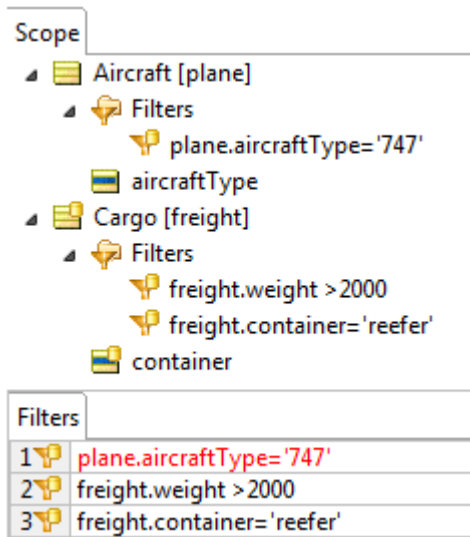
When you choose **Rulesheet > Logical Analysis > Execution Sequence Diagram**, the graphic that is generated distinguishes a database filter from local filter by its shape:



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.)

Error Conditions

It is important to note that you could set a database filter on an entity that is not Datastore Persistent or extended to database, but if you do so, the filter is marked in red, as shown. The error notes that the filter cannot be processed by a database.



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.

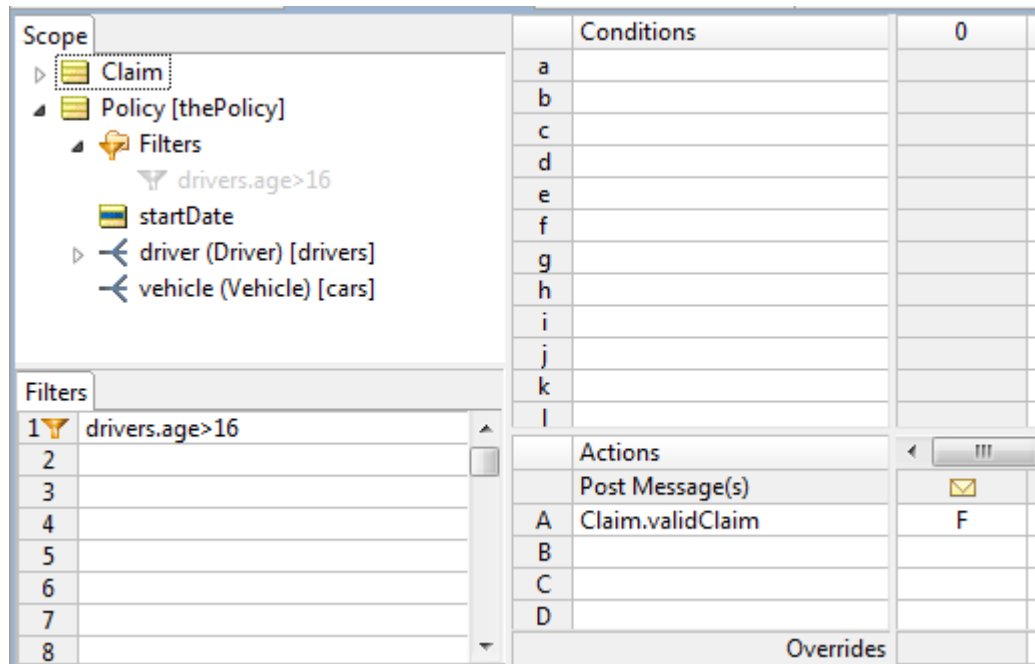
Precondition behavior of a Filter ensures that execution of a Rulesheet **stops** unless *at least one* piece of data survives the Filter. If execution of a Rulesheet stops because no data survived the Filter, then execution moves on to the next Rulesheet (in the case where the Rulesheet is part of a 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 174: 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 175: Rulesheet for an Action Unaffected by a Filter

Input	Output
<ul style="list-style-type: none"> Policy [1] <ul style="list-style-type: none"> driver (Driver) [1] <ul style="list-style-type: none"> age [13] name [Jacob] driver (Driver) [2] <ul style="list-style-type: none"> age [14] name [John] driver (Driver) [3] <ul style="list-style-type: none"> age [10] name [Lisa] Claim [1] <ul style="list-style-type: none"> validClaim 	<ul style="list-style-type: none"> Policy [1] <ul style="list-style-type: none"> driver (Driver) [1] <ul style="list-style-type: none"> age [13] name [Jacob] driver (Driver) [2] <ul style="list-style-type: none"> age [14] name [John] driver (Driver) [3] <ul style="list-style-type: none"> age [10] name [Lisa] Claim [1] <ul style="list-style-type: none"> validClaim [false]

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 176: Rulesheet for Precondition

Scope		Conditions	0
Claim		a	
Policy [thePolicy]		b	
Filters		c	
drivers.age>16		d	
startDate		e	
driver (Driver) [drivers]		f	
vehicle (Vehicle) [cars]		g	
		h	
		i	
		j	
		k	
		l	

Filters		Actions	
1	drivers.age>16	Post Message(s)	
2		A Claim.validClaim	F
3		B	
4		C	
5		D	
6			
7			
8			

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:

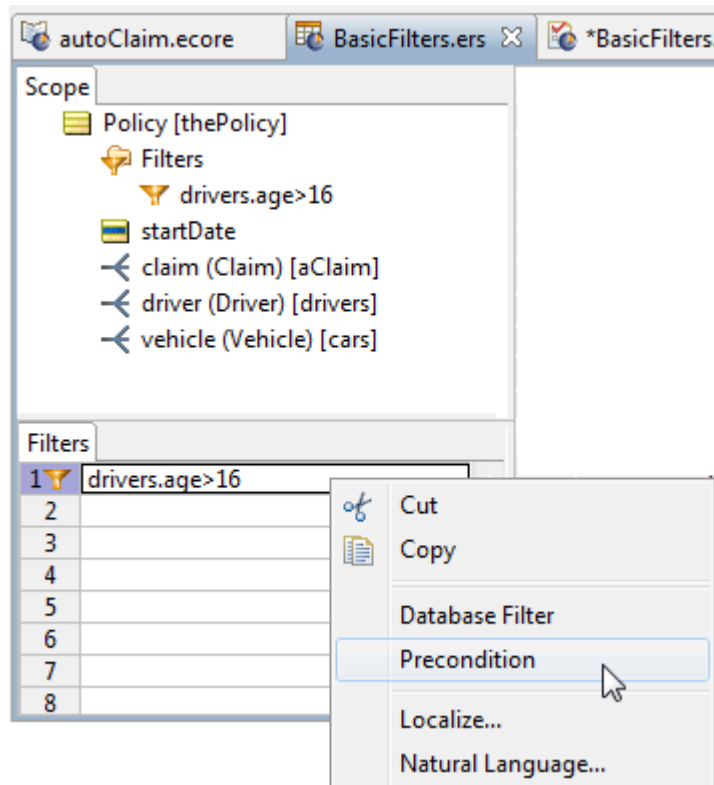
Figure 177: Ruletest for Precondition

Input	Output
Policy [1]	Policy [1]
driver (Driver) [1]	driver (Driver) [1]
age [13]	age [13]
name [Jacob]	name [Jacob]
driver (Driver) [2]	driver (Driver) [2]
age [14]	age [14]
name [John]	name [John]
driver (Driver) [3]	driver (Driver) [3]
age [10]	age [10]
name [Lisa]	name [Lisa]
Claim [1]	Claim [1]
validClaim	validClaim [false]

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



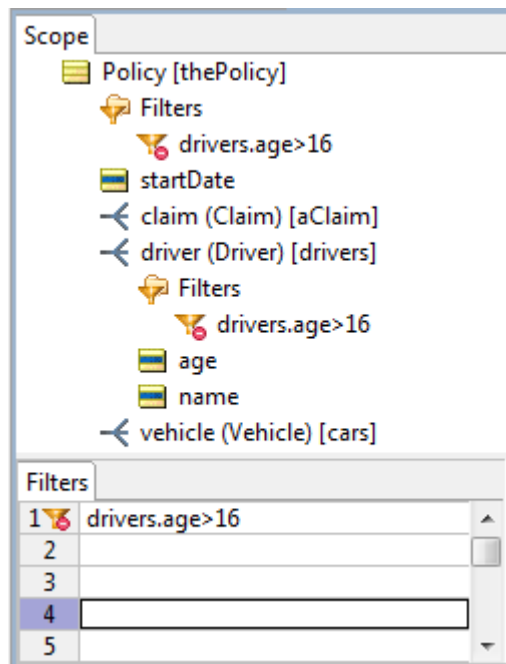
Note that the two options **Precondition** and **Limiting Filter** are mutually exclusive: turning one on turns the other off. A Filter cannot be both a Precondition AND a limiting Filter because at least one piece of data ALWAYS survives a limiting filter, so a Precondition would never stop execution.

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

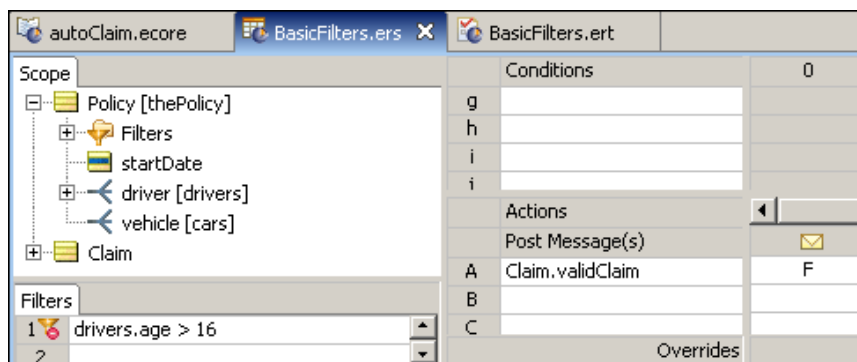
The following figure shows a Filter in **Precondition** mode.

Figure 179: A Filter in Precondition Mode



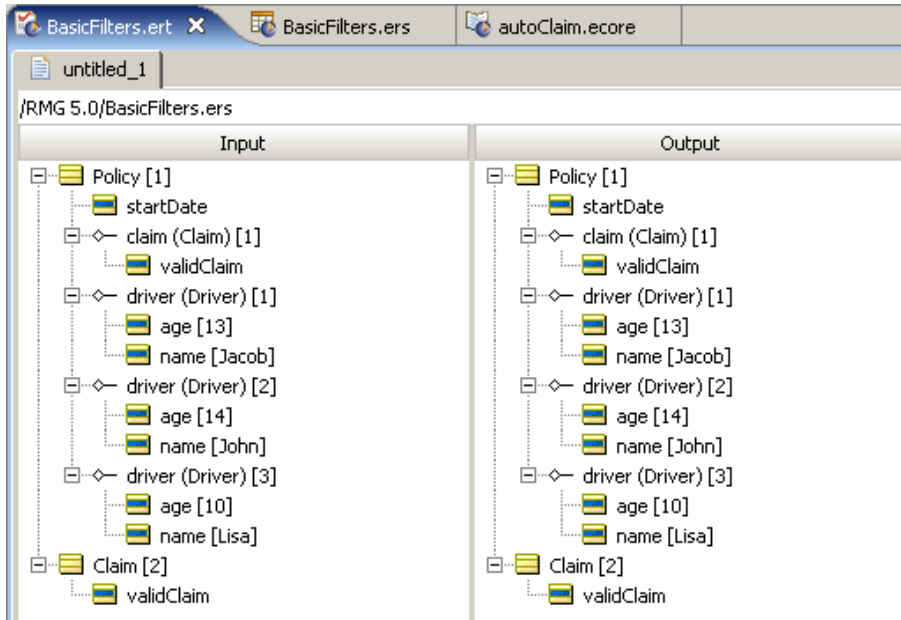
As described before, the precondition behavior of the Filter will cause Rulesheet execution to stop whenever no data survives the Filter. So in the original case where *Policy* and *Claim* were unassociated, a Filter in precondition mode, as shown:

Figure 180: 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 181: 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 has.

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 Rulesheets provides the same result:

Figure 182: A Condition/Action rule column with 2 Conditional rows

Scope		Conditions	0	1
Person [p]	a	p.skydiver		T
	b	p.age > 40		T
	c			
	r			

Filters		Actions	0	1
1	p.skydiver = T			
2				
3				
4				

Rule Statements		Rule Messages		
Ref	ID	Post	Alias	Text
1				A person over 40 who skydives is high risk

Figure 183: Rulesheet with one Condition row moved to Filters row

Scope		Conditions	0	1
Person [p]	a	p.age > 40		T
	b			
	c			
	r			

Filters		Actions	0	1
1	p.skydiver = T			
2				
3				
4				

Rule Statements		Rule Messages		
Ref	ID	Post	Alias	Text
1				A person over 40 who skydives is high risk

Figure 184: Rulesheet with Filter and Condition rows swapped

Scope		Conditions	0	1
Person [p]	a	p.skydiver		T
	b			
	c			
	r			

Filters		Actions	0	1
1	p.age > 40			
2				
3				
4				

Rule Statements		Rule Messages		
Ref	ID	Post	Alias	Text
1				A person over 40 who skydives is high risk

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: $4 \times 5 = 20$ and $5 \times 4 = 20$. Aliases work perfectly well in a declarative language (like Corticon'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 Rulesheets, notice that one of the conditional expressions uses the collection operator `->size`, and therefore must use an alias to represent the collection `Person`.

Figure 185: Collection Operator in Condition row

Scope		Conditions	0	1
+	Person [person]	a	person -> size > 3	-
		b		
		c		
		d		

Filters		Actions	
1	person.skydiver	Post Message(s)	
2		A	person.riskRating
3		B	high

Figure 186: Collection Operator in Filter row

Scope		Conditions	0	1
+	Person [person]	a	person.skydiver	-
		b		
		c		
		d		

Filters		Actions	
1	person -> size > 3	Post Message(s)	
2		A	person.riskRating
3		B	high

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

Figure 187: Ruletest with 3 Skydivers

Input	Output
Person [1] skydiver [true]	Person [1] skydiver [true]
Person [2] skydiver [true]	Person [2] skydiver [true]
Person [3] skydiver [false]	Person [3] skydiver [false]
Person [4] skydiver [true]	Person [4] skydiver [true]

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:

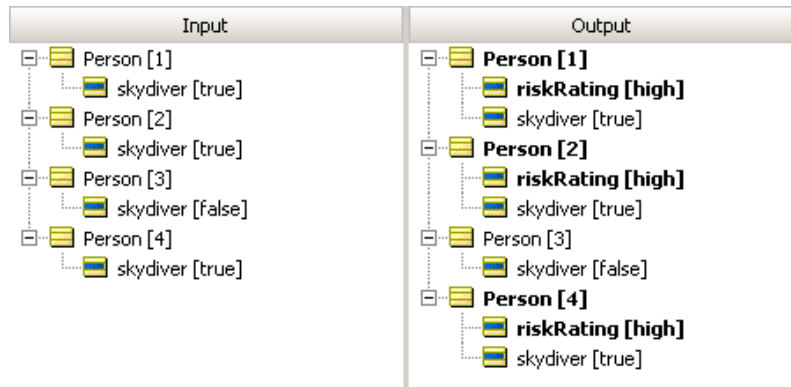
Figure 188: Ruletest with 4 Skydivers

Input	Output
Person [1] skydiver [true]	Person [1] riskRating [high] skydiver [true]
Person [2] skydiver [true]	Person [2] riskRating [high] skydiver [true]
Person [3] skydiver [false]	Person [3] skydiver [false]
Person [4] skydiver [true]	Person [4] riskRating [high] skydiver [true]
Person [5] skydiver [true]	Person [5] riskRating [high] skydiver [true]

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 189: 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 Rulesheets in [Collection Operator in Condition row](#) and [Collection Operator in Filter row](#) really implement two different business rules. When we built our Rulesheets, we neglected to write the plain-language business rule statements (violating our methodology!). The rule statements for these two Rulesheets 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 190: Rulesheet with 2 Conditions

Scope		Conditions		0	1
+	Person [person]	a	person -> size > 3		T
		b	person.gender = 'F'		T
		c			
		.			
Filters		Actions			
1	person.skydiver = true		Post Message(s)		
2		A	person.riskRating		'high'
3		-			
		Overrides			

Figure 191: Rulesheet with 2 Filters

Scope		Conditions		0	1
+	Person [person]	a	person -> size > 3		T
		b			
		c			
		.			
Filters		Actions			
1	person.skydiver = true		Post Message(s)		
2	person.gender = 'F'	A	person.riskRating		'high'
3		-			
		Overrides			

Once again, our Rulesheets 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 192: Ruletest

Input	Output
Person [1] gender [F] skydiver [true]	Person [1] gender [F] riskRating [high] skydiver [true]
Person [2] gender [M] skydiver [true]	Person [2] gender [M] skydiver [true]
Person [3] gender [F] skydiver [true]	Person [3] gender [F] riskRating [high] skydiver [true]
Person [4] gender [F] skydiver [true]	Person [4] gender [F] riskRating [high] skydiver [true]
Person [5] gender [M] skydiver [false]	Person [5] gender [M] skydiver [false]

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 Corticon 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 Corticon Server checking to make sure there are more than three 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 193: Ruletest

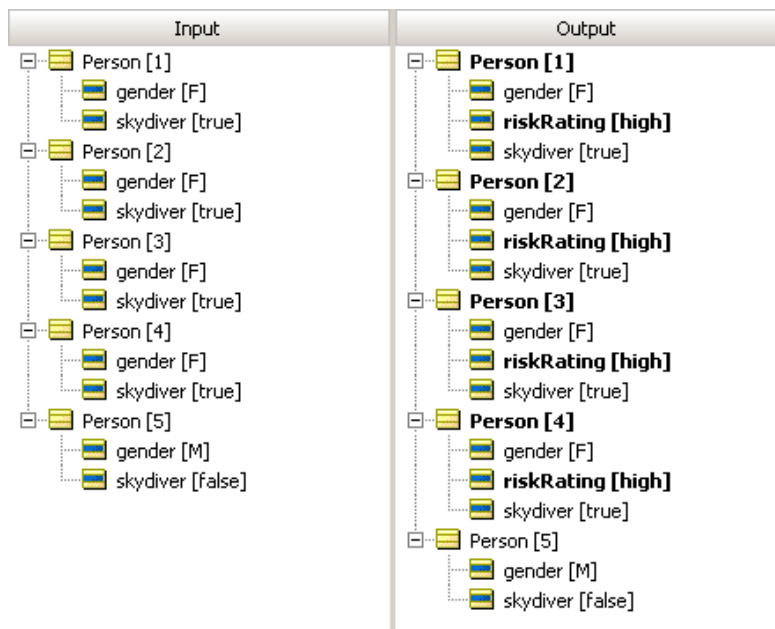
Input	Output
<div>Person [1]</div> <div>gender [F]</div> <div>skydiver [true]</div>	<div>Person [1]</div> <div>gender [F]</div> <div>skydiver [true]</div>
<div>Person [2]</div> <div>gender [M]</div> <div>skydiver [true]</div>	<div>Person [2]</div> <div>gender [M]</div> <div>skydiver [true]</div>
<div>Person [3]</div> <div>gender [F]</div> <div>skydiver [true]</div>	<div>Person [3]</div> <div>gender [F]</div> <div>skydiver [true]</div>
<div>Person [4]</div> <div>gender [F]</div> <div>skydiver [true]</div>	<div>Person [4]</div> <div>gender [F]</div> <div>skydiver [true]</div>
<div>Person [5]</div> <div>gender [M]</div> <div>skydiver [false]</div>	<div>Person [5]</div> <div>gender [M]</div> <div>skydiver [false]</div>

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 Corticon Server is executing this Rulesheet, let's modify our Ruletest and rerun:

Figure 194: 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 and preconditions

Note: Try this test, and then go to [Test yourself answers: Filters and preconditions](#) on page 355 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.

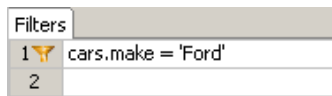
some	all	no	at least one
------	-----	----	--------------

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

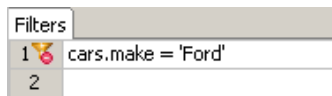
limiting	coffee	full	extreme
----------	--------	------	---------

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
<i>example: movies.studio = 'RKO'</i>	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			

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 kgs.
2. DC-10 aircraft must not exceed maximum cargo weight of 150,000 kgs.

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 kilograms) are different in each rule, the basic form of the rule is the same. In fact, we can generalize the rule as follows:

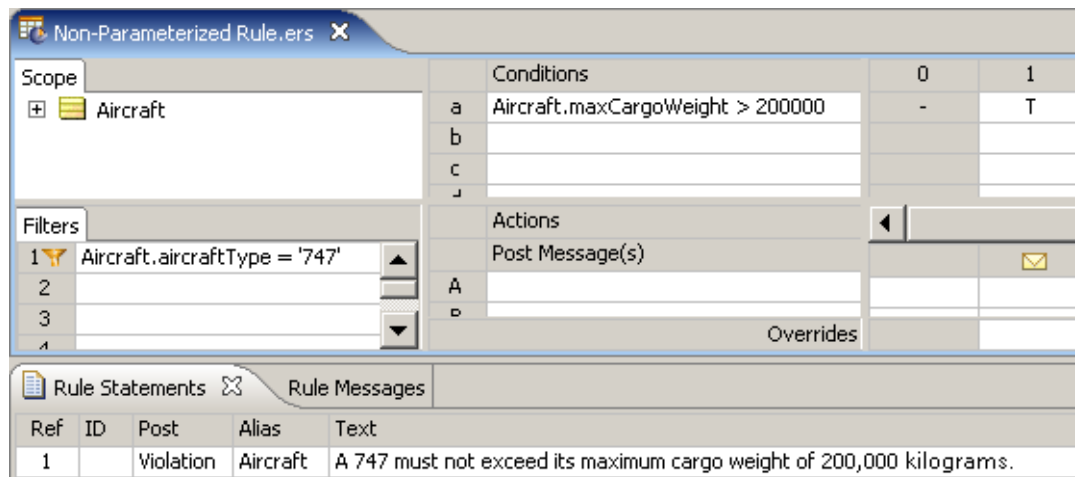
3. **X** aircraft must not exceed maximum cargo weight of **Y** kilograms.

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

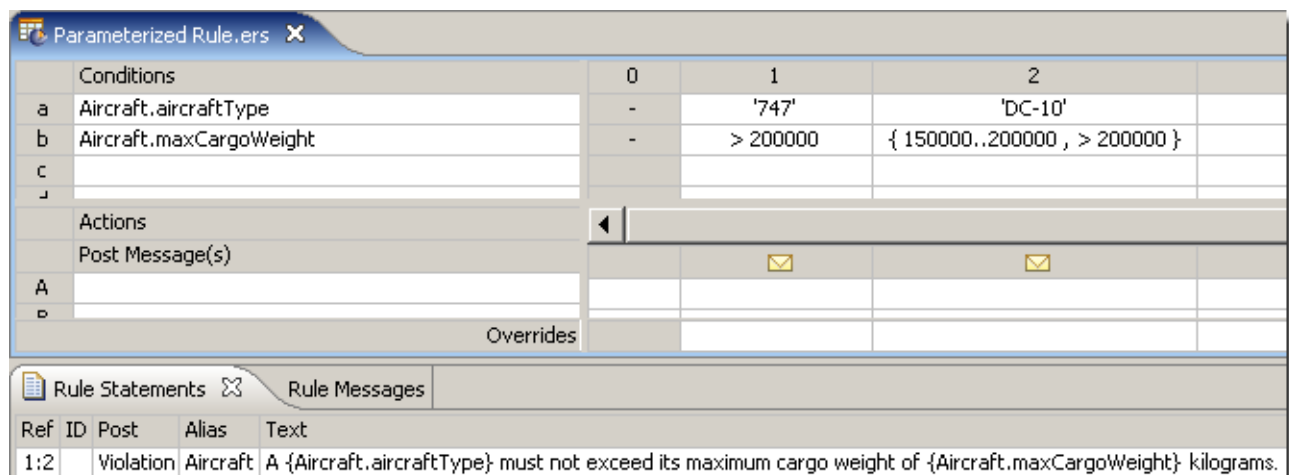
It is important to recognize these patterns because they can drastically simplify rule writing and maintenance in Corticon Studio. As shown in the following figure, we could build these two rules as a pair of Rulesheets, each with a Filter expression that filters data by `aircraftType`.

Figure 195: 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 196: 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 Corticon Studio 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:

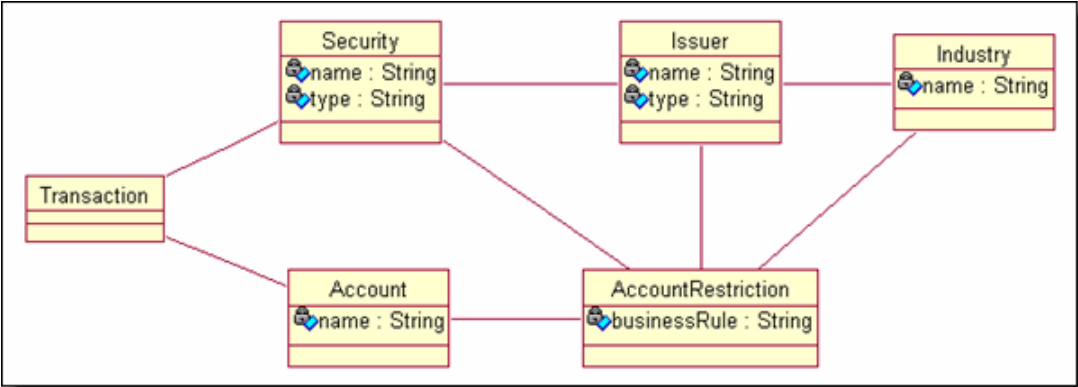
5. An **Account** may restrict holding a **type of Security** for a **specific reason**

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 197: 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

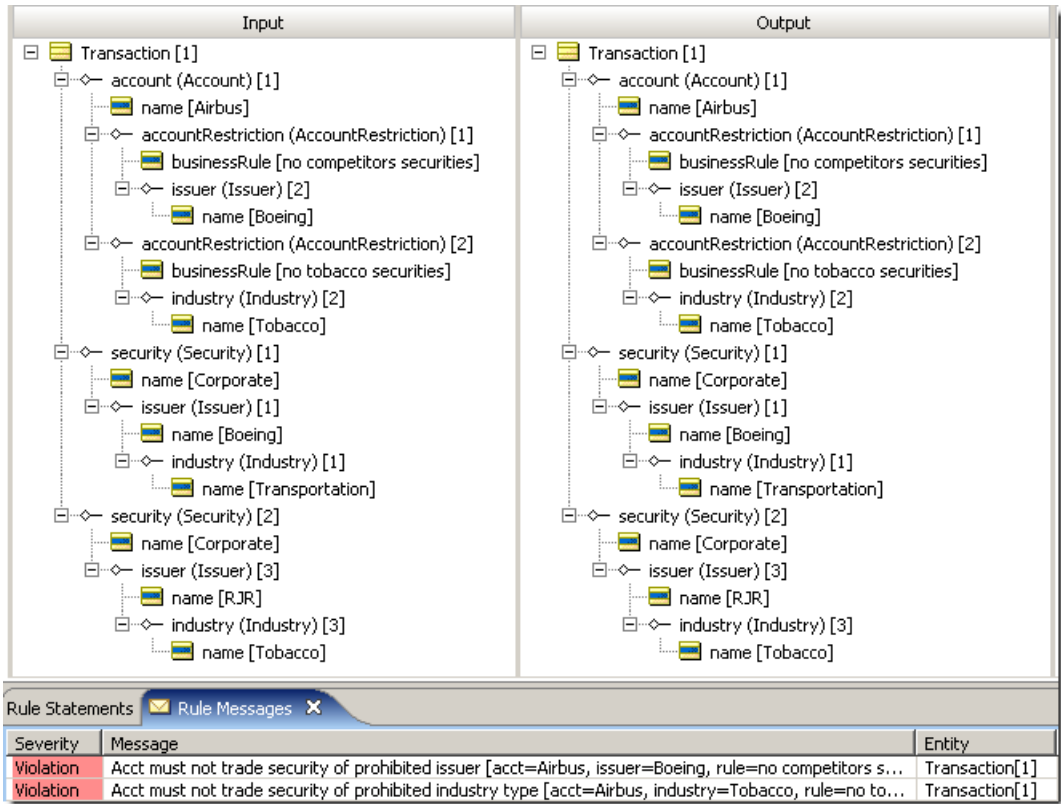
Figure 198: Parameterized Rule Example

The screenshot shows the Progress Corticon Rulesheet Editor. On the left, the 'Scope' tree shows a hierarchy: Transaction [tx] -> account [ac] -> Filters -> ac.name='Airbus' -> name -> accountRestriction [ar] -> security -> type. The 'Filters' section at the bottom shows two filters: 1. ac.name='Airbus' and 2. (empty). The 'Conditions' section has four conditions: a. tx.security.type = ar.security.type, b. tx.security.issuer.name = ar.issuer.name, c. tx.security.issuer.industry.name = ar.industry.name, and d. (empty). The 'Actions' section has a 'Post Message(s)' table with four rows: A, B, C, and D. Row A has a yellow envelope icon in column 1, and rows B, C, and D have yellow envelope icons in column 2. The 'Rule Statements' section at the bottom shows three statements: 1. Violation tx Acct must not trade security of prohibited type [acct={ac.name}, type={ar.security.type}, rule={ar.businessRule}], 2. Violation tx Acct must not trade security of prohibited issuer [acct={ac.name}, issuer={ar.issuer.name}, rule={ar.businessRule}], and 3. Violation tx Acct must not trade security of prohibited industry type [acct={ac.name}, industry={ar.industry.name}, rule={ar.businessRule}].

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 199: 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 200: 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 [Test yourself answers: Recognizing and modeling parameterized rules](#) on page 356 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?

Writing Rules to access external data

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

Note:

Online tutorials for EDC that use Microsoft SQL Server 2014 as the EDC database:

- [Modeling Progress Corticon Rules to Access a Database using EDC](#) is aimed at rule modelers. It covers how to model and test rules that read/write to a relational database
- [Connecting a Progress Corticon Decision Service to a Database using EDC](#) is aimed at integration developers. It covers how to set up the Vocabulary to connect and map to a database, configure EDC settings in the Web Console, and other deployment-related tasks.

Documentation topics on EDC:

- *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.
 - *"Packaging and deploying Decision Services"* in the *Integration and Deployment Guide* describes the Decision Service parameters for EDC and the associated deployment tools.
 - *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 is always enabled in Studio; however, Corticon Servers need explicit licensing for EDC, even in evaluation installations. Contact Progress Software technical support or your Progress Software representative for additional details on licensing and RDBMS support.

Note: When using the Vocabulary editor, setting it to show details exposes properties related to database interaction. When you enter data in properties when showing details and then switch to hide those properties, the properties are just hidden, not changed or cleared.

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](#)
- [Showing vocabulary details in a Rulesheet](#)
- [Clearing database mappings in the vocabulary](#)
- [Validation of database properties](#)
- [Enabling Database Access for Rules using root-level Entities](#)
- [Precondition and Filters as query filters](#)
- [Defining database caching in Studio](#)
- [Test yourself questions: Writing rules to access external data](#)

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. 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 – bring every instance of the entity into scope. This means that a rule using root-level `FlightPlan` acts on every instance of `FlightPlan`, including `Cargo.flightPlan`, `Aircraft.flightPlan`, or any other role using `FlightPlan` that may exist in our Vocabulary.

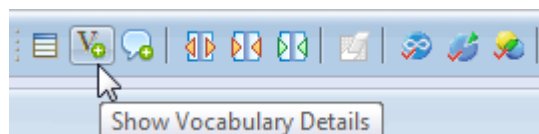
When we add the ability for the Corticon Server and Studio to dynamically retrieve data from a database, rule scope determines which data to retrieve. This is exactly the same concept as Studio determining which data in an Input Ruletest to process and which to ignore based upon a rule's scope. So if 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.

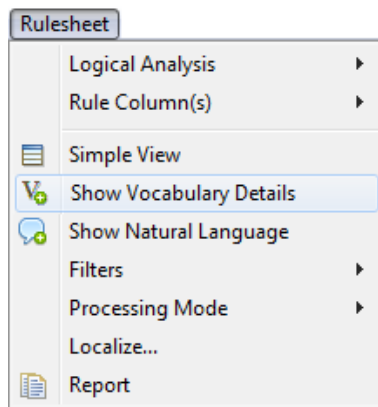
Showing vocabulary details in a Rulesheet

What's new

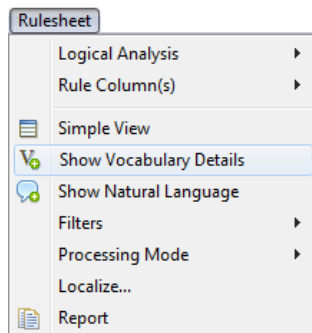
When working on a Rulesheet with the Vocabulary view active, the details of the Vocabulary hover help information might provide too much or too little information. While you could open the Vocabulary in its editor just to toggle the information to show or hide details, Rulesheets provide an additional toggle to change the hover help view in the Vocabulary view, as shown in the toolbar:



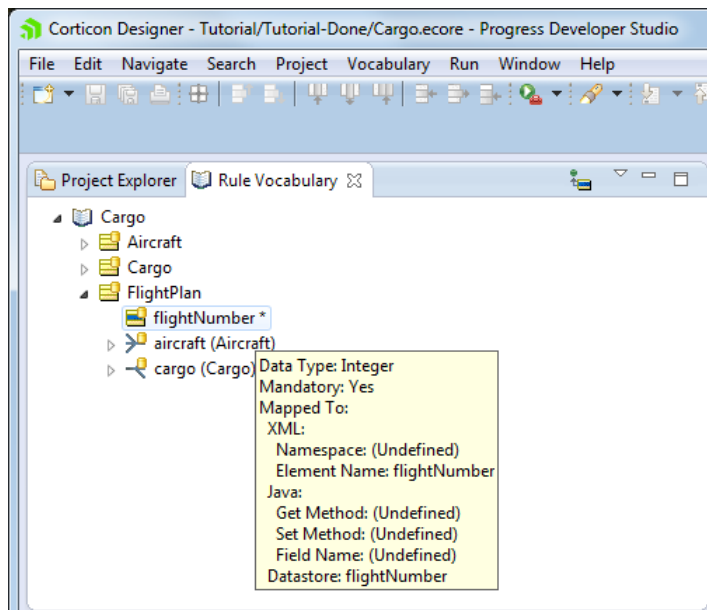
The menu items also offer the opportunity to toggle to the alternate view:



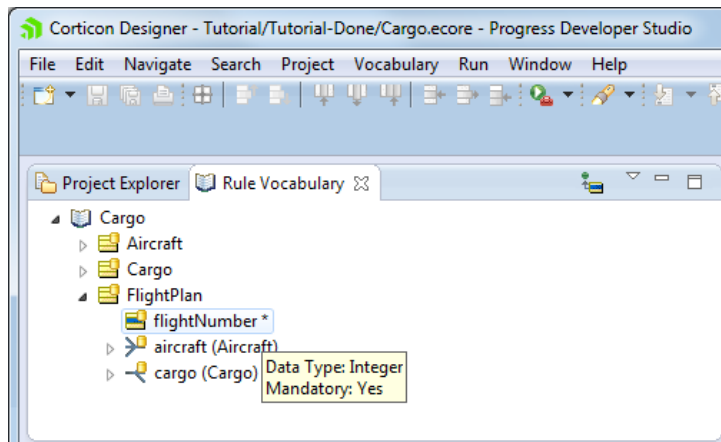
The menu items also offer the opportunity to toggle between showing and hiding details:



The following illustration shows the Rule Vocabulary showing its details. The Rulesheet is the currently active menu/toolbar, and its Vocabulary details button displays a minus sign -- it will toggle to hiding the details:



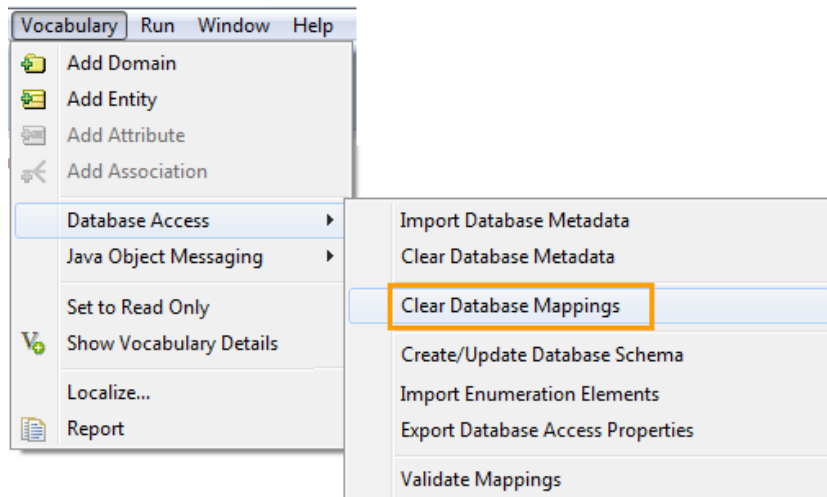
When switched to hiding details, the Vocabulary details button displays a plus sign -- it will toggle to showing the details:



Clearing database mappings in the vocabulary

Imported database metadata and manual adjustments can be cleared with just one command.

In the Vocabulary editor, select the menu command **Vocabulary > Database Access > Clear Database Mappings**, as shown:



The highlighted Entity properties are cleared:

*Cargo.ecore	Property Name	Property Value
<ul style="list-style-type: none"> Cargo <ul style="list-style-type: none"> Aircraft Cargo <ul style="list-style-type: none"> FlightPlan <ul style="list-style-type: none"> flightNumber * aircraft (Aircraft) cargo (Cargo) 	Entity Name	FlightPlan
	Entity Identity	flightNumber
	Inherits From	
	XML Namespace	
	XML Element Name	FlightPlan
	Java Package	
	Java Class Name	
	Datastore Persistent	Yes
	Table Name	Freight.dbo.FlightPlan
	Datastore Caching	
	Identity Strategy	
	Identity Column Name	
	Identity Sequence	
	Identity Table Name	
	Identity Table Name Colu...	
	Identity Table Value Colu...	
	Version Strategy	
	Version Column Name	

The highlighted Attribute properties are cleared:

*Cargo.ecore	Property Name	Property Value
<ul style="list-style-type: none"> Cargo <ul style="list-style-type: none"> Aircraft Cargo <ul style="list-style-type: none"> FlightPlan <ul style="list-style-type: none"> flightNumber * aircraft (Aircraft) cargo (Cargo) 	Attribute Name	flightNumber
	Data Type	Integer
	Mandatory	Yes
	Mode	Base
	XML Namespace	
	XML Element Name	flightNumber
	Java Object Get Method	
	Java Object Set Method	
	Java Object Field Name	
	Column Name	flightNumber
	Value Strategy	
	Value Sequence	
	Value Table Name	
	Value Table Name Colum...	
	Value Table Value Colum...	

The highlighted Association properties are cleared:

*Cargo.ecore	Property Name	Property Value
<ul style="list-style-type: none"> Cargo <ul style="list-style-type: none"> Aircraft Cargo <ul style="list-style-type: none"> FlightPlan <ul style="list-style-type: none"> flightNumber * aircraft (Aircraft) cargo (Cargo) 	Association Role Name	cargo
	Source Entity Name	FlightPlan
	Target Entity Name	Cargo
	Cardinalities	1->*
	Navigability	Bidirectional
	Mandatory	Yes
	XML Property Name	cargo
	Java Object Get Method	
	Java Object Set Method	
	Java Object Field Name	
	Join Expression	Freight.dbo.FlightPlan.flightNumber=Freight.dbo.Cargo.RflightP...

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.

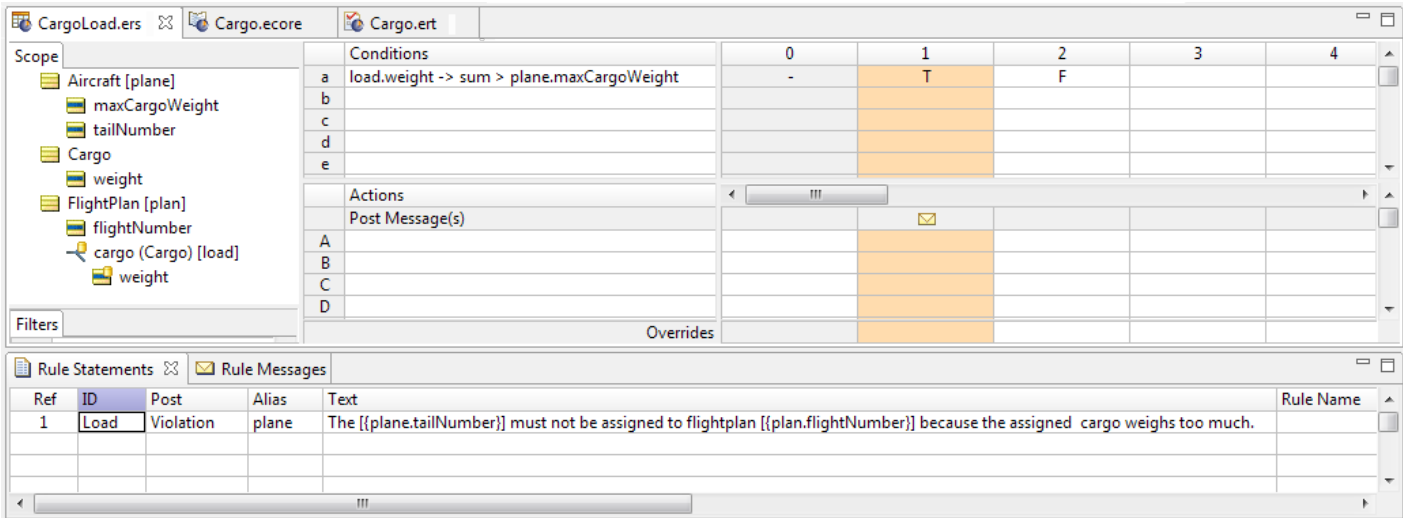
Note: Deployment-time validation check can be turned off in a server property file in (`CcServer.properties`) to "force" deployment despite mapping errors if circumstances warrant it.

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 Cargo project's Vocabulary, `Cargo.ecore`.

The process of connecting this particular Vocabulary to an external database is discussed in detail in the EDC tutorials. 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 201: Sample Rulesheet

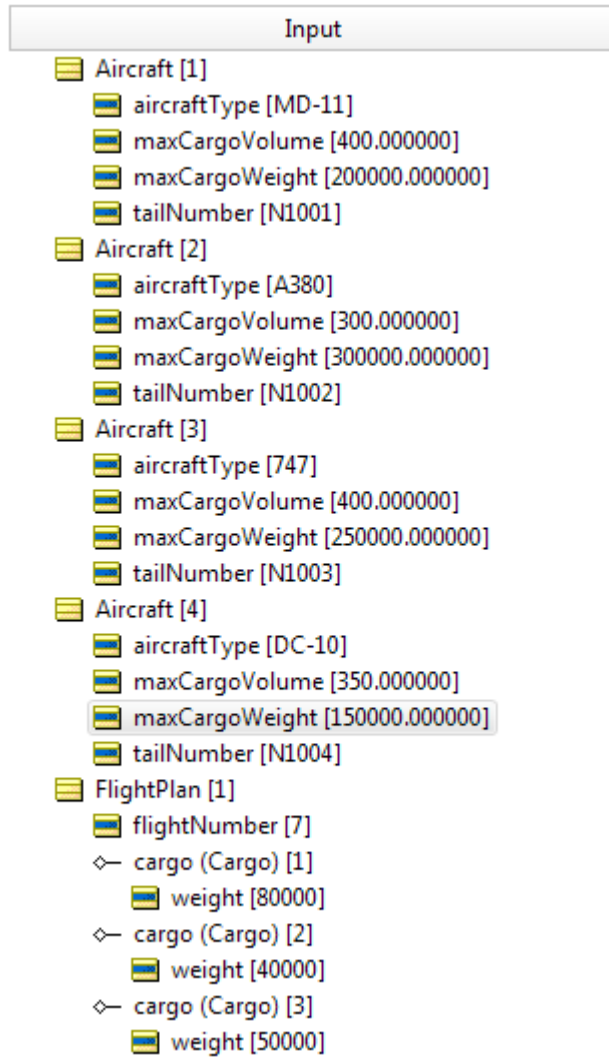


The Rulesheet shown above simply adds up (sums) the collection (you may want to review the Collections chapter for a refresher on this operator or collections in general) of Cargo weights associated with a FlightPlan (load.weight) and compares this to the maxCargoWeight of the root-level Aircraft. Our 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 202: 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 kilograms, and the `Aircraft` with `tailNumber` N1004 can only carry 150000 kilograms, we receive a `Violation` message for that `Aircraft` and that `Aircraft` alone. All other `Aircraft` have `maxCargoWeight` values of 200000 kilograms or more, so they fail to fire the rule.

Figure 203: Ruletest Violation Message

Rule Messages	
Severity	Message
Violation	[N1004] must not be assigned to flightplan [7] because the assigned cargo weighs too much.

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,
```

```

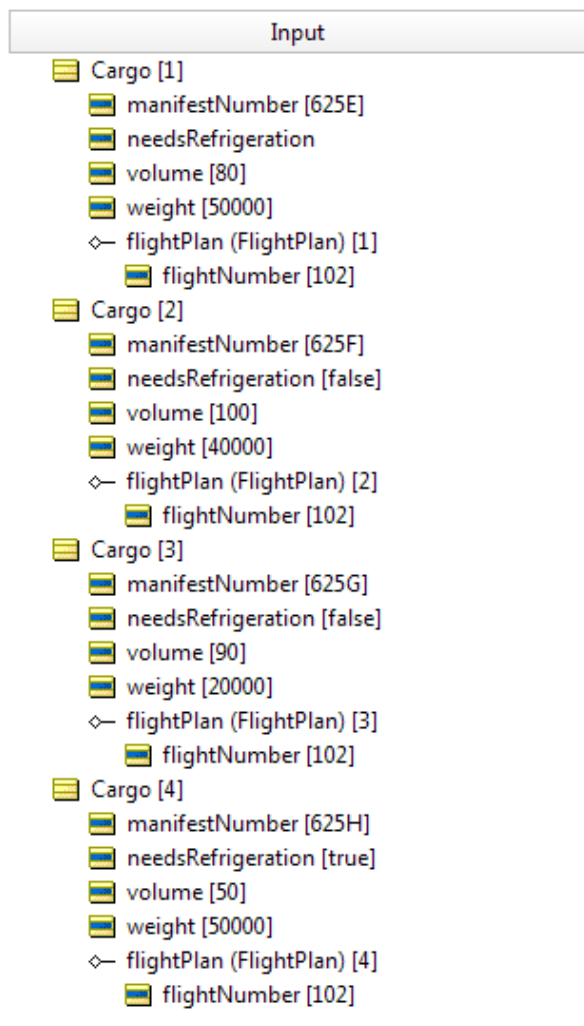
        needsRefrigeration,container,volume,weight)
VALUES ('625E',102,null,null,80,50000);
INSERT INTO Cargo.dbo.Cargo
    (manifestNumber,RflightPlanAssoc_flightNumber,
    needsRefrigeration,container,volume,weight)
VALUES ('625F',102,0,null,100,40000);
INSERT INTO Cargo.dbo.Cargo
    (manifestNumber,RflightPlanAssoc_flightNumber,
    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 204: 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 205: Cargo Table from Database

	manifestNumber	container	needsRefrigeration	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 `CargoLoaders` 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 206: Ruletest of FlightPlan Seed Data

Input	Output
FlightPlan [1] flightNumber [102]	FlightPlan [1] flightNumber [102]

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.

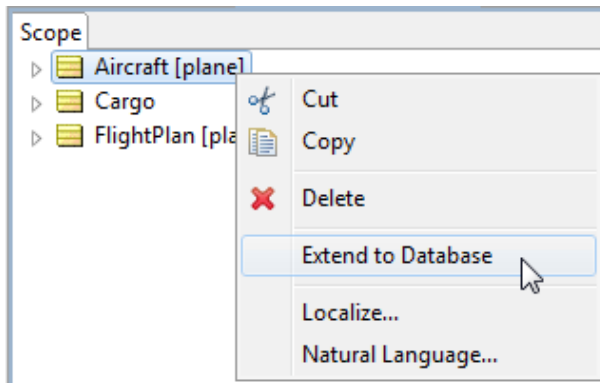
Let's set the Ruletest to read data from the database and return everything that it finds. Toggle the menu options in the **Ruletest > Testsheet** menu as shown:

Database Access	<input type="checkbox"/> None <input checked="" type="checkbox"/> Read Only Read/Update <input checked="" type="checkbox"/> Return All Entity Instances Return Incoming/New Entity Instances Only
-----------------	---

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

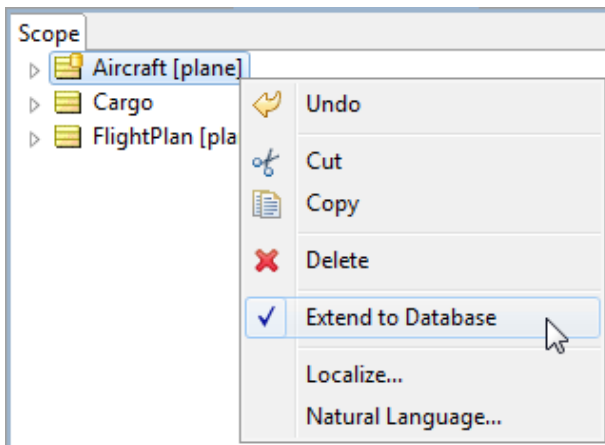
Extend to Database

What we want to do now is set the Rulesheet to **Extend to Database** (provided the entity is set to **Datastore Persistent** in the Vocabulary), 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:



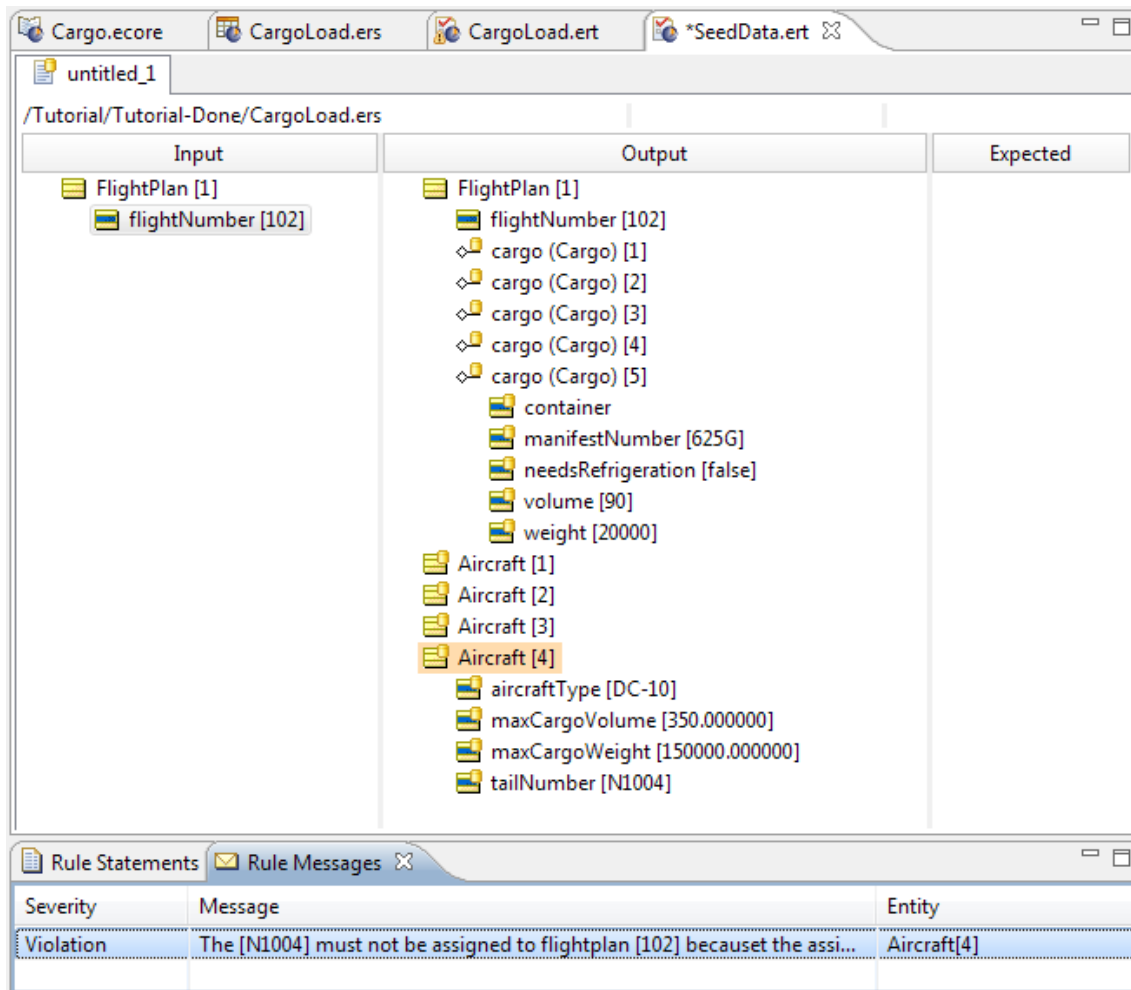
Once selected, the option shows a checkmark, and the `Aircraft` icon in the **Scope** is decorated with a database icon, as shown:

Figure 207: 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:

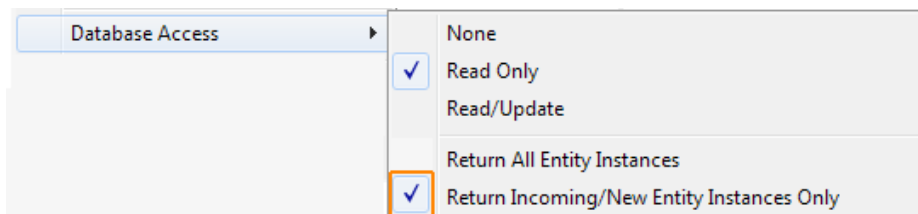
Figure 208: Results Ruletest Showing a Successful Extend-to-Database Retrieval



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:



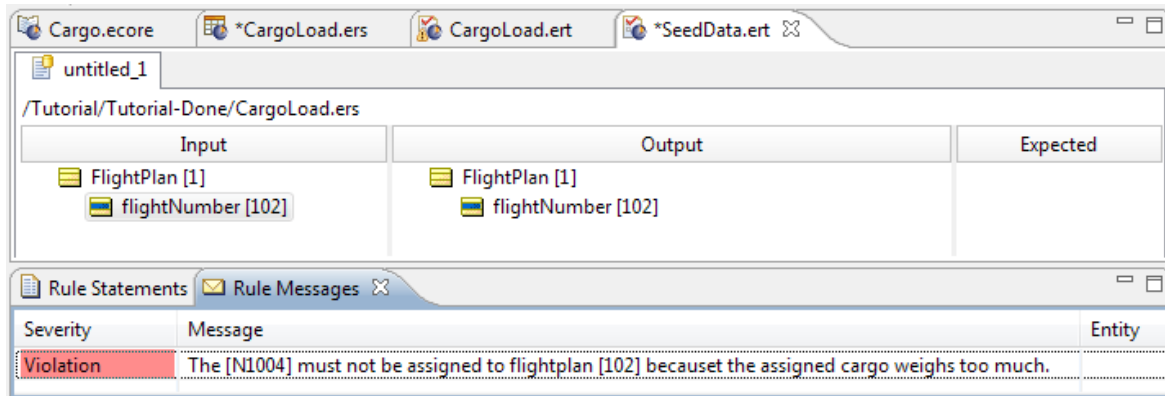
The data that returns is drawn only from those entities that were:

- 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.

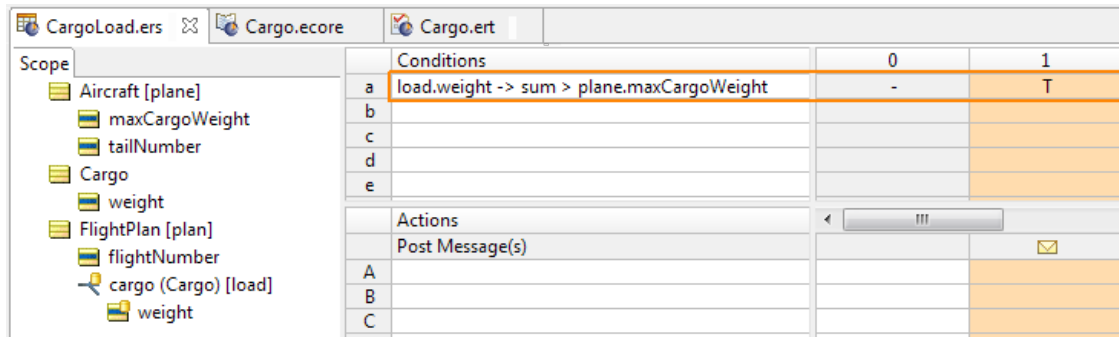
When you run the Ruletest now, the output is unchanged yet we got the Violation message as to which plane cannot be assigned that flight plan.



That result is concise, providing what could be all we really wanted to know in this test.

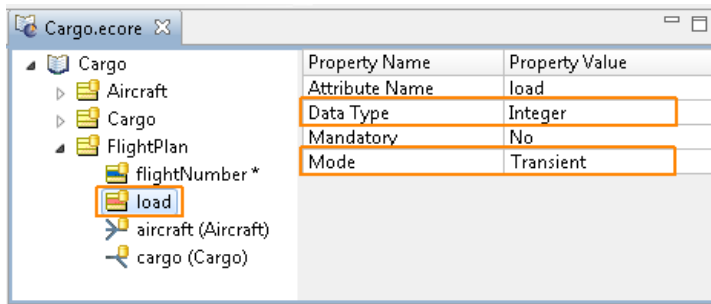
Optimizing Aggregations that Extend to Database

Our rulesheet used a condition statement that did a calculation and a difference, calling a statement when it evaluated as `true`, as shown:

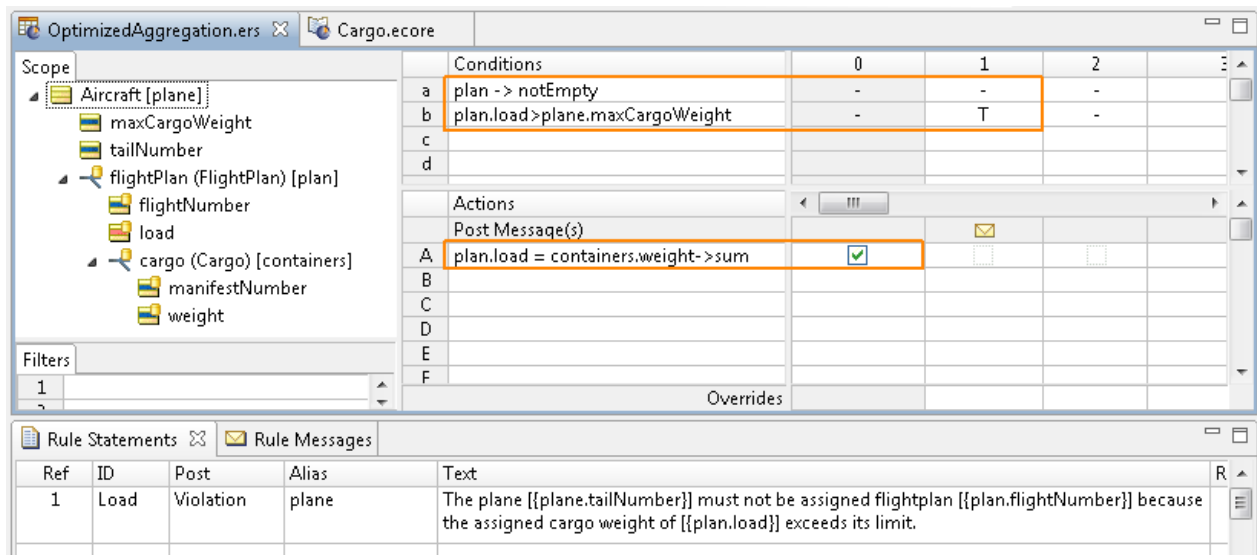


As written, `load.weight ->sum > plane.maxCargoWeight`, the condition will copy all the relevant cargo records into Corticon's memory to perform its `sum`, and then evaluate whether total weight is greater than the plane's capacity. As we are extended to database, the number of values could be large. 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 `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 150 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 has relational operators with Boolean operands.
4. If it uses an operator not supported by databases (see list below)
5. 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 8: 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
Less Than or Equal To	<=	DateTime, Decimal, Integer, String
Greater Than or Equal To	>=	DateTime, Decimal, Integer, String
Absolute Value	.absval	Decimal, Integer

Operator Name	Operator Syntax	Datatypes Supported
Character Count	<code>.size</code>	String
Convert to Upper Case	<code>.toUpper</code>	String
Convert to Lower Case	<code>.toLowerCase</code>	String
Substring	<code>.substring</code>	String
Equal To (comparison)	<code>.equals</code>	String
Collection is Empty	<code>->isEmpty</code>	Collection
Collection is not Empty	<code>->notEmpty</code>	Collection
Size of Collection	<code>->size</code>	Collection
Sum	<code>->sum</code>	Collection
Average	<code>->avg</code>	Collection
Minimum	<code>->min</code>	Collection
Maximum	<code>->max</code>	Collection
Exists	<code>->exists</code>	-

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
```

Defining database caching in Studio

As you move Decision Services that use a database connection toward production, you might want to get the performance benefits of caching entities and queries used in Decision Services. Even though the first cache use has some performance overhead, subsequent references that use that cached content will see excellent performance.

There are two ways caching can be used in Corticon rules:

An *Entity cache* is appropriate when common database-enabled entities are used in the input messages sent to a Decision Service.

- **Case for using entity caching:** Consider a Decision Service that expedites shipping requests. The Decision Service might receive a Order entity that has a one-to-many association with Supplier entities. When the Decision Service receives an Order entity it would query the database to get the associated Supplier entities. When using an entity cache, the Suppliers could be queried once and used in expediting other Order entities.

A *Query cache* optimizes lookup (query) of database data, such as when a cached entity is extended to the database in a Rulesheet. A Query Cache is read-only -- a lock is put on the records(returned from the query) associated with that decision service. If an update is attempted on a entity contained in a query cache an exception occurs. A Query Cache is **optimistic** -- that means that updates from outside of the Decision Service will not modify or invalidate the cache contents.

- Case for using query caching: Consider a Decision Service that prices online orders. With a query cache, it could query state sales tax rates once and use these when calculating the price of all orders it receives.
- Case for using query caching: Consider a Decision Service that prices insurance policies. With a query cache, it could query an actuarial table once and use the results in pricing multiple insurance policies.

First, you specify the caching settings in the Vocabulary and Rulesheets, and then enable caching in tests and deployed Decision Services.

Specifying Entity caching

To use entity caching:

1. Identify the Vocabulary entities that you want cached.
2. Edit the Vocabulary, and then choose **Show Vocabulary Details**.
3. Confirm (or set) the entity's **Datastore Persistent** property to **Yes**
4. Choose the preferred **Datastore Caching** value:
 - `No Cache` or blank (default) - Disable caching.
 - `Read Only` - Caches data that is never updated.
 - `Read/Write` - Caches data that is sometimes updated while maintaining the semantics of "read committed" isolation level. If the database is set to "repeatable read," this concurrency strategy almost maintains the semantics. Repeatable read isolation is compromised in the case of concurrent writes.
 - `Nonstrict Read/Write` - Caches data that is sometimes updated without ever locking the cache. If concurrent access to an item is possible, this concurrency strategy makes no guarantee that the item returned from the cache is the latest version available in the database.

Property Name	Property Value
Entity Name	Cargo
Entity Identity	
Inherits From	
XML Namespace	
XML Element Name	Cargo
Java Package	
Java Class Name	
Datastore Persistent	Yes
Table Name	Cargo.dbo.Cargo
Datastore Caching	<div> No Cache Read Only Read/Write Nonstrict Read/Write </div>
Identity Strategy	
Identity Column Name	
Identity Sequence	
Identity Table Name	
Identity Table Name Column Name	
Identity Table Value Column Name	
Version Strategy	
Version Column Name	

Specifying Query caching

There are two ways to use query caching: on an entity in a Rulesheet, and on a database filter in a Rulesheet. You can use either or both as needed once you have enabled the Vocabulary.

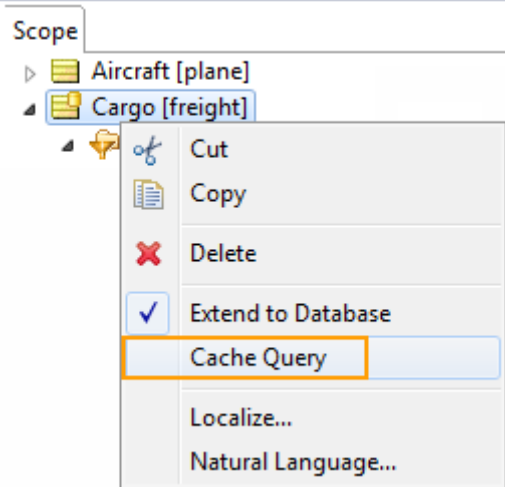
To enable query caching:

1. Identify the Vocabulary entities that you want cached.
2. Edit the Vocabulary, and then choose **Show Vocabulary Details**.
3. Confirm (or set) the entity's **Datastore Persistent** property to **Yes**

Note: Query caching is independent of entity caching so the Datastore Caching setting is not relevant.

To use query caching on an entity in a Rulesheet:

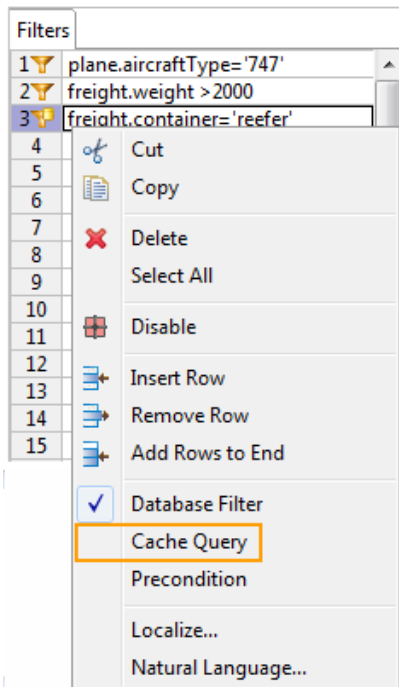
1. In a Rulesheet, choose **Advanced** view to show the **Scope** tab.
2. On the **Scope** tab, datastore-persistent entities show a database decoration, and each one offers the **Extend to Database** option in its dropdown menu. Choose that option to select it.
3. Once extended to database, the option to Cache Query is offered on the dropdown menu, as shown:



4. Choose the option to **Cache Query** on this entity in this Rulesheet.

To use query caching on a database filter in a Rulesheet:

1. In a Rulesheet, choose **Advanced** view to show the **Scope** tab.
2. On the **Scope** section, datastore-persistent entities show a database decoration, and each one offers the **Extend to Database** option in its dropdown menu. Choose that option to select it.
3. On the **Filters** tab, choose a filter that references an entity extended to database, the right-click to display its dropdown menu.
4. Choose **Database Filter**. The filter is decorated with a database symbol.
5. Dropdown the menu on that filter again. The option to **Cache Query** is now available, as shown:

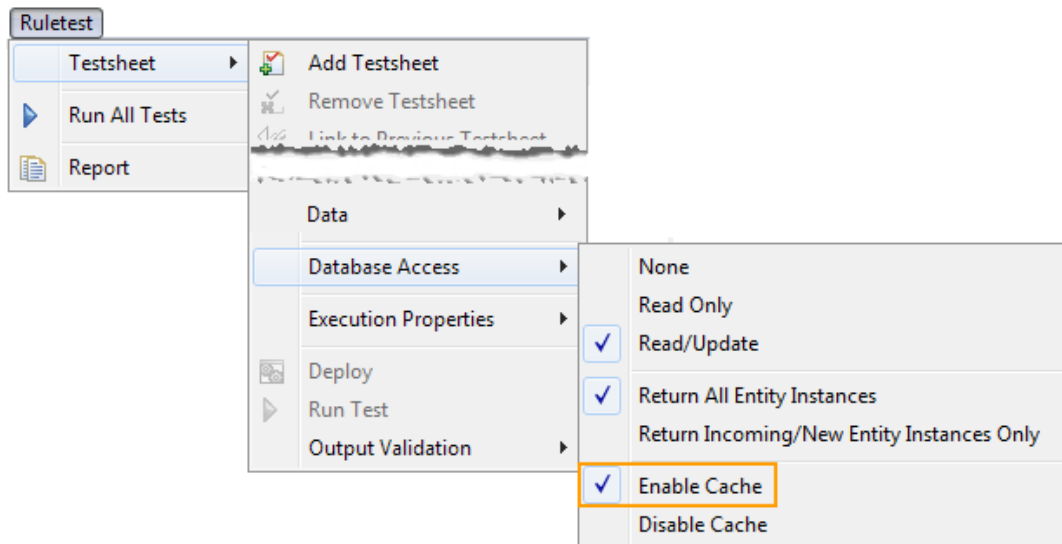


6. Choose the option to **Cache Query** on this filter in this Rulesheet.

Enabling Caching on a Ruletest

Once Rulesheets and the Vocabulary have enabled database caching, each Decision Service has to enable the caching. You can enable caching in Studio through options in Ruletests, and then set parameters that will enable caching on the Decision Services in deployment.

In Studio, use a Ruletest that specifies a test subject that is (or includes) a Rulesheet that specifies entity or query caching, and then choose **Ruletest > Testsheet > Database Access > Enable Caching**, as shown:



Note: For information on how enhanced database caches are implemented in deployments, as well the cache settings and behaviors for both Ruletests and deployed Decision Services, see the topic *"Working with database caches"* in the *Integration and Deployment Guide*.

Test yourself questions: Writing rules to access external data

Note: Try this test, and then go to [Test yourself answers: Logical analysis and optimization](#) on page 357 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?

Logical analysis and optimization

For details, see the following topics:

- [Testing, analysis, and optimization](#)
- [Traditional means of analyzing logic](#)
- [Validating and testing Rulesheets in Corticon Studio](#)
- [Testing rule scenarios in the Ruletest Expected panel](#)
- [Optimizing Rulesheets](#)
- [Precise location of problem markers in editors](#)
- [Test yourself questions: Logical analysis and optimization](#)

Testing, analysis, and optimization

Corticon Studio provides the rule modeler with tools to test, validate, and optimize rules and Rulesheets 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 Ruletest provides the capability to build test cases using real data, which may then be submitted as input to a set of rules for evaluation. The actual output produced by the rules is then compared to the output we expected 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.

For complete details on settings and analysis for scenario testing, see [Testing rule scenarios in the Ruletest Expected panel](#) on page 275

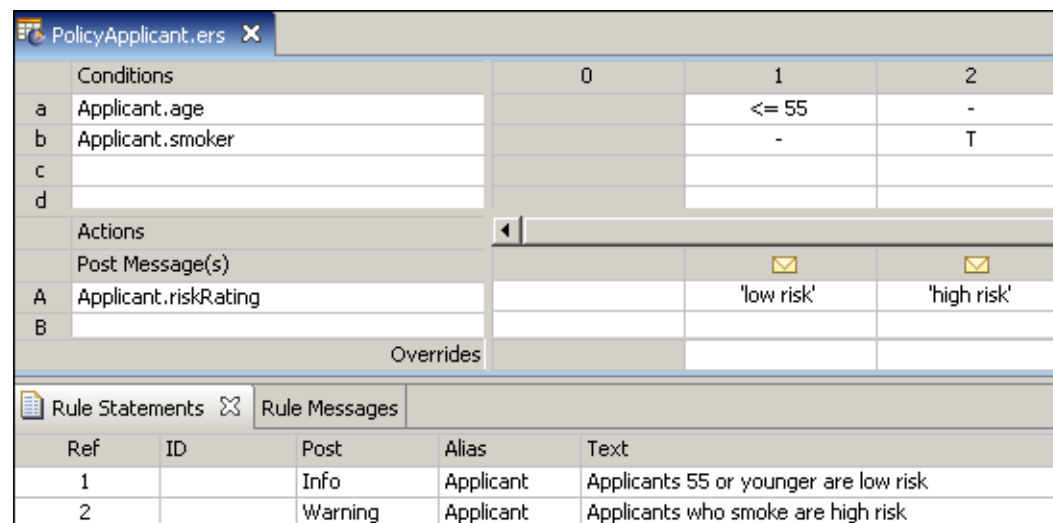
Rulesheet analysis and optimization

Analysis and optimization is the process of examining and correcting or improving the logical construction of Rulesheets, *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 Rulesheets. 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 209: Simple Rules for Profiling Insurance Policy Applicants

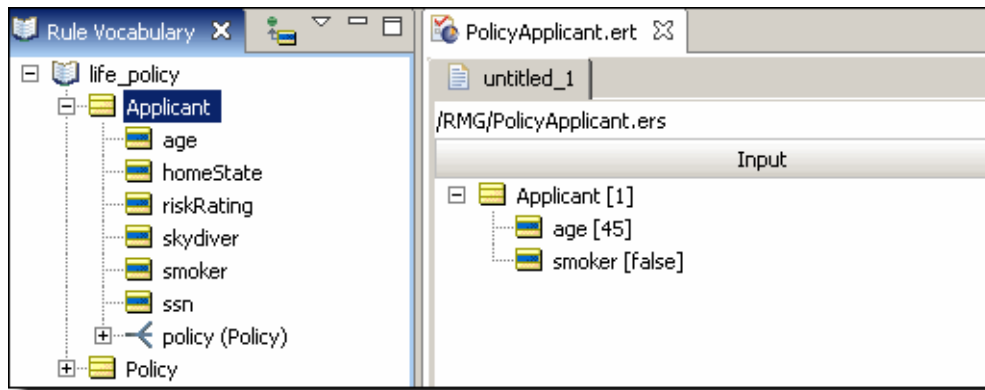


Conditions		0	1	2
a	Applicant.age		<= 55	-
b	Applicant.smoker		-	T
c				
d				

Actions				
Post Message(s)			✉	✉
A	Applicant.riskRating		'low risk'	'high risk'
B				

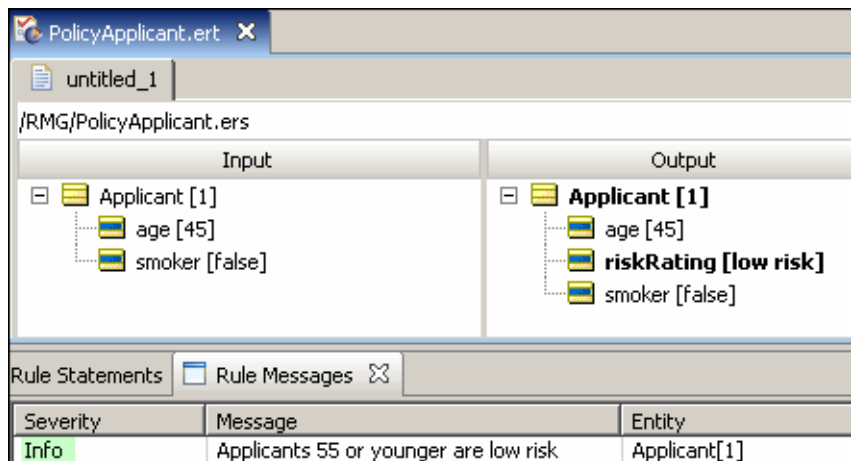
Rule Statements		Rule Messages		
Ref	ID	Post	Alias	Text
1		Info	Applicant	Applicants 55 or younger are low risk
2		Warning	Applicant	Applicants who smoke are high risk

To test these rules, we create a new scenario in a Ruletest, as shown:



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 210: 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?

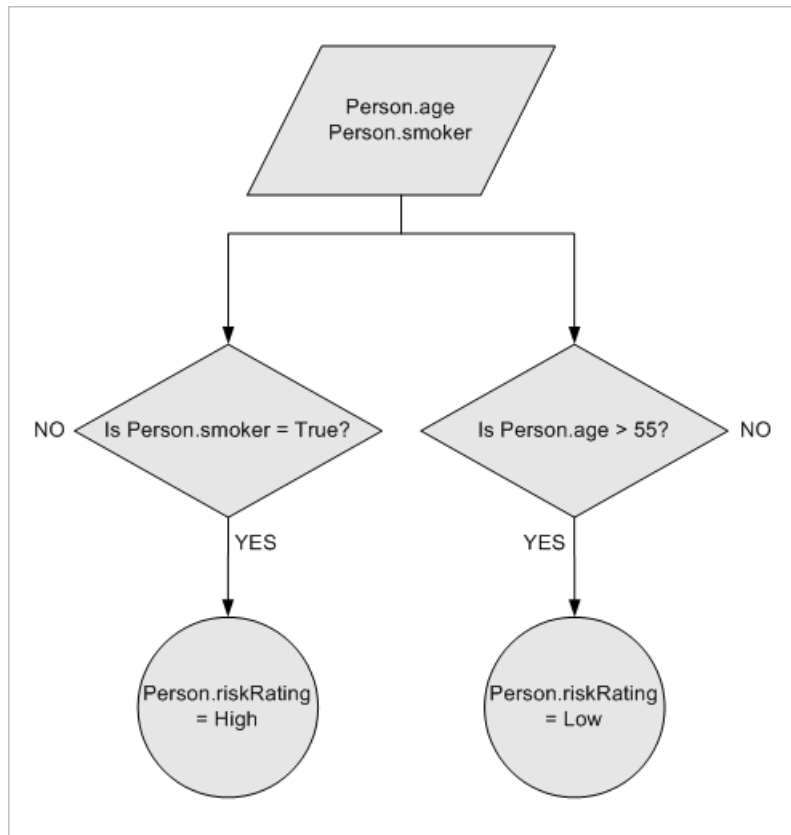
Traditional means of analyzing logic

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 a single rule. More commonly, a decision has dozens or even 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:

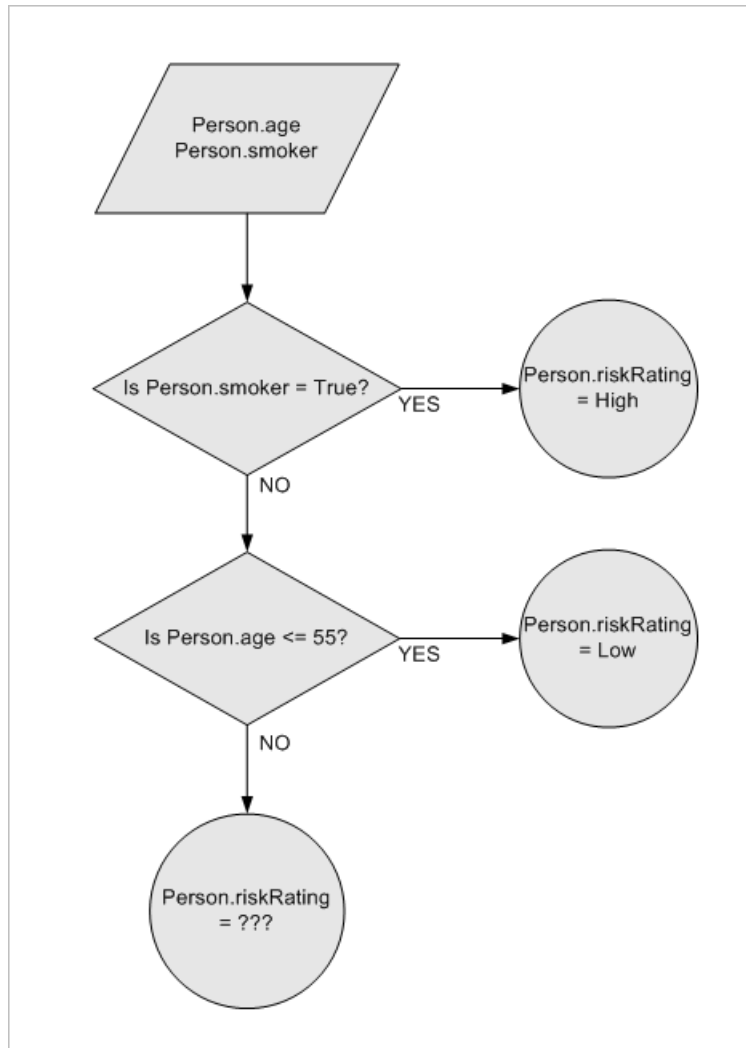
Figure 211: Flowchart with 2 Rules



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 212: 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 9: 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 213: 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 214: 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 215: 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	

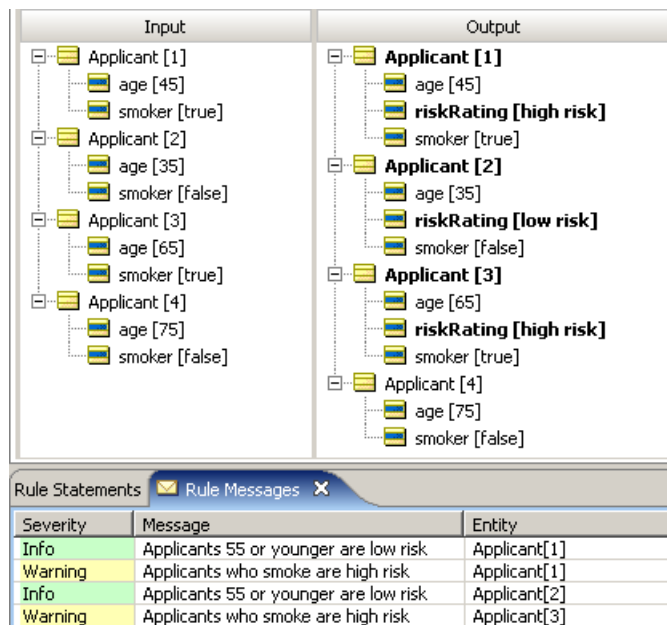
The table format also highlights two problems we encountered earlier with flowcharts. In the figure **Rule 2 Expected Outcome**, row 1 and column 1 intersect in the upper left cell (this cell corresponds to test case #1 in the figure above). As a result, each rule tries to assert its own action – one 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 216: 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.

Validating and testing Rulesheets in Corticon Studio

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. We will also link you to the section that discusses both planned and unintentional looping, [Rule dependency: Chaining and looping](#) on page 173.

Expanding rules

Returning to our original rules (reproduced from [Simple Rules for Profiling Insurance Policy Applicants](#)):

Figure 217: Simple Rules for Profiling Insurance Policy Applicants

Conditions		0	1	2
a	Applicant.age		<= 55	-
b	Applicant.smoker		-	T
c				
d				

Actions		0	1	2
A	Applicant.riskRating		'low risk'	'high risk'
B				

Ref	ID	Post	Alias	Text
1		Info	Applicant	Applicants 55 or younger are low risk
2		Warning	Applicant	Applicants who smoke are high risk

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 218: 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


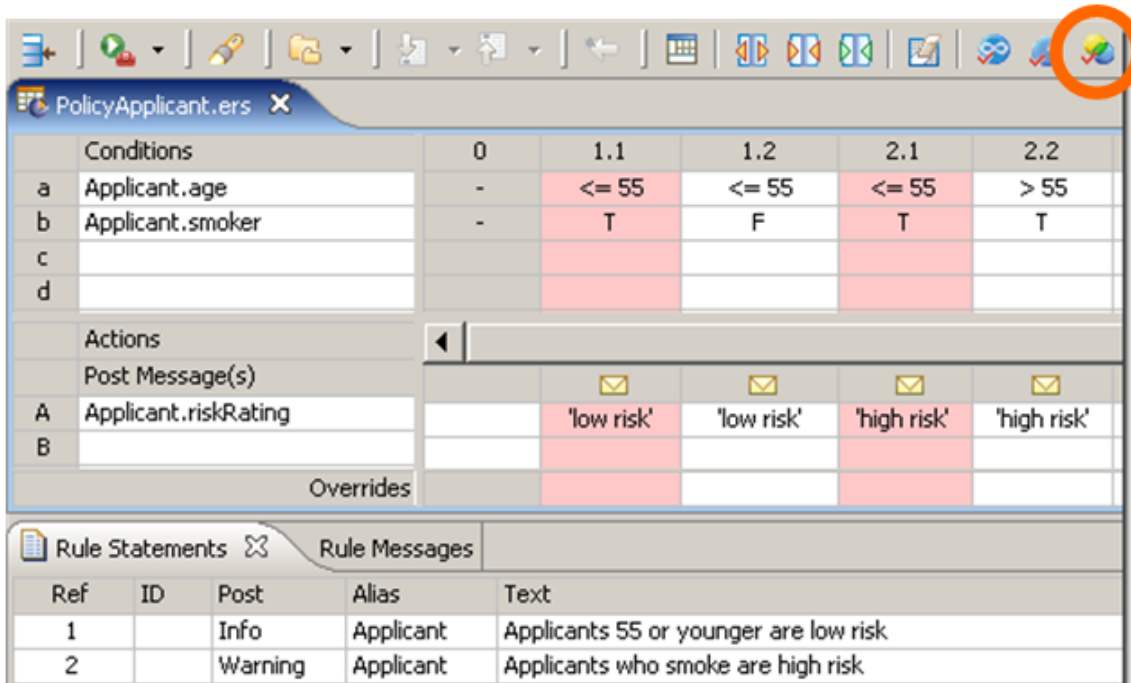
With our two rules expanded into four sub-rules as shown in [Expanding Rules to Reveal Components](#)), most of the Cross Product is displayed for us. Click the **Check for Conflicts**  button in the toolbar.



Figure 219: A Conflict Revealed by the Conflict Checker



Conditions	0	1.1	1.2	2.1	2.2
a Applicant.age	-	<= 55	<= 55	<= 55	> 55
b Applicant.smoker	-	T	F	T	T
c					
d					
Actions					
Post Message(s)					
A Applicant.riskRating		'low risk'	'low risk'	'high risk'	'high risk'
B					
Overrides					

Ref	ID	Post	Alias	Text
1		Info	Applicant	Applicants 55 or younger are low risk
2		Warning	Applicant	Applicants who smoke are high risk

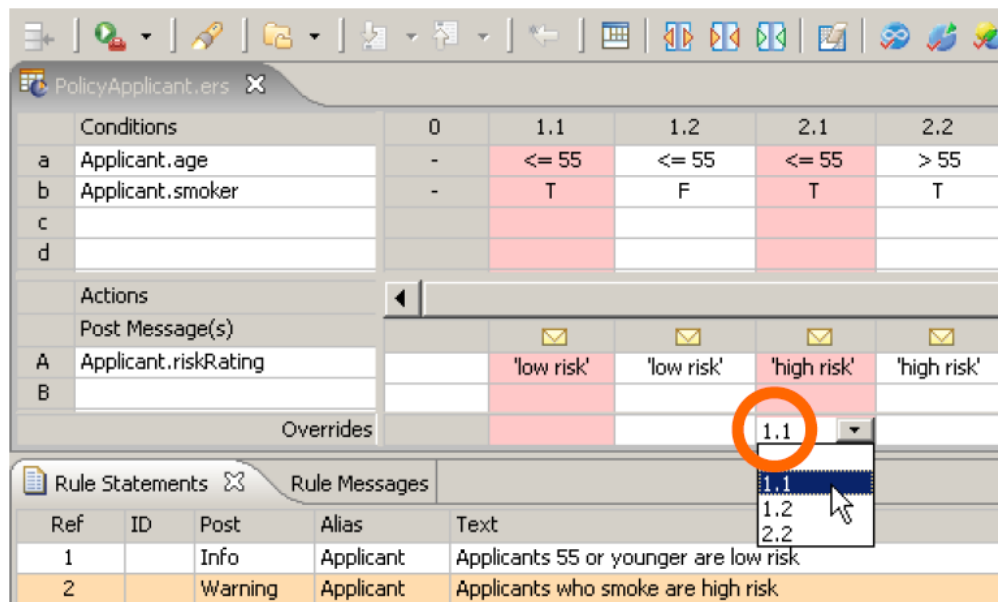
The mechanics of stepping through and resolving each conflict are described in detail in the *Corticon Studio - Basic Tutorial*.

Note: Refresher on conflict discovery and resolution -- On a Rulesheet, click **Check for Conflicts** , and then expand the rules by clicking **Expand Rules** . Expansion shows all of the logical possibilities for each rule. To resolve conflict, either change the rules, or decide that one rule should override another. To do that, in the **Overrides** row at each column intersection where an override is intended, select the one or more column numbers that will be overridden when that rule fires. Click **Check for Conflicts** again to confirm that the conflicts are resolved.

In this topic, 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 219: A Conflict Revealed by the Conflict Checker](#) on page 263, 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 Rulesheets 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 220: Override Entered to Resolve Conflict



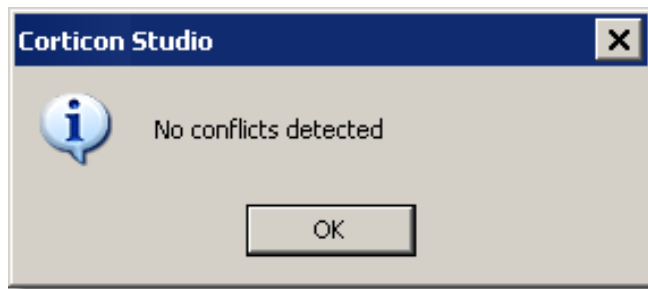
An override is essentially another business rule, which should to be expressed somewhere in the *Rule Statements* section of the Rulesheet. To express this override in plain English, the rule writer might choose to modify the rule statement for the *overridden* rule:

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 221: Conflict Resolution Complete



Note: How does one rule override another rule? - To understand overrides, the first concept to learn is *condition context*. The condition context of a rule is the set of all entities, aliases, and associations that are needed to evaluate all the conditional expressions of a rule. The second concept is the *override context*. The override context is defined using set algebra. The override context of two rules is the intersection of the two rule's condition contexts. To evaluate the override, the set of entities that fulfill the overriding rule's conditions are trimmed to the override context and recorded. Before the conditions of the overridden rule are evaluated, the entities that are part of the override context are tested to determine if they have been recorded; if so, the rule is overridden and processing of the rule with those entities is halted. If the override context is empty, then any execution of the overriding rule will stop all executions of the overridden rule.

Overrides can take effect even if there are no conflicts - Overrides can be used for more than just conflicting rules. While the basic use of overrides is in cases where rules are in conflict to allow the modeler to control execution, it is not the only use. The more advanced usage applies cases where there is a *logical dependency* -- cases where a rule might modify the data so that another rule can also execute. This type of conflict is not detected by the conflict checker.

Consider a simple Cargo Rulesheet:

Conditions		0	1	2
a	Cargo.volume		100	200
b				
c				
Actions		<div> <div></div> <div> </div> </div>		
Post Message(s)				
A	Cargo.volume		200	150
R				
Overrides				

When tested, the first rule is triggered and its action sets a value that triggers rule 2:

Input	Output
<div> <div>Cargo [1]</div> <div> <div>container</div> <div>volume [100]</div> <div>weight</div> </div> </div> <div> <div>Cargo [2]</div> <div> <div>container</div> <div>volume [200]</div> <div>weight</div> </div> </div>	<div> <div>Cargo [1]</div> <div> <div>container</div> <div>volume [150]</div> <div>weight</div> </div> </div> <div> <div>Cargo [2]</div> <div> <div>container</div> <div>volume [150]</div> <div>weight</div> </div> </div>

The Ruletest result shows that the value set in the first rule's action modified the data so that the change in the condition's value triggered the second rule. If this effect is not what is intended, an override can be used. The use of an override here ensures that the modification of data will not trigger execution of the second rule -- they are *mutually exclusive* (mutex). When an override is set on rule 1 that specifies that, if it fired, it should skip rule 2...

Conditions		0	1	2
a	Cargo.volume		100	200
b				
c				
Actions		<div> <div></div> <div> </div> </div>		
Post Message(s)				
A	Cargo.volume		200	150
R				
Overrides			2	

... the rules produce the preferred output:

Input	Output
<div> <div>Cargo [1]</div> <div> <div>container</div> <div>volume [100]</div> <div>weight</div> </div> </div> <div> <div>Cargo [2]</div> <div> <div>container</div> <div>volume [200]</div> <div>weight</div> </div> </div>	<div> <div>Cargo [1]</div> <div> <div>container</div> <div>volume [200]</div> <div>weight</div> </div> </div> <div> <div>Cargo [2]</div> <div> <div>container</div> <div>volume [150]</div> <div>weight</div> </div> </div>

If these rules were re-ordered, the override would be unnecessary.

Using overrides to handle conflicts that are logical dependencies

Overrides can be used for more than just conflicting rules. While the basic use of overrides is in cases where rules are in conflict to allow the modeler to control execution, it is not the only use. An advanced usage applies to cases where there is a *logical dependency* -- cases where a rule might modify the data so that it forces another rule to execute. That can be what you want but it could be unintended. This type of conflict is not detected by the conflict checker.

Consider a simple Cargo Rulesheet:

Conditions		0	1	2
a	Cargo.volume		100	200
b				
c				
Actions				
Post Message(s)				
A	Cargo.volume		200	150
R				
Overrides				

When tested, the first rule is triggered and its action sets a value that triggers rule 2:

Input	Output
<div> Cargo [1]<ul style="list-style-type: none"> container volume [100] weight</div> <div> Cargo [2]<ul style="list-style-type: none"> container volume [200] weight</div>	<div> Cargo [1]<ul style="list-style-type: none"> container volume [150] weight</div> <div> Cargo [2]<ul style="list-style-type: none"> container volume [150] weight</div>

The Ruletest result shows that the value set in the first rule's action modified the data so that the change in the condition's value triggered the second rule. If this effect is not what is intended, an override can be used. The use of an override here ensures that the modification of data will not trigger execution of the second rule -- they are *mutually exclusive* (mutex). When an override is set on rule 1 that specifies that, if it fired, it should skip rule 2...

Conditions		0	1	2
a	Cargo.volume		100	200
b				
c				
Actions		< III >		
Post Message(s)				
A	Cargo.volume		200	150
B				
Overrides			2	

... then the rules produce the preferred output:

Input	Output
<ul style="list-style-type: none"> ▲ Cargo [1] <ul style="list-style-type: none"> container volume [100] weight ▲ Cargo [2] <ul style="list-style-type: none"> container volume [200] weight 	<ul style="list-style-type: none"> ▲ Cargo [1] <ul style="list-style-type: none"> container volume [200] weight ▲ Cargo [2] <ul style="list-style-type: none"> container volume [150] weight

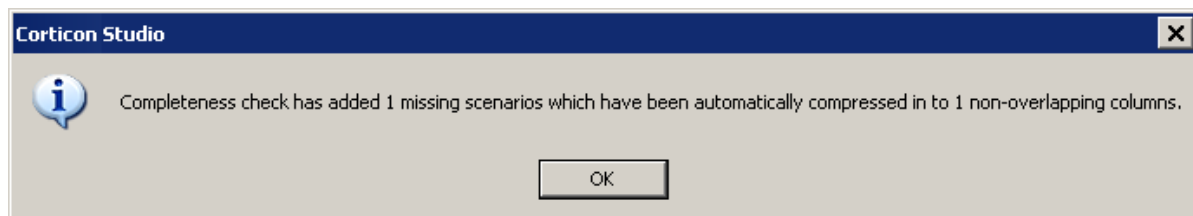
Note that in this example, if these rules are re-ordered, the override is not necessary.

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 222: 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 223: New Rule Added by Completeness Check

Conditions		0	1.1	1.2	2.1	2.2	3
a	Applicant.age	-	<= 55	<= 55	<= 55	> 55	> 55
b	Applicant.smoker	-	T	F	T	T	F
c							
d							

Actions						
Post Message(s)						
A	Applicant.riskRating		'low risk'	'low risk'	'high risk'	'high risk'
B						

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

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 224: Example Prior to Completeness Check

CompletenessCheckerLimitations.ers				
Conditions		0	1	2
a	Patient.gender		'female'	'female'
b	Patient.age		<= 40	> 40
c	Patient.pregnant		T	T
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, we will click on the Completeness Checker:

Figure 225: Example after Completeness Check

Conditions	0	1	2	3	4
a Patient.gender	-	'female'	'female'	<> 'female'	'female'
b Patient.age	-	<= 40	> 40	-	-
c Patient.pregnant	-	T	T	-	F
d					

Actions	0	1	2	3	4
Post Message(s)		✉	✉		
A Patient.riskFactor		'normal'	'elevated'		
B					

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

Progress Corticon Studio

Completeness check has added 6 missing scenarios which have been automatically compressed in to 2 non-overlapping columns.

OK

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 226: 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 227: Sub-Rules Renumbered and Converted to General Rules

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									

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 228: Final Rulesheet with impossible rules disabled

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									

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
{ 4 , 6 }		Warning	Patient	Non-pregnant, non-females not considered by this decision
{ 3 , 5 }		Violation	Patient	Pregnant non-females are not possible: these rules have been disabled
7		Warning	Patient	Non-pregnant females not considered by this decision

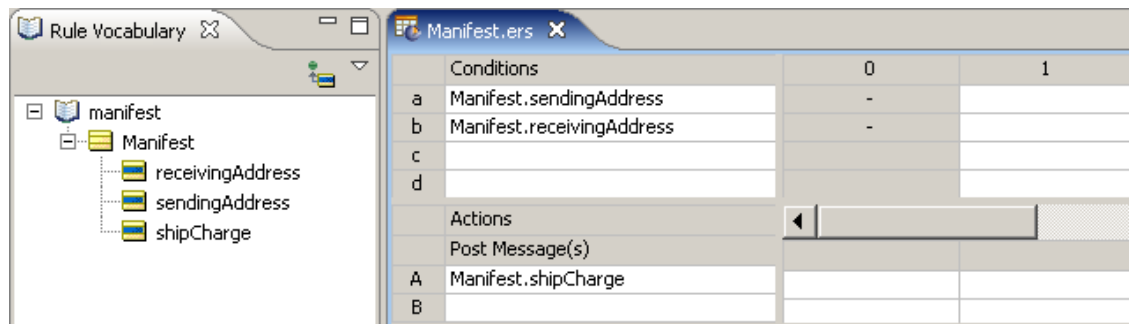
Letting the expansion tool work for you: tabular rules

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 Kilogram					
From/To	zone 1	zone 2	zone 3	zone 4	zone 5
zone 1	\$1.25	\$2.35	\$3.45	\$4.55	\$5.65
zone 2	\$2.35	\$1.25	\$2.35	\$3.45	\$4.55
zone 3	\$3.45	\$2.35	\$1.25	\$2.35	\$3.45
zone 4	\$4.55	\$3.45	\$2.35	\$1.25	\$2.35
zone 5	\$5.65	\$4.55	\$3.45	\$2.35	\$1.25

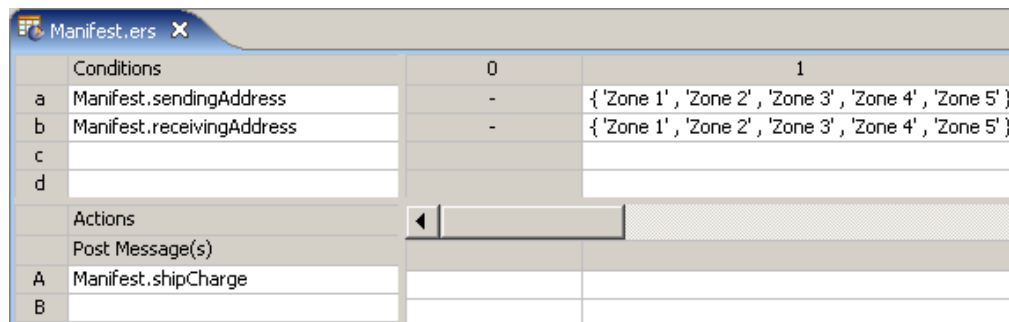
In the following figure, we have built a simple Vocabulary with which to implement these rules. Because each cell in the table represents a single rule, our Rulesheet will contain 25 columns (the Cross Product equals 5x5 or 25).

Figure 229: 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 230: 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 following figure due to page width limitations in this document):

Figure 231: Rule 1 Expanded to Show Sub-Rules

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											
B											

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 232: Rulesheet with Actions Populated

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 233: Rulesheet with Renumbered Rules

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, get details on increasing Corticon Studio's memory allotment in *"Changing the Studio's memory allocation" in the Corticon Installation Guide*.

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 section [Rule dependency: Chaining and looping](#) on page 173.

Testing rule scenarios in the Ruletest Expected panel

Using Ruletests you can submit request data as input to Rulesheets or Ruleflows to see how the rules are evaluated and the resulting output. You can make Ruletests even more powerful by specifying the results you expected, and then seeing how it reconciles with the output. Running the test against a specified Rulesheet or Ruleflow automatically compares the actual **Output** data to your **Expected** data, and color codes the differences for easy review and analysis.

You can establish the expected data either by copying the output column or creating its data directly from the Vocabulary.

To create expected data from test output:

1. Create or import a request into a Ruletest, and run the test against an appropriate Rulesheet or Ruleflow.
2. Choose the menu command **Ruletest > Testsheet > Data > Output > Copy to Expected**, or click  button in the Corticon Studio toolbar.

To create expected data directly from the Vocabulary:

1. Drag and drop nodes from the **Rule Vocabulary** window to create a tree structure in the **Expected** panel that is identical to the input tree.
2. Enter expected values for the **Input** attributes as well as the attributes that will be added in the **Output** panel.

Note:

When expected values are Strings, they might have leading or trailing blanks (often called *whitespace*) that can cause imprecise matching to the output from rules. While the default behavior of trimming whitespace is typically preferred, you can set a Studio property that does *not* perform trimming, and thus reduce these validation mismatches. The default behavior is apparent when copying output to the Expected column, as that action strips whitespaces, and often reveals such mismatches immediately.

To retain leading and trailing whitespaces when output is added to the **Expected** column, add the line:


```
com.corticon.tester.trimstringvalues=false
```

to Studio's `brms.properties` file (located at `[CORTICON_WORK_DIR]\Studio`), and then restart Studio.

Navigating in Ruletest Expected comparison results

When reviewing the results of a test run, two navigation features help you focus your attention :

- **Synchronized scrolling** - When you slide the scroll tab in the Ruletest panels, the three columns do not move together, making alignment of data points difficult. You can set (or unset) synchronized scrolling of

the columns by either right-clicking any of the Ruletest panels and then choosing **Scroll Lock**, or clicking  in the Corticon Studio toolbar. Once set to synchronize, all panels will synchronize their scrolling, even advancing across collapsed entities and associations to stay synchronized on the first displayed line.

- **Navigation to differences** - The Ruletest window provides a set of controls that report the number of discovered differences and controls to navigate across the items. In the upper right of the Ruletest window, the following image shows that the test results have identified six differences:

Differences: 6 

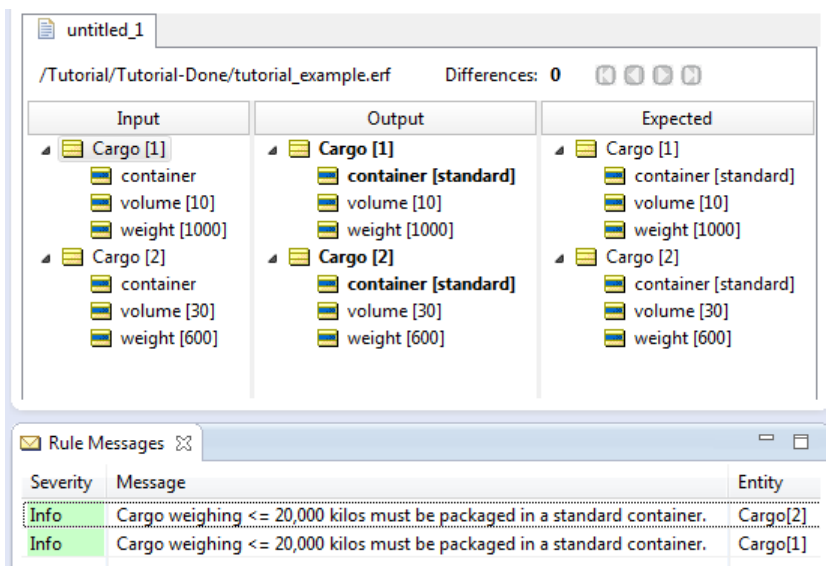
The four buttons, as appropriate, take you to the first, previous, next, and last discovered difference.

Reviewing test results when using the Expected panel

The following topics present a variety of test results.

Expected panel: Output results match expected exactly

In the example below, both `packaging` values are shown in **bold** text, indicating that these values were changed by the rules. Because all colors are black and the differences count is **0**, the **Output** data is consistent with the **Expected** data.



The screenshot shows the Ruletest window for a file named `untitled_1`. The top bar indicates `/Tutorial/Tutorial-Done/tutorial_example.erf` and `Differences: 0`. The main area is divided into three panels: **Input**, **Output**, and **Expected**. Each panel shows a tree structure of data for two cargo items, `Cargo [1]` and `Cargo [2]`. The **Output** panel shows values in bold text, indicating they were changed by the rules. The **Expected** panel shows the original values. The **Rule Messages** panel at the bottom shows two information messages: "Cargo weighing <= 20,000 kilos must be packaged in a standard container." for `Cargo[2]` and `Cargo[1]`.

Severity	Message	Entity
Info	Cargo weighing <= 20,000 kilos must be packaged in a standard container.	Cargo[2]
Info	Cargo weighing <= 20,000 kilos must be packaged in a standard container.	Cargo[1]

Expected panel: Different values output than expected

In the example shown below, one difference has been identified. The expected value of `Cargo [2]` packaging value is `standard`, but the Ruletest produced an actual value of `oversize`. Since the **Output** does not match the **Expected** data, the text is colored red.

The screenshot shows the Ruletest Expected panel for a file named 'untitled_1'. The path is '/Tutorial/Tutorial-Done/tutorial_example.erf'. It indicates 'Differences: 1'. The panel is divided into three columns: Input, Output, and Expected. Below these columns is a 'Rule Messages' section.

Input	Output	Expected
<ul style="list-style-type: none"> Cargo [1] <ul style="list-style-type: none"> container volume [10] weight [1000] Cargo [2] <ul style="list-style-type: none"> container volume [30] weight [600] 	<ul style="list-style-type: none"> Cargo [1] <ul style="list-style-type: none"> container [standard] volume [10] weight [1000] Cargo [2] <ul style="list-style-type: none"> container [oversize] volume [30] weight [600] 	<ul style="list-style-type: none"> Cargo [1] <ul style="list-style-type: none"> container [standard] volume [10] weight [1000] Cargo [2] <ul style="list-style-type: none"> container [standard] volume [30] weight [600]

Severity	Message	Entity
Info	Cargo weighing <= 20,000 kilos must be packaged in a standard container.	Cargo[2]
Info	Cargo weighing <= 20,000 kilos must be packaged in a standard container.	Cargo[1]

In this example, notice that it is the value determined by the rule that changed, not the input values. Research indicates that the designer changed the rule for volume from >30 to ≥ 30 thereby triggering the different container requirement.

Expected panel: Fewer values output than expected

In the example below, Cargo [2] has no input attribute values in the **Input** panel. The rule test failed because of inadequate input data, and the two missing attributes (and their expected values) are colored green.

The screenshot shows the Ruletest Expected panel for a file named 'untitled_1'. The path is '/Tutorial/Tutorial-Done/tutorial_example.erf'. It indicates 'Differences: 3'. The panel is divided into three columns: Input, Output, and Expected. Below these columns is a 'Rule Messages' section.

Input	Output	Expected
<ul style="list-style-type: none"> Cargo [1] <ul style="list-style-type: none"> container volume [10] weight [1000] Cargo [2] <ul style="list-style-type: none"> container 	<ul style="list-style-type: none"> Cargo [1] <ul style="list-style-type: none"> container [standard] volume [10] weight [1000] Cargo [2] <ul style="list-style-type: none"> container 	<ul style="list-style-type: none"> Cargo [1] <ul style="list-style-type: none"> container [standard] volume [10] weight [1000] Cargo [2] <ul style="list-style-type: none"> container [standard] volume [30] weight [600]

Severity	Message	Entity
Info	Cargo weighing <= 20,000 kilos must be packaged in a standard container.	Cargo[1]

Expected panel: More values output than expected

In the example below, Cargo [3] was added in the **Input**, and shown correctly in the **Output** panel. But because it was not anticipated by the **Expected** panel, it is colored blue as one difference at the entity level.

/Tutorial/Tutorial-Done/tutorial_example.erfDifferences: 1

Input	Output	Expected
<div>Cargo [1]</div> <div>container</div> <div>volume [10]</div> <div>weight [1000]</div>	<div>Cargo [1]</div> <div>container [standard]</div> <div>volume [10]</div> <div>weight [1000]</div>	<div>Cargo [1]</div> <div>container [standard]</div> <div>volume [10]</div> <div>weight [1000]</div>
<div>Cargo [2]</div> <div>container</div> <div>volume [30]</div> <div>weight [600]</div>	<div>Cargo [2]</div> <div>container [standard]</div> <div>volume [30]</div> <div>weight [600]</div>	<div>Cargo [2]</div> <div>container [standard]</div> <div>volume [30]</div> <div>weight [600]</div>
<div>Cargo [3]</div> <div>container</div> <div>volume [75]</div> <div>weight [22000]</div>	<div>Cargo [3]</div> <div>container [oversize]</div> <div>volume [75]</div> <div>weight [22000]</div>	

Rule Messages

Severity	Message	Entity
Info	Cargo with volume > 30 cubic meters must be packaged in an oversize container.	Cargo[3]
Info	Cargo weighing <= 20,000 kilos must be packaged in a standard container.	Cargo[2]
Info	Cargo weighing <= 20,000 kilos must be packaged in a standard container.	Cargo[1]

Expected panel: All problems

In this example, there are three differences. The designer changed the trigger point for volume so `Cargo[1]` chose a container that is different from what was previously expected. `Cargo[3]` is on the input and likewise in the output, but `Cargo[2]` was expected and is missing from the output.

/Tutorial/Tutorial-Done/tutorial_example.erfDifferences: 3

Input	Output	Expected
<div>Cargo [1]</div> <div>container</div> <div>volume [10]</div> <div>weight [1000]</div>	<div>Cargo [1]</div> <div>container [oversize]</div> <div>volume [10]</div> <div>weight [1000]</div>	<div>Cargo [1]</div> <div>container [standard]</div> <div>volume [10]</div> <div>weight [1000]</div>
<div>Cargo [3]</div> <div>container</div> <div>volume [75]</div> <div>weight [22000]</div>	<div>Cargo [3]</div> <div>container [oversize]</div> <div>volume [75]</div> <div>weight [22000]</div>	<div>Cargo [2]</div> <div>container [standard]</div> <div>volume [30]</div> <div>weight [600]</div>

Rule Messages

Severity	Message	Entity
Info	Cargo with volume > 30 cubic meters must be packaged in an oversize container.	Cargo[1]
Info	Cargo with volume > 30 cubic meters must be packaged in an oversize container.	Cargo[3]

Techniques that refine rule testing

The following settings help you tune the results of comparing the output data and expected data so that irrelevant errors are minimized:

Setting selected attributes to ignore validation.

When different values are output than what was expected, it could mean that the **Expected** panel data created from **Output** data were reflecting dynamic values such as dates and time. If your Rulesheets use `now` or `today`, the **Expected** values will evaluate as errors very soon. To handle that situation, you can choose to ignore validation for selected values in the **Expected** panel.

Consider the following example:

The selected attribute in this test has no input value and no expected value

Input	Output	Expected
<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date id newlyCreated [true] premium [0] type [Standard] 		<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date id newlyCreated [true] premium [2220.000000] type [Standard]

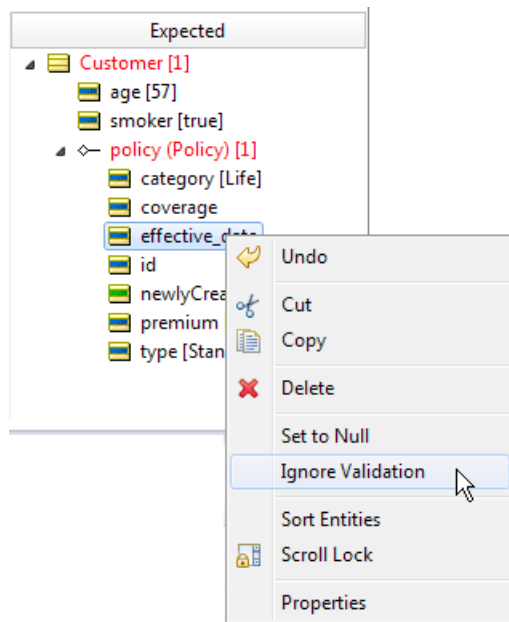
When the test runs, it is valid.

Input	Output	Expected
<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date id newlyCreated [true] premium [0] type [Standard] 	<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date id newlyCreated [true] premium [2220.000000] type [Standard] 	<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date id newlyCreated [true] premium [2220.000000] type [Standard]

But when the input gets a value and the output still has no value (or a different value), the test fails.

Input	Output	Expected
<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date [7/4/2014] id newlyCreated [true] premium [0] type [Standard] 	<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date [7/4/2014] id newlyCreated [true] premium [2220.000000] type [Standard] 	<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date id newlyCreated [true] premium [2220.000000] type [Standard]

Clicking on the expected attribute, you can choose **Ignore Validation**.



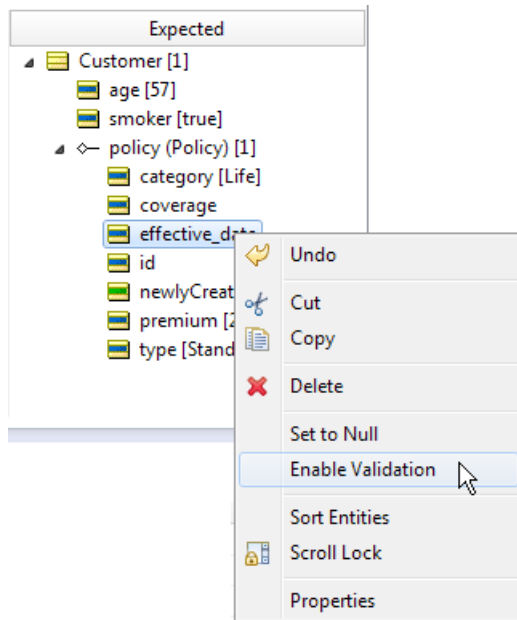
An attribute that will be ignored is greyed out.

Input	Output	Expected
<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date [7/4/2014] id newlyCreated [true] premium [0] type [Standard] 		<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date id newlyCreated [true] premium [2220.000000] type [Standard]

Running the same test, the test passes.

Input	Output	Expected
<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date [7/4/2014] id newlyCreated [true] premium [0] type [Standard] 	<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date [7/4/2014] id newlyCreated [true] premium [2220.000000] type [Standard] 	<ul style="list-style-type: none"> Customer [1] <ul style="list-style-type: none"> age [57] smoker [true] policy (Policy) [1] <ul style="list-style-type: none"> category [Life] coverage effective_date id newlyCreated [true] premium [2220.000000] type [Standard]

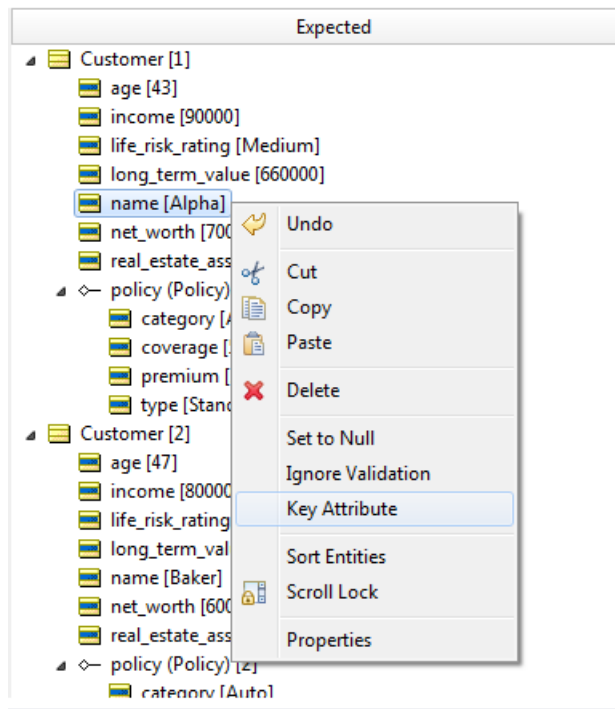
The setting can revert by selecting the attribute and then choosing **Enable Validation**.



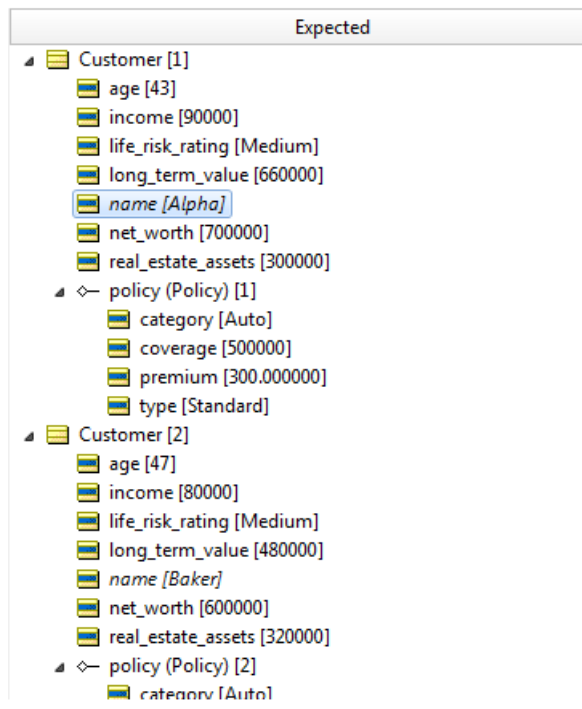
Using key attributes to improve difference detection in Ruletests

The execution of Ruletests can, in certain cases, erroneously detect differences between the Output and Expected results. This typically occurs in Rulesheets which add new entities to collections. The unsorted nature of collections makes it impossible to match the collections in the Output and Expected results with complete accuracy. An optional feature is available when you encounter problems with test failures due to the randomness of entity ordering. To avoid this problem, you can specify certain attributes as *key attributes* that will assist the comparison algorithm, that the validation linking entities in both panels are chosen based on the key values.

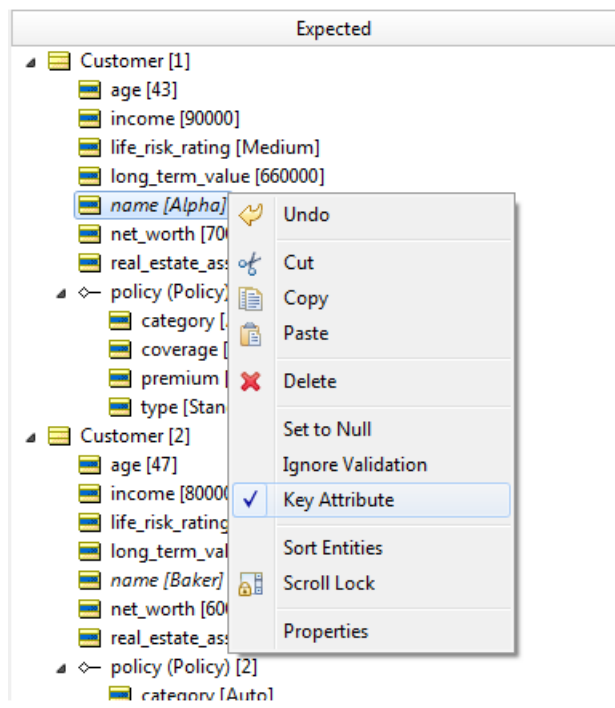
To set a key attribute, right click on it in the Expected panel, and then choose **Key Attribute**, as shown:



Key attributes are shown in italics in the current entity as well as all other corresponding entities in the **Expected** panel, as shown:



To remove a key attribute, right click on it again in the **Expected** panel, and then choose **Key Attribute** to clear the setting, as shown:




Setting multiple key attributes will attempt to match the full set.

Setting how whitespace is handled

Leading or trailing blanks on String values (often called *whitespace*) might cause imprecise matching to the output from rules. While the default behavior of trimming whitespaces is often preferred, you can add `com.corticon.tester.trimstringvalues=false` to your `brms.properties` file to tell Corticon Studio to not perform trimming, and thus reduce validation mismatches. The default behavior is apparent when copying the output to the Expected column, as that action strips whitespaces, and often reveals apparent mismatches immediately.

Optimizing Rulesheets

The compress tool

Corticon Studio helps improve performance by removing redundancies within Rulesheets. 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 234: 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 235: Redundancy Problem Exposed

PolicyApplicant.ers		0	1.1	1.2	2.1	2.2	3	4
Conditions								
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			

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 236: Rulesheet After Compression

PolicyApplicant.ers		0	1	2	3
Conditions					
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.

- Combining Values sets to simplify and shorten Rulesheets. 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 237: 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 greater than 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 238: Implementing the 747 Rule

Conditions		0	1	2
a	Aircraft.maxCargoVolume	-	> 300	
b	Aircraft.maxCargoWeight	-	> 200000	
c	Aircraft.tailNumber	-	'N123UA'	
d				
Actions				
Post Message(s)				
A	Aircraft.aircraftType		'747'	
B				
Overrides				

Ref	ID	Post	Alias	Text
1		Info	Aircraft	An aircraft with max cargo volume greater than 300 and max cargo weight of greater than 200,000 kilogms and a tail number of N123UA is assigned to a 747.

Now, letting the Completeness Checker populate the missing columns:

Figure 239: Remaining Columns Produced by the Completeness Checker

Conditions

	0	1	2	3	4	5
a	Aircraft.maxCargoVolume	-	> 300	> 300	> 300	{ <= 300 , null }
b	Aircraft.maxCargoWeight	-	> 200000	> 200000	{ <= 200000 , n...	-
c	Aircraft.tailNumber	-	'N123UA'	{ <> 'N123UA' , ...	-	-
d						

Actions

	0	1	2	3	4	5
A	Aircraft.aircraftType		'747'			
B						

Rule Statements

Ref	ID	Post	Alias	Text	Rule Name	Rule Link
1		Info	Aircraft	An aircraft with max cargo volume greater than 300 and max cargo weight of greater than 200,000 kilograms and a tail number of N123UA is assigned to a 747.		

Progress Corticon Studio

Completeness check has added 26 missing scenarios which have been automatically compressed in to 3 non-overlapping columns.

OK

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 240: Underlying Sub-Rules Produced by the Completeness Checker

0	1	2.1	2.2	3.1	3.2	3.3	3.4	3.5
-	> 300	> 300	> 300	> 300	> 300	> 300	> 300	> 300
-	> 200000	> 200000	> 200000	<= 200000	<= 200000	<= 200000	null	null
-	'N123UA'	<> 'N123UA'	null	<> 'N123UA'	'N123UA'	null	<> 'N123UA'	'N123UA'

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 ($3 \times 3 \times 3$ or 3 cubed). Subtracting the combination already present in column 1, we expect 26 new columns to be added.

Now, **Compress**

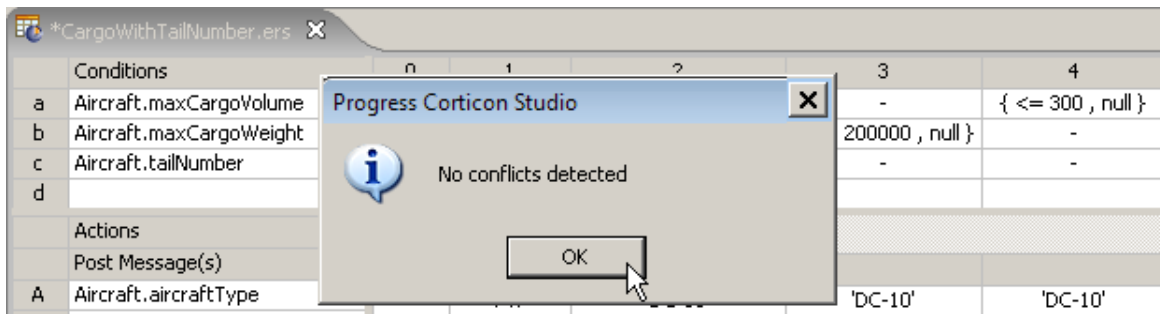
the Rulesheet and fill in the Actions for the new columns as shown in [Missing Rules with Actions Assigned](#):

Figure 241: Missing Rules with Actions Assigned

CargoWithTailNumber.ers		0	1	2	3	4
Conditions						
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 242: 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 243: Expanding Rules *Following* Compression

[illegible]

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.

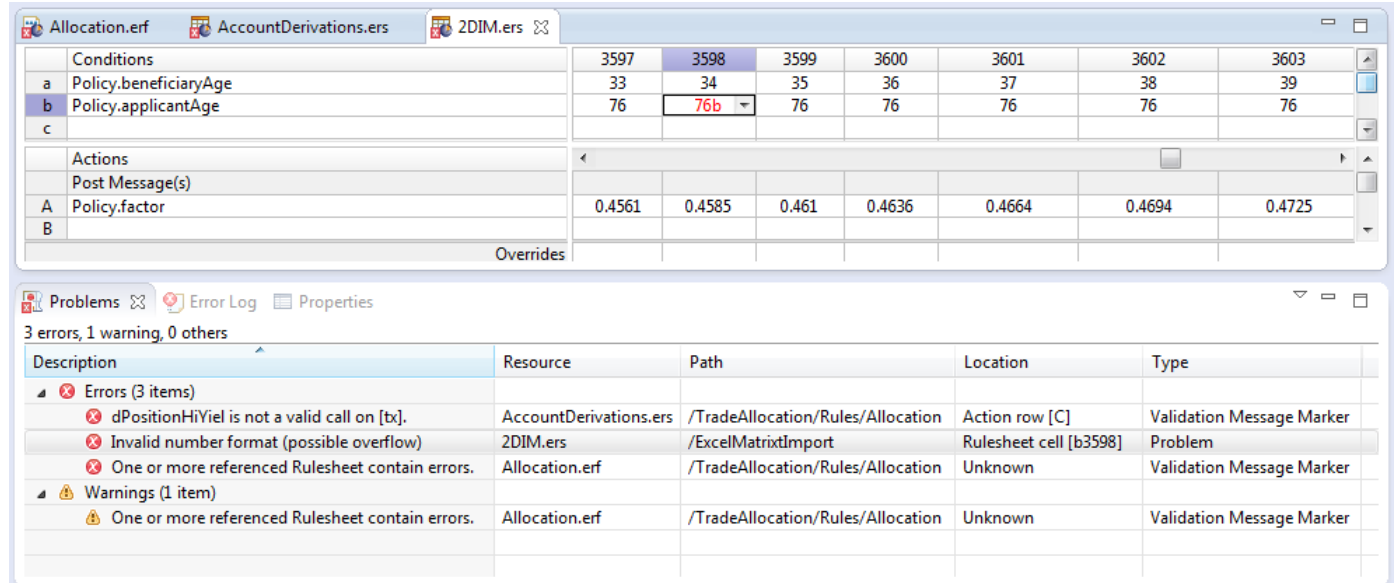
On the other hand, the compression performed by the Compression Tool is designed to reduce the number of rules into the smallest set of general rules (columns with dashes), even if the total number of sub-rules is larger than that produced by the Completeness Checker. This is important for three reasons:

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 Rulesheets (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 Rulesheets 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.

Precise location of problem markers in editors

Problems experienced in Corticon editors are easily located when you click on each annotated error line in the **Problems** view to open the corresponding file in its editor, then bring the specific location into view and give it the focus.

In the following illustration, the problem location is Rulesheet cell [b3598] of the 2DIM Rulesheet. Double-clicking the problem line opened the file to that precise location, as shown:



This functionality applies to Vocabularies, Rulesheets, Ruleflows, and Ruletests.

Note: When migrating projects from earlier releases, the marker metadata has not been captured. When you clear the existing problem list, and then perform a full build of the project, the location metadata that enables this feature will be established.

Test yourself questions: Logical analysis and optimization

Note: Try this test, and then go to [Test yourself answers: Logical analysis and optimization](#) on page 357 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 Rulesheets 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?

Advanced Ruleflow techniques and tools

Ruleflows provide techniques for combining, branching, and graphing. You can also use versioning and effective dating to precisely manage your Ruleflows when they are deployed as Decision Services.

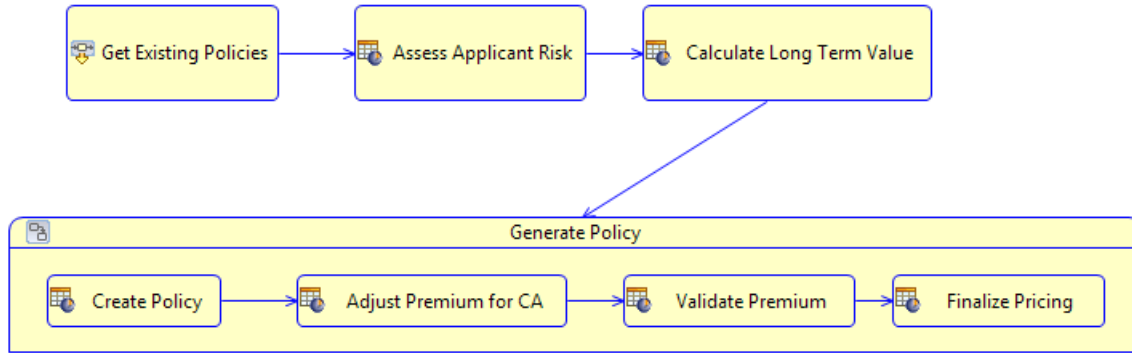
For details, see the following topics:

- [Using a Ruleflow in another Ruleflow](#)
- [Conditional Branching in Ruleflows](#)
- [Generating Ruleflow dependency graphs](#)
- [Ruleflow versioning and effective dating](#)

Using a Ruleflow in another Ruleflow

You can reduce the complexity and testing of large Ruleflows by breaking a Ruleflow into smaller Ruleflows, and then constructing the larger Ruleflow from them. The resulting modularity simplifies unit testing and collaboration.

Consider the following Ruleflow from the Life Insurance sample project:



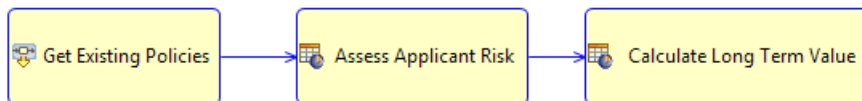
The Ruleflow editor shows the `iSample_policy_pricing.erf` canvas with seven Rulesheets in sequence. The first three apply the risk assessment rules and the other four Rulesheets are in a Subflow to generate a policy based on that assessment. (The Subflow has no processing impact.)

To use this sample to demonstrate Ruleflow within a Ruleflow, perform the following steps:

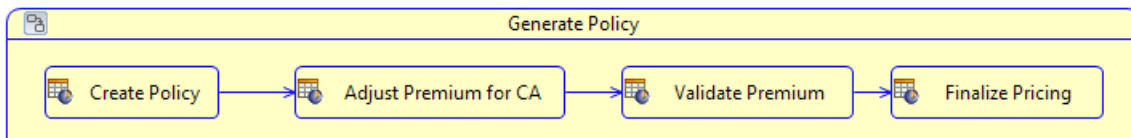
1. In the Life Insurance project in Project Explorer, right-click on `iSample_policy_pricing.erf`, and then choose **Copy**.
2. Right-click on that file again, and then choose **Paste**. Name the file `iSample_AssessRisk.erf`.
3. Again, right-click on that file, and then choose **Paste**. Name the file `iSample_GeneratePolicy.erf`.
4. Open `iSample_AssessRisk.erf`, click on the Subflow, press **Delete**, and then save the file.
5. Open `iSample_GeneratePolicy.erf`, select the three Rulesheets above the Subflow, press **Delete**, and then save the file.

The original Ruleflow has now been cleaved into two separate Ruleflows as shown:

iSample_AssessRisk



iSample_GeneratePolicy



6. In the project, create a new Ruleflow named `iSample_policy_pricing_Ruleflows.erf`.
7. Drag `iSample_AssessRisk.erf` onto the canvas, and then drag `iSample_GeneratePolicy.erf` to its right.
8. Add a Connection from `iSample_AssessRisk.erf` to `iSample_GeneratePolicy.erf`, as shown:

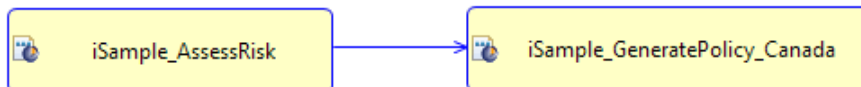


9. Save all the Ruleflow files.

Note: The original files are referenced and you are allowed to browse to choose a different Ruleflow that uses the same Vocabulary. You can change the name of the Ruleflow in this context so that it provides meaning, and you can add comments. None of these actions change the Ruleflow properties of the original Ruleflow.

You can now test the result. Because we only added Ruleflows and changed none of the original project files, you can run Ruletest in the project, alternating the test subject from `iSample_policy_pricing.erf` to `iSample_policy_pricing_Ruleflows.erf` to see that they both produce the same expected output.

Once you have these two Ruleflows, you can update and test each independently, and -- as long as you ensure that the Vocabulary stays consistent -- separate teams can collaborate on developing risk rules and policy rules. That makes it easy to *reuse* either of these Ruleflows. For example, if risk assessment applies to all markets but policy pricing varies, you can create a new Ruleflow that brings in the same AssessRisk module to provide the data to process against a modified policy pricing Ruleflow for another market, as illustrated:



Ruleflow Activity

A Ruleflow file's **Properties** can provide settings for versioning and effective date stamping of the Decision Service that will be created. (See the topic [Ruleflow versioning and effective dating](#) on page 316 for details.) However, when a Ruleflow is added to another Ruleflow's canvas, it ignores its **Ruleflow Properties** and takes on **Ruleflow Activity Properties** that are local to its role as a component of another Ruleflow, as illustrated:

The screenshot shows the 'Properties' window with the 'Ruleflow Activity' tab selected. The 'Name' field is 'iSample_AssessRisk'. The 'Ruleflow' field is '/Life Insurance/iSample_AssessRisk.erf' with a 'Browse...' button. The 'Comments' field is empty.

The parent Ruleflow provides its own settings for versioning and effective date stamping of the Decision Service that will be created, as illustrated:

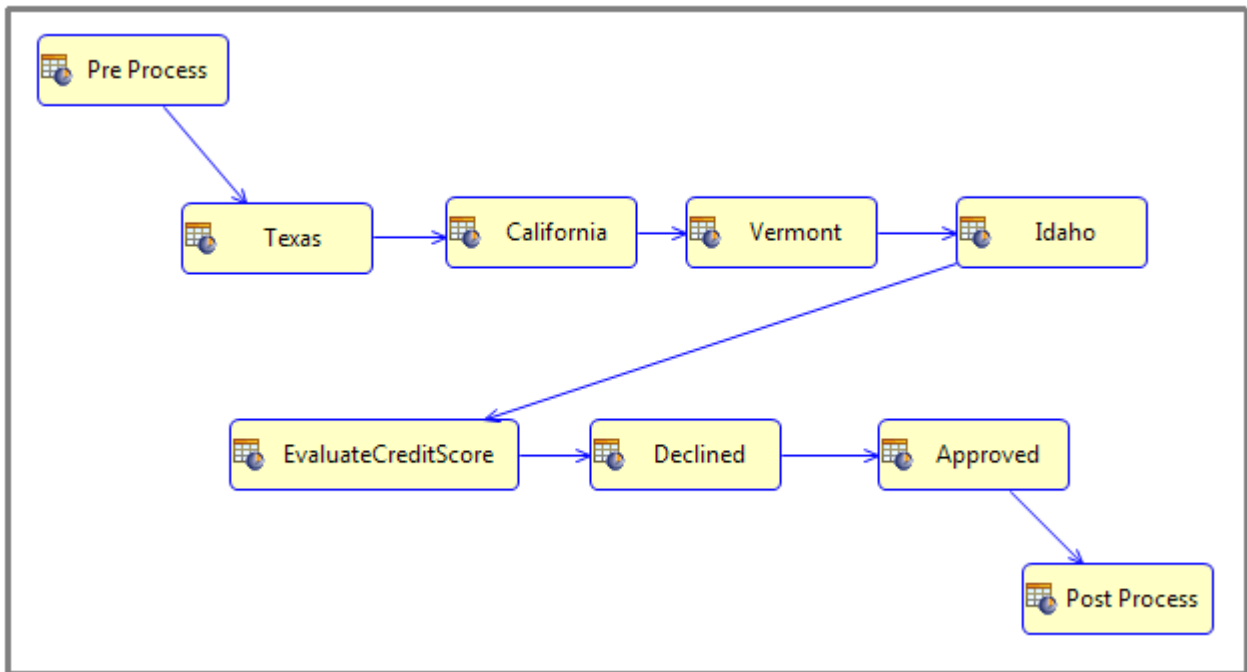
The screenshot shows the 'Properties' window with the 'Ruleflow Activity' tab selected. The 'Rule Vocabulary' field is '/Life Insurance/iSample_Vocabulary.ecore' with a 'Browse...' button. The 'Major Version' is 1, 'Minor Version' is 0, and 'Version Label' is empty. The 'Effective Date' is set to '/'/'/' and 'Time' is 0:00 AM. The 'Expiration Date' is set to '/'/'/' and 'Time' is 0:00 AM. The 'Total Number of Rules' is 33.

Note: Deploying Ruleflows within a Ruleflow - When this Ruleflow is deployed, the generated Decision Service will include the content of both Ruleflows. However, when either of the included Ruleflows changes, Ruleflows that include one of them are not automatically updated -- each must be redeployed to include the changes.

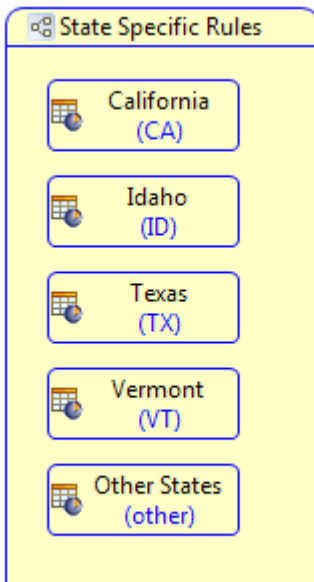
For more information, see *the "Ruleflows" section of the Quick Reference Guide*

Conditional Branching in Ruleflows

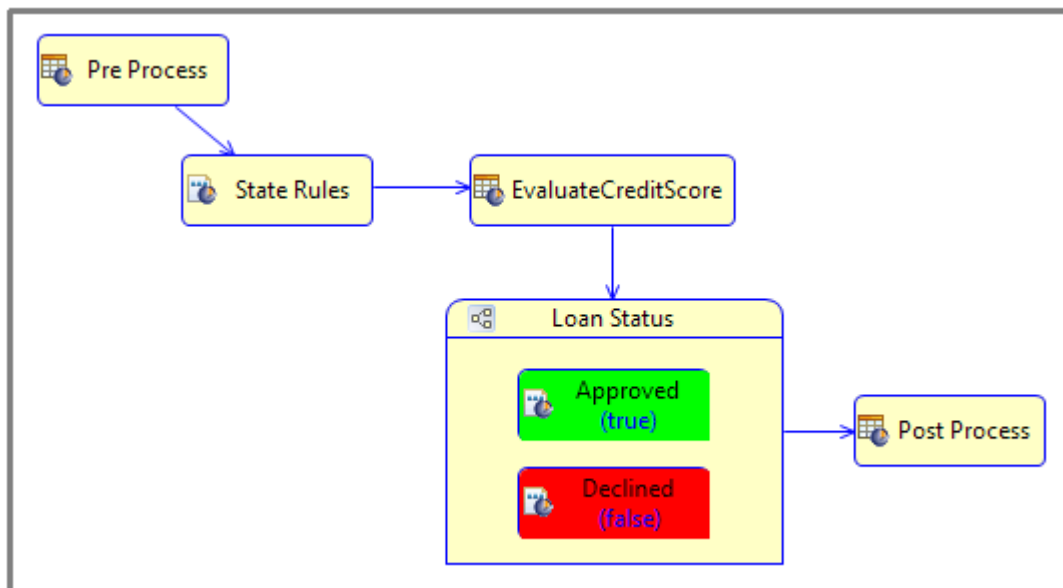
In a Ruleflow, you often have steps which should only process an entity with a specific attribute value. You can accomplish this by using *preconditions* on a Rulesheet, but the resulting logic, or flow, is difficult to perceive when looking at the Ruleflow. The following Ruleflow shows a progression of processing from the upper left to the lower right. But the rules to decide whether a loan is approved or declined are one-or-the-other, and the Rulesheets for the US states do not really represent a progression because the applicant's state is going to trigger only one of these Rulesheets to fire its rules:



Looking at this Ruleflow the real flow is somewhat hidden. If the Rulesheets for Texas, California, Vermont, and Idaho each had a precondition such that only matching states were processed, then they represent a set of mutually exclusive options, not the linear flow depicted in the Ruleflow. We'll see how we can create a branch in a Ruleflow like this:



And then bring that Ruleflow into another Ruleflow where we will also create a branch for the Declined and Approved Rulesheets that also might have needed to use preconditions. The completed Ruleflow looks like this:



A branch node can be Rulesheet, Ruleflow, Service Call Out, Subflow, or another Branch container.

Note: Multiple branches can be assigned to the same target activity. These values are shown as a set in the Ruleflow canvas.

Refresher on enumerations and Booleans

Branching can occur on either enumerated or boolean attribute types. Only these are allowed because they have a set of known possible values. These possible values can be used to identify a branch. Using branches in a Ruleflow lets you clearly identify the set of options, or branches, for processing an entity based on an attribute value. In our example, using branching for the set of state options and whether the loan is approved or declined makes the flow more apparent. It will also be easier to create and maintain.

This topic covers general concepts of branching. First, let's review enumerations and Booleans as they are essential to branching definitions.

When defining elements of a Vocabulary, each attribute is specified as one of seven data types in the Corticon Vocabulary.

Property Name	Property Value
Attribute Name	state
Data Type	String

Boolean
 Decimal
 DateTime
 Date
 Integer
 String
 Time

These can be extended by Constraints or Enumerations. In this illustration, States are extending their String type to be qualified as a list of labels and corresponding values that delimit the expected values yet offer the listed items in pulldowns when you are defining Ruletests. Notice that the Boolean Data Type is not listed as it is implicitly an enumeration.

The screenshot shows the 'Custom Data Types' tab in the Corticon Vocabulary editor. On the left, a tree view shows the hierarchy: mortgage > Applicant > state. The 'state' attribute is selected. The 'Custom Data Types' table has columns: Data Type Name, Base Data Type, Enumeration, and Constraint... The 'States' data type is listed with a base data type of 'String' and an enumeration of 'Yes'. A dropdown menu is open for the 'Base Data Type' column, showing options: Decimal, DateTime, Date, Integer, String (selected), and Time. On the right, a table lists state abbreviations and their corresponding values:

Label	Value
AK	'AK'
AL	'AL'
AR	'AR'
AZ	'AZ'
CA	'CA'
CO	'CO'
CT	'CT'
DC	'DC'
DE	'DE'
FL	'FL'
GA	'GA'
HI	'HI'
IA	'IA'
ID	'ID'
IL	'IL'
IN	'IN'

The Vocabulary definition then chooses the States data type, a subset of String, as its data type.

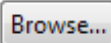
Property Name	Property Value
Attribute Name	state
Data Type	States
	<div> Boolean Decimal DateTime Date Integer String States Time </div>

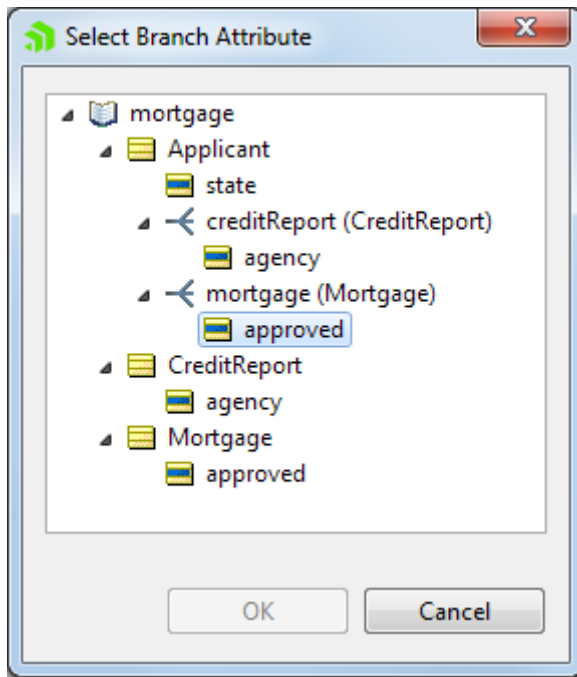
Every attribute that is an enumerated data type or a Boolean is available for branching. For more information, see [Enumerations](#) on page 30 in this guide.

Example of Branching based on a Boolean

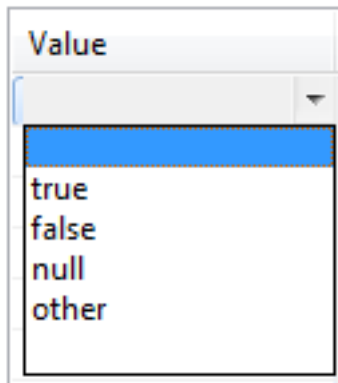
In the example, loan status does not pass through being declined on its way to being approved; it is one or the other. This true/false decision point (actually Ruleflows that might contain several Rulesheets) provides an easy introduction to branching.

To create a Branch on a Ruleflow canvas for a Boolean attribute:

1. On the Ruleflow canvas where you want to create a branch, click **Branch** on the Palette, and then click on the canvas where you want to place the branch. A Branch container is created with your cursor in the name label area.
2. Enter a name such as `Loan Status`, and press **Enter**. You can change the name later.
3. Drag the Rulesheets `Approved.erf` and `Declined.erf` from the Project Explorer to the branch compartment.
4. On the Branch's **Properties** tab for **Branch Activity**, click . The **Select Branch Attributes** for the Ruleflow's Vocabulary identifies three attributes that are candidates for branching (state, agency, and approved), and the associations that apply to these attributes. For this branch, `approved` is the Boolean attribute appropriate for loan status. More specifically the attribute preferred is, `Applicant.mortgage.approved`. Click on that attribute as shown:

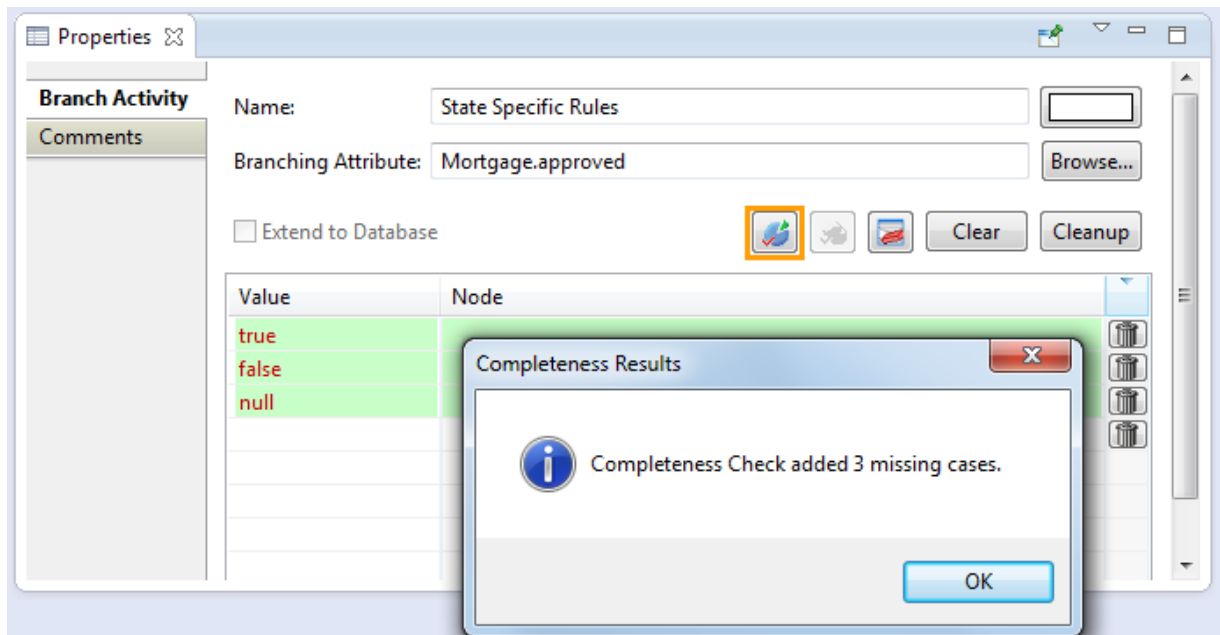


5. Click **OK**.
6. You can define the Boolean branches in a few ways:
 - Click on the **Value** pulldown, as shown:



Notice that there four choices for a Boolean. The null value is offered the attribute is not set as Mandatory so `null` is allowable. The other value is demonstrated below.

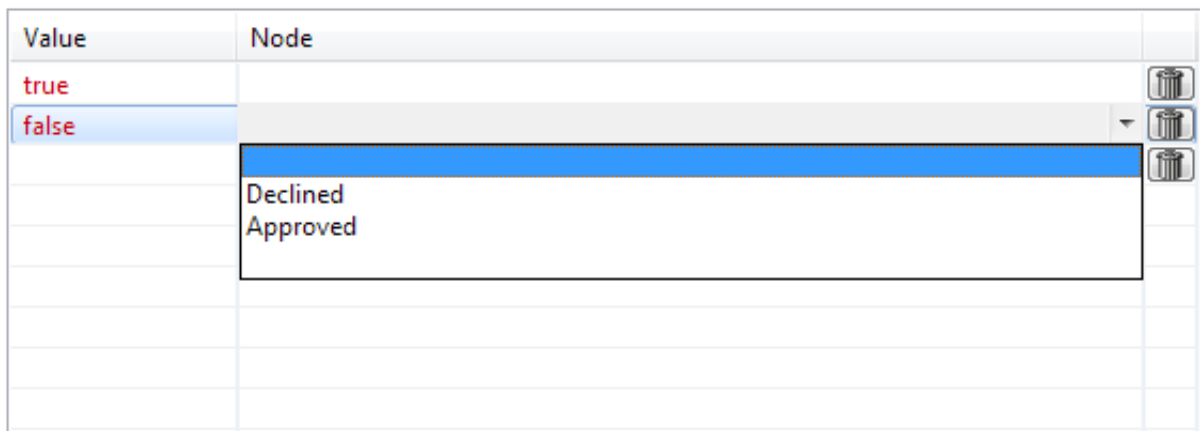
- Choose `true` on the first line and then choose `other` on the second line.
- Click **Check for completeness**, as shown, to populate the **Value** list from the attribute:



Notice that it does not add `other` to the list. If you set `true` and `other` as shown above, clicking **Check for completeness** would have nothing to add because `other` implies completeness. You can clear green highlights by clicking the **Clear analysis results** button.

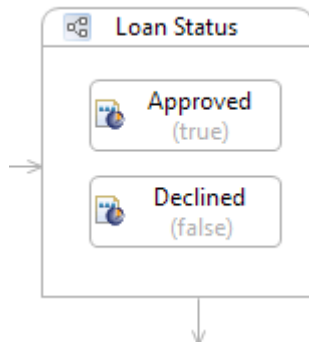
The values listed are in red until we bind each one to a node. You can delete any or all but a minimal number of these lines if you do not have nodes that will handle specific cases. For this example, keep only `true` and `false`. Then click **Cleanup** to remove lines that no assigned node.

7. In the **Branch Activity** section, the **Node** column lets you click on a Value line and then use the pulldown to choose the appropriate target node for the value. When the request in process matches this value, it will be passed to this branch in the branch container:

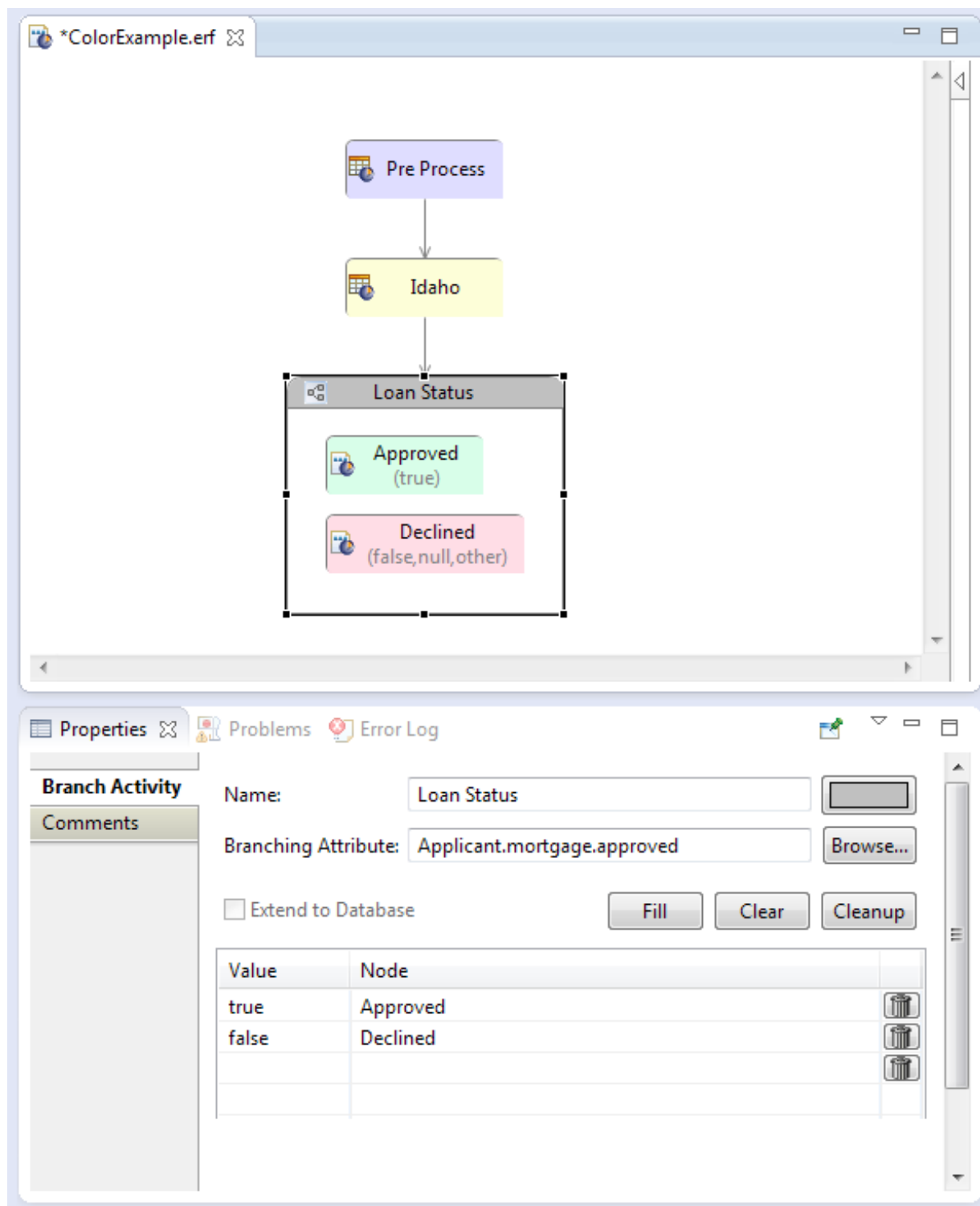


When both `true` and `false` have nodes specified, we have defined the required branches for this rule flow.

8. Connect the incoming and outgoing connections to the branch to complete the flow on the canvas.



Multiple values can direct to the same target node, as shown in this colorized examples, where all the 'not true' possibilities are assigned to the **Declined** node:

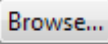


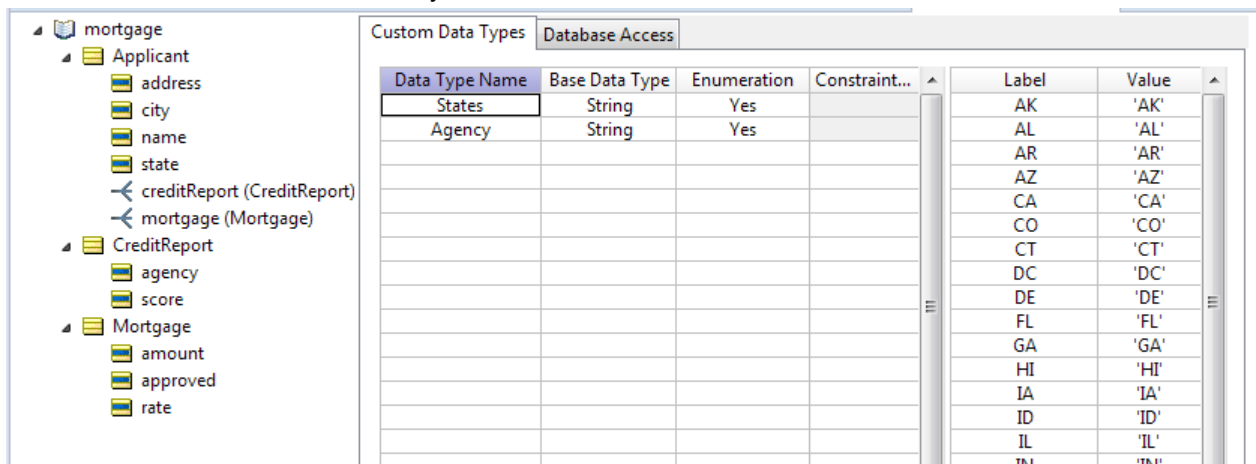
That completes the creation of this Boolean-based branch.

Example of Branching based on an Enumeration

In the example, four US states each have specific rules defined. Processing policy might require graceful rejection of requests that do not specify one of these four states. And, over time, the included states might expand or contract. This branch for State Specific Rules will be created as a separate Ruleflow, `State Rules`, so that it can be reused in other Ruleflows.

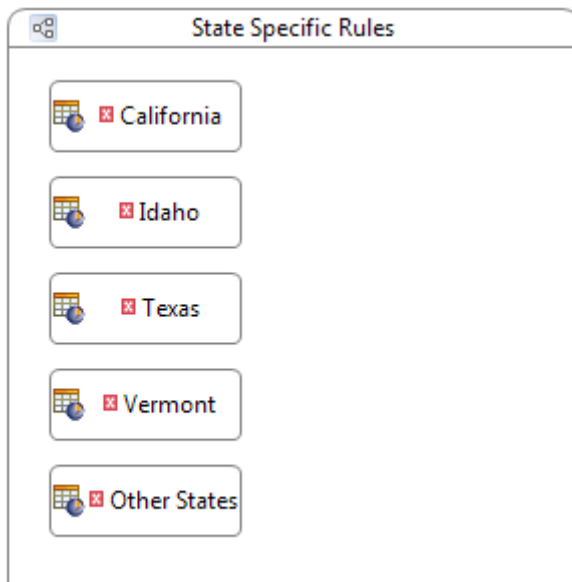
To create a Branch on a Ruleflow canvas for an attribute that is an enumerated list:

1. On the Ruleflow canvas, click **Branch** on the Palette, and then click on the canvas where you want to place the branch. A Branch compartment is created with your cursor in the name label area.
2. Enter a name such as `State Specific Rules`, and press **Enter**.
3. On the Branch's **Properties** tab for **Branch Activity**, click . The **Select Branch Attributes** for the Ruleflow's Vocabulary identifies three attributes that are candidates for branching (state, agency, and approved), and the associations that apply to these attributes.
4. Choose `Applicant.state`. The list of all US state abbreviations that is used by this attribute defines the enumeration in the Vocabulary, as shown:



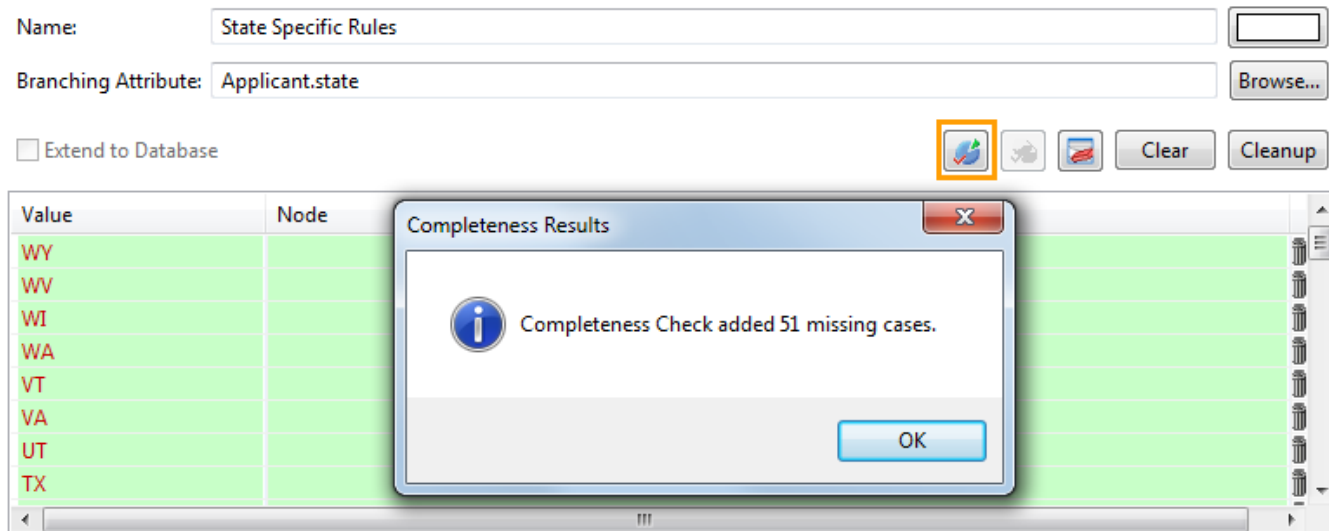
Note: See [Enumerations](#) on page 30 for information on entering or pasting enumeration labels and values as well importing them from a connected database.

5. Drag the Rulesheets `California.ers`, `Idaho.ers`, `Texas.ers`, `Vermont.ers`, and `Other States.ers` into the branch compartment on the canvas. You can use **Ctrl+Click** to select multiples and then drag them as a group. Each Rulesheet is marked with a error flag at this point, as shown:



6. On the canvas, click on the branch to open its **Properties** tab. You can define the enumeration branches in a few ways:

- Click on the **Value** pulldown. On separate value lines, choose each of the defined states and then `other`.
- Click **Check for completeness**, as shown, to populate the **Value** list from the attribute:



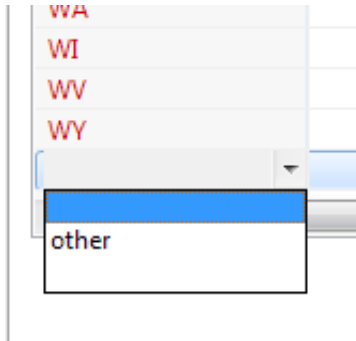
Notice that it does not add `other` to the list. If you set `true` and `other` as shown above, clicking **Check for completeness** would have nothing to add because `other` implies completeness. You can clear green highlights by clicking the **Clear analysis results** button.

The values listed are in red until we bind each one to a node.

7. Click on state value, then use the pulldown to select the appropriate node. In the following image, notice that the California node was assigned to the CA value, so that value turned black, the node on the canvas cleared the error, and the branching value is indicated in parentheses.

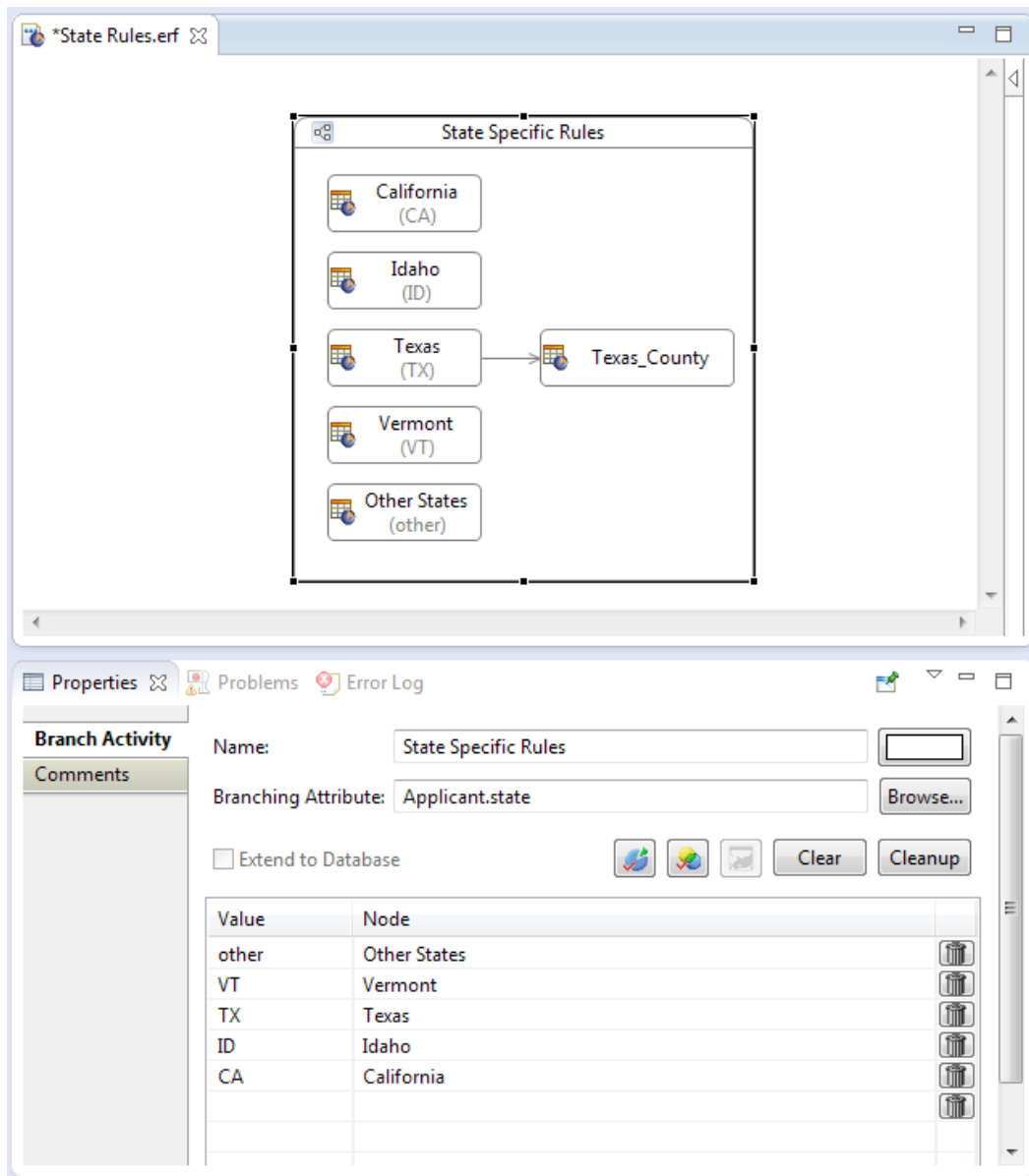
Note: An additional node was added to the canvas but because it is connected to a node, it is not offered in the pulldown list as a branch.

8. After matching the states with appropriate nodes, the `Other States` Rulesheet is unassigned. To handle this, a special purpose value is added. At the bottom of the value list, click on the down arrow and choose `other`.



Assign the `Other States` Rulesheet to that value.

9. Once all the nodes have been assigned to values, click **Cleanup** to clear all the unassigned values, as shown:



The unassigned values that were removed will all be handled by the `other` value's node. If you click **Check for Completeness** now, you will get the branch is complete.

That completes the creation of this enumeration-based branch.

Note: Other features of the user interface for defining Branch Activity are:

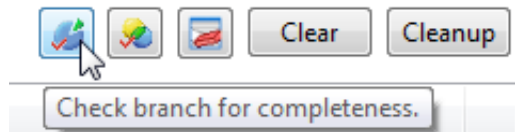
- Clicking a trashcan button on the right side of a branch line, deletes that line.
- Clicking the **Clear** button removes all lines (the branch and components on the canvas are not removed).
- The **Extend to Database** option is offered when the branching attribute is defined to connect to a database table and columns. The option is enabled in the Vocabulary editor by setting the attribute's Entity property **Datastore Persistent** to **Yes**. Choosing the option when it is available pulls the entities out of the defined and connected database and then processes the branch; when cleared, it tests against only the payload.

Logical analysis of a branch container

A Ruleflow branch container is subject to two significant types of logical errors: *completeness* and *conflicts*.

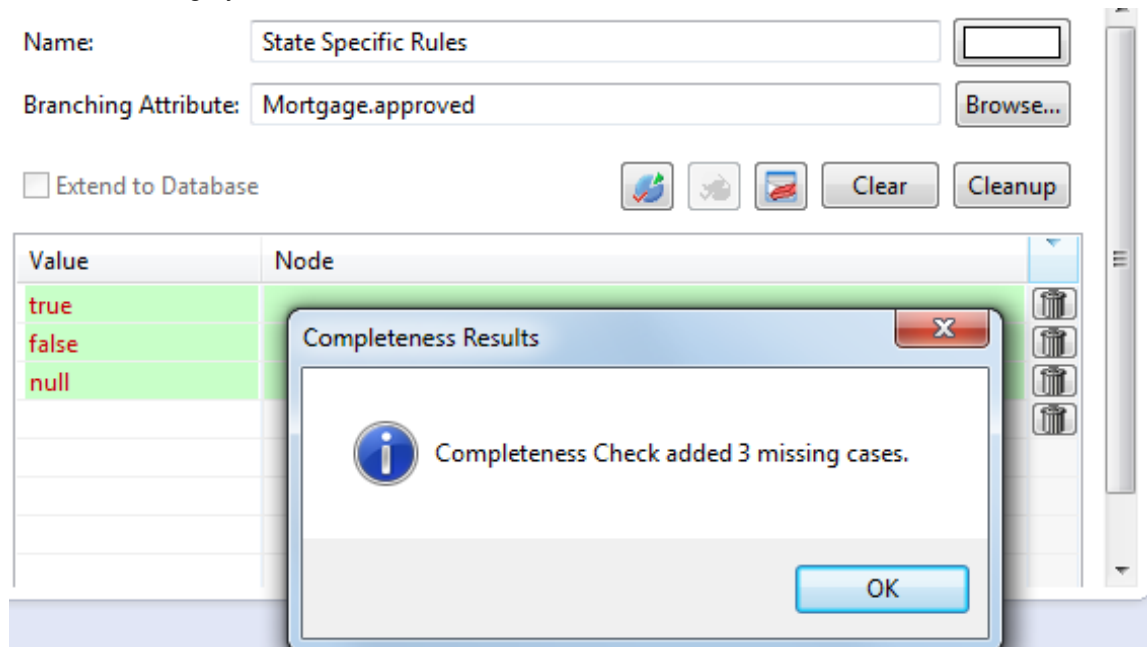
Completeness in a branch

A branch is complete when all of its possible values have been accounted for in branch nodes. When first defining branch activity, instead of selecting each possible value on each line, you can click **Check branch for completeness**, as shown:

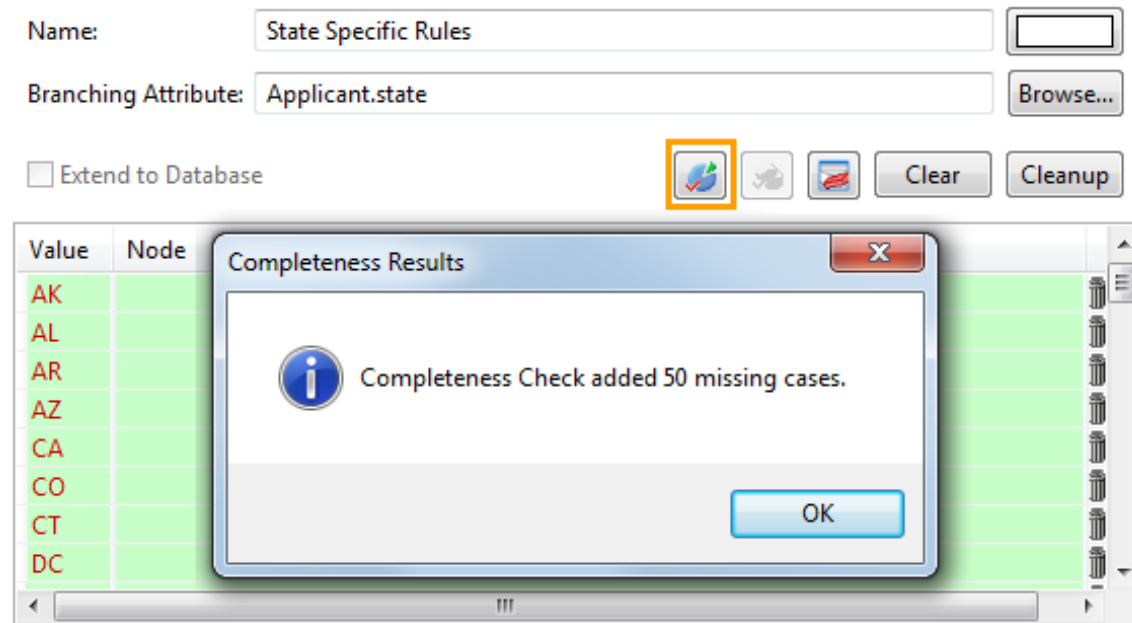


This will add all missing values as branch targets.

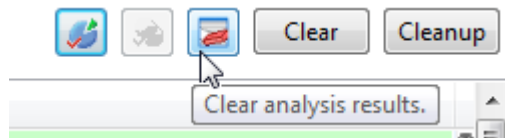
When branching by a Boolean attribute, three values are added, as shown:



When branching by an enumerated Custom Data Type attribute, each label in the enumeration is added, as illustrated:



If the completeness check adds additional branch values, these will be highlighted in green. Clicking **Clear analysis results** removes color highlighting:



Assign nodes in the branch to appropriate listed value values. When you are done, click **Cleanup** to remove any branch values which do not have corresponding branch nodes. Unless you specify the keyword `other` as a branch value and assign it a branch node, your branch would be incomplete; you have not accounted for some of the possible branch values.

Conflicts in a branch

When branch nodes include logic that creates conflicts or ambiguities, those conflicts are difficult to identify. You can evaluate whether there are logical conflicts in a branch by clicking **Check branch for conflicts**, as shown:



Conflict or ambiguity in a Ruleflow branch container might be:

- **Different branches modify a shared entity** - You are informed of the attribute/association being modified.
- **A branch accesses the branch entity through an association that is not being filtered by the branch** - For example, the branch is on `Policy.type` while some rules act on `Customer.policy.type`. That creates a conflicting branch node, each of which is highlighted in red, as illustrated:

The screenshot displays the Ruleflow tool interface with three main panels:

- DirectAccounts.ers**: Shows a condition `Policy.type` highlighted in orange.
- PartnerAccounts.ers**: Shows a condition `Customer.policy.type` highlighted in orange.
- *AccountDistribution.erf**: Shows a branch container named **Accounts** containing two sub-branches: **DirectAccounts (Elite, Preferred)** and **PartnerAccounts (Standard)**.

At the bottom, the **Properties** panel shows the **Branch Activity** for the **Accounts** container. The **Branching Attribute** is set to `Policy.type` (highlighted in orange). Below this, a table lists the values and nodes:

Value	Node
Elite	DirectAccounts
Preferred	DirectAccounts
Standard	PartnerAccounts

The **Standard** row and the **PartnerAccounts** node are highlighted in red, indicating a conflict.

Note: For more about this type of conflict, see the topic, *"How branches in a Ruleflow are processed"*.

Click the **Clear analysis results** button to remove the highlights.

How branches in a Ruleflow are processed

Branch selection

Data is assigned to each branch before any branch execution occurs, so if an attribute in the branch condition changes value during a branch activity execution, it will not change the branch assignment. Further downstream, the new value is presented for subsequent branch activity execution.

Consider the following example. When branching by `Customer.smoker`, the value of `smoker` determines which branch is executed. Changing the value of `smoker` within a branch will not alter which branch processes the Customer.

Suppose you had the payload:

```
Customer 1 (smoker = "Yes")
Customer 2 (smoker = "No")
```

Changing the `smoker` for Customer 1 from "Yes" to "No" would not, within the current branch condition, cause it to be passed to the "No" `smoker` branch. Subsequent branching by `smoker` would use its current value.

Branching by associated attributes

When associations are involved, the data passed into the branch activity is the full association traversal of the branch condition. The entity (with possible associated parents) that satisfies the branch condition is passed into the branch activity. Child associations will be available during activity execution. Unrelated entities are part of the branch payload.

Consider the following example of branching by `Customer.policy.type`. All the policies for an order of some `type` will be passed into the matching branch.

Suppose you had the payload:

```
- Customer 1
- policy 1 (type="standard")
- policy 2 (type="preferred")
- Customer 2
- policy 3 (type="standard")
- policy 4 (type="preferred")
```

The branch for "standard" would be passed:

```
- Customer 1
- policy 1 (type="standard")
- Customer 2
- policy 3 (type="standard")
```

The branch for "preferred" would be passed:

```
- Customer 1
- policy 2 (type="preferred")
- Customer 2
- policy 4 (type="preferred")
```

Branch consistency

When a root entity is used for the branch and the branch activities use associations, care must be taken to ensure consistent results in a Ruleflow branch. It is important to use the same association traversals in the branch Rulesheets as used in the branch attribute. Thus, if the branch Rulesheets reference entities like `Customer.policy.type` and the branch attribute is on entity `policy.type`, the branch attribute in the branch container properties should be defined as `Customer.policy.type`, not `Policy.type`. If the branch container is the root entity `Policy.type`, the branch Rulesheets will still allow for references through the association `Customer.policy.type` to `Policy` entities that did not survive the branch.

Consider the following example of branching on `Policy.type`.

Suppose the payload had `Policy.type`:

```
- Customer 1
- policy 1 (type="standard")
- policy 2 (type="preferred")
- Customer 2
- policy 3 (type="standard")
- policy 4 (type="preferred")
```

The branch for "standard" would be passed:

```
- Policy 1 (type="standard")
- Policy 3 (type="standard")
```

The branch for "preferred" would be passed:

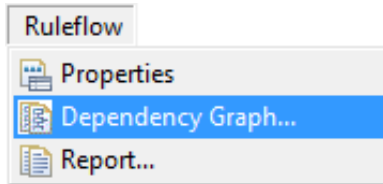
```
- Policy 2 (type="preferred")
- Policy 4 (type="preferred")
```

However in both branches, `Customer 1` and `Customer 2` (with associations) will also be available. So if rules in those branches reference `Customer.policy`, the rules will execute on every `Customer.policy`, not just the branched ones. Since the branch was on `Policy`, rules that reference `Policy` will only execute on the branched ones.

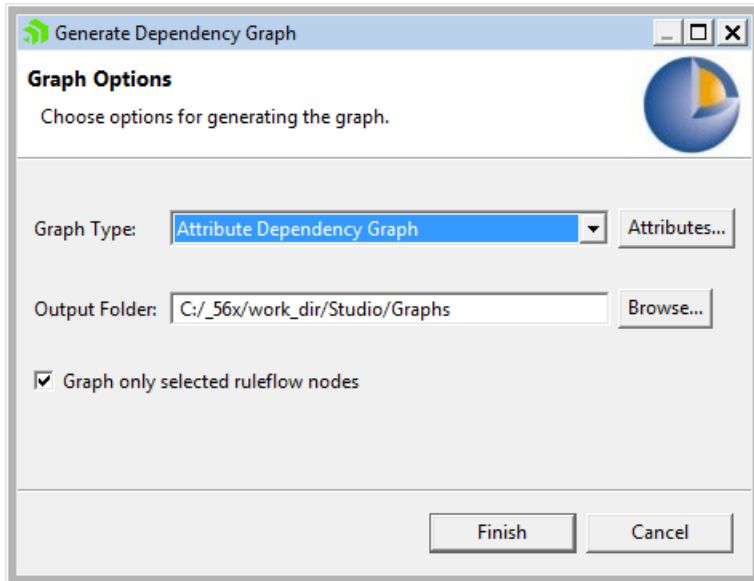
Generating Ruleflow dependency graphs

When working on large Ruleflows you often want to know the dependencies between the nodes in the Ruleflow. This can help you determine how best to order the nodes or detect unanticipated dependencies. Dependencies are identified by the attributes that are set or referenced in the nodes of a Ruleflow. You also often want to know how one or more attributes are used in a Ruleflow. Ruleflow graphing lets you see the dependencies and where attributes are used. This is useful for understanding a Ruleflow, debugging problems, and performing impact analysis when changing a vocabulary.

With the Ruleflow you want to graph open in its Studio editor, select the **Ruleflow** menu command **Dependency Graph**, as shown:



The **Generate Dependency Graph** dialog box opens:



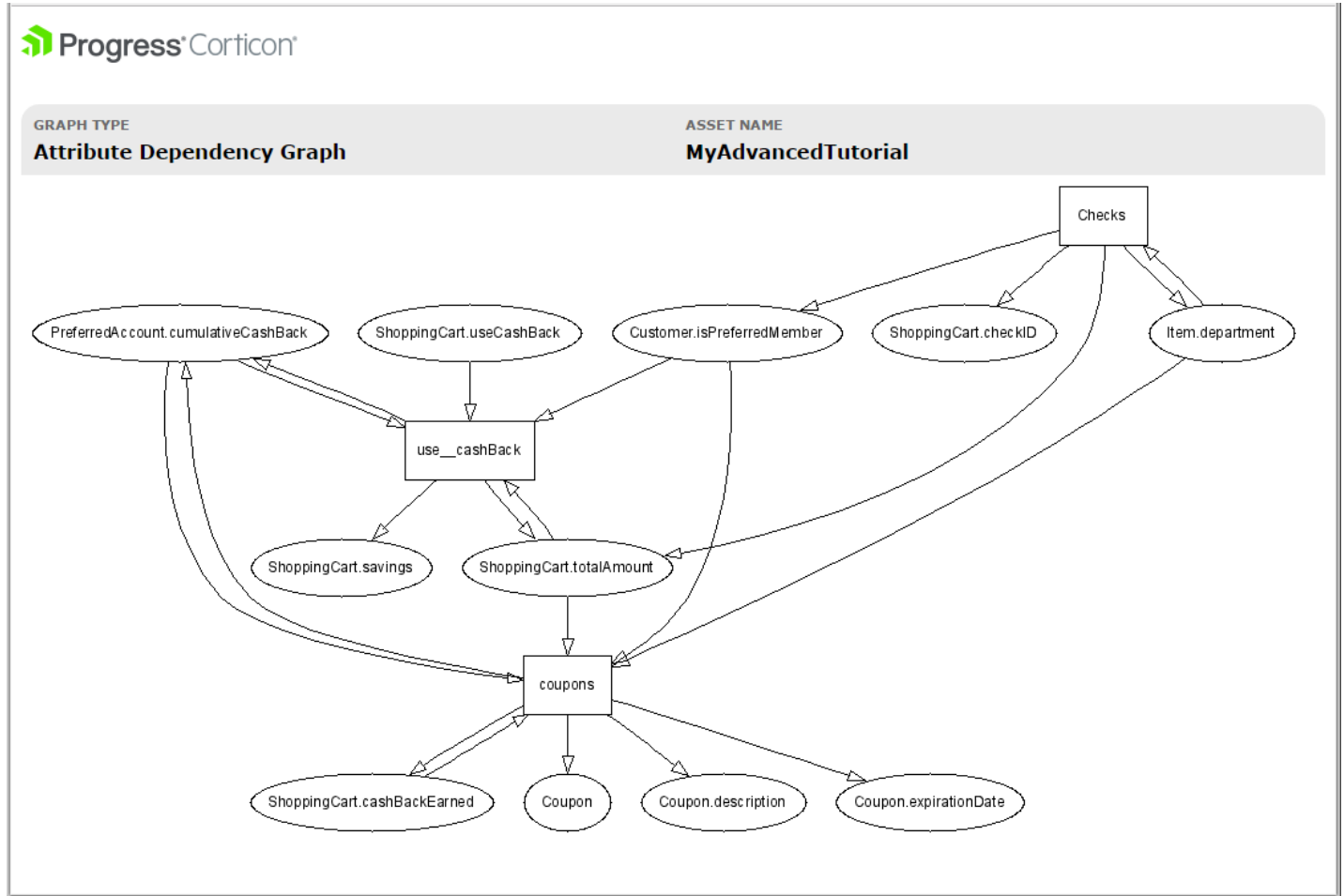
Choose the type of graph you want, and the output folder. You can focus the analysis on just nodes you selected before opening the dialog, or all nodes on the Ruleflow canvas.

Note: When no objects on the Ruleflow canvas are preselected, the option to graph only selected nodes has no effect.

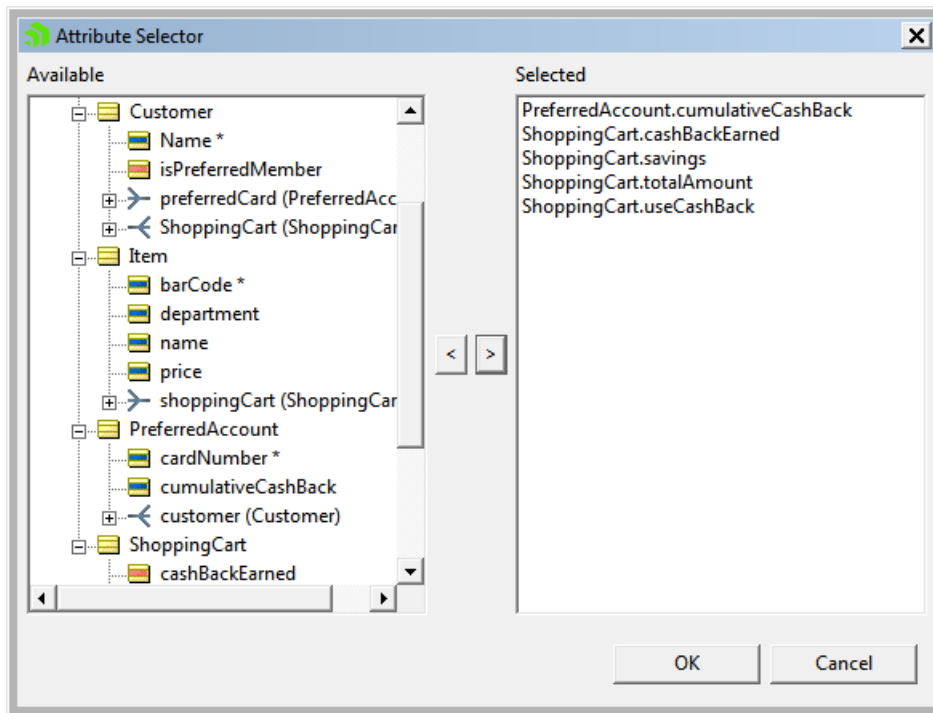
Attribute Dependency Graph

An *attribute dependency graph* shows the attributes that establish dependencies – that is, when a Rulesheet uses an attribute set by another Rulesheet, the former has a dependency on the latter.

When you just generate a graph right away, all the attributes are included, as in this graph of the advanced tutorial's Ruleflow:

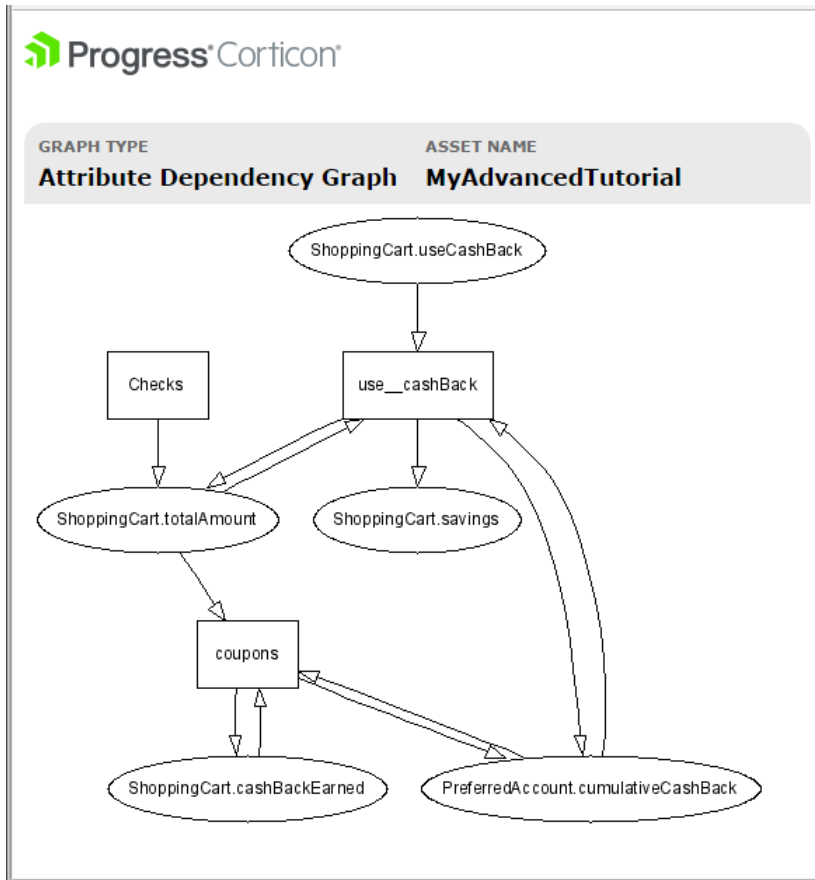


For large projects, graphs with all the attributes and dependencies can be difficult to work with. You can specify that only selected attributes are to be analyzed. Click **Attributes** to open the **Attribute Selector** dialog box, as shown:



In this illustration, five attributes were selected, so clicking **OK** returns to the graph options where clicking **Finish** generates the graph.

The graph opens in your default browser, as illustrated:

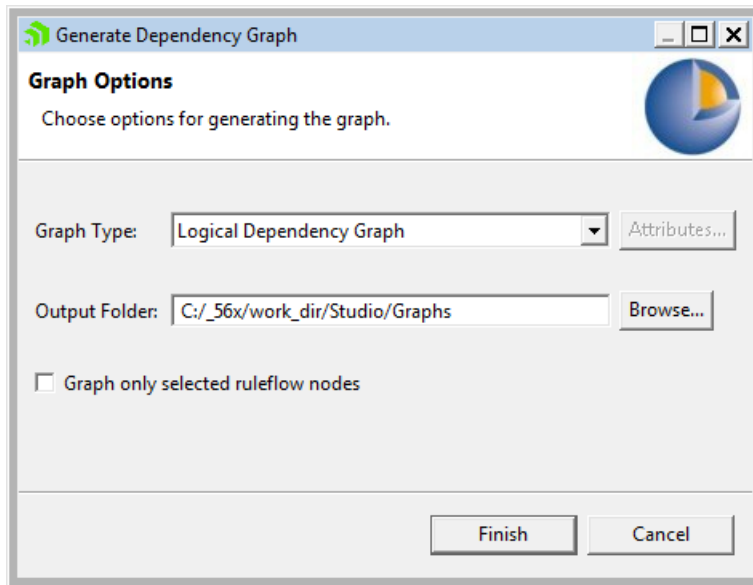


The graph image and its supporting files are saved in the output folder.

Note: When you next generate an attribute graph from the same Ruleflow, it overwrites the existing file unless you relocate generated files or specify unique output folders.

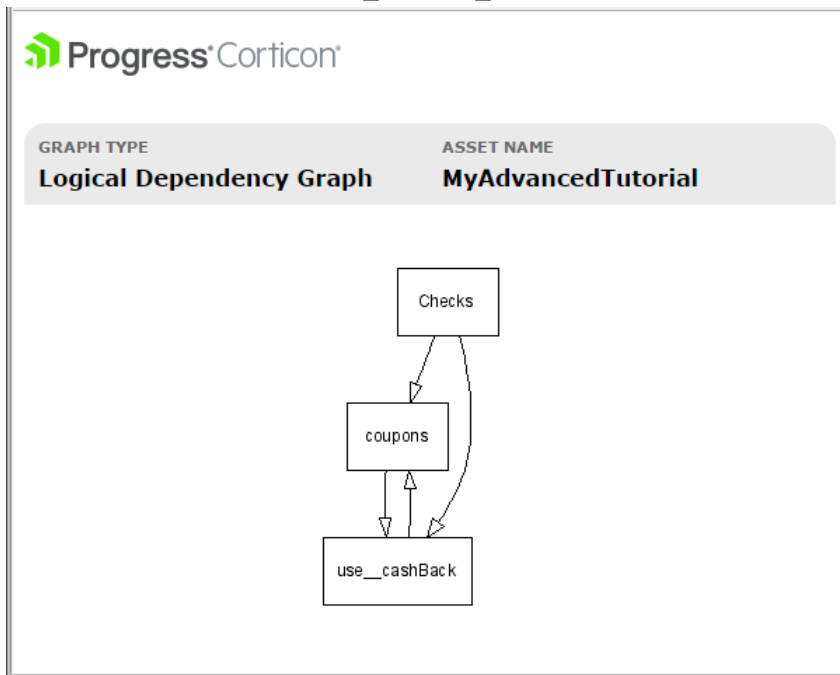
Logical Dependency Graph

A *logical dependency graph* shows the dependency between the Rulesheets in a Ruleflow. Change the graph type to **Logical Dependency Graph**, as shown:



You can set the output folder to your preference and if Ruleflow nodes were selected before opening the dialog box, the analysis is limited to those nodes. The option to specify attributes is not relevant and not available.

Clicking **Finish** generates the graph. The following illustration is the logical dependency graph for the **Life Insurance** sample's `iSample_policy_pricing.erf`:



The graph image and its supporting files are saved in the output folder.

Note: When you again generate a dependency graph from the same Ruleflow, it overwrites the existing file unless you relocate generated files or specify unique output folders.

Ruleflow versioning and effective dating

Setting a Ruleflow version

Major and minor version numbers for Ruleflows 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 244: Assigning a Version Number to a Ruleflow

The screenshot shows the 'Properties' dialog box for a Ruleflow. The 'Ruleflow' tab is selected. The 'Rule Vocabulary' is set to '/Training/Advanced/lifePolicy.ecore'. The 'Work Document Entity' is empty. The 'Major Version' is set to 1, and the 'Minor Version' is set to 0. The 'Version Label' is empty. The 'Effective Date' is set to / /, and the 'Time' is set to 0:00 AM. The 'Expiration Date' is set to / /, and the 'Time' is set to 0:00 AM. The 'Total Number of Rules' is set to 3.

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 discussed 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.

Version numbers can be incremented, but not decremented.

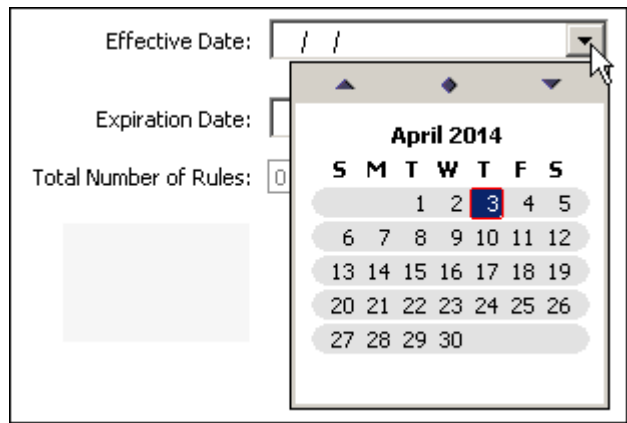
For details on how to invoke a Ruleflow by version number, see the topic *“Decision Service versioning and effective dating” in the Integration & Deployment Guide*.

Setting effective and expiration dates

Effective and Expiration dateTimes are optional for Ruleflows and can be assigned singly or in pairs. When we use different Effective and Expiration dateTimes to describe identically named Ruleflows, the Corticon 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 Corticon 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 245: Setting Effective and Expiration Dates



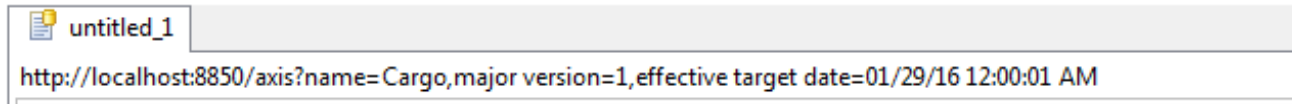
Setting a specific target date for a Ruletest

When you execute a Ruletest against a corresponding Decision Service that is deployed and running on a Corticon Java or .NET Server that was deployed with effective and expiration dates, the day you are testing the Decision Service could be impacted by the data constraints. The ability to set a *target date* lets you execute the test as though it were sent at a specific date and time. Using this feature enables setting the clock back to see how past date ranges would have handled a request, as well as setting the clock forward to test deployed Decision Services in pre-production staging.

To set a version and effective target date for a Ruletest:

1. With the Ruletest in its editor, choose the menu command **Ruletest > Testsheet > Select Test Subject**.
2. Select the **Run against Server** tab, select a Server URL, and then click **Refresh**.
3. Click on a Decision Service in the list.
4. In the **Optional Overrides** section, specify the Decision Service's version identity and effective target date to use for the Ruletest, as illustrated:

5. Click **OK**. The dialog closes. The details of the deployed Decision Service and its overrides are displayed at the top of the Testsheet:



6. Run the Ruletest.

The test executes against the specified Decision Service on the selected server using the overrides you entered.

Test yourself questions: Ruleflow versioning and effective dating

Note: Try this test, and then go to [Test yourself answers: Ruleflow versioning and effective dating](#) on page 358 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 Corticon 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.

Localizing Corticon Studio

Localizing your rule modeling and processing environment involves three related functions:

- Displaying the Studio **program** in your locale of choice. This means switching the Corticon Studio user interface (menus, operators, system messages, etc.) to a new language. Consult with Progress Corticon support or your Progress representative to learn more about available and upcoming Corticon localization packages.
- Displaying your Studio **assets** in your locale of choice. This means switching your Vocabularies, Rulesheets, Ruleflows, and Ruletests to a new language. This part is described here.
- Requests submitted to a Corticon Server can specify an execution property that indicates the **locale of the incoming payload** so that the server can transform the payload's decimal and date values to the decimal delimiter and month literal names of the server, run the rules, and return the output formatted to the submitter's preference. See *"Handling requests and replies across locales" in the Server Integration and Deployment Guide*.

For details, see the following topics:

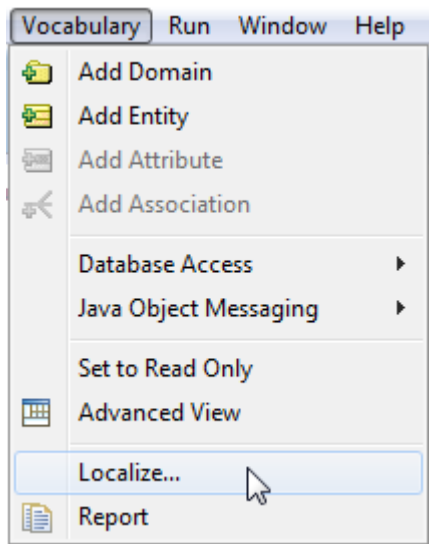
- [Localizing the Vocabulary](#)
- [Localizing the Rulesheet](#)

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:

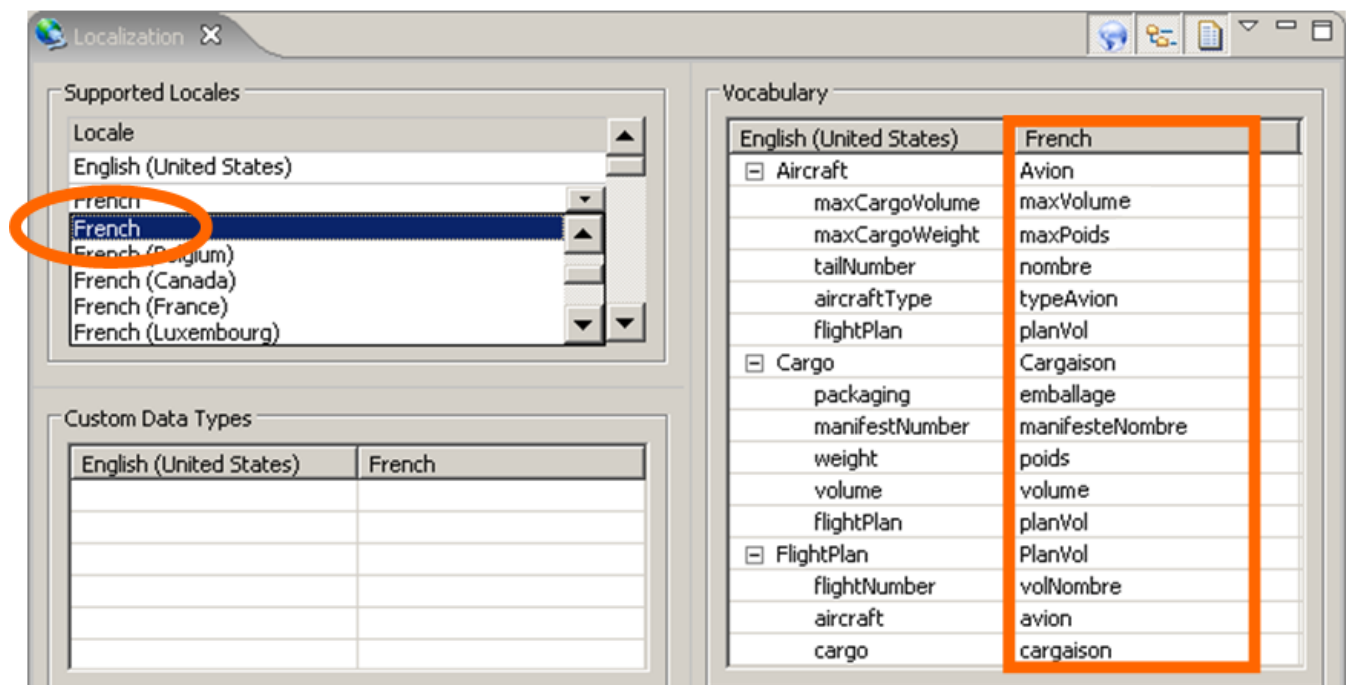
Figure 246: Localize a Vocabulary



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).

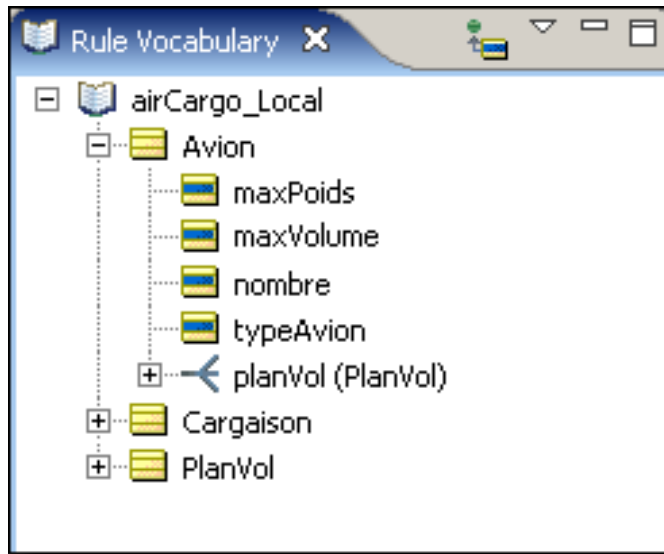
Figure 247: Selecting and populating a second locale



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.

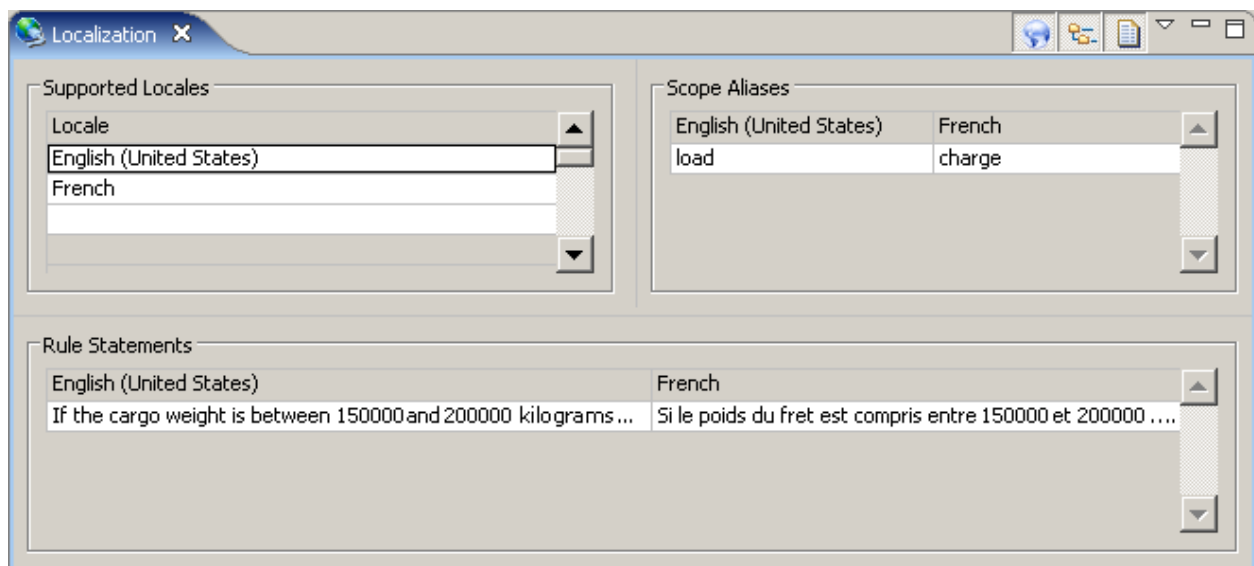
Figure 248: A Vocabulary displaying its French translation



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:

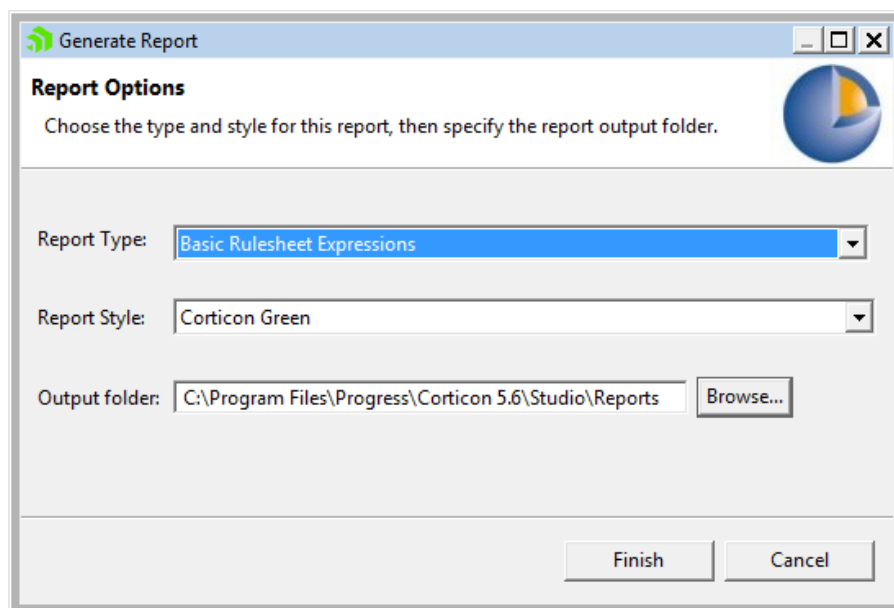
Figure 249: A Rulesheet displaying its French translation



The Corticon Studio reporting framework

Corticon Studio lets you create reports on each of the assets in a project: Vocabulary, Rulesheets, Ruleflows, and Ruletests.

You generate reports from a dialog box that lets you choose from standard types and styles of reports, and then set the output folder, as illustrated here for a Rulesheet:



The standard reports provide summaries at different levels of detail for the different asset types.

For example:

Progress Corticon

REPORT TYPE	PROJECT NAME	REPORT DATE
Basic Rulesheet Natural Language	Tutorial	2016-09-14

ASSET FILE NAME
.../Tutorial/Cargo.ers

COMMENTS
The basic rules for determining the appropriate container for a package.

► SCOPE

▼ FILTERS
Reject any package that exceeds the assigned aircraft weight capacity

RULES

#	Conditions (incl. Values)	Actions (incl. Values)	Rule Statements	Alias	Overrides
1	What is the weight (in kilograms) of the package? <= 20000	Then use this type of container... standard	INFO: Cargo weighing <= 20,000 kilos must be packaged in a standard container.	Cargo	
2	What is the volume (LxWxH in cubic meters) of the package? > 30	Then use this type of container... oversize	INFO: Cargo with volume > 30 cubic meters must be packaged in an oversize container.	Cargo	1
3	What is the weight (in kilograms) of the package? > 20000 What is the volume (LxWxH in cubic meters) of the package? <= 30	Then use this type of container... heavyweight	INFO: Cargo weighing > 20,000 kilos, with volume <= 30 cubic meters, must be packaged in a heavyweight container.	Cargo	

The standard report types are:

Report Type one of the XSLT files for the asset type:

- **Vocabulary**
 - Basic Vocabulary
 - Detailed Vocabulary
- **Rulesheets**
 - Basic Rulesheet Expressions
 - Basic Rulesheet Natural Language
 - Detailed Rulesheet Expressions
 - Detailed Rulesheet Natural Language
- **Ruleflows**
 - Basic Ruleflow Expressions
 - Basic Ruleflow Natural Language
 - Detailed Ruleflow Expressions
 - Detailed Ruleflow Natural Language
- **Ruletests**
 - Basic Ruletest

The type files are located at [CORTICON_WORK_DIR]\Studio\Reports\XSLT\ in folders according to the asset types. You can copy the files to use as templates or change them to create report types that are then offered in the **Report Type** dropdown menu for the asset type.

Report Style is the CSS stylesheet to use for the report. The basic stylesheets are:

- Corticon Blue
- Corticon Green

The style files are located at [CORTICON_WORK_DIR]\Studio\Reports\CSS\. You can copy a stylesheet file to use as a template to create custom report styles that are then offered in the **Report Style** dropdown menu.

Output Folder is the location where the report will be stored on disk. The default location is [CORTICON_WORK_DIR]/Studio/Reports. You can create a root location such as C:\CorticonStudioReports and then append subfolder names to sort out your projects, tasks, clients, or versions.

When you have set the parameters of the report, and then click **Finish** the processing takes place as follows:

1. Generates an XML file using its built-in XML template
2. Uses the selected XSLT transformation to move the XML into HTML
3. Applies the selected CSS to render the HTML file in a web browser page
4. Copies the XML, HTML, and CSS files to the specified output folder.

Customizing the XSLT stylesheet transforms

Corticon Studio's initial XSLT stylesheets can be modified or copied to create preferred XSLT files to generate custom Studio reports. Name and save your modified XSLT file in the appropriate XSLT subfolder so that it will be listed and callable when you run reports.

Customizing the CSS stylesheets

Corticon Studio's initial CSS stylesheet can be modified or copied to create a preferred CSS files to render standard and custom Studio reports. Name and save your file in the CSS folder so that it will be listed and callable when you run reports.

See also:

- *"Creating a Vocabulary report" section of the Quick Reference Guide*
- *"Creating a Rulesheet report" section of the Quick Reference Guide*
- *"Creating a Ruleflow report" section of the Quick Reference Guide*
- *"Creating a Ruletest report" section of the Quick Reference Guide*

Applying logging and override properties to Corticon Studio and its built-in Server

The Studio and its built-in Corticon Server can change behaviors based on properties that you can customize in the **Override properties file** `[CORTICON_WORK_DIR]/brms.properties`. The most commonly used properties that are overridden by Studio users are provided so you can just uncomment them. You can specify name-value pairs for any setting described in the section *"Configuring Corticon properties and settings" in the Integration and Deployment Guide*. If you later want to revert to the original Server behavior, you can change your settings, comment out all your overrides, or even clear all the content of the override properties file. Also see the topic *"Changing logging configuration" in the Using Corticon Server logs section of Integration and Deployment Guide*.

Applying override changes

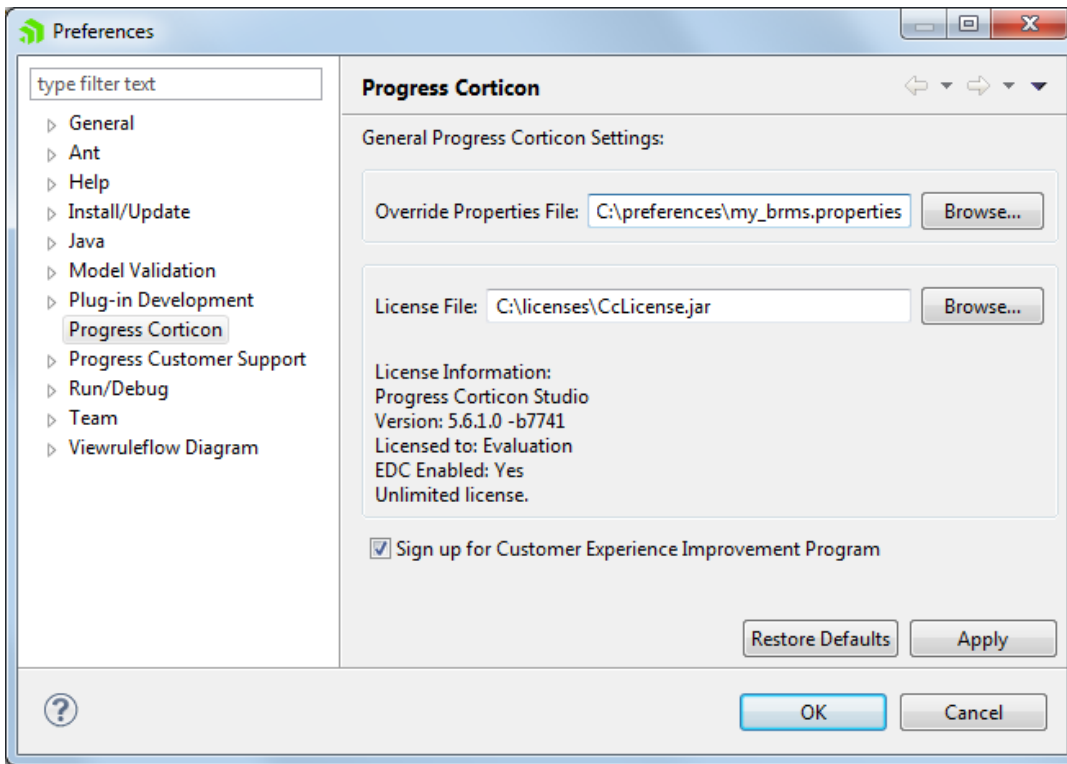
When you save the changed settings files, and then restart Studio, your preferred settings are in effect.

Using an alternate location for the override properties file

You can choose to point Corticon Studio to a preferred location of the `brms.properties` file.

To set an alternate location of the override properties in Studio:

1. The file must exist before you can point to it. You can copy the default file located at the Server's installation root, or you can just create a new file and give it your preferred name, such as `my_brms.properties`. Save the file at your preferred location, such as `C:\preferences\`.
2. In Studio, choose **Window > Preferences**, and then click on **Progress Corticon**.
3. Enter or browse to the override properties file you created, as shown:



When Corticon starts up, it will read your properties file last, and thus override default settings with your settings.

Note: Property settings you list in your `brms.properties` *replace* corresponding properties that have default settings. They do not *append* to an existing list. For example, if you want to add a new `DateTime` mask to the built-in list, be sure to include *all* the masks you intend to use, not just the new one. If your `brms.properties` file contains only the new mask, then it will be the only mask Corticon uses.

After editing and saving your overrides file, restart Corticon components for their changes to take effect. The properties in these files are described in detail in the following topics.

Troubleshooting Rulesheets and Ruleflows

In addition to being a convenient way to test your Rulesheets 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?](#)
- [Using Corticon Studio to reproduce the behavior](#)
- [Test yourself questions: Troubleshooting rulesheets and ruleflows](#)

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 Rulesheets (.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” Ruletests (.ert files) for the Corticon Rulesheets and Ruleflows 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 you run a Corticon Ruletest, it might produce error messages in Corticon Studio. Error messages are distinct from **Post** messages you specified in Rulesheet **Rule Statements** to generate info, warning, and violation statements that are 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 these techniques, 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 Constraint Violation

Figure 250: A Constraint Violation in a Ruletest

Severity	Message
Violation	An unexpected error occurred in Input Data: com.corticon.cdo.ConstraintViolationException: constraint violation setting Cargo.volume to value [-1]

A constraint violation indicates that values in test attributes are not within numeric constraint ranges or not included in enumerated lists that have been set in the Vocabulary's Custom Data Types. In the following example, the constraint is shown, and its violation is marked on the attribute and its entity in the Input column:

Constraint Expression	 Cargo [1]
value >=1	 volume [-1]
	 weight [100]

Running the test halts at the first constraint violation. The log lists the first constraint exception and its detailed trace. No response is generated.

You can revise the input to have valid values, or choose to relax enforcement of such violations through a setting in the Corticon `brms.properties` file,
`com.corticon.vocabulary.cdt.relaxEnforcement=true`.

When the option is enabled, a response is generated that includes each of constraint violation warnings. For example:

```
<CorticonResponse xmlns="urn:Corticon"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  decisionServiceName="Cargo.ers_null_ALL">
  <WorkDocuments>
    <Cargo id="Cargo_id_1">
      <weight>0</weight>
      <volume>-1</volume>
      <container>standard</container>
    </Cargo>
  </WorkDocuments>
  <Messages version="0.0">
    <Message postOrder="cc00000001">
      <severity>Warning</severity>
      <text>constraint violation setting Cargo.weight to value [0]</text>
      <entityReference href="Cargo_id_1" />
    </Message>
    <Message postOrder="cc00000002">
      <severity>Warning</severity>
      <text>constraint violation setting Cargo.container to value [secure]</text>
      <entityReference href="Cargo_id_1" />
    </Message>
    <Message postOrder="cc00000003">
      <severity>Warning</severity>
      <text>constraint violation setting Cargo.volume to value [-1]</text>
      <entityReference href="Cargo_id_1" />
    </Message>
    <Message postOrder="cc00000004">
      <severity>Info</severity>
      <text>[1] Cargo weighing <= 20,000 kilos must be packaged in a standard
container.</text>
      <entityReference href="Cargo_id_1" />
    </Message>
  </Messages>
</CorticonResponse>
```

See [Relaxing enforcement of Custom Data Types](#) on page 42 for details about constraints and the option to relax enforcement.

Note: The output example shown reflects the execution properties in a Ruletest output file. If you extract the same response for a Ruletest from your Studio log when the RULETRACE logging filter is enabled, you would see several additional execution properties such as `inputTransients`, `returnTransients`, `invokedByTester`, `inBytes`, `newOrModified`, `manuallyCreated`, `dataSource`, `new_or_modified_attributes`, `new_or_modified_association_ids`. These properties can be helpful in support efforts but they are otherwise not meaningful to users and should never be included in requests (and will not be in responses) other than the logs of those run as Ruletests. When you export files through the **Rulesheet > Testsheet > Data** commands, these internal properties are removed.

The Null Pointer Exception

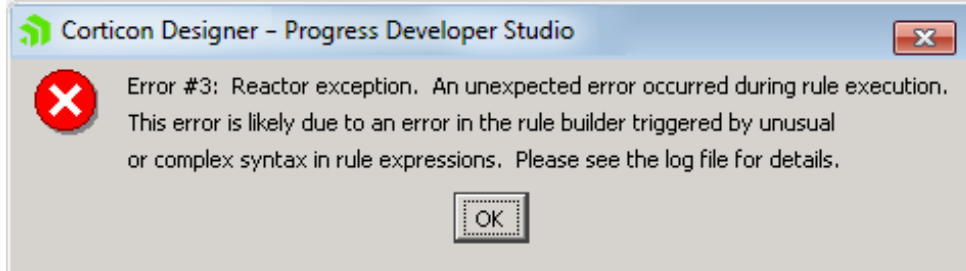
Figure 251: A Null Pointer Exception as Shown in a Ruletest

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 252: The Reactor Exception Window

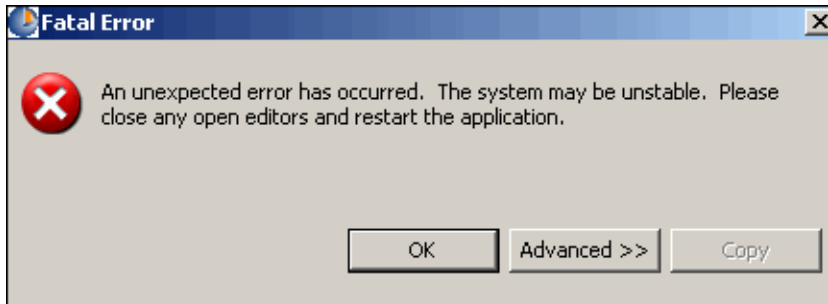


This error is also indicative of a possible rule engine problem, but it may also indicate improper or unnecessarily complex language usage in your rule expressions. The Rule Language's flexibility may permit workarounds to the limitation(s) that produced this message; Progress Corticon Technical Support 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

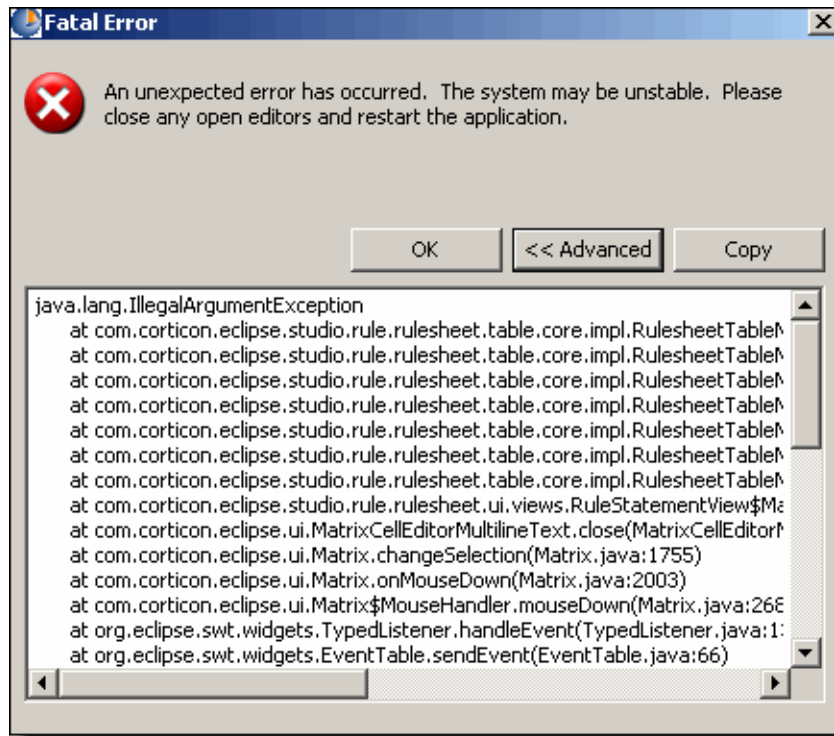
On very rare occasions, you may experience a full crash of the Corticon Studio application. In these cases, you will see a window like the following:

Figure 253: 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 254: 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 Corticon Server deployment.

The Corticon Server log captures errors and exceptions caused by certain rule and request errors. These log messages are detailed in the *Using Corticon Server logs section of the Integration and Deployment Guide*.

- If not, the problem is with the rules themselves. Continue in this section.

Tracing rule execution

A first step in analyzing results of executing Decision Services is to gain visibility to the rules that fired. With rule tracing, you can see what rules and Rulesheets fired in processing a work document. Rule tracing can be used during development time in the Studio Tester or with deployed Decision Services. The following example uses the Advanced Tutorial's Ruleflow as the test subject. The Ruleflow has three Rulesheets, each with conditional and non-conditional rules.

Figure 255: Rule messages when metadata is enabled in Studio

Severity	Message	Entity
Info	[Checks,2] The customer is a Preferred Cardholder	Customer[1]
Info	[coupons,2] \$2 off next purchase when 3 or more Soda/Juice items are purchased in a single visit.	ShoppingCart[1]
Info	[coupons,3] 10% off next gas purchase when total is over \$75.	ShoppingCart[1]
Info	[coupons,B0] \$1.649800 cashBack bonus earned today, new cashBack balance is \$10.889800.	ShoppingCart[1]
Info	[use__cashBack,1] cashback.bonus has been deducted from the total. New total = \$71.600200. Today's savings = \$10.889800.	ShoppingCart[1]

The metadata can be expressed in natural language, such as, for line 1: "In Rulesheet Checks.ers, rule 2 generated this statement", and, for line 4 "In Rulesheet coupons.ers, line B's non-conditional (column 0) action generated this statement."

To enable this function, rule statement metadata, set the `CcServer` property as
`com.corticon.reactor.rulestatement.metadata=true`

Note: It is recommended that you create or update the standard last-loaded properties file `brms.properties` to list override properties such as this for Corticon Studios and Servers. See the introductory topics in *"Configuring Corticon properties and settings" in the Server Integration and Deployment Guide* for information where to locate this properties file.

Note: For Decision Service deployments, set this property where the Deployment Console runs, and then regenerate each ESD file in the Deployment Console. For CDD deployments, set this property on the Server, and then regenerate and compile each Decision Service.

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.

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 Rulesheets. 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.

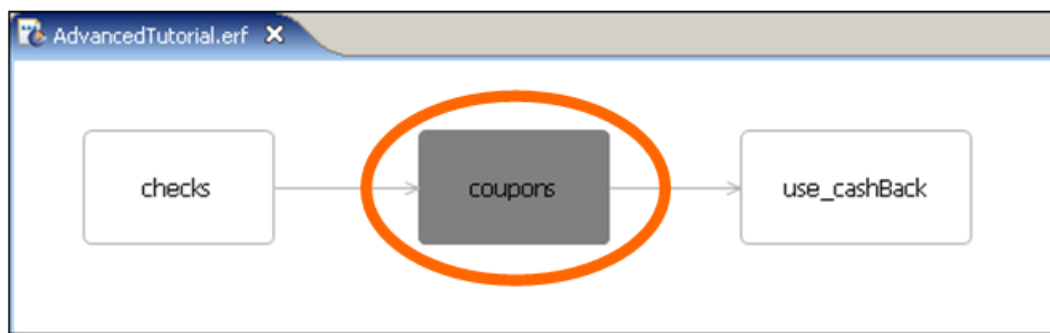
Figure 256: Rulesheet with Rule Column #2 Disabled.

Conditions		0	1	2
a	Cargo.weight	-	150000 .. 200000	-
b	Cargo.volume	-	-	< 300
c				
d				

Actions				
Post Message(s)				
A	Cargo.packaging		Container	Pallet
B				

Ref	ID	Post	Alias	Text
1		Info	Cargo	Cargo weighing between 150,000 and 200,000 pounds must be packaged in a container
2		Info	Cargo	Cargo with a total volume less than 300 cubic yards must be packaged on a pallet

Figure 257: 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.

Incorrect results in Studio

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.

Partial rule firing

A Condition/Action rule column might 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.

There are two exceptions to this rule:

Nonconditional actions

In the special Nonconditional rule column, column 0, each Action row in column 0 acts as its own separate, independent rule, so Action row A may fire even if Action row B does not.

Partial execution of rules with relationships and null attributes

When a *relationship* is null, the rule does not fire. When an *attribute* is null, and the relationship aspects of the rule can be evaluated, the rule fires partially: The actions related to the association do fire but the action related to a null attribute does not. Consider a Rulesheet and test on the Cargo sample where the `Aircraft` information is set from its ID, and the total cargo weight computed. If there is no associated `Aircraft.flightPlan` or `Aircraft.flightPlan.cargo` in the test, then the rule does not execute (even though those associations are not referenced in the Rulesheet's Conditions section). However, if the associations exist but the attribute `Aircraft.flightPlan.cargo.volume` is null, the rule *does* fire partially -- all the `Aircraft` values are computed but the weight is not computed from the null value of the attribute.

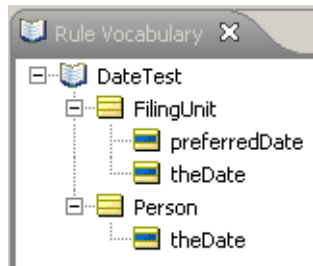
Initializing null attributes

Attributes that are used in calculations must have a non-null value to prevent test rule failure. More specifically, attributes used on the right-hand-side of equations (that is, an attribute on the right side of an assignment operator, such as `=` or `+=`) are *initialized* prior to performing calculations. It is not necessary for attributes on the left-hand-side of an equation to be initialized -- it is assigned the result of the calculation. For example, when you are calculating `Force=Mass*Acceleration`, you must provide values for `Mass` and `Acceleration`. `Force` is the result of a valid calculation.

Initialization of attributes is often performed in Nonconditional rules, or in rules expressed in Rulesheets that execute beforehand. That was often because an Attribute that was set to Transient mode could not be added as Input to Ruletests. The limitation has been removed: You can add Transients to the Input column of a Ruletest. Then, as stated, you must provide a value to such attributes in their Input locations in Ruletests to enable valid firing of the rule.

Handling nulls in compare operations

Unless the application that formed the request ensured that a value was provided before submission, one (or both) of the attributes used in a comparison test might have a null value. You might need to define rules to handle such cases. An example that describes the workaround for these cases uses the following Vocabulary:



Here are two scenarios:

1. Two dates are passed from the application and one of them is null. When given the rule ``[If FilingUnit.theDate is null] or [[FilingUnit.theDate = Null] and [FilingUnit.theDate >= Person.theDate]]'`, then the appropriate action triggers.
2. In Actions, one date value is set to another date's value which happens to be null. If the date is null, then it is used in the subsequent Rulesheets in their Conditions section. However, since the value is null, a warning will be generated in the Corticon logs.

For the first scenario, the logic in subsequent Rulesheets needs to determine whether a value is null, so it can apply appropriate actions. The following Rulesheet shows that you can avoid the error message by only setting the preferred date when you have a non-null filing date or person date.

Conditions		0	1	2	3	4	5
a	FilingUnit.theDate = null		T	F	T	F	F
b	Person.theDate = null		F	T	T	F	F
c	FilingUnit.theDate >= Person.theDate		-	-	-	T	F
d							
e							
f							
Actions							
Post Message(s)							
A	FilingUnit.preferredDate = FilingUnit.theDate						
B	FilingUnit.preferredDate = Person.theDate						
C							
D							
Overrides							
<div> <div>Rule Statements</div> <div>Rule Messages</div> <div>Natural Language</div> <div>Properties</div> <div>History</div> </div>							
Ref	Post	Alias	Text				
1	Warning	FilingUnit	Filing unit date is null - use person date as the preferred date				
2	Warning	Person	Person date is null - use filing unit date as the preferred date				
3	Violation	FilingUnit	Both dates are null - unable to determine preferred date				
4	Info	FilingUnit	Filing data is greater than or equal to the person date - use filing date				
5	Info	FilingUnit	Filing date is less than person date - use person date				

Note: If null values would prevent subsequent rules from continuing reasonable further processing, then perhaps validation sheets should be used before rule processing to check the data, and then terminate execution of the decision if the data is bad. That could be accomplished by setting an attribute that can be tested in the filter section of subsequent Rulesheets. Then, every subsequent Rulesheet is assured of dealing only with "clean" data.

For the scenario where both values being compared are null, you could set the resulting value to a default value or to null, as shown here:

Conditions		0	1	2	3	4	5
a	FilingUnit.theDate = null		T	F	T	F	F
b	Person.theDate = null		F	T	T	F	F
c	FilingUnit.theDate >= Person.theDate		-	-	-	T	F
d							
Actions							
Post Message(s)							
A	FilingUnit.preferredDate = FilingUnit.theDate						
B	FilingUnit.preferredDate = Person.theDate						
C	FilingUnit.preferredDate = null						
D							
Overrides							
<div> <div>Rule Statements</div> <div>Rule Messages</div> <div>Natural Language</div> <div>Properties</div> <div>History</div> </div>							
Ref	Post	Alias	Text				
1	Warning	FilingUnit	Filing unit date is null - use person date as the preferred date				
2	Warning	Person	Person date is null - use filing unit date as the preferred date				
3	Violation	FilingUnit	Both dates are null - set preferred date to null				
4	Info	FilingUnit	Filing data is greater than or equal to the person date - use filing date				
5	Info	FilingUnit	Filing date is less than person date - use person date				

As highlighted, Rule 3 explicitly sets the preferred date to null when both incoming dates are null.

Test yourself questions: Troubleshooting rulesheets and ruleflows

Note: Try this test, and then go to [Test yourself answers: Troubleshooting rulesheets](#) on page 358 to correct yourself.

1. Troubleshooting is based on the principle that Rulesheets 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 are the Corticon Studio logging features set?
11. Where are the Corticon Studio override properties set?
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

Standard Boolean constructions

For details, see the following topics:

- [Boolean AND](#)
- [Boolean NAND](#)
- [Boolean OR](#)
- [Boolean XOR](#)
- [Boolean NOR](#)
- [Boolean XNOR](#)

Boolean 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

PolicyApplicantBoolean.ers		0	1	
Conditions				
a	Applicant.age >=45	-	T	
b	Applicant.smoker	-	T	
Actions				
Post Message(s)				
A	Applicant.riskRating		'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

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:

PolicyApplicantBoolean.ers		0	1	2	3	4
Conditions						
a	Applicant.age >=45	-	T	T	F	
b	Applicant.smoker	-	T	F	-	
Actions						
Post Message(s)						
A	Applicant.riskRating		'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		

Completing the truth table and the Rulesheet requires the definition of 2 additional business rules:

PolicyApplicantBoolean.ers					
Conditions		0	1	2	3
a	Applicant.age >=45	-	T	T	F
b	Applicant.smoker	-	T	F	-
Actions					
Post Message(s)					
A	Applicant.riskRating		'high'	'low'	'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 low risk
3				If an applicant is NOT 45 or older, then ignore whether or not the applicant smokes and classify the applicant as low risk

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:

Boolean 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		

Boolean 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		

Boolean 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

PolicyApplicantBooleanXOr.ers		0	1	2	3	4
Conditions						
a	Applicant.age < 45	-	F	T	F	T
b	Applicant.smoker	-	F	T	T	F
Actions						
Post Message(s)						
A	Applicant.riskRating		'low'	'low'	'high'	'high'
Overrides						
Rule Statements	Rule Messages					
Ref	ID	Post	Alias	Text		
1				If an applicant is 45 or older AND does NOT smoke, then classify the applicant as low risk		
2				If an applicant is younger than 45 AND smokes, then classify the applicant as low risk		
3				If an applicant is 45 or older 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		

Boolean 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

PolicyApplicantBooleanNOR						
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		'low'	'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 low risk		
2				If an applicant is younger than 45 AND does NOT, then classify the applicant as low risk		
3				If an applicant is younger than 45 AND smokes, then classify the applicant as low risk		
4				If an applicant is younger than 45 AND does NOT smoke, then classify the applicant as high risk		

Boolean 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

PolicyApplicantBooleanXOR.ers						
Conditions		0	1	2	3	4
a	Applicant.age < 45	-	F	T	F	T
b	Applicant.smoker	-	F	T	T	F
Actions						
Post Message(s)						
A	Applicant.riskRating		'high'	'high'	'low'	'low'
Overrides						
Rule Statements		Rule Messages				
Ref	ID	Post	Alias	Text		
1				If an applicant is 45 or older AND does NOT smoke, then classify the applicant as high risk		
2				If an applicant is younger than 45 AND does smoke, then classify the applicant as high risk		
3				If an applicant 45 or older AND smokes, then classify the applicant as low risk		
4				If an applicant is younger than 45 AND does NOT smoke, then classify the applicant as low risk		



Answers to test-yourself questions

For details, see the following topics:

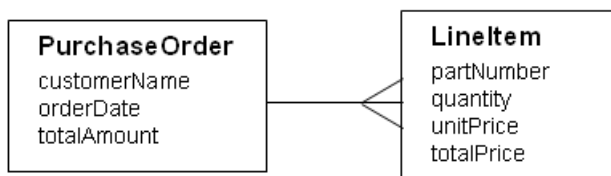
- [Test yourself answers: Building the vocabulary](#)
- [Test yourself answers: Rule scope and context](#)
- [Test yourself answers: Rule writing techniques and logical equivalents](#)
- [Test yourself answers: Collections](#)
- [Test yourself answers: Rules containing calculations and equations](#)
- [Test yourself answers: Rule dependency: dependency and inferencing](#)
- [Test yourself answers: Filters and preconditions](#)
- [Test yourself answers: Recognizing and modeling parameterized rules](#)
- [Test yourself answers: Writing rules to access external data](#)
- [Test yourself answers: Logical analysis and optimization](#)
- [Test yourself answers: Ruleflow versioning and effective dating](#)
- [Test yourself answers: Troubleshooting rulesheets](#)

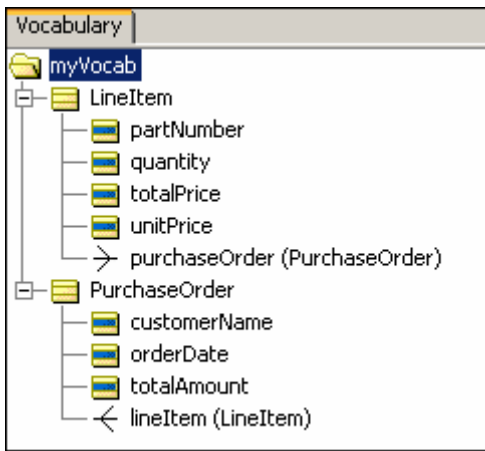
Test yourself answers: Building the vocabulary

[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. A 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 Bicycle = 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.

Test yourself answers: Rule scope and context

[Show me this set of test questions.](#)

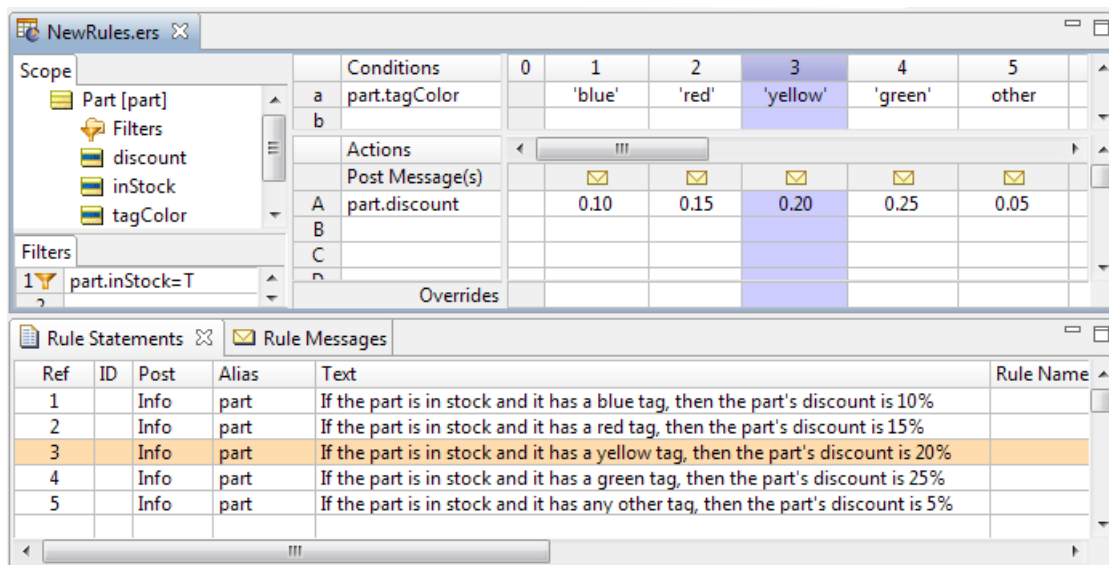
1. 7 root-level entities are present
2. all terms are allowed **except** DVD.actor
3. Movie.supplier

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.

Test yourself answers: Rule writing techniques and 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:



13. True

14. False. Nonconditional rules are governed by Preconditions on the same Rulesheet only if they share the same scope as the Preconditions.

Test yourself answers: Collections

[Show me this set of test questions.](#)

- Children of a Parent entity are also known as **elements** of a collection.
- False. A collection can contain root-level entities.
- True
- True
- Refer to the Rule Language Guide for a full list and description of all collection operators.
- Rule Language Guide
- Order total is equal to the sum of the line item prices on the order.
- items
- one-to-many (1->*)
- It is not an acceptable replacement since the use of any collection operator requires that the collection be represented by an alias.
- set the navigability of the association between Order and Lineltem to Order->lineltem. In other words, make the association one-directional from Order to Lineltem.
- Optional, Convenient
- A collection alias is not required in this case because no collection operator is being applied to the collection.
- >forAll
- >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.dVD
 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.

Test yourself answers: Rules containing calculations and equations

[Show me this set of test questions.](#)

1. comparison in Preconditions and Conditions, assignment in Nonconditionals and Actions
2. The results of the equations are:
 - 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. The data types are:
 - a. Integer
 - b. String
 - c. Boolean
 - d. Decimal
 - e. Boolean

- f. Boolean
 - g. Boolean
5. The validity of the assignments are:
- 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 \rightarrow 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. The sections of a Rulesheet that accept equations and calculations are:
- a. Scope: False
 - b. Rule statements: False
 - c. Condition rows: True
 - d. Action rows: True
 - e. Column 0: True
 - f. Condition cells: False
 - g. Action cells: False
 - h. Filters: True

Test yourself answers: Rule dependency: dependency and 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*.

Test yourself answers: Filters and preconditions

[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

Test yourself answers: Recognizing and 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..100000], >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.

Test yourself answers: 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.

Test yourself answers: Logical analysis and 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 Rulesheets 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 Rulesheets 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 made up 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.

Test yourself answers: Ruleflow versioning and 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.

Test yourself answers: Troubleshooting rulesheets

[Show me this set of test questions.](#)

1. Troubleshooting is based on the principle that Rulesheets 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. 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. In the `brms.properties` file at `[CORTICON_WORK_DIR]` root.
11. In the `brms.properties` file at `[CORTICON_WORK_DIR]` root.
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

Access to Corticon knowledge resources

[Complete online documentation for the current release](#)

Corticon online tutorials available in the [Corticon Learning Center](#):

- [Tutorial: Basic Rule Modeling in Corticon Studio](#)
- [Tutorial: Advanced Rule Modeling in Corticon Studio](#)
- [Modeling Progress Corticon Rules to Access a Database using EDC](#)
- [Connecting a Progress Corticon Decision Service to a Database using EDC](#)
- [Deploying a Progress Corticon Decision Service as a Web Service for Java](#)
- [Deploying a Progress Corticon Decision Service in process for Java](#)
- [Using Corticon Business Rules in a Progress OpenEdge Application](#)

Corticon guides (PDF):

- [What's New in Corticon](#)
- [Corticon Installation Guide](#)
- [Corticon Studio: Rule Modeling Guide](#)
- [Corticon Studio: Quick Reference Guide](#)
- [Corticon Studio: Rule Language Guide](#)
- [Corticon Studio: Extensions Guide](#)
- [Corticon Server: Integration and Deployment Guide](#)

- [Corticon Server: Web Console Guide](#)
- [Corticon Server: Deploying Web Services with Java](#)
- [Corticon Server: Deploying Web Services with .NET](#)

Corticon JavaDoc API reference (HTML):

- [Corticon Server API](#)
- [CorticonExtensions API](#)

See also:

- [Introducing the Progress® Application Server](#)
- Corticon documentation for this release on the [Progress download site](#): What's New Guide (PDF), Installation Guide (PDF), PDF download package, and the online Eclipse help installed with Corticon Studio.