Corticon
Rule Modeling

Copyright

Copyright

Table of Contents

Collections...115

Rules containing calculations and equations...153

Rule dependency in chaining and looping...171

Filters and preconditions 195

How to recognize and model parameterized rules....................223

How to write rules to access external data...............................229

Logical analysis and optimization..245

Contents

Appendix A: Studio properties and settings..327

Appendix B: Answers to TestYourself questions....................................333

1

Introduction to Corticon rule modeling

This set of topics describes the core of Corticon. Here you construct the logic and patterns in vocabularies that are assembled in row-and-column rule sheets where the diverse operators enable spreadsheet layouts of readable rule patterns. The modeling topics are supported by guides to the modeling language and a quick reference to the user interface's basic tooling functions.

2

Build 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.

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 documentation, these things are referred to as *entities,* and the 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 are referred to as *persistent*) in a database, or communicated to other applications external to Corticon. An example of this might be the following two simple computational rules:

In these two rules, `itemSubTotal` is the intermediate or transient term. You may never use `itemSubTotal` by itself; instead, you may only create it for purposes of subsequent derivations, as in the calculation of `orderTotal` in rule #2. Because 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. Progress believes that rule modelers should 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. This concept is called *abstraction*—the complexities of an enterprise's data storage and retrieval systems were 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 Figure 1: Operator Vocabulary on page 14, is called the Operator Vocabulary because it provides many of the verbs (the operators) needed for 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
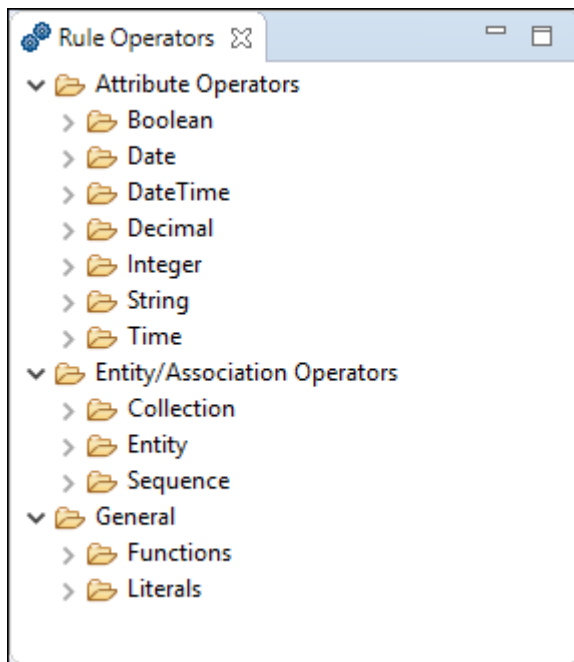
- When XML messaging is used to carry data to and from the rules for evaluation, data must be organized in a predefined structure that can be understood and processed by the rules. A schema supplies the contract for sending data to and from a Corticon Decision Service. An XML schema, generated directly from the Vocabulary, accomplishes this purpose. This schema is called a Vocabulary-Level service contract and details can be found in the *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 do not need to be included in the Vocabulary. Conversely, the Vocabulary may include terms (such as transient attributes) that are used only in rules. These terms do not need to 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. Terms can exist only within the Vocabulary. These are the transient attributes that were introduced previously.

- 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 and is not included in this manual.

There are two basic starting points for building a Vocabulary: construct one, or generate one from a REST or database source.

For details, see the following topics:

- Build a Vocabulary by hand

- Populate a Vocabulary from a Datasource

- Extend a Vocabulary

- TestYourself questions for Build the vocabulary

Build a Vocabulary by hand

An alternative to generating a vocabulary is to create one by hand. Creating a vocabulary by hand requires more effort then generating one, yet has the potential advantage of forcing you to carefully consider the elements to include in your vocabulary.

The first step in creating a Vocabulary is to collect information about the specifics of the business problem you are trying to solve. This step 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 are often the best source of information about how business is conducted. 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 than those who actually perform it.

- **Company policies and procedures**—Any written policies and procedures are 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 actually happens can provide valuable insight into problems.

- **Existing systems and data sources**— Systems 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 used often provides clues about the core problems. Also, business logic contained in these legacy systems often captures business policies and procedures (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 or existing business rules in order to identify the relevant terms and the relationships among these terms. Statements that express the relevant terms and relationships are called facts, and Progress recommends developing a Fact Model to more clearly illustrate how they fit together. A simple example shows you the creation of a Fact Model and its subsequent development into a Vocabulary for use in Corticon Studio.

Step 1: Design the Vocabulary

Example

An air cargo company has a manual process for generating flight plans. These flight plans assign cargo shipments to a specific aircraft. Each flight plan is assigned a flight number. The cargo company owns a small fleet of three planes: 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 airplane type also has a tail number that identifies it. A cargo shipment has characteristics like weight, volume and a manifest number.

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

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

Step 2: Identify the terms

Identify the terms (entities and attributes) for our Vocabulary by circling or highlighting those nouns that are used in the business rules you want to automate. Example on page 16 is marked up:

> An air cargo company has a manual process for generating flight plans. These flight plans assign cargo shipments to a specific aircraft. Each flight plan is assigned a flight number. The cargo company owns a small fleet of three planes: 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 airplane type also has a tail number that identifies it. A cargo shipment has characteristics like weight, volume and a manifest number.

Step 3: Separate the generic terms from the specific

Why circle *aircraft* 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 is an attribute of the generic aircraft entity. Several cargo shipments and flight plans can exist. Like the specific aircraft, these are *instances* of their respective generic terms. For the Vocabulary, you identify the generic (and therefore reusable) terms. But, ultimately, you 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 accomplishes this goal, discussed later.

Step 4: Assemble and relate the terms

None of the circled terms exists in isolation. They all relate to each other in one or more ways. Understanding these relationships is the next step in Vocabulary construction. The following facts are 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.

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

Also notice that a fact "has a" relationship. 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, the Vocabulary contains 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

Attributes: weight, volume, manifest number, packaging

Entity: FlightPlan

Attributes: flight number

Step 5: Diagram the Vocabulary

Using this breakdown, sketch a simple Fact Model that illustrates the entities and their relationships, or *associations*. In the Fact Model, entities are rectangular boxes, associations between entities are straight lines connecting the entity boxes, and entity-to-attribute relationships are diagonal lines from the associated entity. The following illustration is the resulting Fact Model:
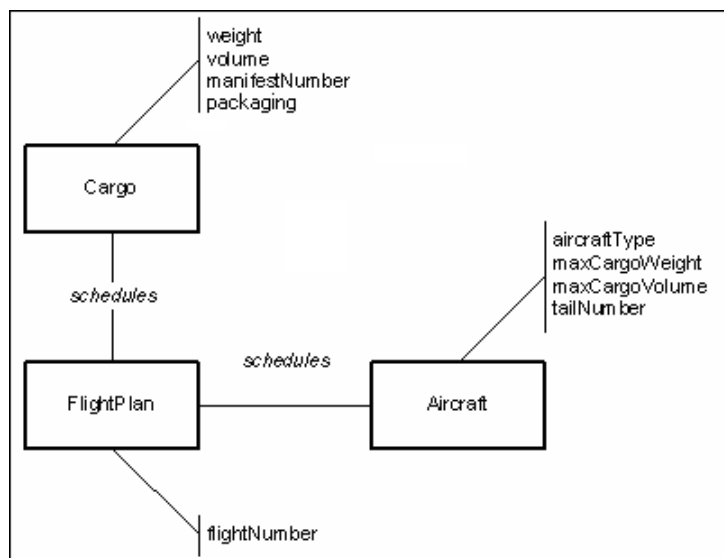
A unified modeling language (UML) class diagram contains the same type of information, and may be more familiar to you:
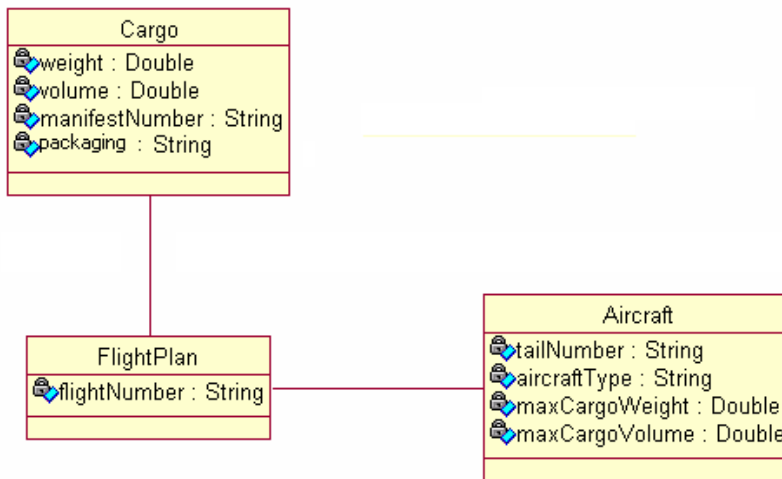
It is not a requirement to construct diagrams or models of the Vocabulary before building it in Corticon. But, it can be very helpful in organizing and conceptualizing the structures and relationships, especially for very large and complex Vocabularies. The BRMS Fact Model and UML Class Diagram are appropriate because they remain sufficiently abstracted from lower-level data models that contain information not typically required in a Vocabulary.

Step 6: Model the Vocabulary in Corticon Studio

The next step is to transform the diagram into your Corticon Vocabulary. This can be done in Corticon Studio using its built-in **Vocabulary Editor**.

In Corticon Studio, choose **New > Rule Project**. Click the Rule Project, and then choose **New > Vocabulary**. Create the entities, attributes, and associations that were defined in the diagram.

Note: See *"Vocabulary topics" in the Quick Reference guide* for complete details on building a Vocabulary.

The naming conventions for the entities and attributes will be used in the Vocabulary:

- All attributes in our Vocabulary must have a data type specified. These data types can 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

  - **Transient**: Created, derived, or assigned by rules in Studio.

Note:

Transient attributes carry or hold values while rules are executing within a single Rulesheet. Because 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, then the attribute 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 for `packDate`:
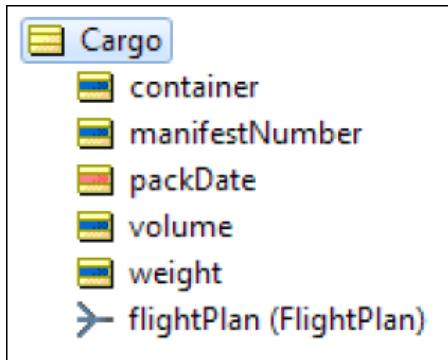
XML response messages created by Corticon Server will not contain the `packDate` attribute.

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`. Do not use names that are cryptic. The intent is to express the names in terms that are understood by business users as well as developers.

- Associations between entities have role names that are assigned when you build 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 that covers this in more detail in the Scope chapter.

- Associations between entities can be directional (one way) or bidirectional (two way). If the association between `FlightPlan` and `Aircraft` were directional (with `FlightPlan` as the *source* entity and `Aircraft` as the *target*), you would only be able to write rules that traverse *from* `FlightPlan` *to* `Aircraft`, but not the other way. This means that a rule <u>can use</u> the Vocabulary term `flightPlan.aircraft.tailNumber` but <u>cannot use</u> `aircraft.flightPlan.flightNumber`. Bidirectional associations allow you to traverse the association in either direction, which allows you more flexibility in writing rules. Therefore, Progress strongly recommends that all associations be bidirectional whenever possible. New associations are bidirectional by default.

- Associations also have cardinality, which indicates how many instances of a given entity can be associated with another entity. For example, in the air cargo scenario, each instance of `FlightPlan` will be associated with only one instance of `Aircraft`, so 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 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.

- 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, the 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 process will most likely be iterative, with multiple Vocabulary changes being made as the rules are built, refined, and tested. It is common to discover, while building rules, that the Vocabulary does not contain all the necessary terms. But, the flexibility of Corticon Studio permits the rule developer to update or modify the Vocabulary immediately, without programming.

The following figure shows the vocabulary modeled in Corticon Studio:

Figure 2: Vocabulary Window in Corticon Studio
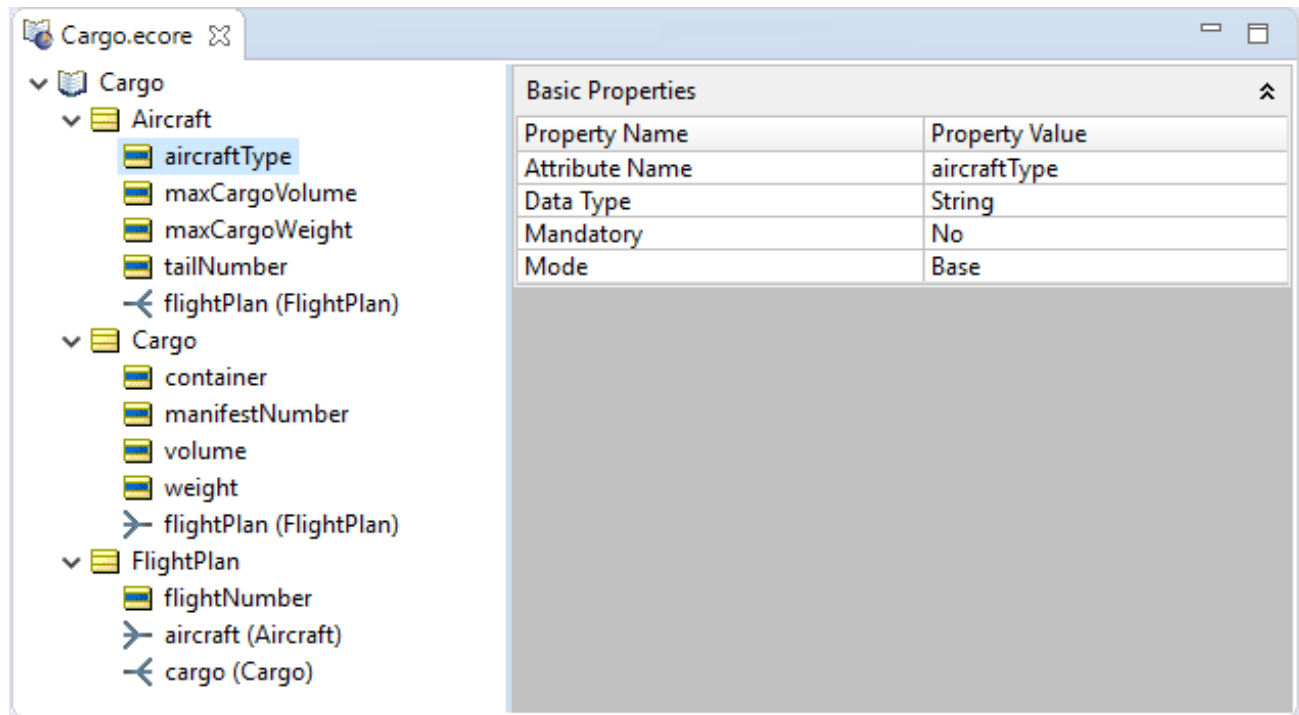
Populate a Vocabulary from a Datasource

Often you have data sources that you want to use as the basis for your rule modeling that might have many tables, each with many columns. You could transcribe each data source's schema to create a Vocabulary, yet the ability to populate the Vocabulary quickly from the schema would expedite the process dramatically.

When you use this built-in Vocabulary generation utility, Corticon sets up the name patterns and defines the data types and associations as best it can. It is important that you review the Vocabulary against the source schema, to validate that the results are correct.

Step 1: How Datasources are transformed into a Corticon Vocabulary

The following are the relationships between relational Datasources and Corticon Vocabulary elements are:

Relational database	Corticon Vocabulary
Schema	Vocabulary
Table	Vocabulary: Entity
Table Column or Field	Vocabulary: Attribute
Relationship between Tables	Vocabulary: Association

After you connect to a Datasource and import its metadata, you can constrain the tables and attributes that will be evaluated. Then, the internal algorithm makes its best effort to populate the Vocabulary.

Assuming that you are creating a new Vocabulary, these are the steps it takes:

1. For each selected Table in the Datasource, create a new Entity in the Vocabulary.

2. For each Column in the Table:

   a. Create a new Attribute in the Entity.

   b. Determine the best Corticon data type for the Attribute by referring to the column's data type information.

   c. If the column is part of the Table's primary key, then mark the Attribute as part of the Entity identity.

3. After all Tables and Columns are processed, Associations are created for each foreign key for each table (if the source and target tables are both mapped in the Datasource).

The creation process tries to be complete and accurate, but it has limited abilities:

- Columns that are referenced by foreign keys are not added as Attributes.

- Tables that do not have any valid columns are not created (such as, Association middle tables or Sequence tables).

- The data type for an attribute is evaluated in this order: Datetime, Time, Date, Decimal, Integer, String, and Boolean. Some Corticon data types might not get picked for attributes because of an overload of possible mappings (such as, Date and Time could always be created as Datetime). Note that these decisions are derived from data when data is in a REST source that has no schema.

Step 2: The Vocabulary generation process for RDBMS sources on page 21 shows the procedure for populating a new Vocabulary.

Step 2: The Vocabulary generation process for RDBMS sources

Relational databases have well-structured schemas that declare every element's data type. The following steps in Corticon Studio populate a new Vocabulary from a relational database Datasource. For an example, use the Patient/Treatment schema that was created in SQL Server from SQL statements in the Data Integration's **ADC Connectivity** sample.

To generate a Vocabulary from a relational data source:

1. In Corticon Studio, create a new Rules Project named **GenMed**.

2. In the new project, create a Vocabulary named **GenMed**.

3. Open the Vocabulary in its editor, and then select the menu command **Vocabulary > Add Datasource > Add ADC Datasource**.

4. Define the Datasource name as **Patient Data**. Connect to SQL Server database **PatientRecords**. Enter credentials, and the click **CONNECTION Test**:

Note: You might want to add a **Schema Filter** value, such as `dbo`, to constrain the results of the next step.

5. Click **METADATA Import**, and then choose the option to choose the tables you want to use. For this example, choose just the two `dbo` tables, as shown:
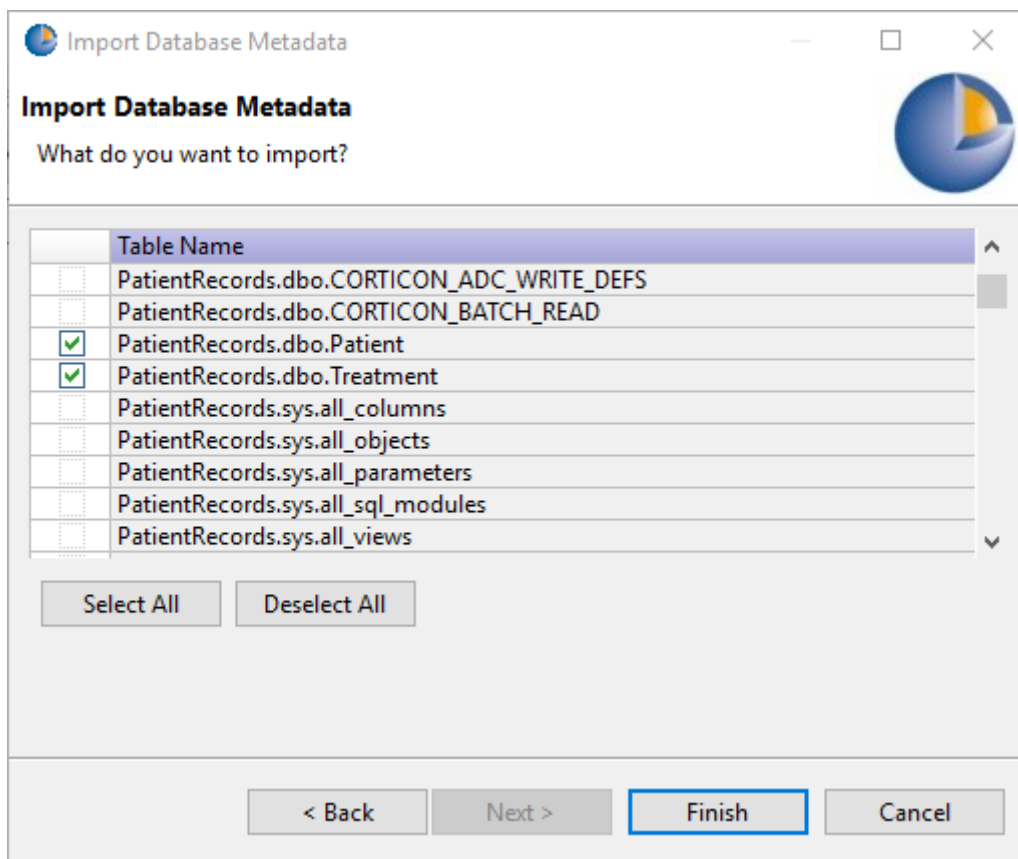
and click **Finish**.

6. Select **Vocabulary > Populate Vocabulary From Datasource**:
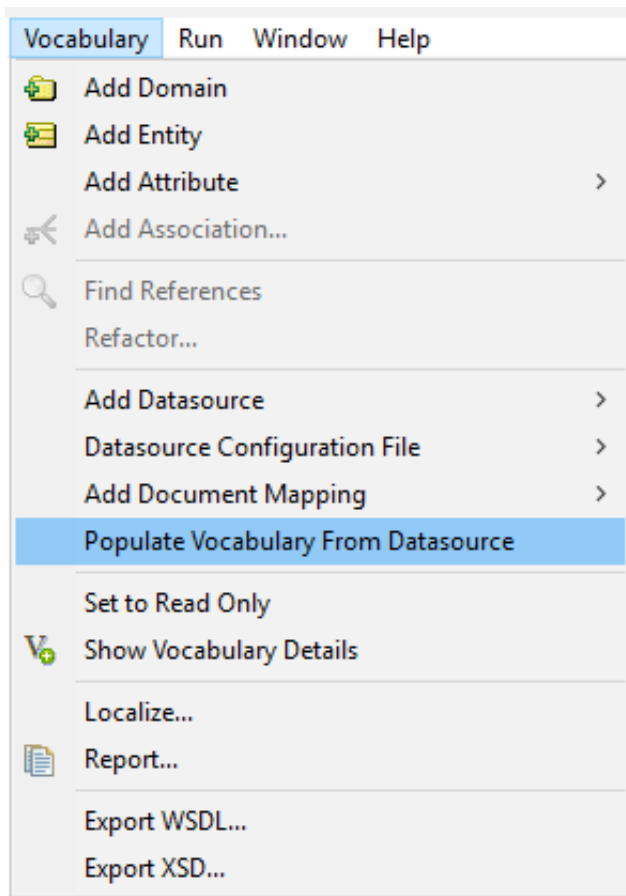
7. Choose the **Patient Data** Datasource. If there were several Datasources defined, choose them one at a time for this process. In this example, there is only one. Click **Next**.

8. A wizard opens to let you review the Datasource prior to creating the Vocabulary elements, where you can select the Tables and Columns that create Entities and Attributes. In the following image, the tree was expanded:

9. Click **Finish**. The Vocabulary is generated, as shown:

Step 3: The Vocabulary generation process from REST sources

REST sources are usually not as clearly structured as relational databases. Some provide a schema, but generally they do not. REST sources can conform to a relational database schema when Corticon uses the Progress® DataDirect® Autonomous Rest Connector to access REST sources. The REST connector maps the JSON in a REST source to a relational database schema, and then translates SQL statements to REST API requests. These steps in Corticon Studio populate a new Vocabulary from the REST Datasource used in the REST connectivity sample from the Data Integration guide. The REST source has no schema; its data looks this:

```
 1 ▾ {
 2 ▾   "results": [
 3 ▾     {
 4         "procedureCode": "B5120ZZ",
 5 ▾       "rates": [
 6 ▾         {
 7            "startDate": "2018-1-1",
 8            "endDate": "2018-6-1",
 9            "rate": 0.85
10         },
11 ▾         {
12            "startDate": "2018-6-2",
13            "endDate": "2018-12-31",
14            "rate": 0.8
15         }
16       ]
17     }
18   ]
19 }
```

To generate a Vocabulary from a REST data source:

1. In Corticon Studio, create a new Rule Project named **GenRates**.

2. In the new project, create a Vocabulary named **GenRates**.

3. Open the Vocabulary in its editor, and then select **Vocabulary > Add Datasource > Add REST Datasource**.

4. Define the Datasource connection for the URL
   `https://bj36i9ki66.execute-api.us-east-2.amazonaws.com/prod/ReimbursementRate?procedureCode=B5120ZZ`
   as shown, and then click **CONNECTION Test**:

5. Click **SCHEMA Discover**. If your REST source has a schema, or is one that you exported in an earlier processing of this source you could import it now. For this source, you need to let the Progress® DataDirect® Autonomous Rest Connector map the JSON in the REST source to a relational database schema, and then translate SQL statements to REST API requests.

6. Select **Vocabulary > Populate Vocabulary From Datasource**

Vocabulary   Run   Window   Help

🗂	Add Domain
🗂	Add Entity
	Add Attribute >
⤨	Add Association...
🔍	Find References
	Refactor...
	Add Datasource >
	Datasource Configuration File >
	Add Document Mapping >
	Populate Vocabulary From Datasource
	Set to Read Only
V⊕	Show Vocabulary Details
	Localize...
📄	Report...
	Export WSDL...
	Export XSD...
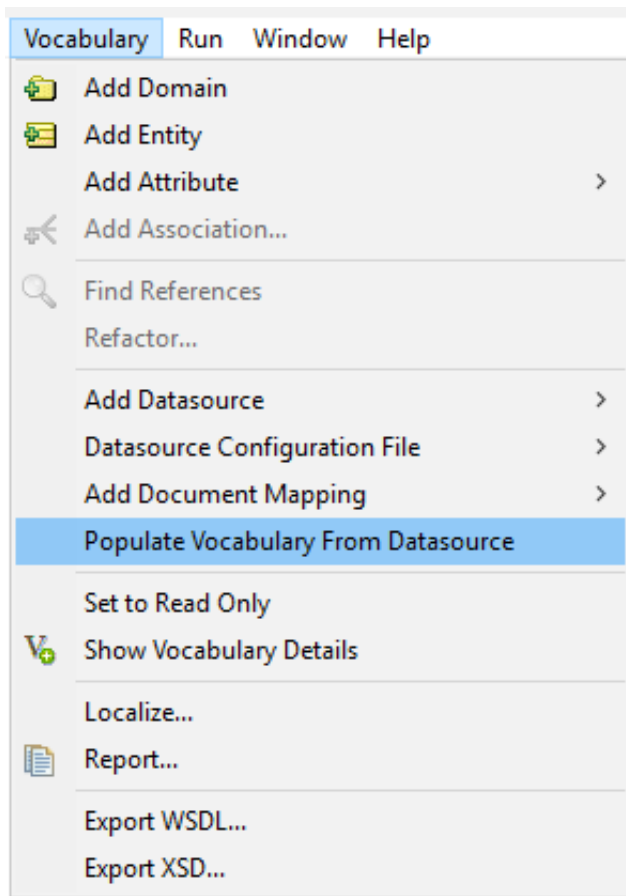
7. Choose the **REST Service** Datasource. If there were several Datasources defined, choose them one at a time for this process. In this example, there is only one, **REST Service**. Click **Next**.

8. A wizard opens to let you review the Datasource prior to creating the Vocabulary elements, where you can select the Tables and Columns to create as Entities and Attributes. Here, the tree was expanded for REST_DATA, the primary table that was unnamed so it was given this default name. You can see links between tables to all the other tables at the bottom of the table.
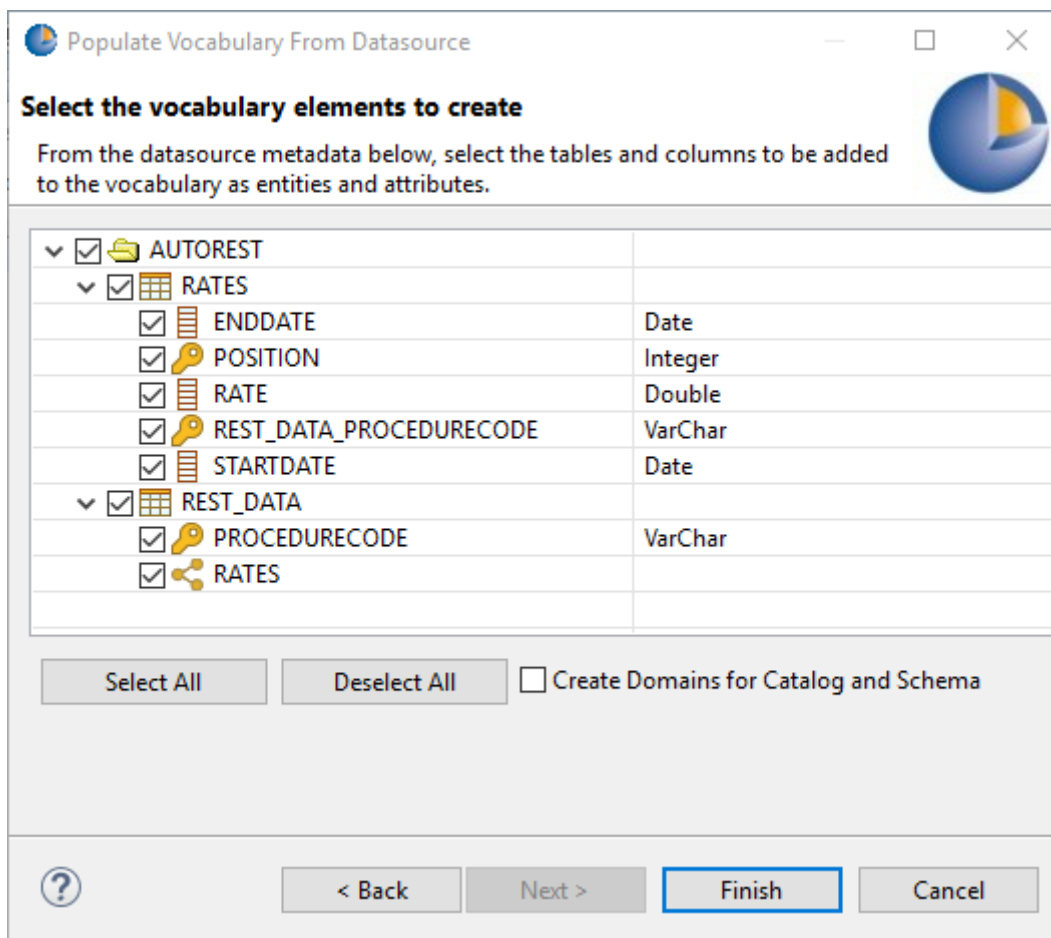
9. Click **Finish**. The Vocabulary is generated, as shown:
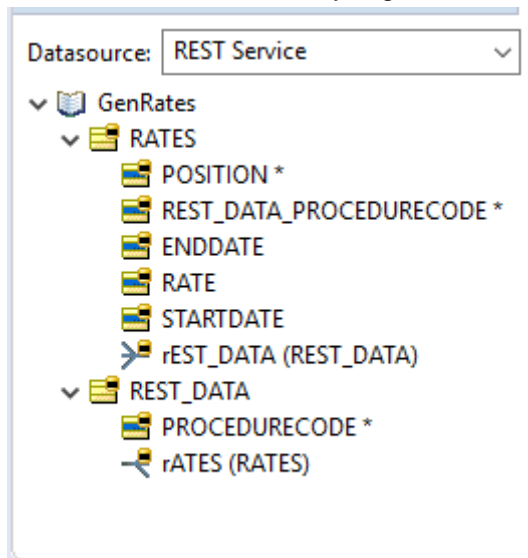
The Primary Key in RATES is POSITION, a standard that ARC uses to ensure keys are unique, plus the REST_DATA_PROCEDURECODE, the default name of the primary entity. The Primary Key in the REST_DATA entity is the single primary key, PROCEDURECODE

Here is an association:

Step 4: Verify and update the generated Vocabulary

Produce a Vocabulary Report

The import of Datasource metadata to build a Vocabulary is a processed through a best-effort algorithm. You should produce a Vocabulary report to review the entity names, attribute names and their data types, and the implied associations.

Adding and deleting

You can delete any entity (which deletes all its attributes) and any attributes and associations. Be careful not to delete primary keys. The effect of deletions in the Vocabulary are local. The deletions do not affect the Datasource. You can add attributes, and because the metadata you imported was not deleted, you can re-add a deleted attribute and bind it to the column in the Datasource.

Doing updates from a Datasource is not a synching operation. There is no provision for removing metadata that is no longer in the Datasource.

Refactoring names

You can rename any entity and attribute even if it is a just a case change. For example, you can refactor dob with DOB. It is important that you *refactor*, not *rename*, so that other instances of the name in the Vocabulary such as associations are also updated.

If you repopulate the Vocabulary from the Datasource, then the name you entered is retained while it is still logically the Datasource column name.

Note: MS Dynamics as a Datasource: Some table names in Dynamics might map to an unexpected name. For example, a Case table might become an Incident entity by default on the initial import.

Data types

The algorithm in the import makes a best effort to map the Datasource's data type to a corresponding Corticon data type. The data type for an attribute is evaluated in this order: Datetime, Time, Date, Decimal, Integer, String, and Boolean. Some Corticon data types might not get picked for attributes because of an overload of possible mappings (such as, Date and Time could always be created as Datetime). Note that these decisions are derived from data when data is in a REST source that does not have a schema. After import, you can revise the data type, for example when you have custom data types that apply constraints, or date of birth imports as Datetime when your rules want just Date, or when `"flight_number":55` is imported as an integer data type when you want it as a String,

Mandatory

Whether an attribute is mandatory is set by you. It is not changed on a re-import.

Associations

The metadata from the Datasource often provides correct associations. When you use multiple Datasources, you need to create the associations between entities. In all cases, review your associations.

Transients

You can add transients. If you change an imported attribute to a transient, then its binding to its Datasource column is dropped.

Foreign keys

When if both the source and target table are mapped in the Datasource, then an association is created for each foreign key for each table.

Domains

You might need multiple domains. If you use REST Datasources, then you need to rename the existing domains before importing a new one.

Extend a Vocabulary

After a Vocabulary is defined, you can extend the design by customizing data types to enforce certain values and constraints, use multiple domains, and implement inheritance.

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. 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 blank spaces. 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.

You already used this feature in the custom data type `containerType`, a `String`, in the *Basic Rule Modeling Tutorial*. The following figure lists its labels and values.

Figure 3: Vocabulary Editor Showing the Custom Data Type containerType
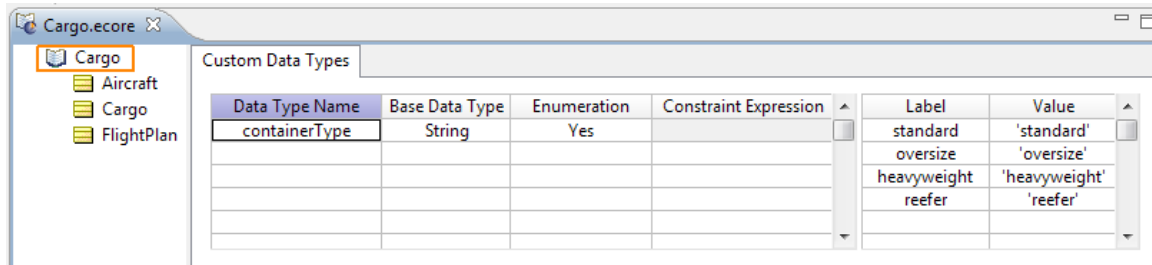
Enumeration or Constraint Expression?

Enumeration—When the **Enumeration** for a Custom Data Type is set to `Yes`, as shown in the preceding figure, 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 of 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 is not necessary because it reduces to a base attribute with a custom name.

All **Constraint Expressions** must be `Boolean` expressions: 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 `value` to represent the Custom Data Type value.

- Logical connectors such as `and` and `or` are supported.

- Parentheses can be used to form more complex expressions

- The expression can include references to Base and Extended Operators which are compatible with the Base Data Type chosen.

- No Collection operators can 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 designated by selecting `Yes` in its **Mandatory** property value.

The following are typical Constraint Expressions:

Constraint Expression	Meaning
value > 5	Integer values greater than 5
value >= 10.2	Decimal values greater than or equal to 10.2
value in (1.1..9.9]	Decimal values between 1.1 (exclusive) and 9.9 (inclusive)
value in ['1/1/2014 12:30:00 PM'..'1/2/2019 11:00:00 AM')	DateTime values between '1/1/2014 12:30:00 PM' (inclusive) and '1/2/2019 11:00:00 AM' (exclusive)
value in ['1:00:00 PM'..'2:00:00 PM']	Time values between '1:00:00 PM' (inclusive) and '2:00:00 PM' (inclusive)
value.size >= 6 and (value.indexOf(1) > 0 or value.indexOf(2) > 0)	String values of minimum 6 characters in length that contain at least a 1 or 2

How to use non-enumerated Custom Data Types in Rulesheets and Ruletests

Non-enumerated custom data types use **Constraint Expressions** and do not cause Rulesheet drop-down lists 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 following figure, `weightRange` is defined as a non-enumerated custom data type with **Base Data Type** of `Decimal`.

Figure 4: Non-enumerated Custom Data Types

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

Figure 5: Using Custom Data Types in a Rulesheet

Notice in the preceding figure 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:

Figure 6: Violating a Custom Data Type's Constraint Expression
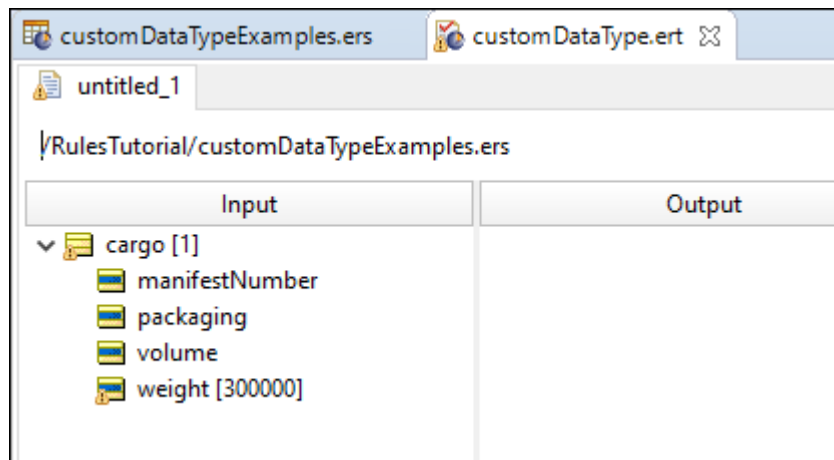
Notice that the small yellow warning icon ⚠ indicates a problem in the attribute, entity, and both Ruletest tabs. Such an error is hard to miss. Also, a Warning message will appear in the **Problems** tab (if open and visible) as shown. If the **Problems** tab is closed, you can display it by selecting **Window** > **Show View** > **Problems** from the **Studio** menubar.

Figure 7: Violating the Constraint Expression of a Custom Data Type
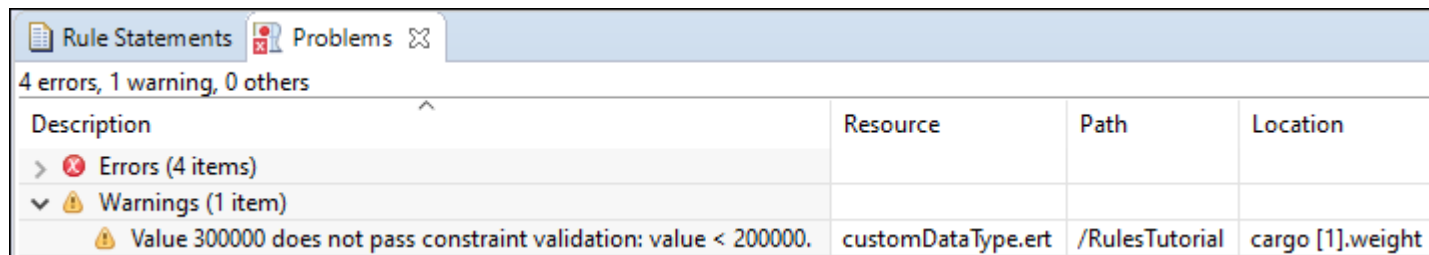
A Warning does not prevent you from running the Ruletest. However, an Error, indicated by a small red icon ⊠, will prevent the Ruletest execution. You must fix any errors before testing.

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, often referred to as *enums*, can be maintained directly in the Vocabulary, or retrieved and updated from a data source.

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, because changing a set of enumerations later to add or remove the labels data will affect any Rulesheets and Ruletests that use that custom data type's enumerations as you can observe in this topic.

At the Vocabulary root, you 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 most other characters. It needs to be set off in plain single quotation marks. If you enter or paste text with the delimiters, they are added for you. Like this:

Data Type Name	Base Data...	Enum...		Label	Value	
colorLabeled	String	Yes			'red'	
colorUnlabeled	String	Yes			'blue'	

If you want to use labels, then 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.

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

Set `Glove.color` to use the `colorUnlabeled` data type:

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

Set `Ball.color` to use the `colorLabeled` data type:

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

(Simple.ecore tree: Simple > Ball > color, size; Glove > color, size)
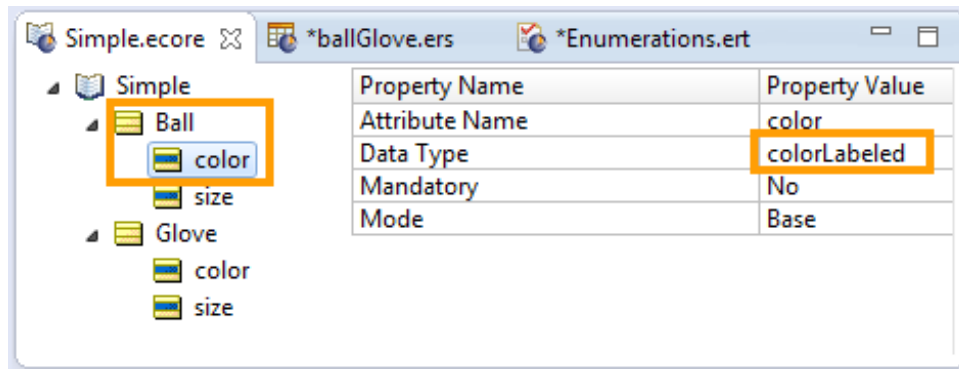
When you create a Rulesheet, the list offered at A1 contains the label (`Ball.color = red`) , while the list offered at B1 contains the value in qoutes (`Glove.color='red'`).

	Conditions	0	1	2	3	
a	Ball.size		1	2		
b	Glove.size		1	2		
c						

	Actions					
	Post Message(s)		✉	✉		
A	Ball.color		red	blue		
B	Glove.color		'red'	'blue'		
	Overrides					

Rule Statements ✕   ✉ Rule Messages

Ref	ID	Post	Alias	Text
1		Info	Ball	{Ball.color} ball and {Glove.color} glove
2		Info	Ball	{Ball.color} ball and {Glove.color} glove
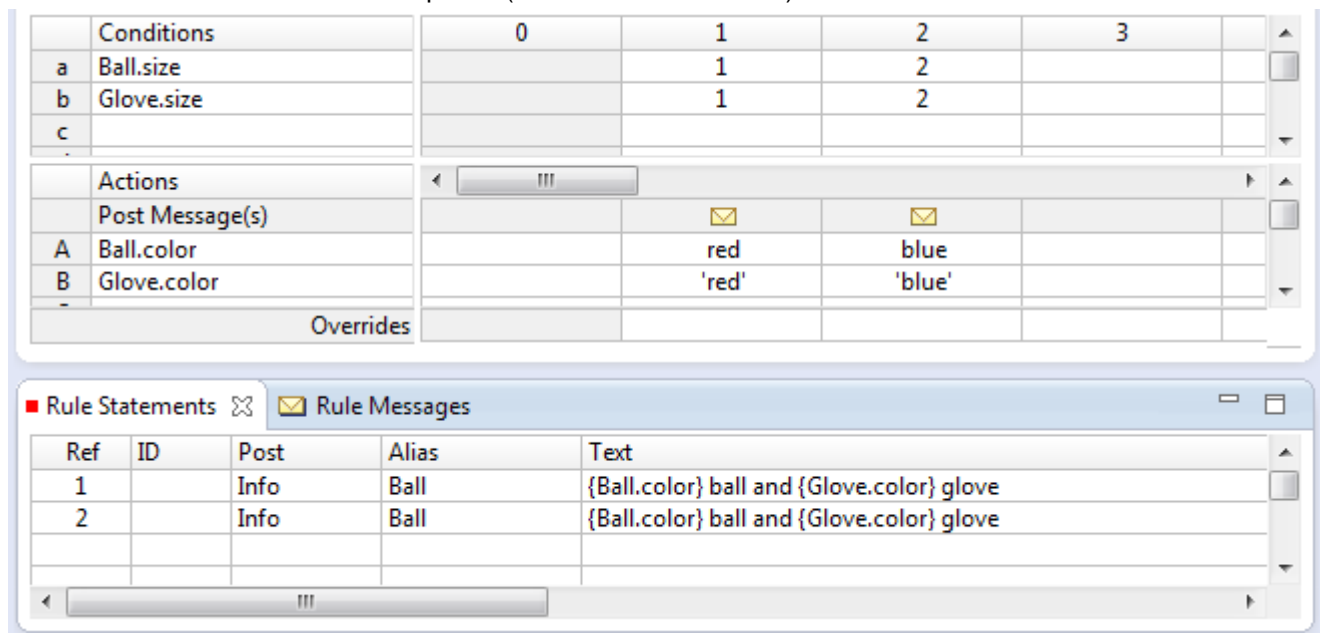
You add Rule Statements so that you can see how the labeled and unlabeled items are handled.

In a simple Ruletest, add some size tests to see what happens. As shown, the labels and values in the resulting **Output** are both unquoted. The **Rule Messages** tab displays 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
▲ 🗒 Ball [1]	▲ 🗒 **Ball [1]**
🗒 size [1]	🗒 color [red]
▲ 🗒 Glove [1]	🗒 size [1]
🗒 size [1]	▲ 🗒 **Glove [1]**
▲ 🗒 Ball [2]	🗒 color [red]
🗒 size [2]	🗒 size [1]
▲ 🗒 Glove [2]	▲ 🗒 **Ball [2]**
🗒 size [2]	🗒 color [blue]
	🗒 size [2]
	▲ 🗒 **Glove [2]**
	🗒 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   ▼
🗒 size [1 red
▲ 🗒 Glove [1] blue
🗒 color
🗒 size [1]
▲ 🗒 Ball [2]
🗒 size [2]
▲ 🗒 Glove [2]
🗒 size [2]

The value-only list has quoted values:

Input
◢ ▤ Ball [1]
▤ color [red]
▤ size [1]
◢ ▤ Glove [1]
▤ color   ▾
▤ size [1   'red'
◢ ▤ Ball [2]   'blue'
▤ size [2]
◢ ▤ Glove [2]
▤ size [2]

Both are reconciled to unquoted values in the displayed **Input** and **Output** columns:

Input	Output
◢ ▤ Ball [1]	◢ ▤ Ball [1]
▤ color [red]	▤ color [red]
▤ size [1]	▤ size [1]
◢ ▤ Glove [1]	◢ ▤ Glove [1]
▤ color [red]	▤ color [red]
▤ size [1]	▤ size [1]
◢ ▤ Ball [2]	◢ ▤ **Ball [2]**
▤ size [2]	▤ **color [blue]**
◢ ▤ Glove [2]	▤ **size [2]**
▤ size [2]	◢ ▤ **Glove [2]**
	▤ **color [blue]**
	▤ **size [2]**

■ Rule Statements   ☑ Rule Messages ⊠

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, affects 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 click its root—`Cargo`, in this example. Then, enter a preferred unique name without spaces, and click 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.
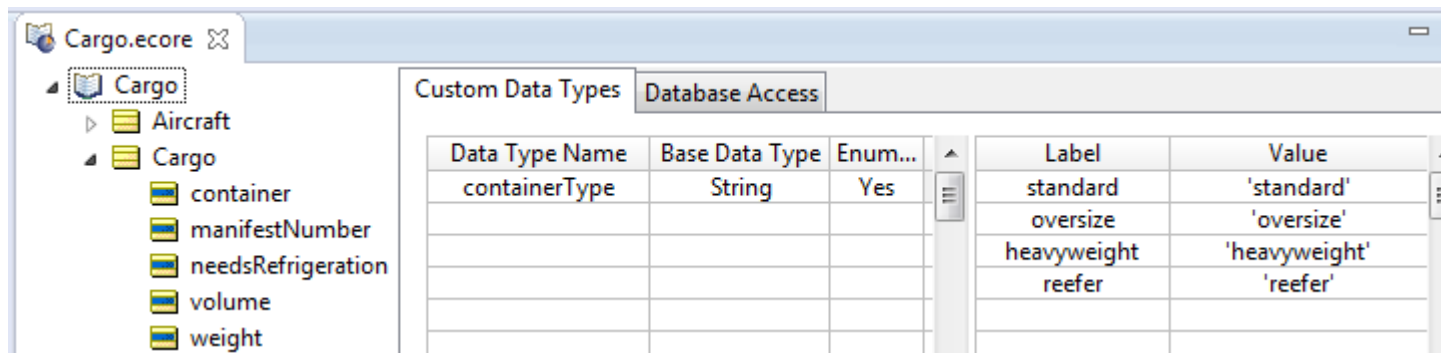
When you complete a valid Custom Data Type, choose the attributes in the Vocabulary that will be constrained to the enumeration.
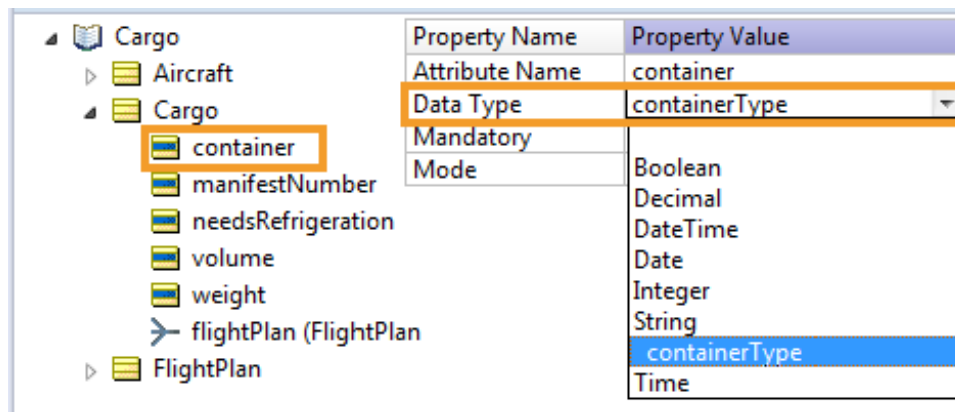
If your custom data type is a local enumeration, then you enter the enumerated values of the base data type into the **Value** column, and, if you intend to use labels, then enter label text into the **Labels** column.

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 define the name, base data type, and chosen yes to enumeration. When you paste two columns of data, click 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 8: Custom Data Type, example 1

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

Figure 9: Custom Data Type, example 2

Data Type Name	Base Data Type	Enumerati...	Constraint Expression	Label	Value
containerType	String	Yes		standard	'standard'
PrimeNumbers	Integer	Yes		oversize	'oversize'
USHolidays2020	Date	Yes		heavyweight	'heavyweight'
ShirtSize	Integer	Yes		reefer	'reefer'
RiskProfile	Integer	Yes			
DevTeam	String	Yes			

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

Figure 10: Custom Data Type, example 3

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 11: Custom Data Type, example 4

Data Type Name	Base Data Type	Enumerati...	Constraint Expression	Label	Value
containerType	String	Yes		S	1
PrimeNumbers	Integer	Yes		M	2
USHolidays2020	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 12: Custom Data Type, example 5

Data Type Name	Base Data Type	Enumerati...	Constraint Expression	Label	Value
containerType	String	Yes		Low	1
PrimeNumbers	Integer	Yes		Medium	2
USHolidays2020	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 13: Custom Data Type, example 6

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

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

Enumerations retrieved from a database

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

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

Note: This functionality uses Corticon's Enterprise Data Connector. For more information, see *"Import possible values of an attribute from database tables" in the "Advanced EDC Topics" section of the Data Integration Guide*

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

Figure 14: Custom Data Type columns for defining database retrieval

These columns are how you specify:

- **Lookup Table Name**—The database syntax that specifies the table that has the enumerations.

- **Labels Column**—The column in the lookup table that holds the label. This is optional because 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 15: SQL Server table with values to use in the Vocabulary
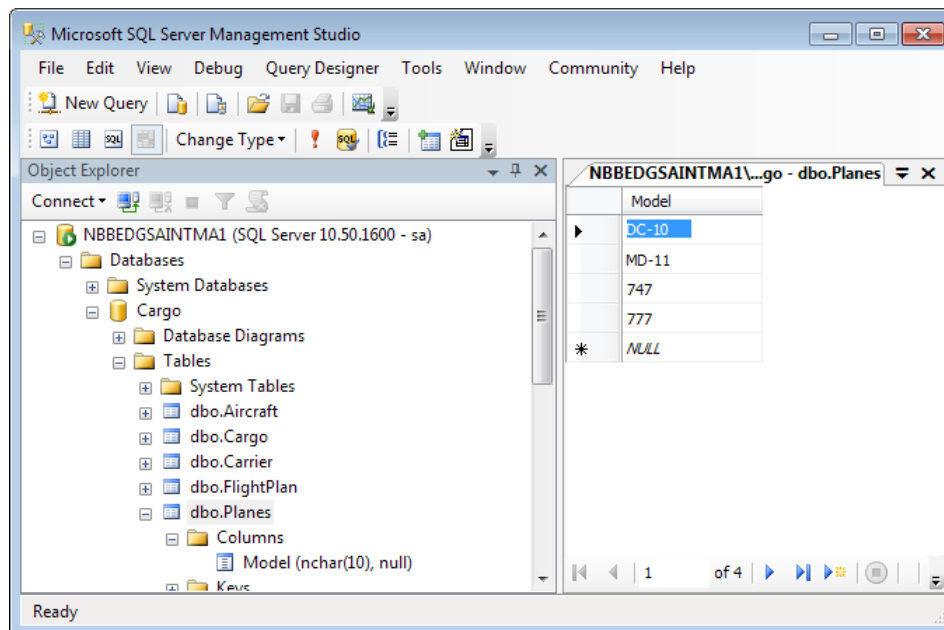
The value data is retrieved into the Vocabulary as highlighted:

Figure 16: Definition and retrieved values in Corticon Studio
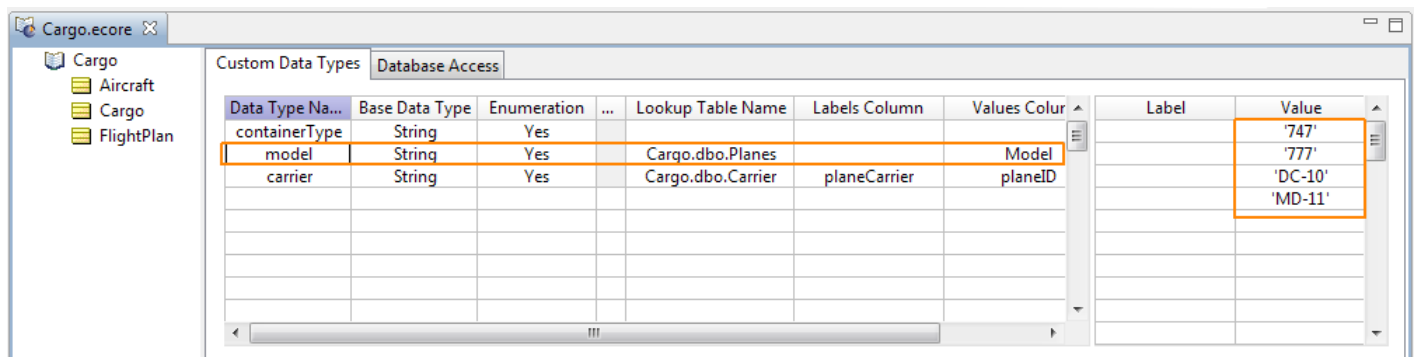
Another example retrieves name-value pairs.

Figure 17: SQL Server table with labels and values to use in the Vocabulary
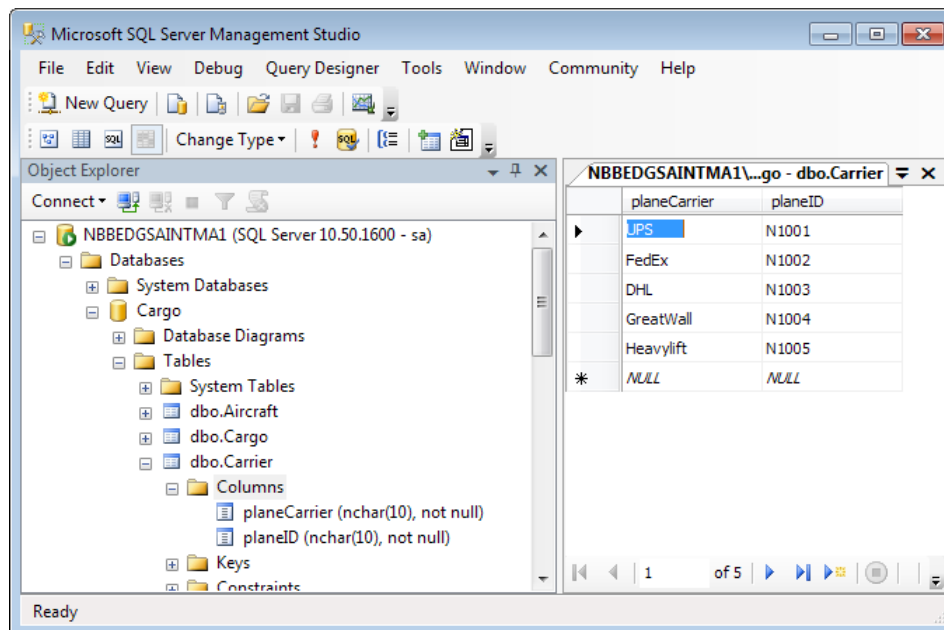
The label/value data is retrieved into the Vocabulary as highlighted:

Figure 18: Definition and retrieved label-values in the Corticon Studio
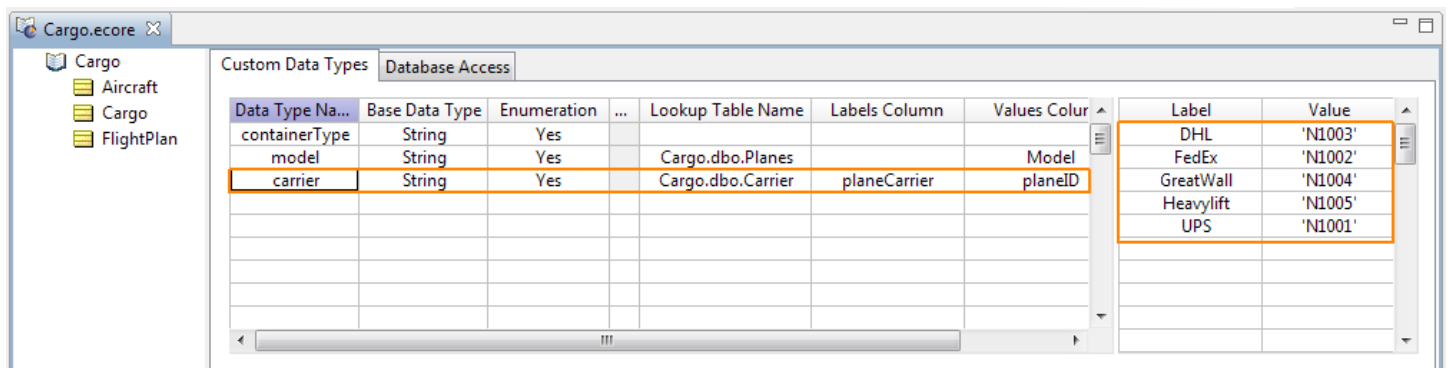
How to use Custom Data Types

Custom data types are powerful additions to your Vocabulary that propagate their effects into Rulesheets and Ruletests.

Use Custom Data Types in a Vocabulary

After a Custom Data Type is defined as shown, it can be used and reused throughout the Vocabulary's attribute definitions.

Figure 19: Using Custom Data Types in the Vocabulary
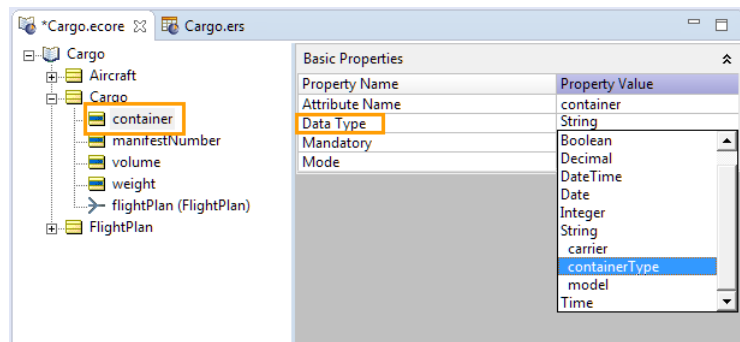
Notice in this figure that multiple attributes can use the same custom data type; the custom data type `containerType` is shown in the drop-down as a sub-category of the String-based data type. The other custom data types will be grouped with their base data types as well.

Use enumerated Custom Data Types in Rulesheets

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

Figure 20: Using Custom Data Types in the Rulesheet
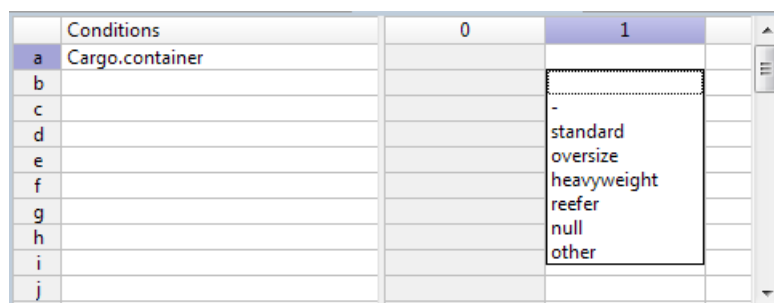
You can test a condition bound to an attribute by evaluating the attribute against a custom data type label using the # tag, as shown:

Figure 21: Using # tag to test a custom data type
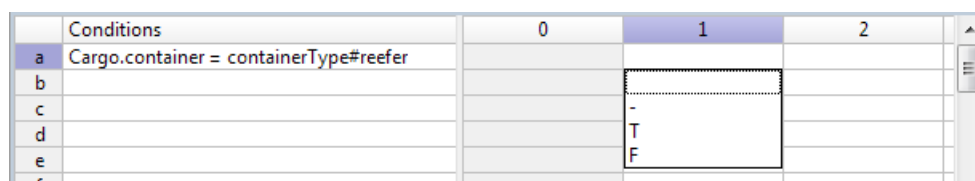
Note: Using a dot instead of a # tag works, but if there is custom data type with the same name as an entity, then the expression will be invalid.

Use enumerated Custom Data Types in Ruletests

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

Figure 22: Ruletest selecting container's containerType list
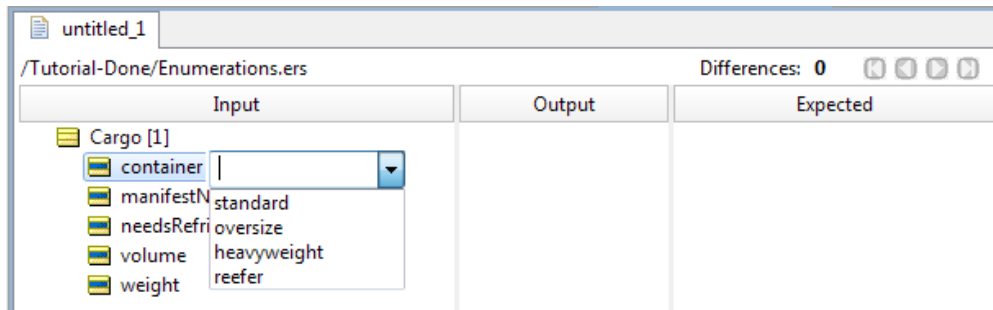
If you want the attribute value to be null, right-click the attribute, and then select **Set to Null**, as shown:
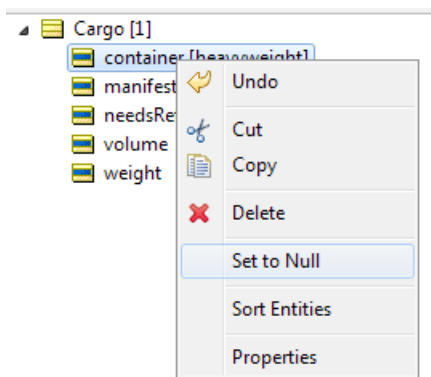
Use IN operator with an enumerated list

When your rule condition or filter is not defined by a range of values, you might have try to 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 Qualify rules with ranges and lists on page 94.

How to relax 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 were set in the Vocabulary's Custom Data Types.

Note: Progress recommends 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 preproduction 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 to 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 stops on the first `Violation`, as shown:

untitled_1

/Tutorial/Cargo.ers

Input	Output	Expected
Cargo [1]	Cargo [1]	
container [secure]	container [secure]	
volume [-1]	volume [-1]	
weight [100]	weight [100]	

Rule Messages

Severity	Message	Entity
Violation	An unexpected error occurred in Input Data: com.corticon.cdo.ConstraintViolationException: constraint violation setting Cargo.container to value [secure]	

Properties | Problems | Error Log

0 errors, 2 warnings, 0 others

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 information into the log.) No further processing occurs.

Note: See the topic *"How to change logging configuration" in the Using Corticon Server logs section of Server Guide.*

You can set the property in `brms.properties` that relaxes enforcement of Custom Data Types, `com.corticon.vocabulary.cdt.relaxEnforcement=true`, and then restart 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.

untitled_1

/Tutorial/Cargo.ers

Input	Output	Expected
Cargo [1]	**Cargo [1]**	
container [secure]	**container [standard]**	
volume [-1]	volume [-1]	
weight [100]	weight [100]	

Rule Messages

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]

Properties | Problems | Error Log

0 errors, 2 warnings, 0 others

Description	Resource	Location	Path	Type
Warnings (2 items)				
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: Progress recommends 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 *"Server properties and settings" in the Server Guide* for more for information about 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*. Domains allow you to bundle one or more entities in a *subset* within the Vocabulary, thus you can reuse entity names as 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**, 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 23: Using domains in the Vocabulary>
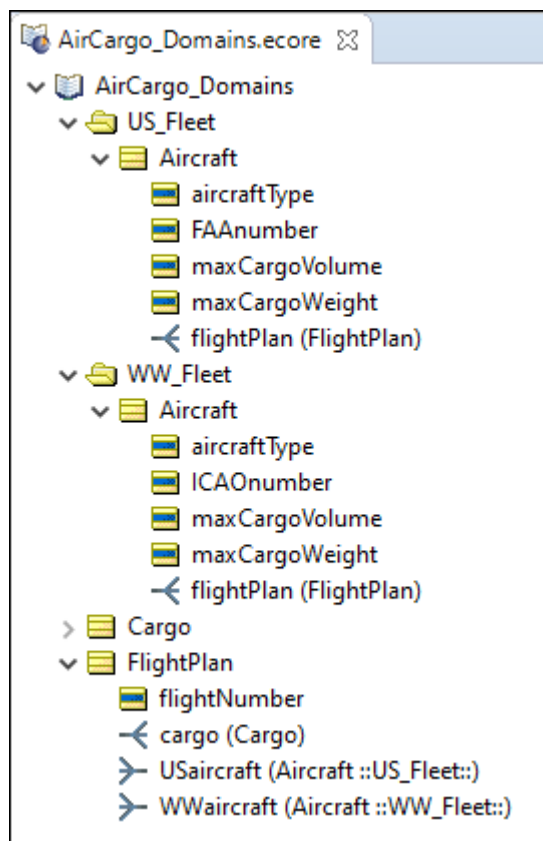
Notice that the entity `Aircraft` appears in each Domain, using the same spelling and containing slightly different attributes (`FAAnumber` vs. `ICAOnumber`). Notice, too, that the association role names from `FlightPlan` to `Aircraft` were named manually to ensure uniqueness: one is now `USaircraft` 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 24: Non-unique Entity names prior to defining Aliases
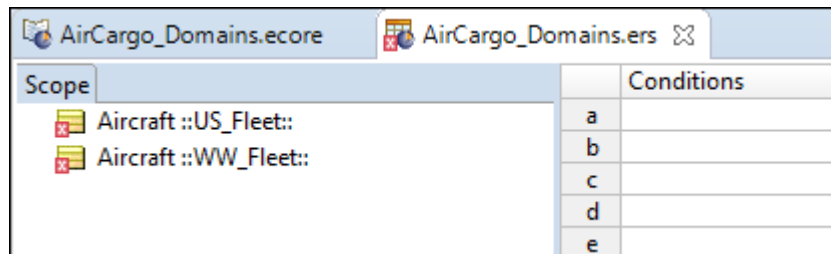
In *Non-unique Entity names prior to defining Aliases*, both `Aircraft` entities have been dropped into the **Scope** section of the Rulesheet. But because their names are not unique, an error icon ⊞ appears. Also, the "fully qualified" domain name has been added after each to distinguish them. By fully 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, you must define a unique alias for each. The aliases will be used in the Rulesheet expressions, as shown:

Figure 25: Non-unique Entity names after defining Aliases
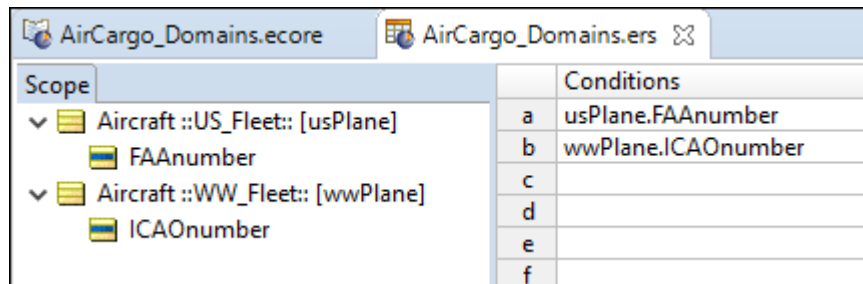
Domains in a Ruletest

When using Vocabulary terms in a Ruletest, drag and drop them as usual. Notice that they are automatically labeled with the fully qualified name, as shown:

Figure 26: Domains in a Ruletest
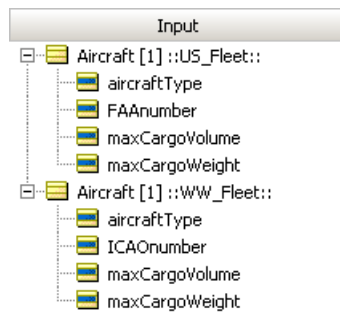
Support for inheritance

UML Class diagrams frequently include a modeling/programming concept called inheritance, where a class can "inherit" attributes and associations from another class, for example:

Figure 27: Rose UML model showing inheritance
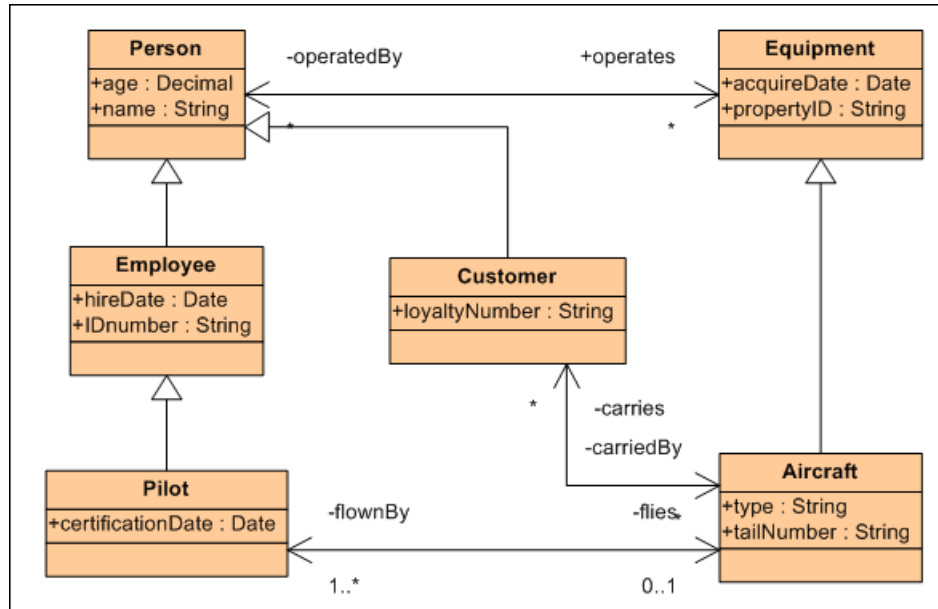
In this diagram, there is a UML model that includes inheritance. The solid-headed arrow indicates that the `Employee` class is a descendant of the `Person` class, and therefore inherits some of its properties. Specifically, the `Employee` class inherits the `age` and `name` attributes from `Person`. In other words, `Employee` has all the same attributes of `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 per normal practice. To incorporate the elements of inheritance, you only need to add one additional setting for each of the descendant entities, as shown:

Figure 28: Selecting Ancestor Entity for Descendant
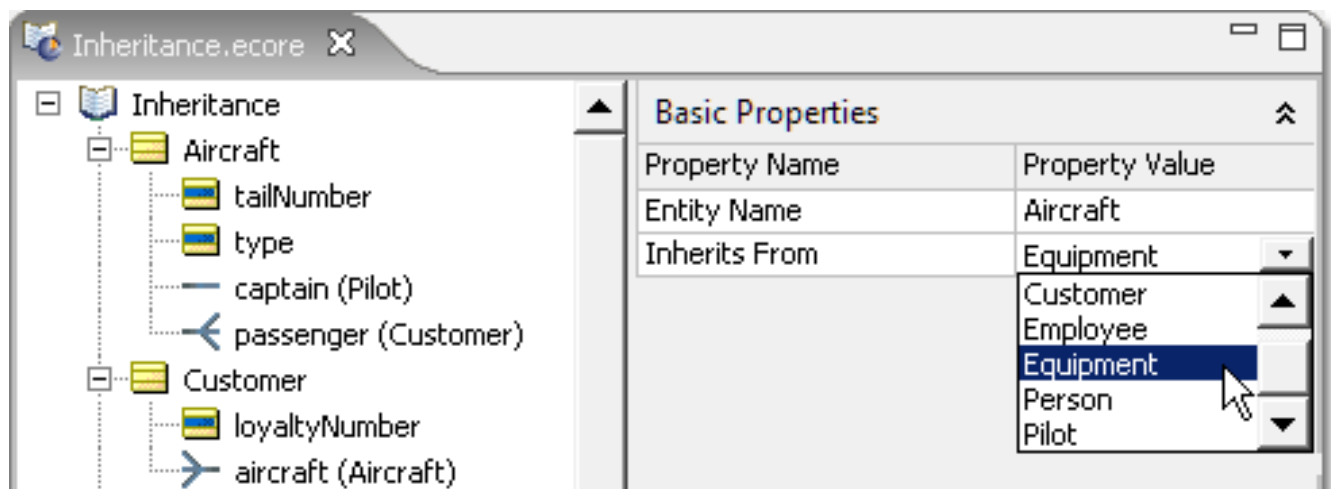
After all descendant entities are configured to inherit from their proper ancestor entities, you can save the Vocabulary and view it in the **Rule Vocabulary** window:

Figure 29: Vocabulary with Inheritance
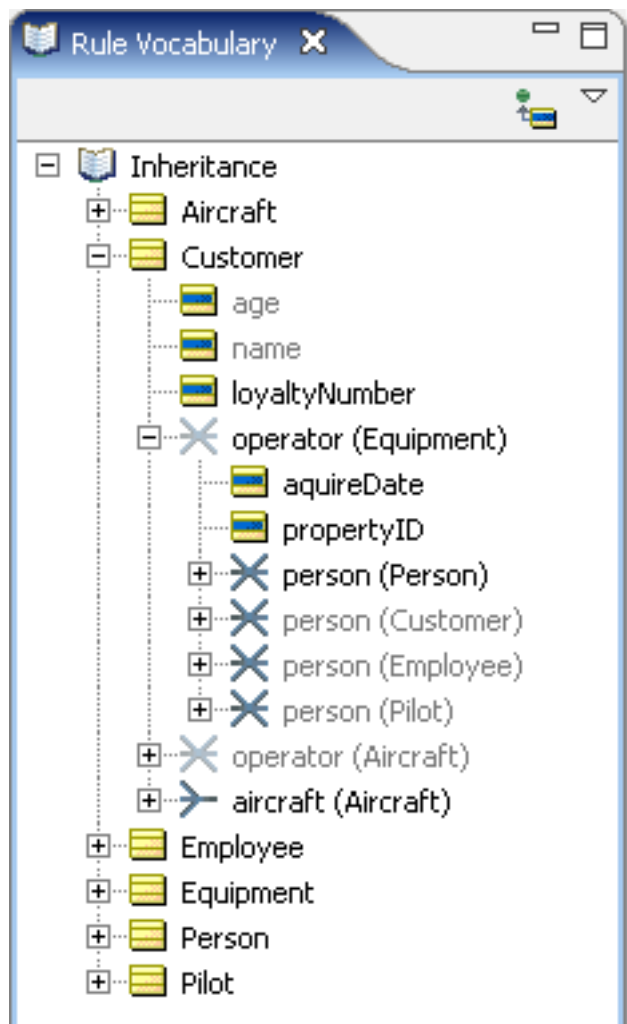
Notice that many of the term names and icons are varying shades of gray. These color codes help you to understand the inherited relationships that exist in the Vocabulary.

Inherited attributes

Attributes with names displayed in **solid black type**, such as `Customer.loyaltyNumber` in , 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 can be directly associated with another entity or that entity's descendants. An entity can also inherit an association from its ancestor.

Using the example shown in and above, each of these combinations is described:

- `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 the `Customer` ancestor entity `Person`, which has a direct association with `Equipment` and the rolename `operator` in our Vocabulary. The UML class Diagram in Figure 28: Selecting Ancestor Entity for Descendant on page 51 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 you 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 the association icon and the rolename are **black**. But, `Equipment` also has associations with descendants of the `Person` entity, specifically `Employee`, `Customer`, and `Pilot`. These are *filtered* associations, and display their rolenames in **dark gray.**

- Finally, `Customer` has another association with `operator` (Aircraft) because `Aircraft` is a descendant of `Equipment`. So, the *inherited* **dark gray** icon and the *filtered* **dark gray** rolename are combined to display this association.

How to control the tree view

In cases where a Vocabulary contains inheritance (and includes the various icons and color schemes previously described), 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 30: Vocabulary with inheritance properties hidden
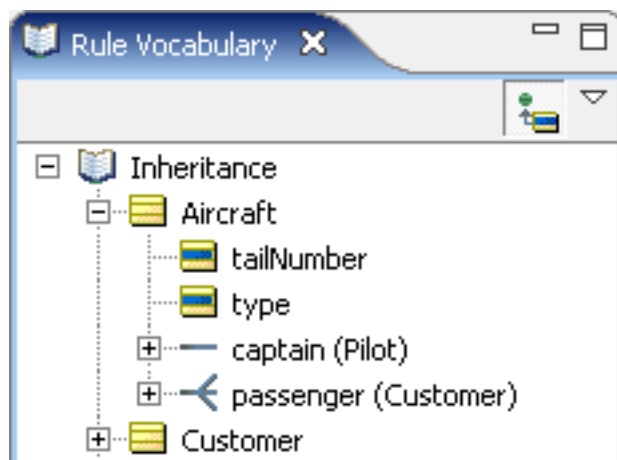
`Person` and `Equipment` are associated (using named roles), but what relationship does `Employee` have with `Equipment` or `Aircraft`, if any?

How to use 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, then they must be assigned unique alias names before they can be used in rules.

For example, in Figure 29: Vocabulary with Inheritance on page 52, 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, they will be assigned error icons ✖ to indicate that their names are not unique, as shown:

Figure 31: Non-Unique nodes used in the Scope window
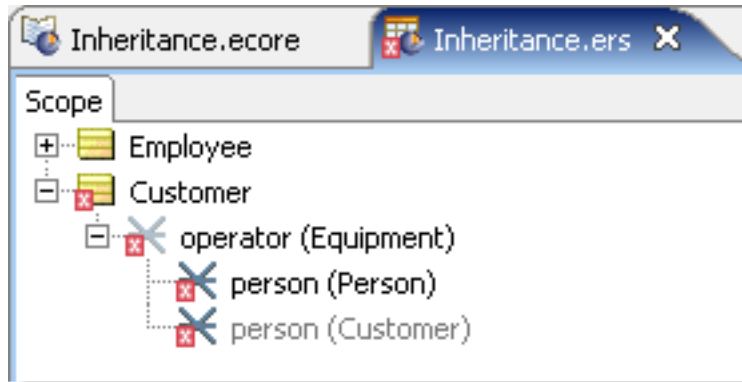
Without unique names, Corticon Studio does not know which one is intended in any rule that uses one of the nodes. To ensure uniqueness, aliases must be assigned and used in rules, as shown:

Figure 32: Uniqueness Established using an Alias
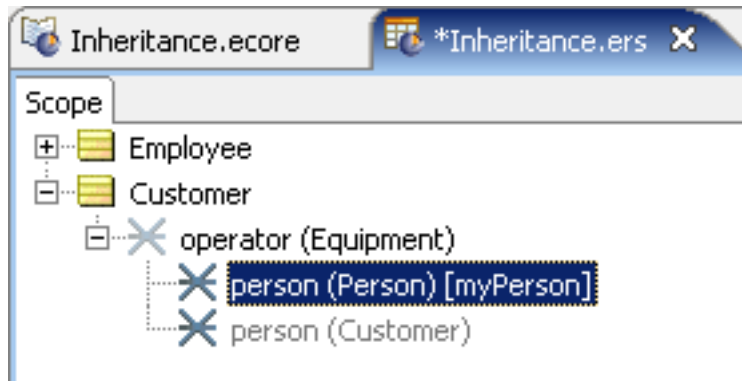
Effects of inheritance on rule execution

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.

Examples of effects of inheritance on rule execution are:

- **Inherited Conditions and Actions**

  The following Rulesheet 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 33: Rules written on Employee

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

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 important from a rule execution perspective, a `Pilot` is also be affected by any rules that affect `Employee`, as shown:

Figure 34: Inheritance in action

Input	Output
⊟ 🗒 Employee [1]	⊟ 🗒 Employee [1]
🟦 age [24]	🟦 age [24.000000]
🟦 hireDate	🟦 hireDate
🟦 IDnumber	🟦 IDnumber
🟦 name [Joe Smith]	🟦 name [Joe Smith]
⊟ 🗒 Pilot [1]	⊟ 🗒 Pilot [1]
🟦 age [12]	🟦 age [12.000000]
🟦 certificationDate	🟦 certificationDate
🟦 hireDate	🟦 hireDate
🟦 IDnumber	🟦 IDnumber
🟦 name [Mary Jones]	🟦 name [Mary Jones]

Rule Statements | ✉ Rule Messages ✕

Severity	Message	Entity
Info	[Joe Smith] is old enough to be an employee	Employee[1]
Warning	[Mary Jones] is too young to be an employee!	Pilot[1]

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, you test `Employee.hireDate` to see who is 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 35: Rulesheet populating the operators collection

  In the following Ruletest, there is a sample `Equipment` entity, one `Employee`, and one `Pilot`. Both hire dates satisfy the rule's Condition (the `Pilot` inheriting `hireDate` from its `Employee` ancestor as before). When `Employee` was 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 36: Inheriting an Association
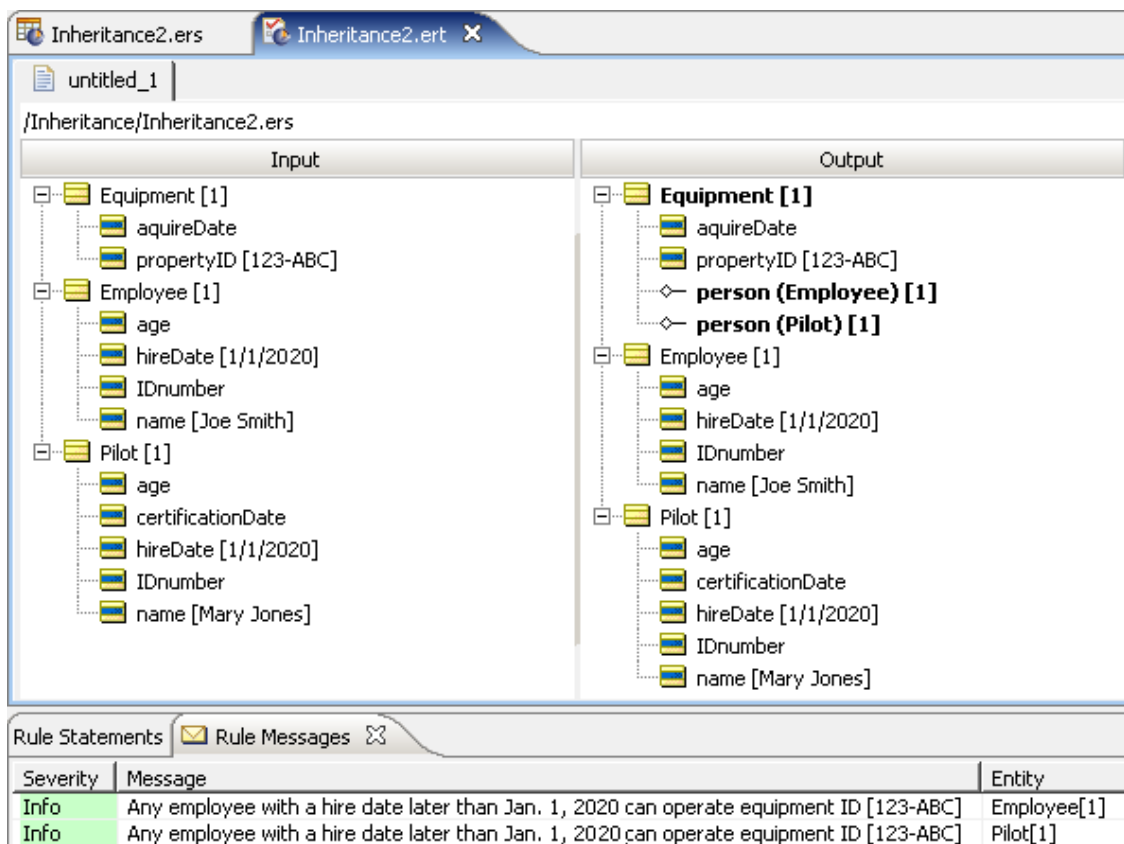
TestYourself questions for Build the vocabulary

Note: Try this test, and then go to TestYourself answers for Building the vocabulary on page 334 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 21.

23. List the six steps in constructing a Vocabulary.

24. Cardinality of an association determines:

    a. The number of possible associated entities.

    b. The number of attributes for each entity.

    c. The number of associations possible within an entity.

    d. The number of attributes for each association.

25. The Vocabulary terms are the nouns of Corticon rules. What are the verbs?

26. What Corticon document contains the complete list of all Vocabulary operators, descriptions of their usage, and actual examples of use in Rulesheets?

27. True or False. In addition to the supported vocabulary data types, you can create *any* type of custom data type you want?

28. You must name your custom data type. Which of the following are *not* custom data type naming convention requirements?

    a. Cannot match any other vocabulary entity names

    b. May match other Custom Data Type Names

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

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

29. True or False. The Enumeration section of the Custom Data Types 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 Contraint Expression. Give an example of a Constraint Expression:

_____

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

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

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

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

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

35. Explain what Domains do in the Vocabulary?

36. True or False. If you use a Domain, then you are 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 have inherited attributes?
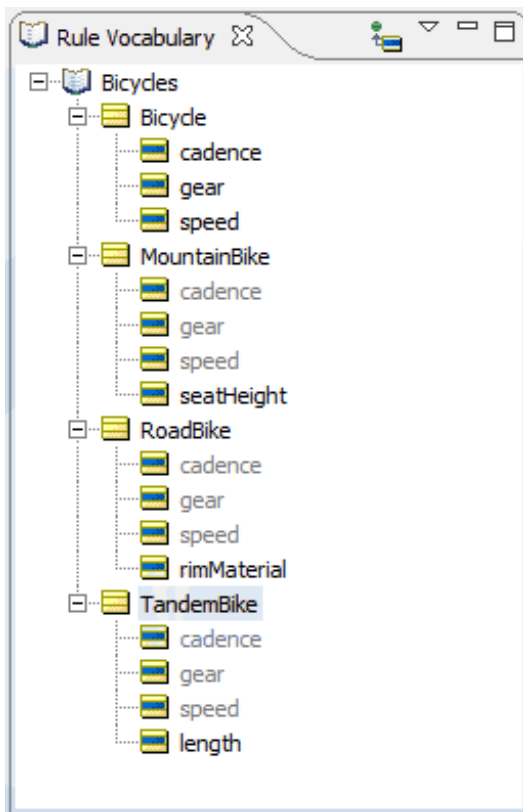
39. Give two examples of inherited attributes from the preceding vocabulary:

_____ _____

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 preceding Vocabulary example.

3

Rule scope and context

The air cargo example that you 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 37: Fact model
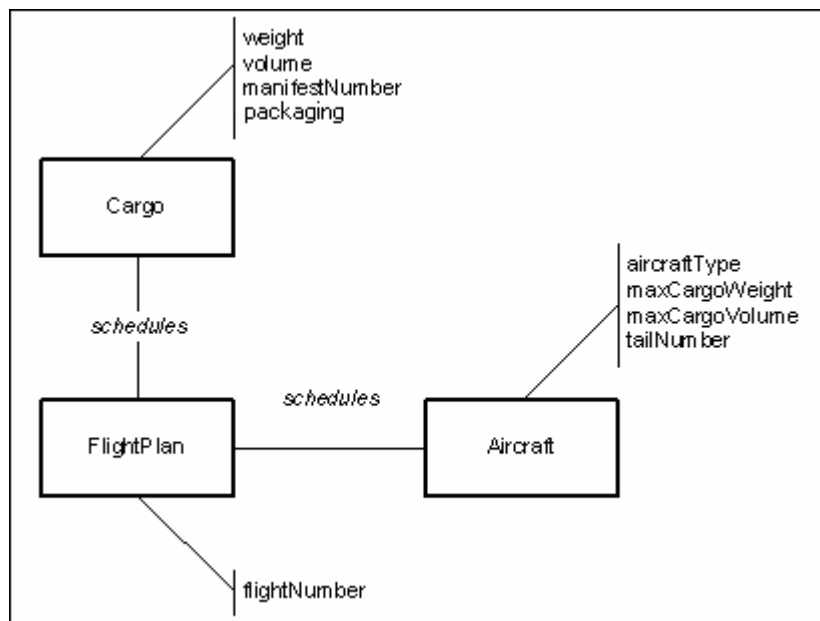
According to this Vocabulary, an aircraft is related to a cargo shipment through a flight plan. In other words, it is the flight plan that connects or relates an aircraft to its cargo shipment. The aircraft, by itself, has *no direct relationship* to a cargo shipment unless it is scheduled by a flight plan; or, no aircraft may carry a cargo shipment without a flight plan. Similarly, no cargo shipment can be transported by an aircraft without a flight plan. These facts constitute business rules in and of themselves and constrain creation of other rules because they define the Vocabulary you 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 your Vocabulary created, you 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 write your rule using the root-level terms in the Vocabulary. In the following figure, column #1 (the **true** Condition) is the rule you should be most interested in. The **false** condition in column #2 was added simply to show a logically complete Rulesheet.

Figure 38: Expressing the rule using root-level Vocabulary terms
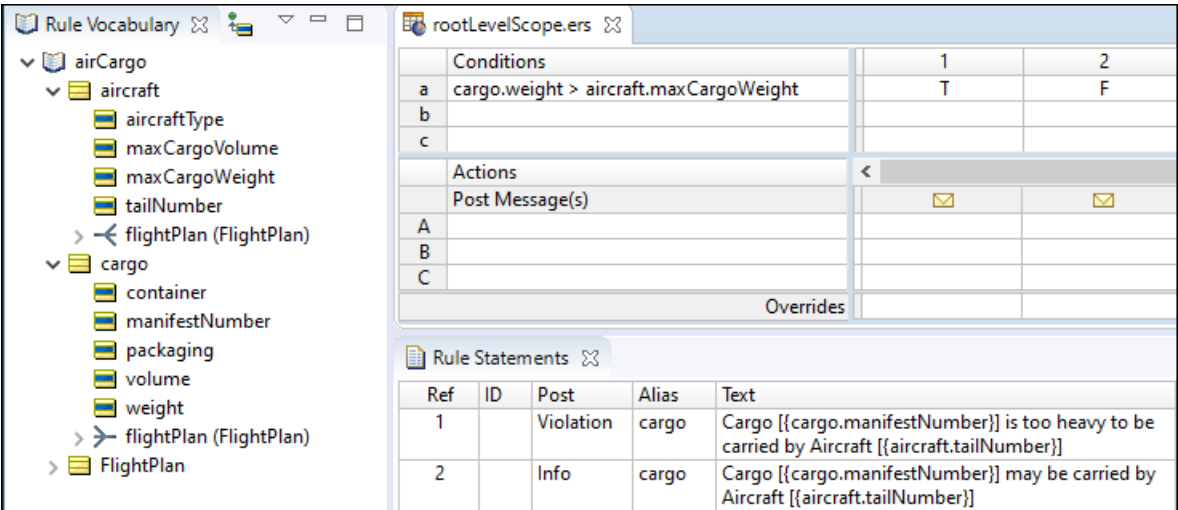
You can build a Ruletest to test the rule using the Cargo company's actual data, as follows:

The company owns 3 Aircraft, 2 747s and a DC-10, each with different tail numbers. Each box represents a real-life example (or *instance*) of the `Aircraft` term from your Vocabulary.

Figure 39: The Cargo Company's 3 Aircraft

```
                Aircraft

  aircraftType: 747
  maxCargoWeight: 200,000 kgs.
  tailNumber: N1001


  aircraftType: 747
  maxCargoWeight: 200,000 kgs.
  tailNumber: N1002


  aircraftType: DC-10
  maxCargoWeight: 150,000 kgs.
  tailNumber: N1003
```
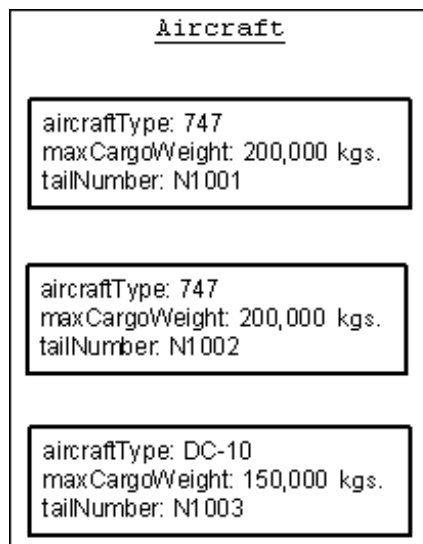
These aircraft give the company the ability to schedule 3 cargo shipments each night. Another business rule is implied:– "an `Aircraft` cannot be scheduled for more than one flight per night". This rule is not addressed now because it is not relevant to the discussion}. On a given night, the cargo shipments look like those shown. Again, like the `Aircraft`, these cargo shipments represent specific *instances* of the generic `Cargo` term from the Vocabulary.

Figure 40: The three cargo shipments for the night of June 25th

```
                Cargo

  weight: 100,000 kgs.
  volume: 300 cubic meters
  manifestNumber: 625A


  weight: 175,000 kgs.
  volume: 300 cubic meters
  manifestNumber: 625B


  weight: 150,000 kgs.
  volume: 300 cubic meters
  manifestNumber: 625C
```
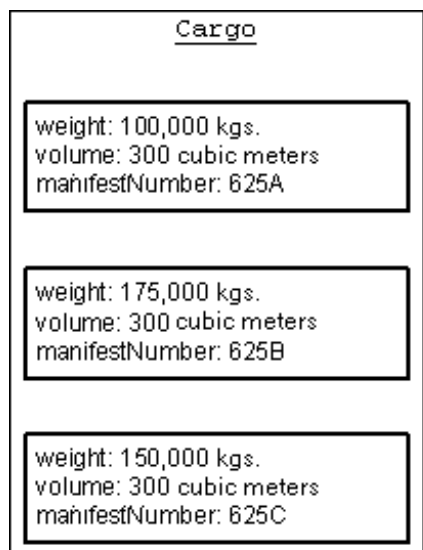
Finally, the sample business process manually matches specific aircraft and cargo shipments together as three flight plans, as shown below. This organization of data is consistent with the structure and constraints implicit in the Vocabulary.

Figure 41: The three FlightPlans with their related Aircraft and Cargo instances
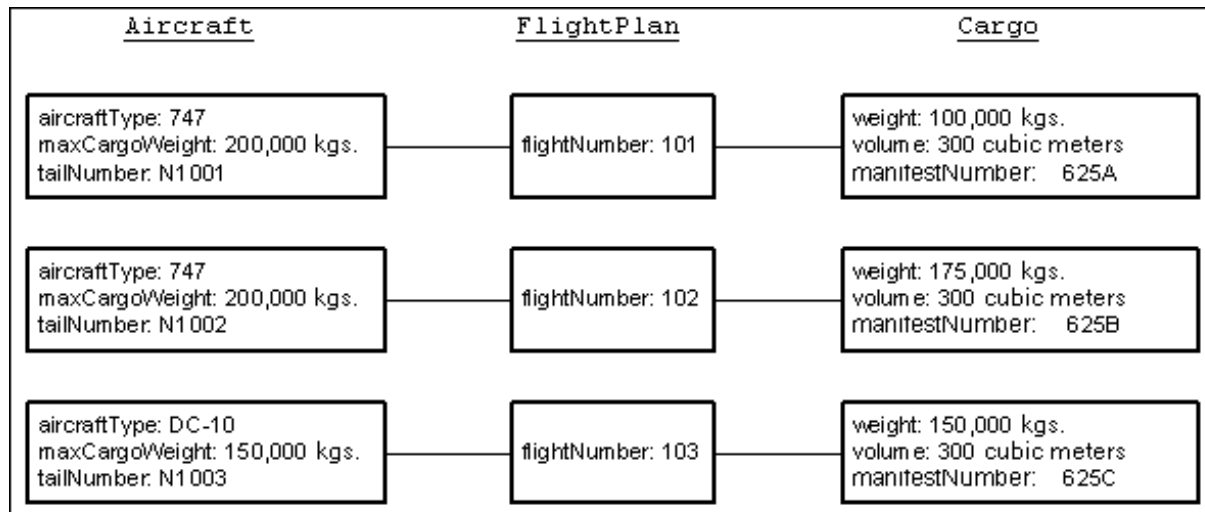
You can construct a Ruletest (in the following figure) so that the company's actual data is evaluated by the rule. Because the rule used root-level Vocabulary terms in its construction, you use root-level terms in the Ruletest:

Figure 42: Test the rule using root-level Vocabulary terms
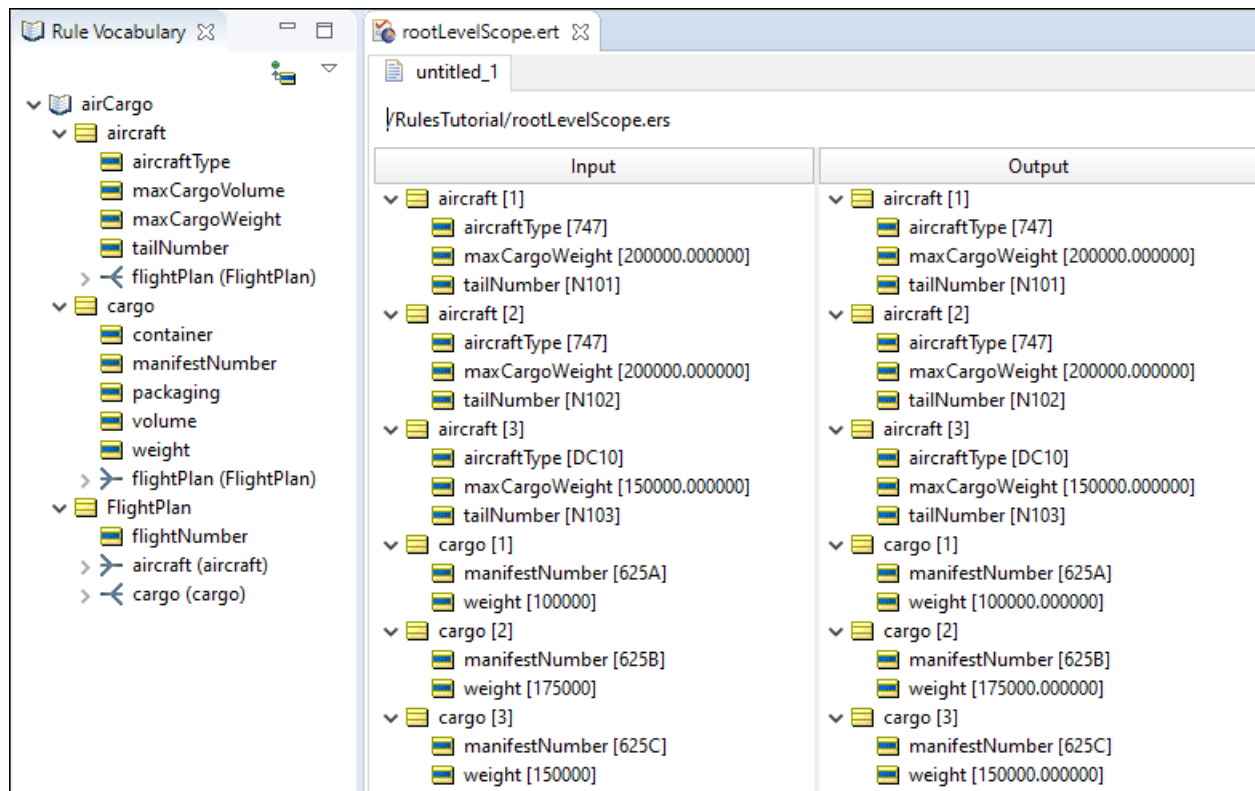
Running the Ruletest:

Figure 43: Results of the Ruletest

airCargo.ecore	rootLevelScope.ers	rootLevelScope.ert ⌗

📄 untitled_1

/RulesTutorial/rootLevelScope.ers

Input	Output
⌄ 🗂 aircraft [1]	⌄ 🗂 aircraft [1]
🔲 aircraftType [747]	🔲 aircraftType [747]
🔲 maxCargoWeight [200000.000000]	🔲 maxCargoWeight [200000.000000]
🔲 tailNumber [N101]	🔲 tailNumber [N101]
⌄ 🗂 aircraft [2]	⌄ 🗂 aircraft [2]
🔲 aircraftType [747]	🔲 aircraftType [747]
🔲 maxCargoWeight [200000.000000]	🔲 maxCargoWeight [200000.000000]
🔲 tailNumber [N102]	🔲 tailNumber [N102]
⌄ 🗂 aircraft [3]	⌄ 🗂 aircraft [3]
🔲 aircraftType [DC10]	🔲 aircraftType [DC10]
🔲 maxCargoWeight [150000.000000]	🔲 maxCargoWeight [150000.000000]
🔲 tailNumber [N103]	🔲 tailNumber [N103]
⌄ 🗂 cargo [1]	⌄ 🗂 cargo [1]
🔲 manifestNumber [625A]	🔲 manifestNumber [625A]
🔲 weight [100000]	🔲 weight [100000.000000]
⌄ 🗂 cargo [2]	⌄ 🗂 cargo [2]
🔲 manifestNumber [625B]	🔲 manifestNumber [625B]
🔲 weight [175000]	🔲 weight [175000.000000]
⌄ 🗂 cargo [3]	⌄ 🗂 cargo [3]
🔲 manifestNumber [625C]	🔲 manifestNumber [625C]
🔲 weight [150000]	🔲 weight [150000.000000]

📄 Rule Statements   ✉ Rule Messages ⌗

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

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`. You already know that there are only three `Flightplans`. And you also know, from examination of Figure 41: The three FlightPlans with their related Aircraft and Cargo instances on page 66, that the combination of aircraft `N1003` and cargo `625C` does not appear in any of the three `Flightplans`. So, why was this combination, one that does not actually exist, evaluated? For that matter, why did the rule fire *nine* times when only *three* sets of Aircraft and Cargo were present? The answer is in the way the rule was defined, and in the way the rules engine evaluated it.

The Ruletest has 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. Because 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. Progress uses cross-product when describing this outcome.

Figure 44: All possible combinations of Aircraft and Cargo
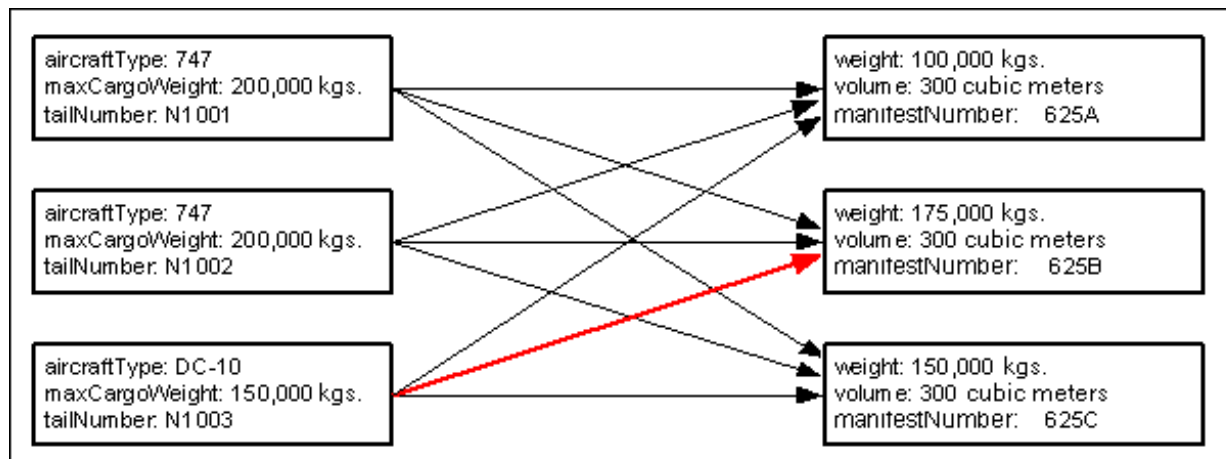
One pair, the combination of manifest `625B` and plane `N1003` (shown as the red arrow in the preceding figure), is illegal, because the plane, a `DC-10`, can only carry `150,000` kilograms, while the cargo weighs `175,000` kilograms. But, this pairing does not correspond to any of the three `FlightPlans` created. Many of the other combinations evaluated (five others) are not represented by real flight plans either. So why did Studio perform three times the necessary evaluations? It is because the rule, as implemented in Figure 38: Expressing the rule using root-level Vocabulary terms on page 64, does not capture the essential elements of **scope** and **context**.

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

Refer to the following figure:

Figure 45: Rule expressed using FlightPlan as the Rule Scope

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

Note: *Inside* means that the `Aircraft` and `Cargo` terms that appear when the `FlightPlan` term is opened in the Vocabulary tree, as shown by the orange highlights in **Rule expressed using FlightPlan as the Rule Scope**.

This is significant. It means that you 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, only those cargo shipments *in the context of* actual flight plans should be evaluated. To put it differently, the rule's application is limited to only those cargo shipments assigned to a specific aircraft using a specific flight plan. 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`, you indicate mandatory *traversals* of the associations between `FlightPlan` and the other two terms, `Aircraft` and `Cargo`. This instructs the rules engine to evaluate the rule using the intended context. When writing rules, it is 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

- TestYourself questions for Rule scope and context

Rule scope

Because the rule evaluates both `Cargo` and `Aircraft` in the context of `FlightPlan`, 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 `cargo.weight` is compared to `aircraft.maxCargoWeight` only **if** both `cargo` and `aircraft` share the same `FlightPlan`. This simplifies rule expression greatly, because it eliminates the need to specify *which* `FlightPlan` is referred to 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* flight plan to be evaluated by the rule. And, because Corticon Studio automatically handles multiple instances as *collections*, it sends *all* pairs to the rule for evaluation.

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

To test this new rule, structure your Ruletest to correspond to the new structure of your rule and reflect the rule's scope. For more information about the mechanics of creating associations in Ruletests, see and *"Add and edit association nodes and their properties" and "Create associations in the test tree" in the Quick Reference Guide.*

Finally, one `FlightPlan` is created for each `Aircraft-Cargo` pair. This means that a total of three `FlightPlans` are generated each night. Using the terms in your Vocabulary *and the relationships between them*, Figure 41: The three FlightPlans with their related Aircraft and Cargo instances on page 66 shows the possibilities. The rule evaluates these combinations and identifies any violations.

Figure 46: New Ruletest using flight plan as the rule scope
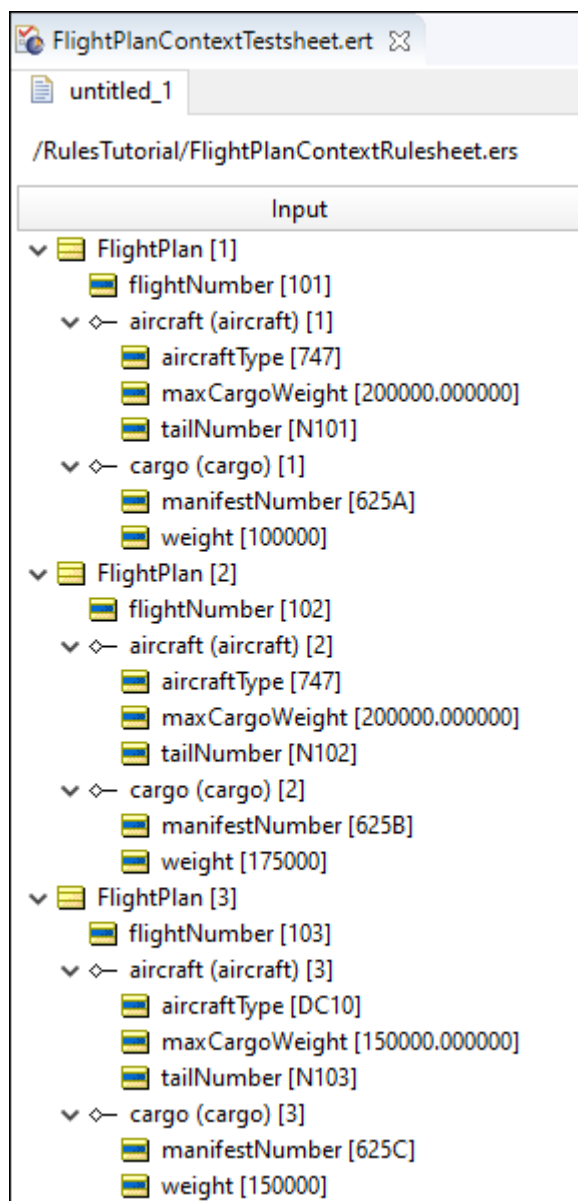
What is the expected result from this Ruletest? If the results follow the same pattern as in the first Ruletest, you might expect the rule to fire nine times (three `Aircraft` evaluated for each of three `Cargo` shipments).

In the following figure you see that the rule fired only three times, and only for those `Aircraft-Cargo` pairs that are related by common flight plans. This is the result that you want. The Ruletest shows that there are no `FlightPlans` in violation of the rule.

Figure 47: Ruletest results using scope – note no violations

One final point about scope: it is critical that the context you choose for your rule supports the intent of the business decision you are modeling. At the beginning of the example, the purpose of the application was to check flightplans *that have already been created.* Therefore, the context of the 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 using a common flight plan.

But what if the business purpose was different? What if the problem trying to be solved is modified to: Of all possible combinations of aircraft and cargo, determine which pairings must **not** be included in the same flight plan. The difference here is subtle but important. Before, you were identifying invalid combinations of pre-existing flight plans. Now, you 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 you built, root-level context, is the appropriate way to implement the rule, because now you are looking at all possible combinations *prior to creating new flight plans*.

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

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

Figure 48: Defining an alias in the Scope window
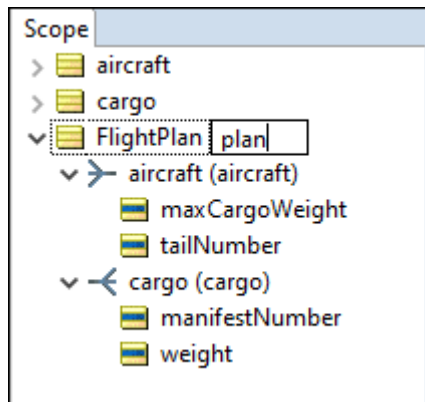
After an alias is defined, any subsequent rule modeling in the Rulesheet automatically substitutes the alias for the Vocabulary term it represents.

In the next illustration, notice that the terms in the Condition rows of the Rulesheet do not show the `FlightPlan` term. That is because the alias `plan` substitutes for `FlightPlan`.

Figure 49: Rulesheet with FlightPlan alias declared in the Scope section

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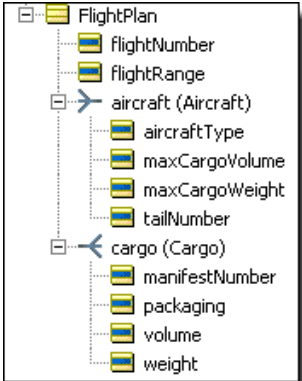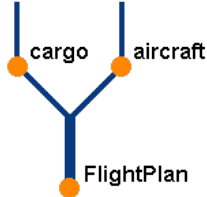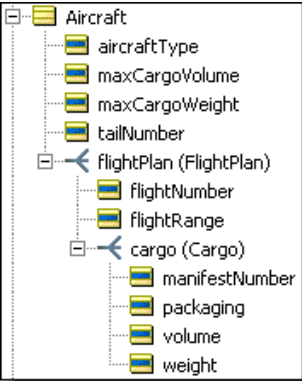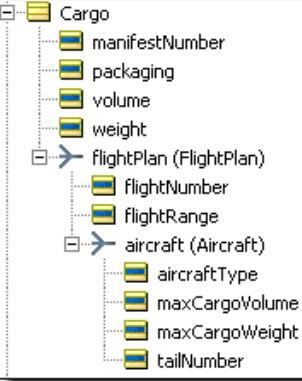
Aliases work in all sections of the Rulesheet, including the **Rule Statement** section. Modifying an alias name defined in the **Scope** section causes the name to update everywhere it is used in the Rulesheet.

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 the Vocabulary is organized as a tree 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: Vocabulary Tree Views and Corresponding Branch Diagrams

Vocabulary Tree	Description	Branch Diagram

FlightPlan
  flightNumber
  flightRange
  aircraft (Aircraft)
    aircraftType
    maxCargoVolume
    maxCargoWeight
    tailNumber
  cargo (Cargo)
    manifestNumber
    packaging
    volume
    weight
	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 `FlightPlan` *root*, the branches also originate with the `FlightPlan` root or trunk. The `FlightPlan`'s associated `cargo` and `aircraft` terms are branches from the trunk.
	
	Any rule expression that uses `FlightPlan`, `FlightPlan.cargo`, or `FlightPlan.aircraft` is using scope from this perspective of the Vocabulary tree.	

Aircraft
  aircraftType
  maxCargoVolume
  maxCargoWeight
  tailNumber
  flightPlan (FlightPlan)
    flightNumber
    flightRange
    cargo (Cargo)
      manifestNumber
      packaging
      volume
      weight
	This portion of the Vocabulary tree begins with `Aircraft` as the root, with its associated `flightPlan` branching from the root. A `cargo`, in turn, branches from its associated `flightPlan`.
	
	Any rule expression that uses `Aircraft`, `Aircraft.flightPlan`, or `Aircraft.flightPlan.cargo` is using scope from this perspective of the Vocabulary tree.	

Cargo
  manifestNumber
  packaging
  volume
  weight
  flightPlan (FlightPlan)
    flightNumber
    flightRange
    aircraft (Aircraft)
      aircraftType
      maxCargoVolume
      maxCargoWeight
      tailNumber
	This portion of the Vocabulary tree begins with `Cargo` as the root, with its associated `flightPlan` branching from the root. An `aircraft`, in turn, branches from its associated `flightPlan`.
	
	Any rule expression that uses `Cargo`, `Cargo.flightPlan`, or `Cargo.flightPlan.aircraft` is using scope from this perspective of the Vocabulary tree.	

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 the root term `Aircraft`, including `Aircraft.flightPlan` and `Aircraft.flightPlan.cargo`, also apply to this data and its associated collections. Likewise, a rule written with the 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 the former are written using different root terms and therefore different scopes, whereas the rules in the latter use the same FlightPlan root and therefore share common scope.

How to use roles

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

Figure 50: UML Class Diagram without Roles
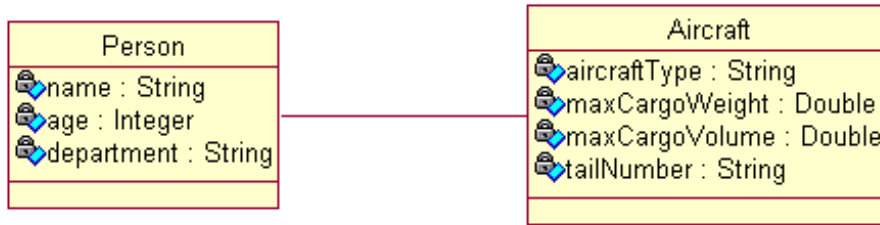
As shown in this class diagram, the entities Person and Aircraft are joined by an association. However, can this single association sufficiently represent multiple relationships between these entities? For example, a prior Fact Model might state that "a pilot flies an aircraft" and "a passenger rides in an aircraft". Both pilot and passenger are descendants of the entity Person. Furthermore, 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 you represent this in the Vocabulary and rules in Corticon Studio?

Assume that you 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

These rules are called *cross-entity* because they include more than one entity (both `Aircraft` and `Person`) in the expression. Unfortunately, with the Vocabulary as it is, you have no way to distinguish between pilots and passengers, so there is no way to unambiguously implement these two rules. This class diagram, when imported into Corticon Studio, looks like this:

Figure 51: Vocabulary without roles
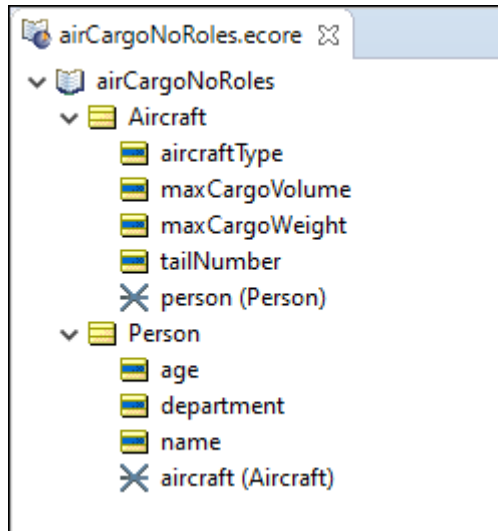
However, there are several ways to modify this Vocabulary to allow you to implement these rules.

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 you can 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, a `Person` 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 52: UML class diagram with roles
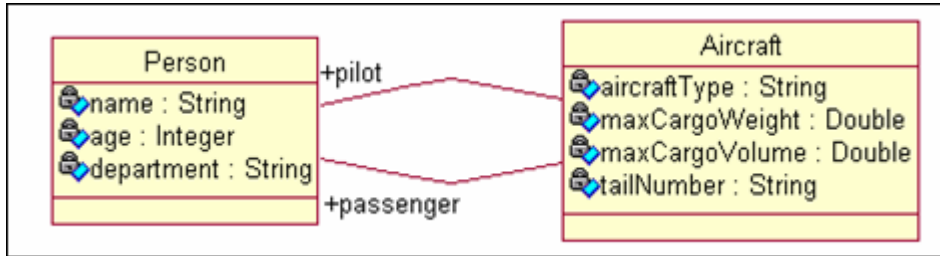
When the class diagram is imported into Corticon Studio, it appears as the Vocabulary below:

Figure 53: Vocabulary with roles
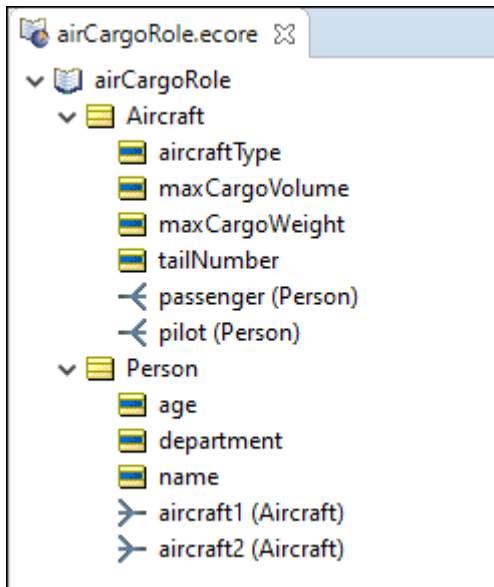
Notice the differences between the two preceding Vocabularies In **Vocabulary with Roles**, `Aircraft` contains 2 associations, one labeled `passenger` and the other `pilot`, even though both associations relate to the same `Person` entity. Also notice that the cardinalities of both `Aircraft-Person` associations have been updated to one-to-many.

Written using roles, the first rule is illustrated 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 you 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 detail in the section on Collections on page 115.

Figure 54: Rule #1 implemented using roles

Conditions	1	2	3
a plane.aircraftType	'747'	'747'	'747'
b pilotOfPlane -> size	{0, 1}	2	> 2

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

Scope: Aircraft [plane], aircraftType, pilot (Person) [pilotOfPlane]

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

Figure 55: Ruletest with no Person entities in pilot role

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

Figure 56: Ruletest with both Person entities in role of pilot

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

Figure 57: Ruletest with one Person entity in each of pilot and passenger roles

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 is not implemented.

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 the cargo example might look like the following:

Figure 58: 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 is 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 (*) to indicate those table columns that are primary keys. If a Vocabulary is connected to an external database using Datasource Configuration features, then you may notice asterisks next to attributes, indicating their designation as primary keys. See *"How Datasource information is viewed in the Vocabulary" in the Data Integration Guide* for complete details.

Notice that the `FlightPlan` table contains columns that did not appear in the Vocabulary. Specifically, `tailNumber` and `manifestNumber` exist in the `Aircraft` and `Cargo` entities, respectively, but you did not include them in the `FlightPlan` Vocabulary entity. Does this mean that your original Vocabulary was wrong or incomplete? No, the extra columns in the `FlightPlan` table are 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, `flightNumber` 101 (the first row or record in the `FlightPlan` 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), then a user could determine the weight of `Cargo` planned for `Aircraft` N1001 by traversing the relationships from the `Aircraft` table to the `FlightPlan` table, you see that `Aircraft` N1001 is scheduled to carry `Cargo` 625A. By traversing the `FlightPlan` table to the `Cargo` table, you can see 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 the Vocabulary, there are "associations" between entities. Traversing an association in the Vocabulary is equivalent to traversing a relationship between database tables. When a term like `Aircraft.tailNumber` is used 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 creates 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 (7 unique rows in the table), but the `FlightPlan` table contains only 3 unique instances of flight plans, the term `FlightPlan.aircraft.tailNumber` creates a collection of only 3 tail numbers—those instances from the `Aircraft` table that 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 the rule, Corticon Studio automatically ensures that the collection of data contains matching foreign keys. That is why, when 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 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 this simple example, for a large company with a fleet of hundreds of aircraft and several thousand unique cargo shipments every day, the system performance differences could be enormous.

TestYourself questions for Rule scope and context

Note: Try this test, and then go to to see how you did.

Use the following Vocabulary to answer the questions.

Rule Vocabulary ⊠

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

1. How many root-level entities are 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. Because the association between Actor and Role is bidirectional, you can use both Actor.roles and _____ in the 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 one role is used to associate two entities, each role name must be:

friendly	unique	colorful	melifluous

12. True or False. Rules evaluate only data that shares the same scope

13. Write a conditional expression in a Rulesheet for each of the following phrases:

- If a movie's DVD has deleted scenes…

- If an actor played a role in a movie winning an Oscar…

- If the DVD is an import…

- If the movie was released more than 50 years before the DVD…

- If the actor ever played a leading role…

- If the movie was nominated for a Golden Globe…

- If the distributor offers any drama DVDs…

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

14. Which term should be used to represent `Movie`?

15. Which term should be used to represent `DVD`?

16. True or False. The following Rulesheet correctly relates the `Movie` and `DVD` entities?

	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		

Rule Statements ✕   ✉ Rule Messages

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

17. Given the business intent, how many times do you want the rule to fire given this Input Testsheet?

scope1.ert ✕

scope1

/scope1.erf

Input

- Movie [1]
  - genre [Classic Animation]
  - studio [Disney]
  - title [Cinderella]
- DVD [1]
  - priceTier
- Movie [2]
  - genre [Animation]
  - studio [Pixar]
  - title [Toy Story]
- DVD [2]
  - priceTier

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

19. Assume that you update the Rulesheet to include another rule, as shown. Answer the following questions:

	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 ✕   ✉ **Rule Messages**

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 17, what result do you *want* for Cinderella?

- What result do you *want* for Toy Story?

- What results do you *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 the result make business sense?

- What changes should be made to the Rulesheet so that it functions as we intend?

20. True or False. Whenever rules contain scope, you 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 can be the same as the Vocabulary entity it represents?

4

Rule writing techniques

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 set of topics.

For details, see the following topics:

- How to work with rules and filters in natural language

- Filters versus conditions

- Qualify rules with ranges and lists

- How to use standard Boolean constructions

- How to embed attributes in posted rule statements

- How to include apostrophes in strings

- TestYourself questions for Rule writing techniques and logical equivalents

How to work 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 **Natural Language**.

2. The Natural Language view opens, and typically places itself above the Rulesheet, as shown:

	Condition Expression	English (United States)
a	Cargo.weight	
b	Cargo.volume	
c	Cargo.needsRefrigeration	

	Action Expression	
A	Cargo.container	

*Cargo.ers

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

	Actions					
	Post Message(s)	✉	✉	✉	✉	
A	Cargo.container	standard	oversize	heavyweight	reefer	
B						
C						
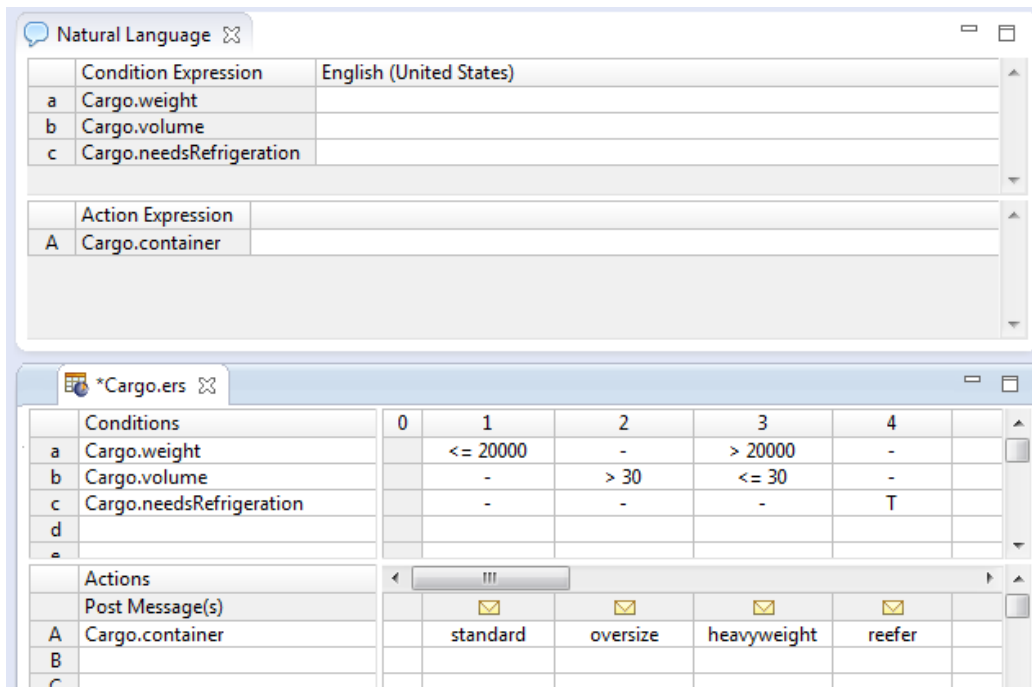
Note: If the **Natural Language** window does not open, choose **Window>Show View>Natural Language**.

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

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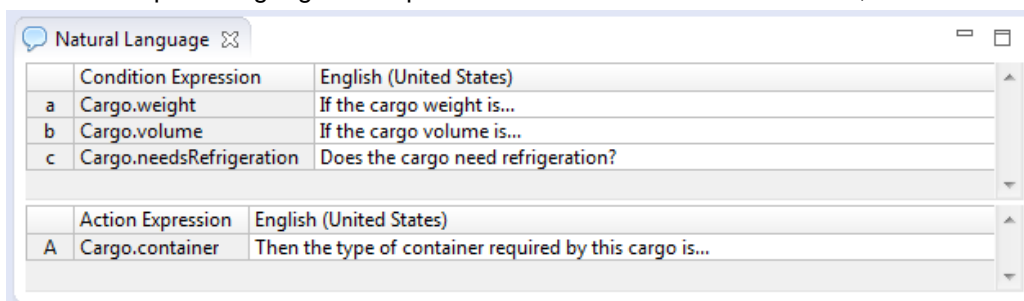
While your use of natural language might vary, it is good practice to use a consistent, clear style. Here are some tips:

- 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 ellipses helps a reader continue the expression with the values in its column cells.

- If you enter no natural language text, then the existing expression is shown.

4. Expose your natural language expressions in the Rulesheet by either clicking the **Show Natural Language** toolbar button , or **Rulesheet > Show Natural Language**. The natural language is displayed as shown:

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

5. Save the Rulesheet to store its expressions as well as its natural language data.

6. You can revert to the actual, editable expressions by clicking the **Hide Natural Language** toolbar button

   , or **Rulesheet > Hide Natural Language**.

7. 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.

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:

	Filter Expression	English (United States)
1	Cargo.weight<Aircraft.maxCargoWeight	Reject any package that exceeds the assigned aircraft weight capacity
2	Cargo.volume<Aircraft.maxCargoVolume	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

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

Filters

1	Cargo.weight<Aircraft.maxCargoWeight
2	Cargo.volume<Aircraft.maxCargoVolume
3	

	Actions			
	Post Message(s)	✉	✉	✉
A	Cargo.container	standard	oversize	heavyweigl
B				
C				
	Overrides		1	

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 shown:

	Condition Expression	English (United States)	French	Portuguese (Brazil)	Spanish
a	Cargo.weight	If the cargo weight is...	Si le poids de la cargaison est ...	Se o peso da carga é ...	Si el peso de la carga es ...
b	Cargo.volume	If the cargo volume is...	Si le volume de chargement est ...	Se o volume de carga é ...	Si el volumen de carga es ...
c	Cargo.needsRefrigeration	If the cargo must be refrigerated...	Si la cargaison doit être réfrigérée ...	Se a carga deve ser refrigerado ...	Si la carga debe ser refrigerad

	Action Expression	English (United States)	French	Portuguese (Brazil)	Spanish
A	Cargo.container	Then the type of container required by this cargo is...	Puis le type de conteneur ...	Em seguida, o tipo de recipiente ...	A continuación, el tipo de

	Condition Expression	English (United States)	French	Portuguese (Brazil)
a	Cargo.weight	If the cargo weight is...	Si le poids de la cargaison est ...	Se o peso da carga é
b	Cargo.volume	If the cargo volume is...	Si le volume de chargement est ...	Se o volume de carga
c	Cargo.needsRefrigeration	If the cargo must be refrigerated...	Si la cargaison doit être réfrigérée ...	Se a carga deve ser re

	Action Expression	English (United States)	French	Portuguese (Brazil)
A	Cargo.container	Then the type of container required by this cargo is...	Puis le type de conteneur ...	Em seguida, o tipo d

Filters versus conditions

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

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

Figure 59: 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 is sometimes called a *nonconditional* or *action-only* 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 `200000`. 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 `200000` kilograms. Using this method, additional Rulesheets can be used to assign different `maxCargoWeight` values for each `aircraftType`. The **Filters** section can be thought of as a convenient way to quickly add the same conditional expression or constraint to all other rules in the same Rulesheet.

You can also achieve the same results without using filters. The following figure shows how you 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** `maxCargoWeight` equals `200000` kilograms.

Figure 60: Rulesheet using a conditional rule

Regardless of how you choose to express logically equivalent rules in a Rulesheet, the results will be equivalent. While the logical result may be identical, the time required to produce those results may not be. See How to optimize Rulesheets on page 276 for information about compression techniques that remove redundancies.

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 the preceding example, 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.

Qualify rules with ranges and lists

You can use values for any data type except Boolean 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.

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 61: 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]

where:

- Filter 8 does no encapsulation.

- Filter 9 uses braces for encapsulation. Its delimiter in the expression is a comma, rather than two dots like the others. Because this syntax defines a set and overloads the syntax for a list, it is a good practice to not use it to encapsulate a range.

- Filters 10 through 13 use (and mix) parentheses and brackets where a bracket on either side expresses that the value on that side also passes the test.

Examples of value ranges in filter rows

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

Figure 62: Rulesheet filters showing the syntax of ranges for each data type

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 63: Rulesheet Filters showing encapsulation of a list

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

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 can 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, as illustrated:

Figure 64: Rulesheet using value ranges in the column cells of a condition row

In this example, a `maxCargoWeight` value is assigned to each `Aircraft` depending on the `flightNumber` value from the `FlightPlan` that `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; 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 not inclusive, 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 shows the same Rulesheet as in the previous figure, but with one difference: the value range was changed from `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. Because `flightNumber` is an integer value, and therefore there are no fractional values allowed, so `201..300` and `(200..300]` are equivalent.

Figure 65: Rulesheet using open-ended value ranges in condition cells

	Conditions	1	2	3	4
a	FlightPlan.flightNumber	<= 100	101..200	(200..300]	> 300
b					

	Actions				
	Post Message(s)				
A	FlightPlan.aircraft.maxCargoWeight	50000	100000	150000	200000
B					
	Overrides				

Rule Statements   **Rule Messages**

Ref	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

All of the possible combinations of parenthesis and bracket notation for value ranges and their meanings are:

Figure 66: 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
```

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. 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 Ranges and lists in conditions and filters on page 95 for details about usage.

Boolean condition versus values set

A simple Boolean Condition that evaluates to either `True` or `False` might look llike this:

Figure 67: Rulesheet using a conditional rule

	Conditions	1
a	aircraft.aircraftType = '747'	T
b		
c		

	Actions	‹
	Post Message(s)	
A	aircraft.maxCargoWeight = 200000	☑
B		
C		
	Overrides	

📄 Rule Statements ⊠   ✉ Rule Messages

Ref	ID	Post	Alias	Text
1				Aircraft max cargo weight must equal 200000 kg if aircraft type is a 747

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 68: Rulesheet illustrating use of multiple values in the same condition row

🔲 ValueSetsInConditionalRules.ers ⊠

	Conditions	1	2	3
a	aircraft.aircraftType	'DC-10'	'A340'	'747'
b				

	Actions	‹		
	Post Message(s)			
A	aircraft.maxCargoWeight	100000	150000	200000
B				
	Overrides			

📄 Rule Statements ⊠   ✉ Rule Messages

Ref	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 this Rulesheet, you can write multiple rules (represented as different columns in the table) for different condition-action combinations. Expressing these same rules using Boolean expressions requires many more Condition and Action rows, and would fail to take advantage of the semantic pattern that these three rules share.

Exclusionary syntax

The following examples are logically equivalent:

Figure 69: Exclusionary logic using Boolean condition, Pt. 1

	ExclusionarySyntax.ers		0	1
	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 100000 when aircraft type is NOT a 747

Figure 70: Exclusionary logic using Boolean condition, Pt. 2

	ExclusionarySyntax.ers		0	1
	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 100000 when aircraft type is NOT a 747

Figure 71: Exclusionary logic using negated value

	ExclusionarySyntax.ers		0	1
	Conditions		0	1
a	aircraft.aircraftType			not '747'
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 100000 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, you can see that the same rule can be expressed in different ways on the Rulesheet, with identical results. The rule modeler decides which way of expressing the rule is preferable in a given situation. Progress recommends, 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 discussion of Boolean logic 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.

How to use 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**, a `maxCargoWeight` is assigned when `aircraftType` is **not** a `747`. But, what would you write in the Conditions cell if you want to specify any `aircraftType` *other than* those specified in *any of the other* Conditions cells? For this, you use a special term in the Operator Vocabulary named `other`, shown in the following figure:

Figure 72: Literal term other in the Operator Vocabulary
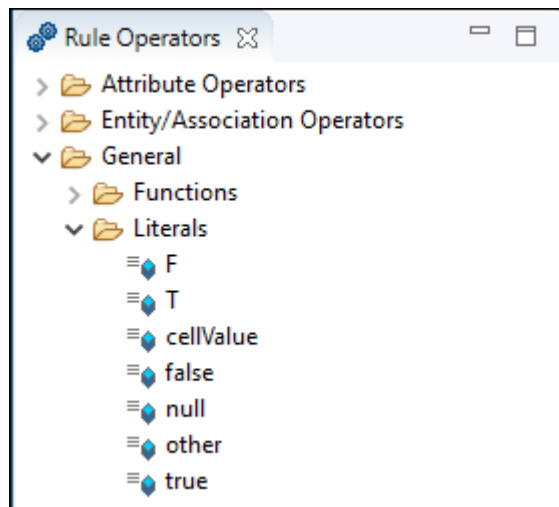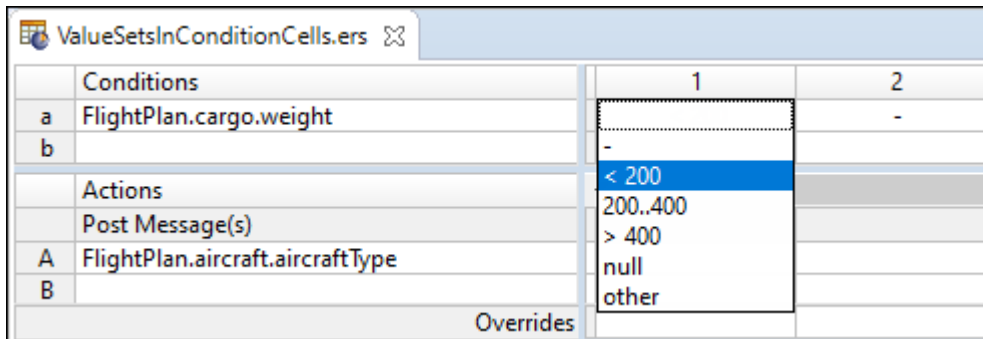
The term `other` provides a simple way of specifying any value *other than* any of those specified in other cells of the same Conditions row. The following figure illustrates how you can use `other` in the example.

Here, a new rule (column 4) was added 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`). The 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 73: Rulesheet using numeric value ranges in condition values set

In this example, an `integer2` value is assigned 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 quotation marks, as shown in the following figure. Corticon Studio will add the single quotation marks for you, but always check to make sure it has interpreted your entries correctly.

Figure 74: Rulesheet using String value ranges in condition values set

Value sets in condition cells

Most conditions implemented in the **Rules** section of the Rulesheet use a single value in a cell, as shown:

Figure 75: Rulesheet with one value selected in condition cell
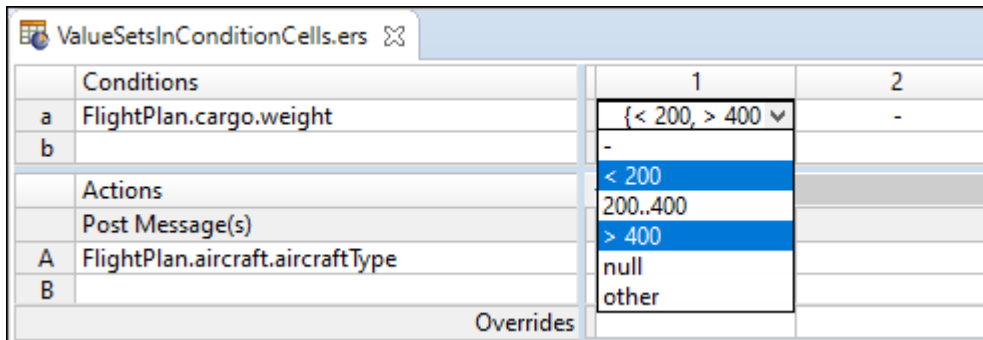
Sometimes, however, it is useful to combine more than one value in the same cell. You do this by holding **CTRL** while clicking multiple values from the Condition cell's drop-down list. Then, pressing **ENTER** encloses the resulting set in braces { . . } in the cell as shown in the sequence of the next two figures. Additional values may also be typed into Cells.
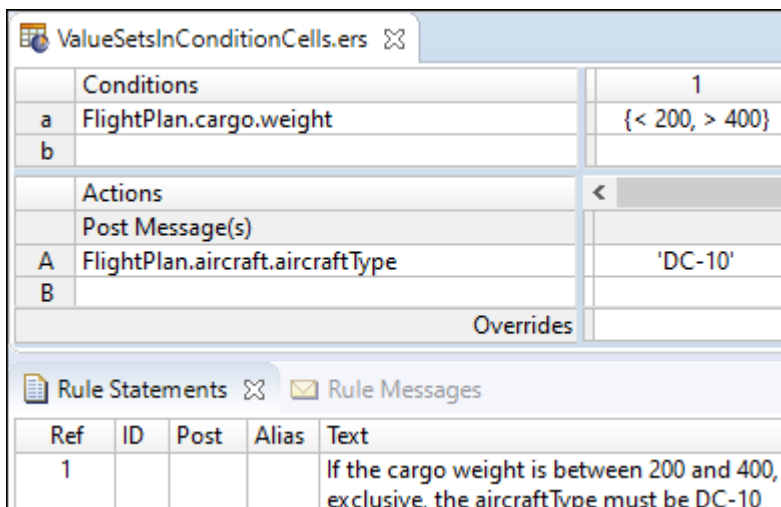
Figure 76: Rulesheet with two values selected in condition cell

Figure 77: Rulesheet with value set in condition cell

The rule implemented in Column 1 of the preceding figure is logically equivalent to the Rulesheet shown in the following figure:

Figure 78: Rulesheet with two rules instead of a value set

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

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

If you write rules using the logical OR operator in separate columns, performing a **Compression** reduces 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 instead of a value set results 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, type `not` in front of the leading brace {, 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

Given the condition cell's value set, the rule statement 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 79: Negating a value set in a condition cell

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

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

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.

Note: The issue with using multiple attributes in a condition row (or attributes mixed with literals) is a warning, not an error; as such, analysis functions are not available.

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}

Incorrect 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 quotation marks, as shown:

Figure 80: Rulesheet using a date value range in condition cells

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 not inclusive, meaning that the starting and 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: the value range 201..300 was changed 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. Because integer1 is an integer value, and therefore no fractional values are allowed, 201..300 and (200..300] are equivalent, and the 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 81: Rulesheet using an integer value range in condition values set

RulesheetUsingAnIntegerValueRange.ers

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

	Actions				
	Post Message(s)				
A	Entity1.integer2	50000	100000	150000	200000
B					
	Overrides				

Rule Statements   Rule Messages

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

All of the possible combinations of parenthesis and bracket notation for value ranges and their meanings are:

(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
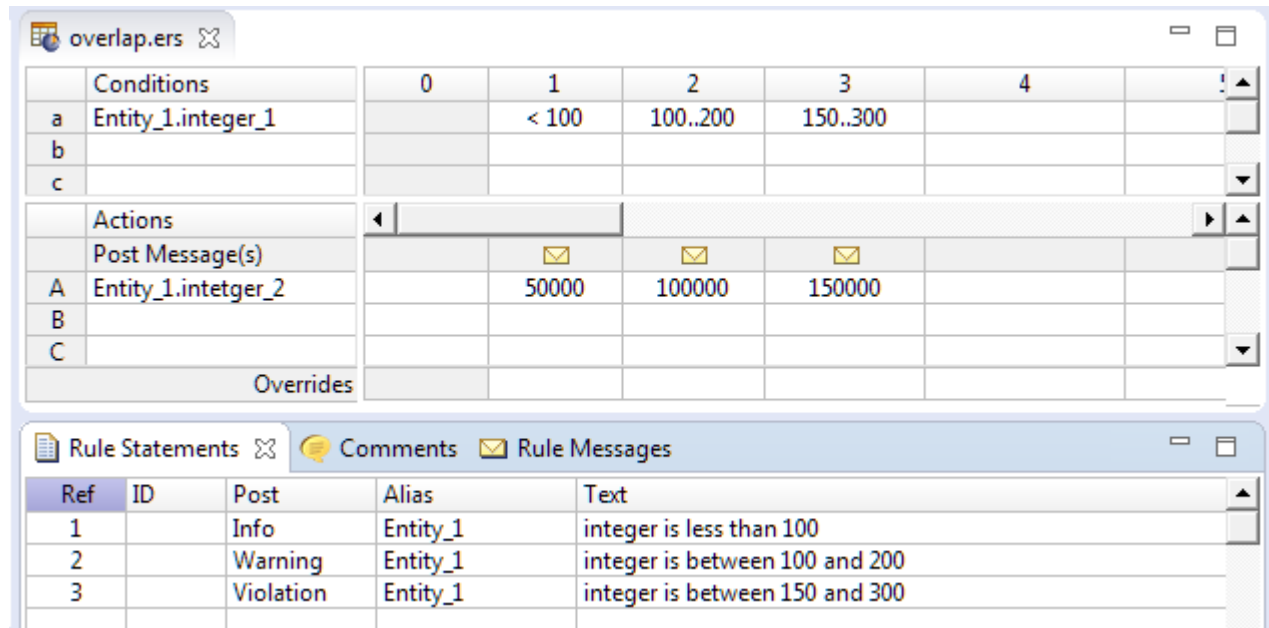
As illustrated in Rulesheet using numeric value ranges in condition values set and Rulesheet using an integer value range in condition values set, if a value range has no enclosing parentheses or brackets, then it is assumed to be closed. It is, therefore, not necessary to use the [..] notation for a closed range in Corticon Studio. In fact, if you try to create a closed value range by entering [..], then the brackets are 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 correctly expressed as [x..y).

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

Value ranges that overlap

One final note about value ranges: they **might overlap**. In other words, condition cells can 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 82: Rulesheet with value range overlap
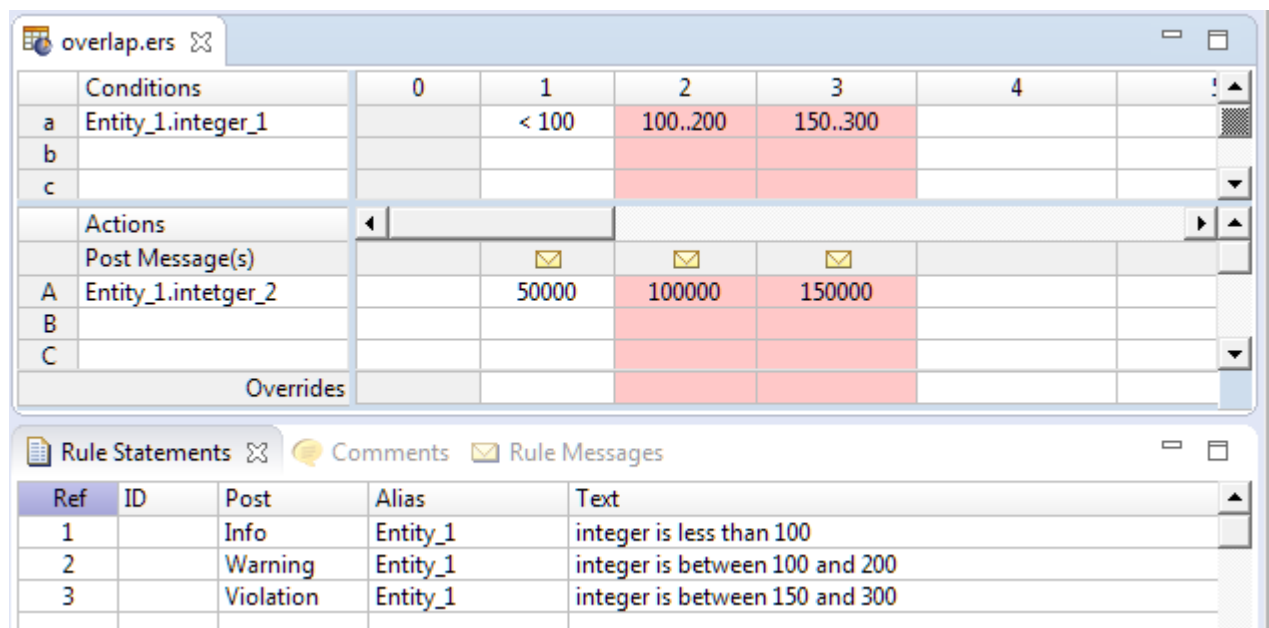
Figure 83: Rulesheet expanded with conflict check applied

Figure 84: Ruletest showing multiple rules firing for given test data

Input	Output
⊟ Entity_1 [1]	⊟ Entity_1 [1]
└ integer_1 [175]	└ integer_1 [175]
	└ **intetger_2 [150000]**

📄 Rule Statements 💬 Comments ✉ Rule Messages ✕

Severity	Message	Entity
Warning	integer is between 100 and 200	Entity_1[1]
Violation	integer is between 150 and 300	Entity_1[1]

Alternatives to value ranges

As you might expect, there is another way to express a rule that contains a range of values. One alternative is to use a series of Boolean conditions that cover the ranges of concern, as illustrated:

Figure 85: Rulesheet using Boolean conditions to express value ranges

	BooleansAsValueRanges.ers ✕				
	Conditions	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	50000	100000	150000	200000
	Overrides				

📄 Rule Statements ✕ ✉ Rule Messages

Ref	ID	Post	Alias	Text
A1				Aircraft max cargo weight must be 50000 kg when flight number is less than or equal to 100
A2				Aircraft max cargo weight must be 100000 kg when flight number is between 101 and 200, inclusive
A3				Aircraft max cargo weight must be 150000 kg when flight number is between 201 and 300, inclusive
A4				Aircraft max cargo weight must be 200000 kg 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 conditions that use the **AND** operator. 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.

The following expresses this rule in friendlier, more natural English:

> 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 the preceding figure, **Rulesheet Using Boolean Conditions to Express Value Ranges**. The same rules can 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 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 simple and elegant way. It is especially useful when dealing with decimal type values.

How to use 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, then you may have seen alternative methods. One such method is called a *truth table*.

The section *"Standard Boolean Constructions" in the Rule Language guide* presents several standard truth tables (AND, NAND, OR, XOR, NOR, and XNOR) with examples of usage in a Rulesheet.

How to embed 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 braces { .. } 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 current data and which part is the standard rule statement. As shown in Sample Rulesheet with Rule Statements Containing Embedded Attributes, brackets are used immediately outside the braces so that the dynamic values inserted into the message at rule execution are enclosed withing brackets. The use of these brackets is optional; other characters can 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 results in a `null` value inserted in the posted message.

Figure 86: Sample Rulesheet with rule statements containing embedded attributes

	Conditions	1	2	3
a	Entity1.integer1	< 18	18..25	> 25
b				

	Actions	‹		
	Post Message(s)	✉	✉	✉
A				
B				
	Overrides			

Rule Statements | Rule Messages

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

Figure 87: Rule Messages window showing bracketed embedded attributes

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

Rule Statements | Rule Messages

Severity	Message
Info	This person is [15] which is less than 18, so they cannot drink or vote
Info	This person is [23] which is between 18 and 25, so they can drink, vote, and be drafted, but not rent a car
Info	This person is [33] which is greater than 25, so they can drink, vote, be drafted, and rent a car

When an attribute uses an enumerated Custom Data Type, the dynamic value embedded in the posted rule message is 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 tables *"Usage restrictions" in the Rule Language Guide*, which specifies that the only parts of the Vocabulary that can be embedded in rule statements are attributes. No operators or expressions are permitted inside rule statements. Often, operators cause error messages when you try to save a Rulesheet. Sometimes the rule statement turns red. Sometimes an embedded equation executes as you intended, but no obvious error occurs, but the rule does not execute as intended. Remember that operators and expressions are not supported in rule statements.

How to include apostrophes in strings

String values in Corticon Studio are always enclosed in single quotation marks. But occasionally, you may want the String value to include single quotation marks, or apostrophes. If you enter the following text in Corticon Studio:

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

The text turns 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 quotation marks or apostrophes, you must use the special character backslash (\) that tells Corticon Studio to ignore the apostrophe, as shown:

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

When preceded by the backslash, the second apostrophe is 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.

TestYourself questions for Rule writing techniques and logical equivalents

Note: Try this test, and then go to TestYourself answers for Rule writing techniques and logical equivalents on page 337 to see how you did.

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. Which operator is used to negate a Boolean expression?

7. If a Boolean expression is written in a condition row, which values are automatically entered in the values set when **Enter** is pressed?

8. A Filter expression written as `Entity.boolean1=T` is equivalent to which of the following? (Circle all that apply.)

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

9. Of all alternatives listed in Question 8, 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` can be used in Action row values sets.

12. Using best practices discussed in this section, 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.

5

Collections

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

For details, see the following topics:

- How Corticon Studio handles collections

- How to visualize collections

- A basic collection operator

- How to filter collections

- How to use 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 EDC database access

- TestYourself questions for Collections

How Corticon Studio handles collections

Support for using collections is extensive in Corticon Studio. The integration of collection support in the Rule Language is so seamless and complete that the rule modeler often discovers that rules are performing multiple evaluations on collections of data beyond what they 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 iterates or cycles through all the available data during rule execution should not be of concern.

As you 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 a message or by a Ruletest (which are really the same thing, because the Ruletest serves as a quick and convenient way to create message 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.

How to visualize collections

Collections of data can be visualized as discrete portions, subsets, or branches of the Vocabulary tree. A parent entity is associated with a set of child entities, which are called call *elements* of the collection. The collection of pilots can be illustrated as:

Figure 88: Visualization of a collection of pilots
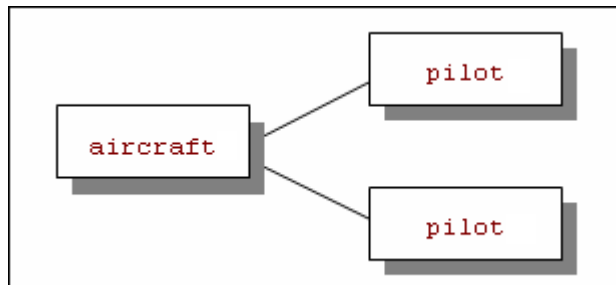
In this figure, the `aircraft` entity is the parent of the collection, while each `pilot` is a child element of the collection. As you saw in the role example, this collection is expressed as `aircraft.pilot` in the Corticon Rule Language. It is important to reiterate that this collection contains scope. You are seeing the collection of pilots as they relate to this aircraft. Or, more simply, you are seeing a plane and its 2 pilots, arranged in a way that is consistent with the Vocabulary. Whenever a rule exists that contains or uses this same scope, it also *automatically* evaluates 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 automatically evaluates all those collections as well. In the Corticon 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 89: Three-level collection
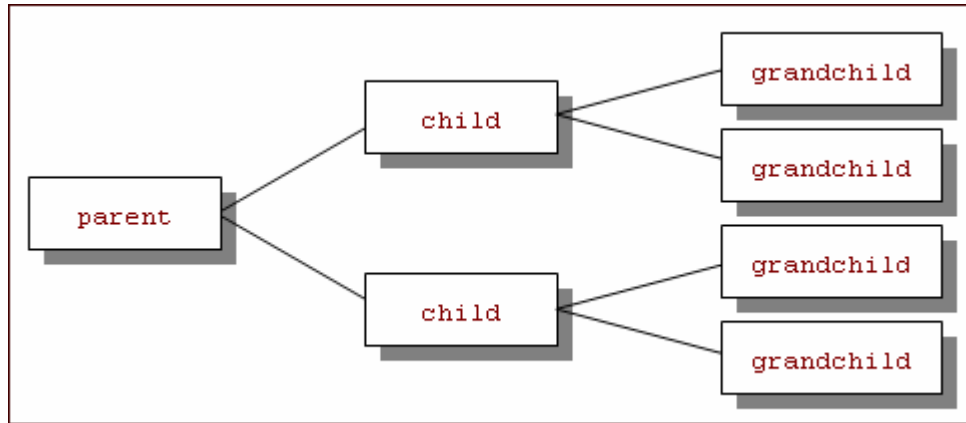
This collection is expressed as `parent.child.grandchild` in the Corticon Rule Language.

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, 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 previous rule becomes:

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

or

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

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

How to filter 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 and preconditions on page 195 topic for more details.

How to use aliases to represent collections

Aliases provide a means of using scope to specify elements of a collection; more specifically, you use 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 multiple instances 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 which collection (or copy) you are referring to in your rules, use a **unique alias** to refer to each collection.

For the purposes of illustration, a new scenario and business Vocabulary will be used. 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, this example is kept simple. The new Vocabulary is illustrated in a UML class diagram:

Figure 90: Security Vocabulary UML class diagram
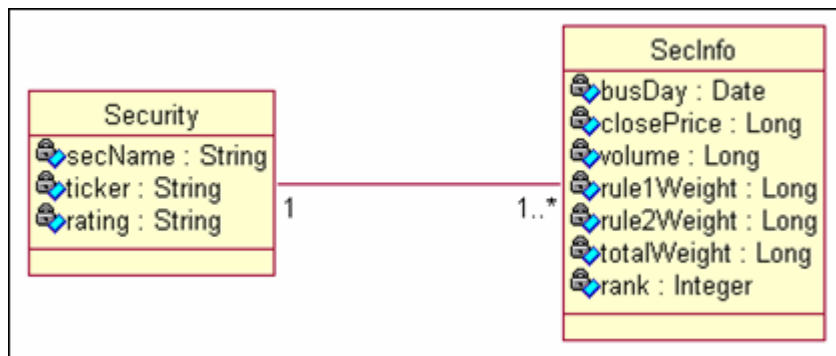
This Vocabulary consists of only two entities:

`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) because there are multiple instances of `SecInfo` data (multiple days of historical data) for each `Security`.

In this scenario, three rules 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 are 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. The Rulesheet that accomplishes these tasks is this:

Figure 91: Rulesheet with ranking model rules 1 and 2

In the preceding figure, two business rules are expressed in a total of four rule models (one for each possible outcome of the two business rules). The rules are straightforward, but the shortcuts (alias values) used in these rules are different than other rules you have seen. In the Scope section, you see that `Security` is the scope for the Rulesheet, which is not a new concept. But then, there are two aliases for the `SecInfo` entities associated with `Security`: `secinfo1` and `secinfo2`. Each of these aliases represents a <u>separate but identical</u> collection of the `SecInfo` entities associated with `Security`. In this Rulesheet, you constrain each alias by using filters. In a later example, you will see how more loosely constrained aliases can represent many different elements in a collection when the rules engine evaluates rules. In this 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 about the `.addDays` operator, see that topic in the *Rule Language Guide*.

After the aliases are created and constrained, you can use them in your rules where needed. In the figure **Rulesheet with Ranking Model Rules 1 and 2**, you 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 that uses a nonconditional rule to calculate the sum of the partial weights from the model rules 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. Because you 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 92: Rulesheet with total weight calculation and rank determination

You can test your new rules using a Ruleflow to combine the two Rulesheets. In a Ruletest that executes the Ruleflow, you expect to see the following results:

1. The `Security.secInfo` collection that contains data for the current business day (the expectation is that this collection reduces to a single `secinfo` element, because 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 93: Ruletest for testing security ranking model rules
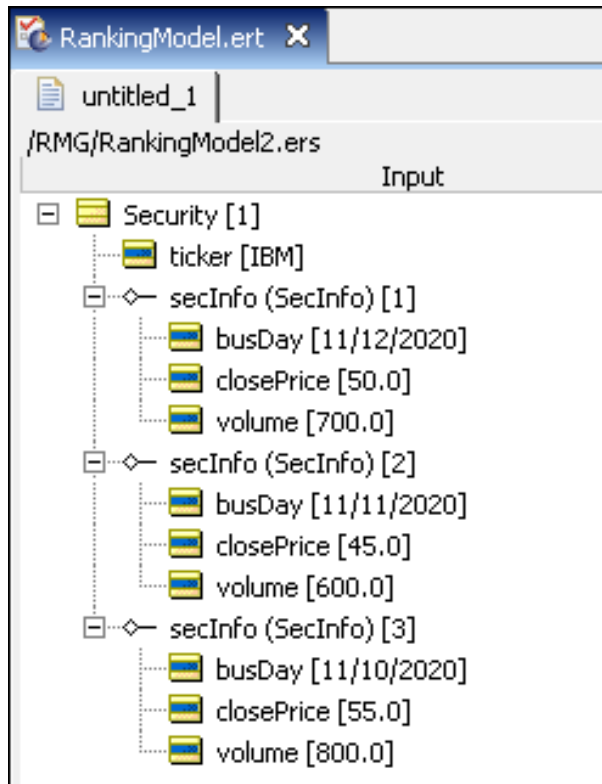
In this figure, one `Security` object and three associated `SecInfo` objects were added from the Vocabulary. The current day at the time of the Ruletest is `11/12/2020`, so the three `SecInfo` objects represent the current business day and two previous business days. The third business day is included 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 the two `SecInfo` objects being tested, you expect to see the highest rank of 4 assigned to your security in the current business day's `SecInfo` object.

Figure 94: Ruletest for security ranking model rules
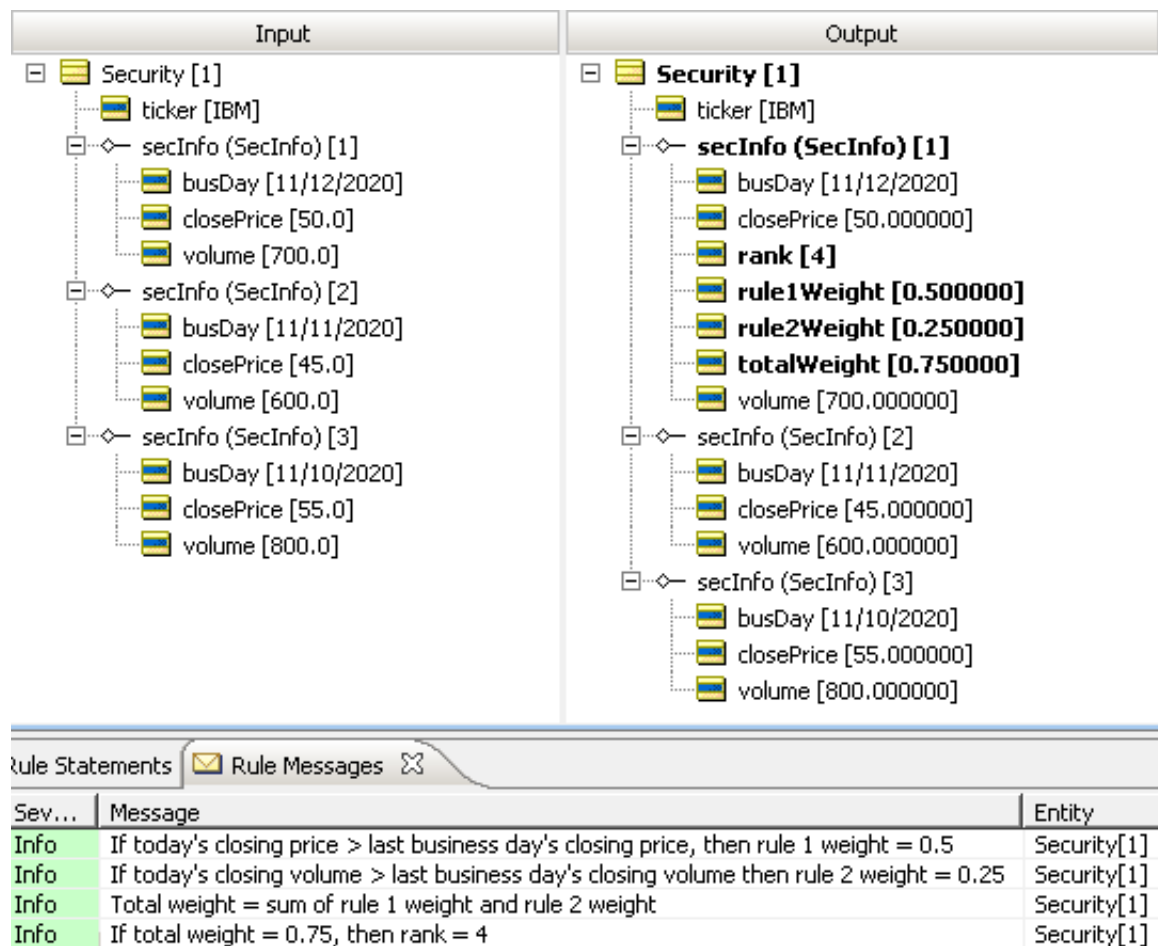
Both `closePrice` and `volume` for 11/12/2020 were higher than the values for those same attributes on 11/11/2020; therefore, both `rule1Weight` and `rule2Weight` attributes were assigned their high values by the rules. Accordingly, the `totalWeight` value calculated from the sum of the partial weights was the highest possible value, and a `rank` of 4 was assigned to this security for the current day.

As previously mentioned, the preceding example was tightly constrained in that the aliases were assigned to two 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?

The second example is also based on the security ranking scenario, but, in this example, the rank assignment that was accomplished will be done in a different way. Instead, you will 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 91: Rulesheet with ranking model rules 1 and 2 on page 119. In the rules for the new example, you 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 95: Rulesheet with alternate rank determination rules

In these new ranking rules, aliases were created 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 `secInfo1` and `secInfo2` are filtered for a specific value of `busDay` (today's date). However, our `Security` instances were loosely constrained. You 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 single elements of `Security` are not assigned to our aliases. Instead, the rules engine is instructed to evaluate all *allowable* combinations (that is, all those combinations that satisfy the first filter) of `Security` elements in the collection in each of the aliases (`sec1` and `sec2`). For each allowable combination of `Security` elements, the `totalWeight` values from the associated `SecInfo` element for `busDay = today` are compared, 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.

Figure 96: Input Testsheet for testing alternate security ranking model rules
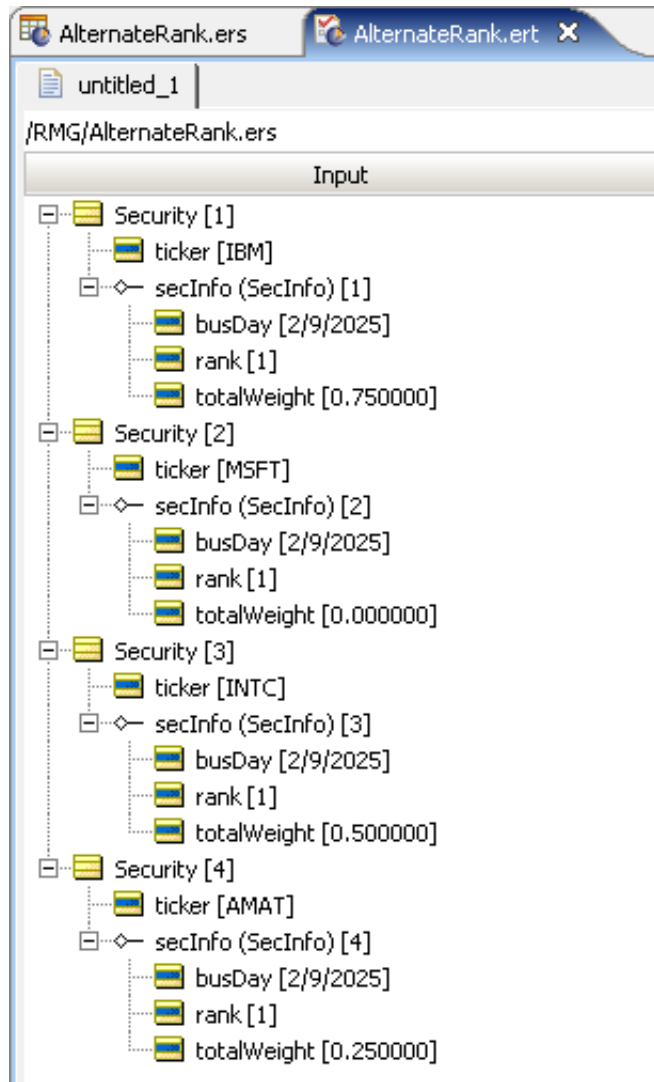
This figure shows a Ruletest constructed to test these ranking rules. In the data, four `Security` elements and an associated `secInfo` element for each were added. Note that each alias represents **all** four `security` elements and their associated `secInfo` elements. The current day at the time of the Ruletest is 2/9/2025, so each `Security.secInfo.busDay` attribute is given the value of 2/9/2025 (if additional `secinfo` elements in each collection were added, they would have earlier dates, and therefore would be filtered out by the preconditions on each alias). Each `Security.secInfo.rank` was initially set equal to 1 so that the lowest ranked security still has a value of 1. The lowest ranked security is 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, each security ranked between 1 and 4 with no identical `rank` values is expected.

Note: If there were multiple `Security.secInfo` elements (multiple securities) with the same `totalWeight` value for the same day, then the final `rank` assigned to these objects is expected 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 all `rank` values are initialized at 1.

Figure 97: Results Testsheet for alternate security ranking model rules

In this figure, the 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 98: Ruleflow to test two Rulesheets in succession
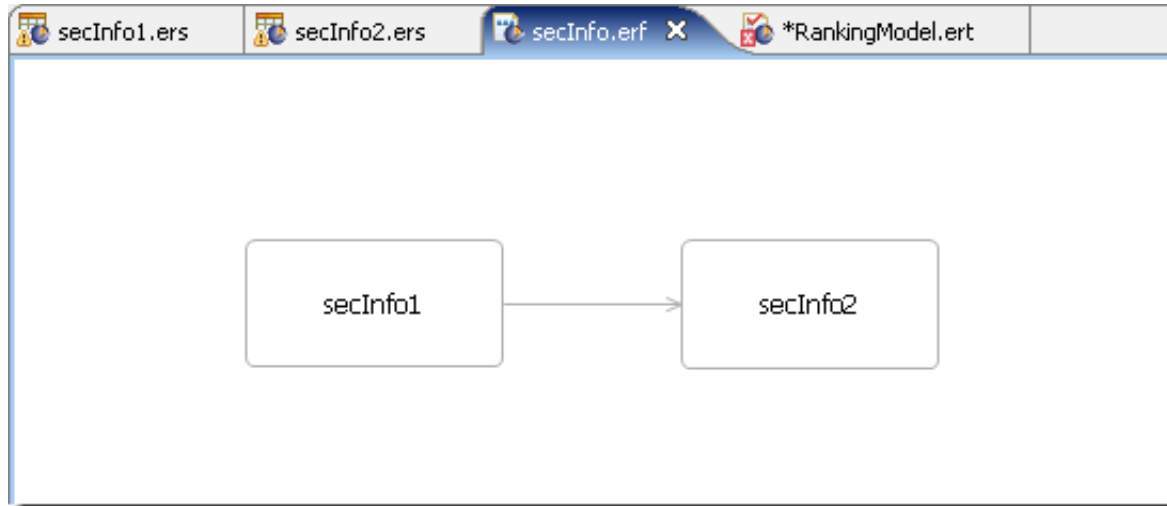
Sorted aliases

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. The `Cargo` entity and its `weight` attribute were brought into the scope:
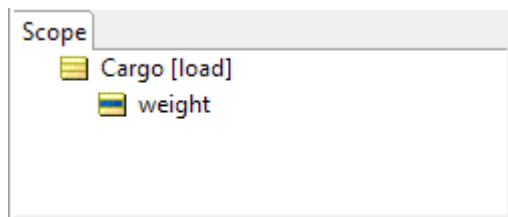
The operators `sortedBy` and `sortedByDesc` enable sorting in ascending or descending order of the numeric or alphabetic values of the attribute in the set of data. Note that an attribute with a Boolean data type is not valid for this operation.
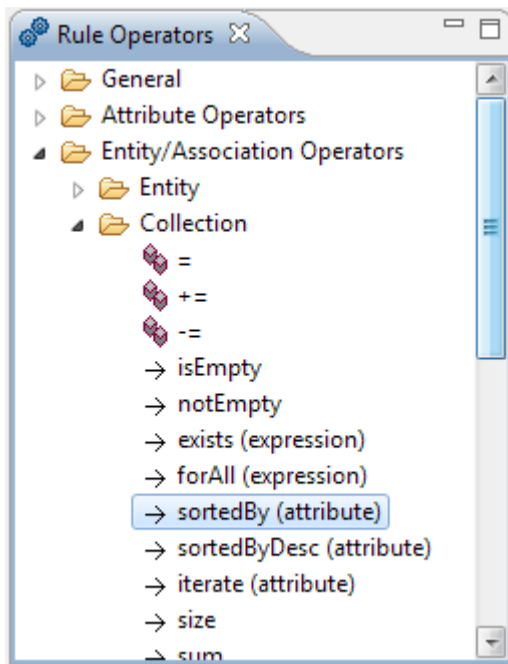
Dragging the `sortedBy` operator and dropping it (you cannot type it in) on the attribute `weight` places it in the scope, yet an error shows:
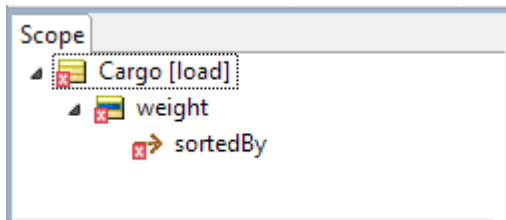
The error message notes that a sorted alias node requires an alias name. When you enter an alias name, the scope is complete.
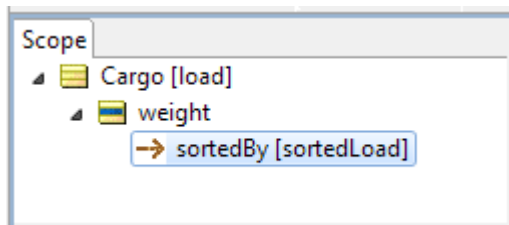
A filter expression is added to establish that, when you iterate through the list, each pass presents the next sequential item in the sorted set. You defined this by dragging `sortedBy` from the scope to filter line 1, and then appended the `->next` operator. A rule statement based on sorted load that echoes the weight was added so you can see the results in the tests.
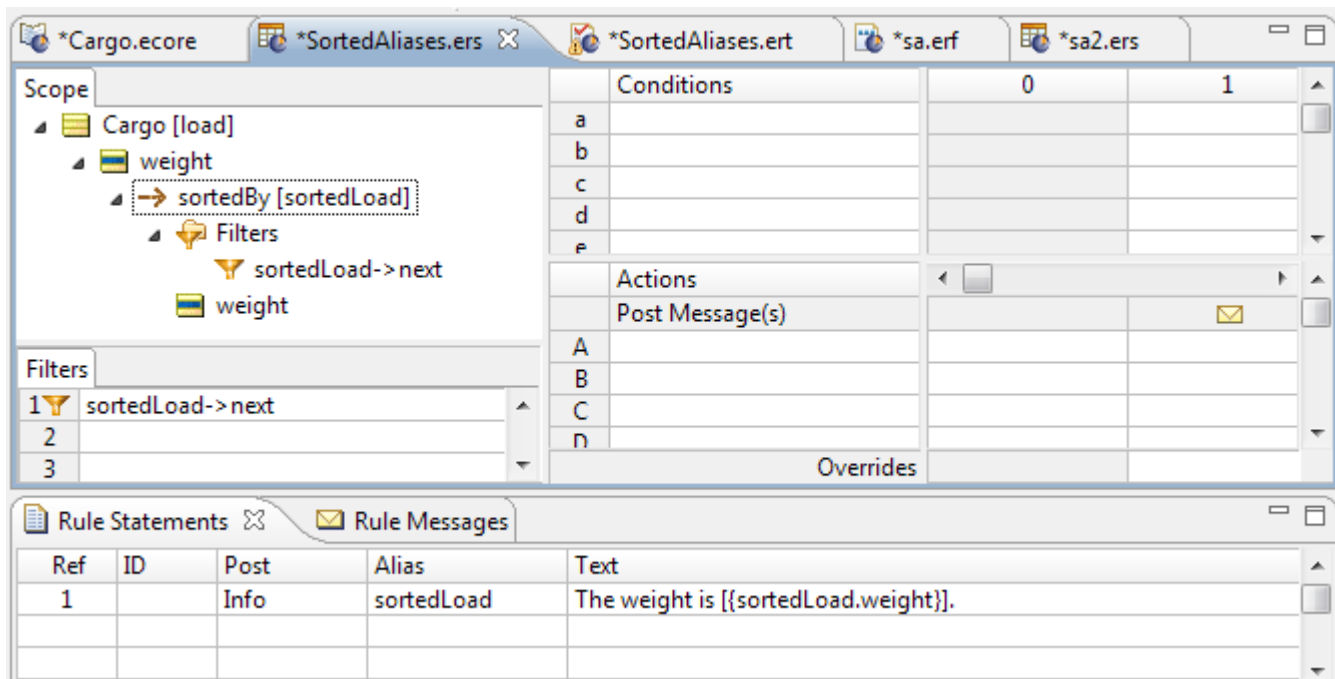
The Rulesheet is saved and a Ruleflow is created, adding in the Rulesheet. Then, you drag an **Iterative** operation to the Rulesheet in the Ruleflow and save it.
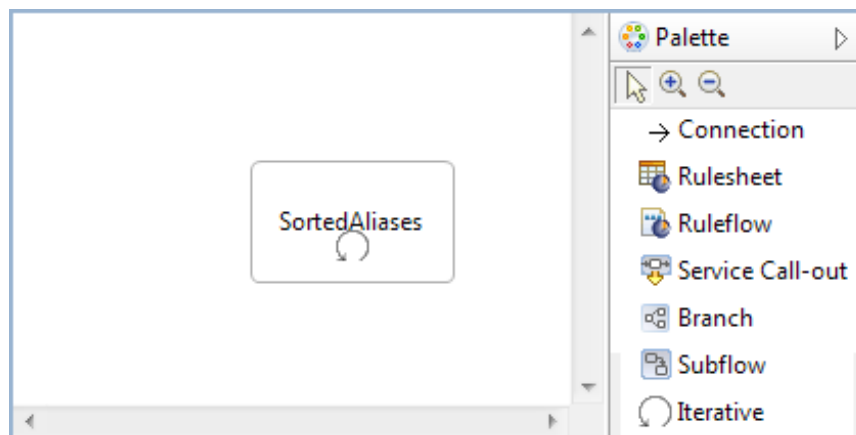
A Ruletest with a few Cargo items was created, each with a weight that is expected to sequence numerically when you run the test. Each iteration posts a message, and that message (the corresponding Rule Statement) contains the embedded attribute load weight. Because each member of the load collection will trigger the nonconditional rule, and even though the elements will be processed in no particular order, you 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.
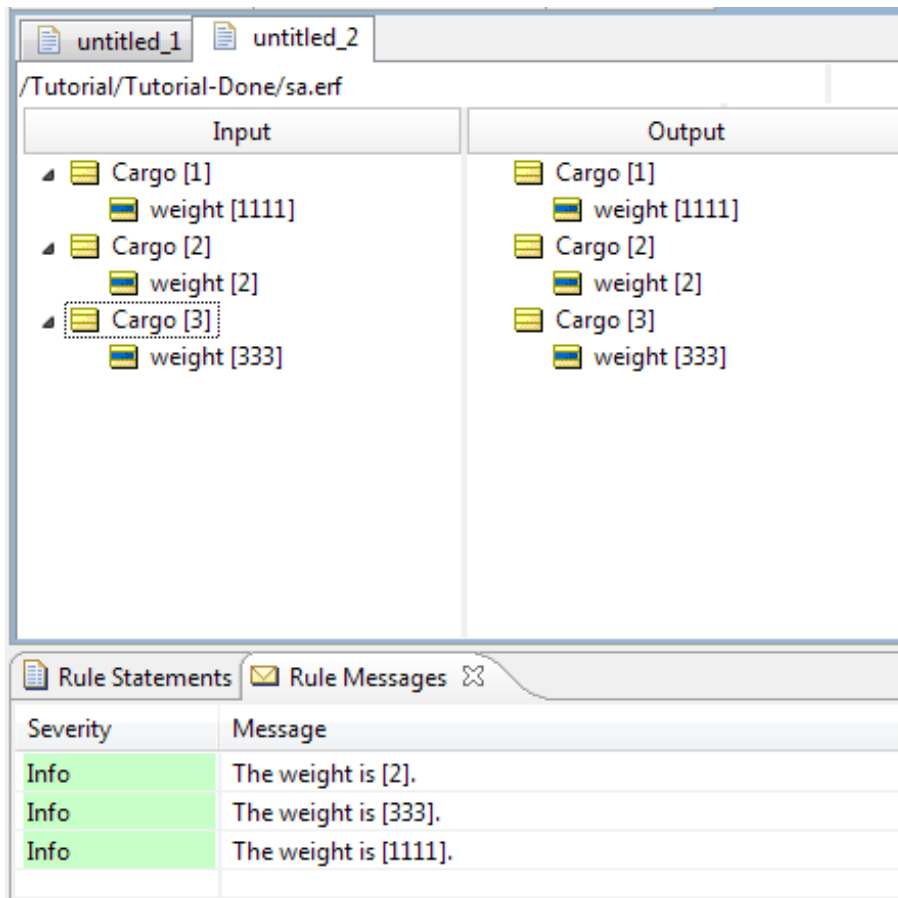
If you change the operator to `sortByDesc`, the results are shown in *descending* order by weight, as expected.

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. Use this approach in order to dissect the following expression:

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

1. `Collection1`

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

2. `Collection1 -> sortedBy(attribute1)`

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

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

    This expression 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 you identify a specific entity, you can easily reference the value of any attribute it contains, as in the following, which returns {e_n.attribute2}:

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

Entity Context

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

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

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

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

This expression returns a single integer value, like 14.

Notice that all you have now is a number, a *value*. You lost the entity context, so you cannot 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 -> sortedBy(attribute1) -> last > 10`

In preceding expression, it is not important that you know which element has the highest value of `attribute1`, all you 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 you 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 you care *which* child is oldest? Do you need to know his or her name? No. You 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 -> sortedByDesc(age) -> first.min(12)`

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

What about this scenario: Name the youngest child Junior.

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

Now return a *specific entity* – that of the youngest child – and assign to its name a value of `Junior`. You need to keep the entity context in order to make this assignment, and the preceding expression 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 question mark character (`?`) to hold or pin an element so that further action can be taken on it, including tagging it by assigning a value to one of its attributes. These special holder variables can be declared when needed, meaning they do not need to be defined anywhere prior to use.

For example, in a sales management system, the performance of sales representatives is analyzed every quarter, and the highest grossing sales representative is awarded *Salesperson of the Quarter*. This special status is then used to automatically increase the representative's commission percentage on sales made in the following quarter. The generic Vocabulary used in previous examples is used, but with 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, construct the following Rulesheet:

Figure 99: Rulesheet using statement block to identify and reward winner

Important Notes about Statement Blocks

As expressed in Action row A in the preceding figure, 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 was declared when it was needed.

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 the example, a value of 'Salesperson of the Quarter' is assigned to the string2 attribute of the salesperson held by ?tag. In effect, the highest grossing salesperson with this award is tagged.

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, then the element represented by the ? variable is lost. In other words, the ?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 a good practice to use semicolons only when there is no alternative, as in this example.

Now that the winner has been tagged, you can use the tagged element (awardee) to take additional actions. In the Conditional rule, the commission percentage of the winner is increased by 5% using the increment operator.

The next figure shows a Ruletest Input and Output pane. As expected, the highest grossing salesperson was awarded Salesperson of the Quarter honors, and their commission was increased by 5%.

Figure 100: Output panel with winner and adjusted commission in bold

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 data type, 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. 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. This behavior was seen 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, the `Aircraft` entity and its `maxCargoWeight` is brought into the scope as well as `Cargo` (with the alias `load`) and its attribute `weight`. The nonconditional action you enter is:

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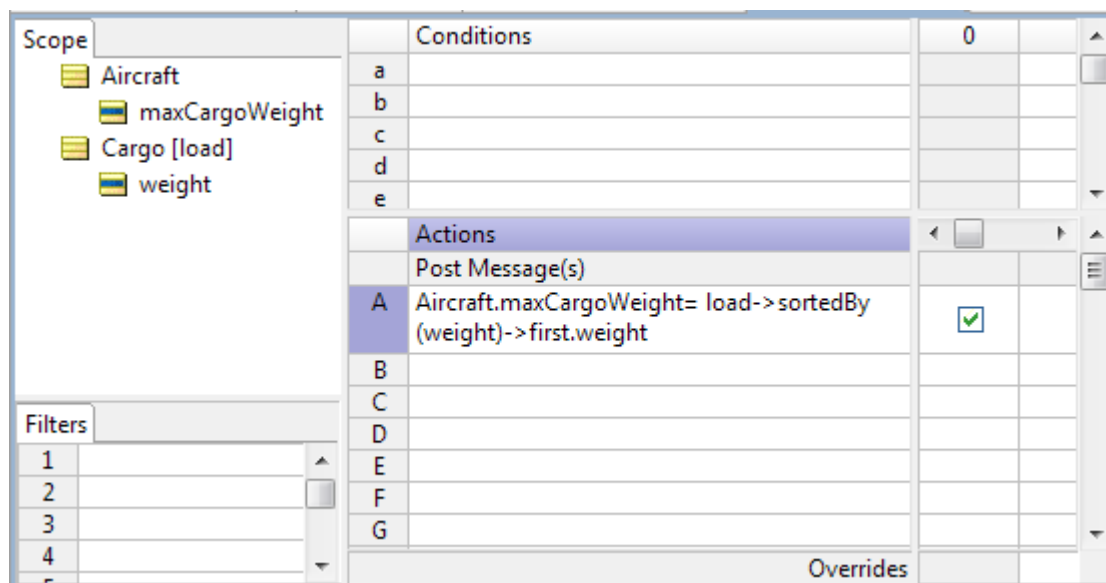
When you extend the test used for sorted aliases, you need to add an `Aircraft` with `maxCargoWeight` to show the result of the test. The result is as expected: the lightest item passed the test.

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

Figure 101:

Now, reverse the test to select the first item when you sort the items by descending weight:

Scope		Conditions	0	
Aircraft	a			
maxCargoWeight	b			
Cargo [load]	c			
weight	d			
	e			

	Actions		
	Post Message(s)		
A	Aircraft.maxCargoWeight= load->sortedByDesc(weight)->first.weight	☑	
B			
C			
D			
E			
F			
G			
	Overrides		

Filters

1	
2	
3	
4	
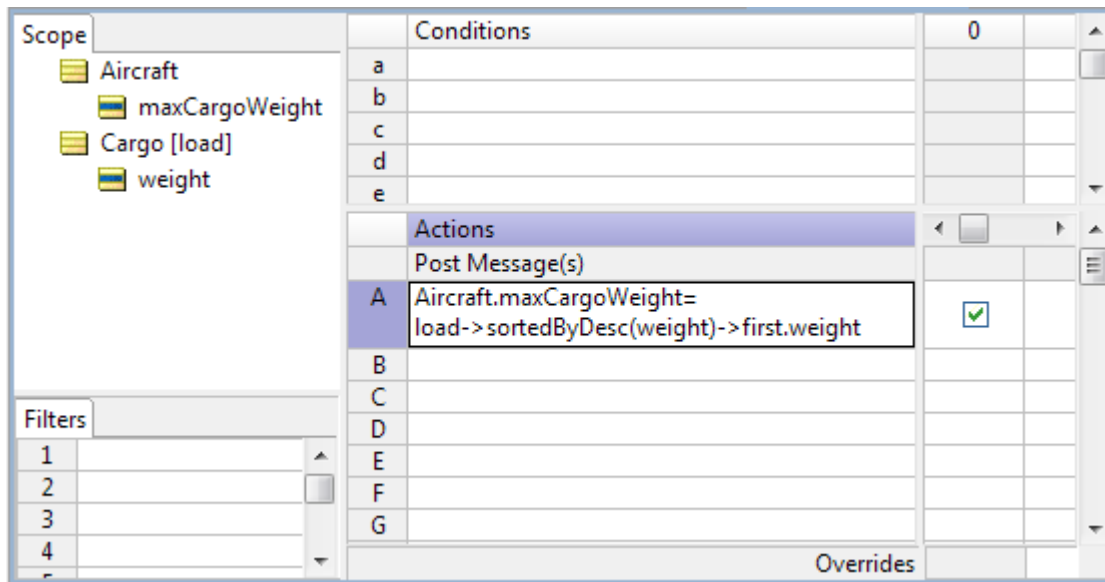
The heaviest item is output:

untitled_1   untitled_2

/Tutorial/Tutorial-Done/sa2.ers

Input	Output	Expected
Cargo [1]	Cargo [1]	
weight [1111]	weight [1111]	
Cargo [2]	Cargo [2]	
weight [2]	weight [2]	
Cargo [3]	Cargo [3]	
weight [333]	weight [333]	
Aircraft [1]	**Aircraft [1]**	
aircraftType	aircraftType	
maxCargoVolume	maxCargoVolume	
maxCargoWeight	**maxCargoWeight [1111.000000]**	
tailNumber	tailNumber	
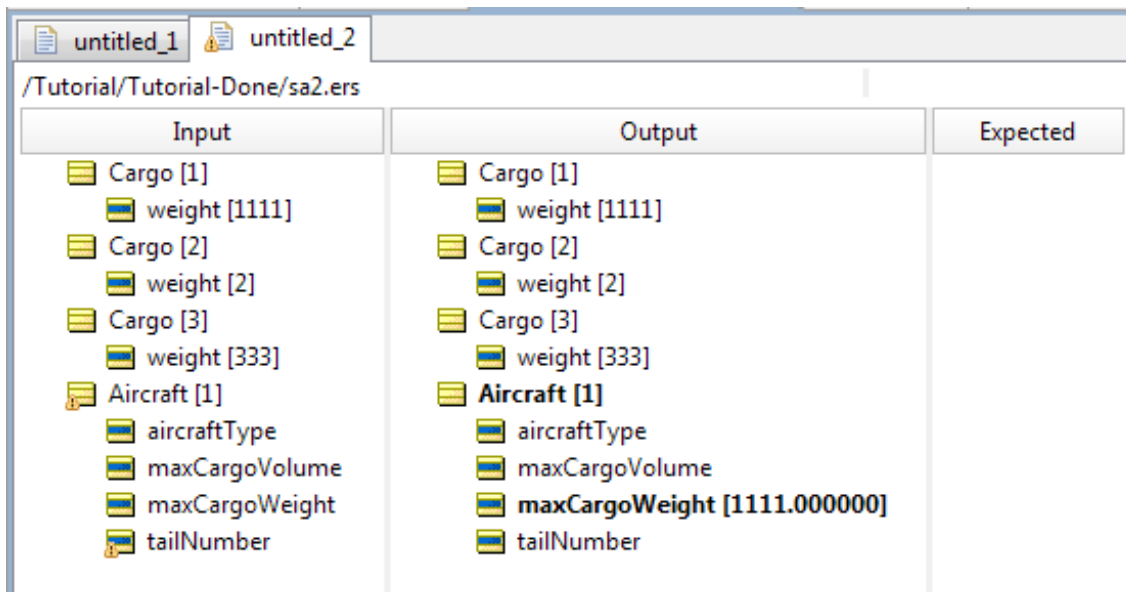
Note: Singletons do not operate against an iterative Ruleflow as was required by Sorted Aliases. The tests apply directly to the Rulesheet.

Special collection operators

There are two special collection operators available in Corticon Studio's Operator Vocabulary that allow you 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 (its *truth value* is determined) *for all* instances of an entity or collection. This is implemented as the ->forAll operator in the Operator Vocabulary. This operator will be demonstrated with an example created using the Vocabulary from the security ranking model. Note that these operators act on collections, so all the examples shown will declare aliases in the **Scope** section.

Figure 102: Rulesheet with universal quantifier ("for all") condition

In this figure, you see the following 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 the 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 the "for all" condition rules.

Figure 103: Ruletest for testing "for all" condition rules
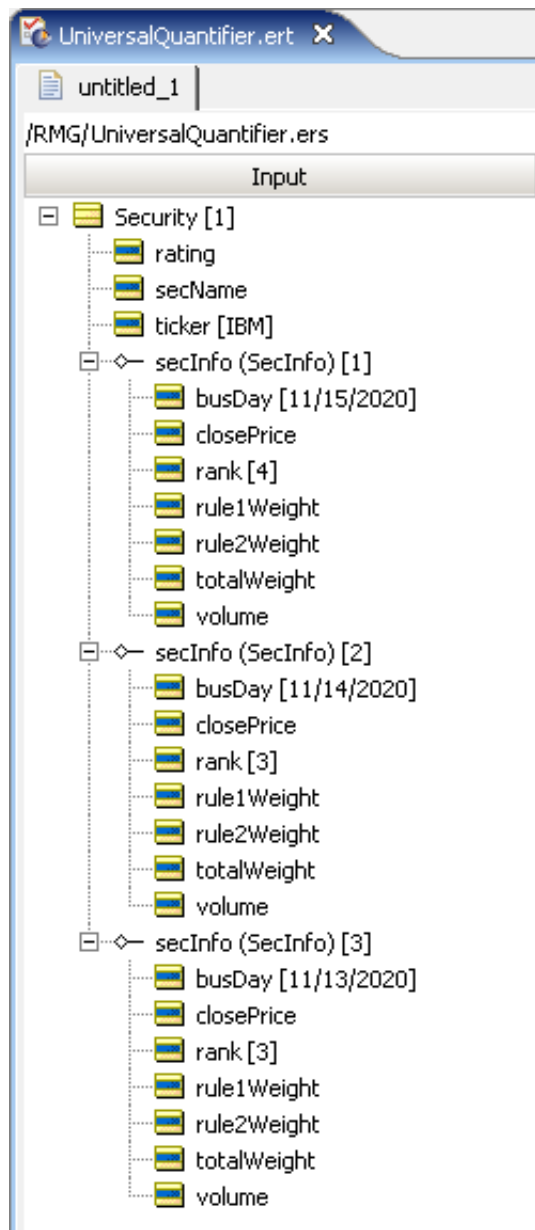
In this Ruletest, a collection of three `SecInfo` elements associated with a `Security` entity is evaluated. Because the `rank` value assigned in each `SecInfo` object is at least 3, you should expect that the "for all" condition will evaluate to `true`, and a rating value of `High` will be assigned to the `Security` object when the Ruletest is run through the rules engine. This outcome is confirmed in the Ruletest results, as shown:

Figure 104: 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 in the Operator Vocabulary.

You 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 example, `volume` rather than `rank` is used to determine the `rating` value for the security. The Rulesheet for this example is shown in the following figure:

Figure 105: Rulesheet with existential quantifier ("exists") condition

In this Rulesheet, you see the following 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 the `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 the `exists` condition rules.

Figure 106: Ruletest for testing ("exists") condition rules
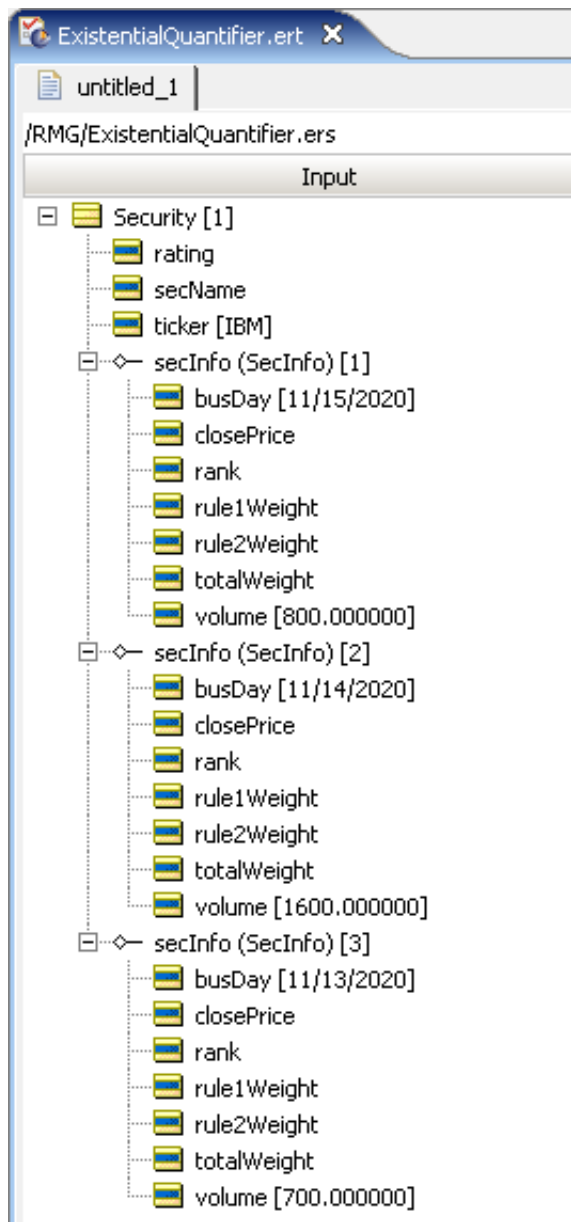
A collection of three `SecInfo` elements associated with a single `Security` entity will be evaluated. Because the `volume` attribute value assigned in at least one of the `SecInfo` objects (`secInfo[2]`) is greater than 1000, you should expect that the `exists` Condition will evaluate to **true** and a `rating` value of `High Volume` will be assigned to our `Security` object when the Ruletest is run through the rules engine. This outcome is confirmed in the Ruletest shown in the following figure:

Figure 107: Ruletest output for ("exists") condition rules

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, another example is provided.

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, extract the 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 the business rule, the following **UML Class Diagram** is proposed. Note the following aspects of the diagram:

- A policy can cover one or more drivers
- A policy can cover one or more vehicles
- A policy can have zero or more claims submitted against it.
- The claim entity was 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 108: UML Class Diagram
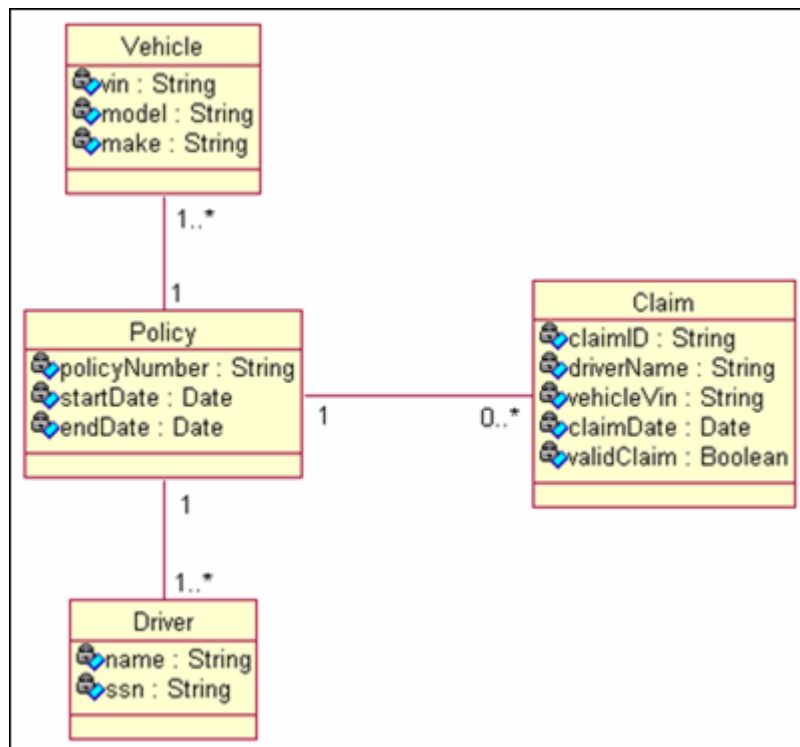
This model is realized in Corticon Studio as:

Figure 109: Vocabulary for insurance claims
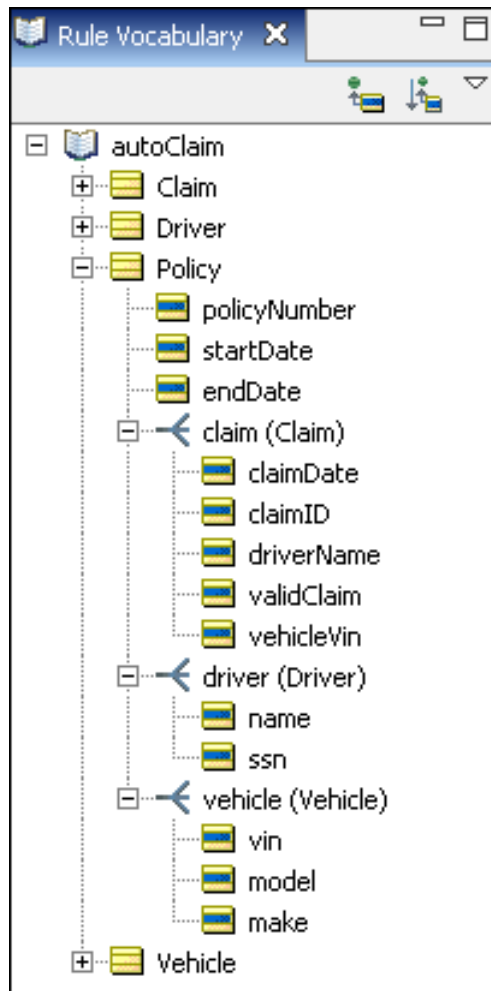
Model the following rules in Corticon Studio, as shown:

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 110: Rulesheet for insurance claims

Scope		Conditions	0	1	2	3
Claim [aClaim]	a	aClaim.driverName = aClaim.policy.driver.name	-	T	F	-
driverName	b	aClaim.vehicleVin = aClaim.policy.vehicle.vin	-	T	-	F
validClaim	c					
vehicleVin	d					
policy		Actions				
Filters		Post Message(s)		✉	✉	✉
1	A	aClaim.validClaim		T	F	F
2	B					
3		Overrides				

Rule Statements / Rule Messages

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 or vehicles listed on the policy. In other words, when the policy contains a collection of drivers or vehicles. The Vocabulary permits this scenario because of the cardinalities that were assigned to the various associations. This problem is demonstrated in the following Ruletest:

Figure 111: Ruletest input for insurance claims
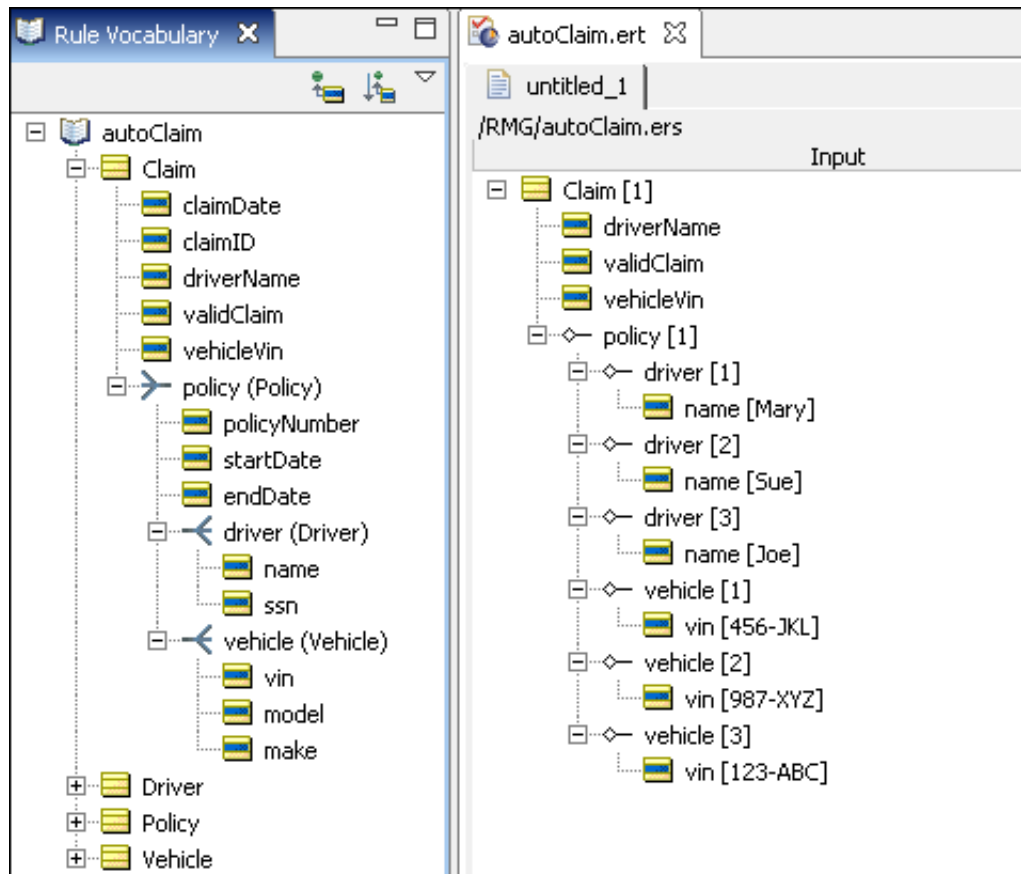
Notice in the Rulestest that there are three drivers and three vehicles listed on (associated with) a single policy. When you run this Ruletest, you see the results:

Figure 112: Ruletest output for insurance claims
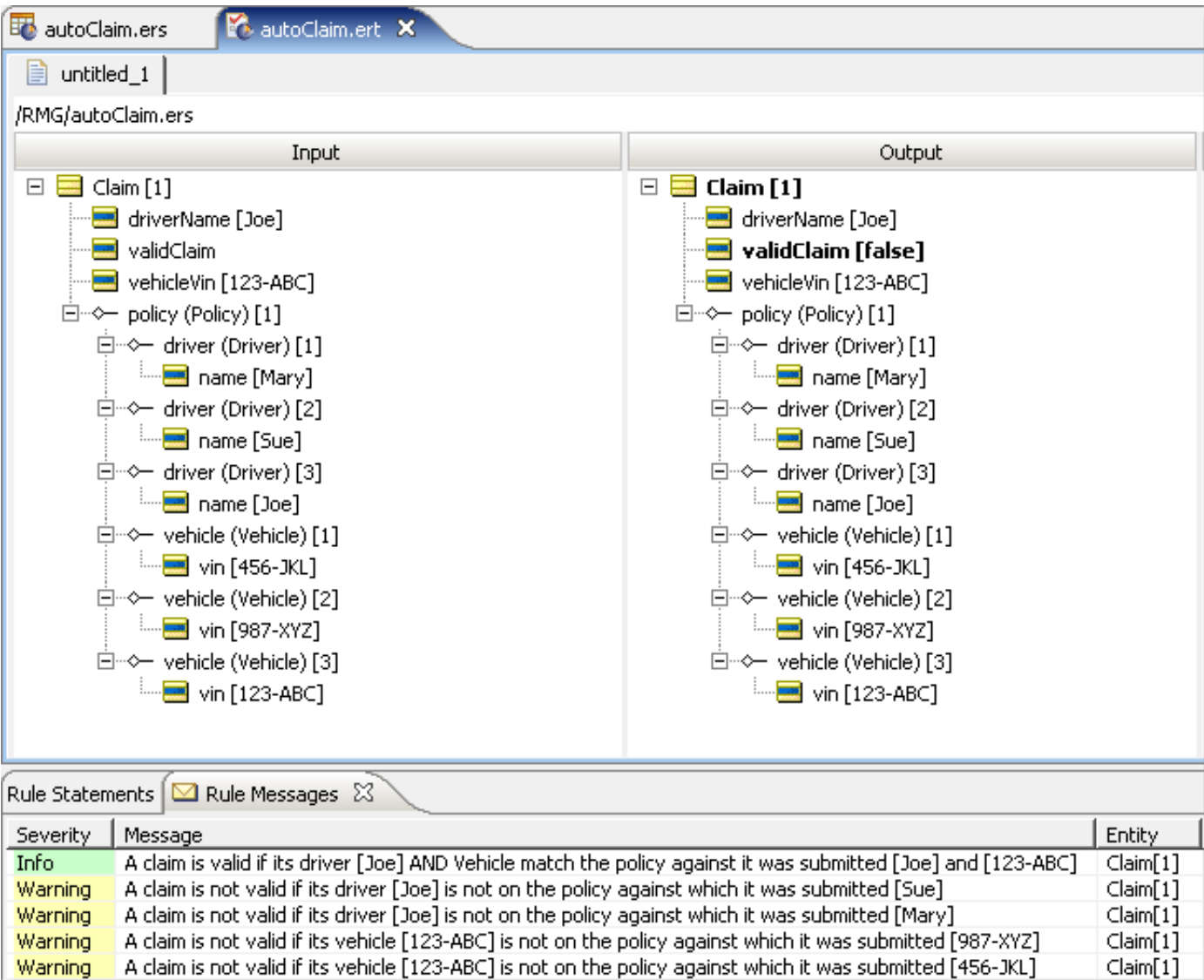
As you can see from the Ruletest results, the way Corticon Studio evaluates rules involving comparisons of multiple collections means that the `validClaim` attribute may have inconsistent assignments – sometimes `true`, sometimes `false` (as in this Ruletest). It can be seen from the following table below that, given the Ruletest data, 4 of 5 possible combinations evaluate to `false`, while only 1 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

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

The existential quantifier will be used to rewrite these rules:

Figure 113: Rulesheet with rules rewritten using the existential quantifier

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

Now the same Ruletest data as before is used to test these new rules. The following figure shows the results:

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

Aggregations that optimize EDC 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:
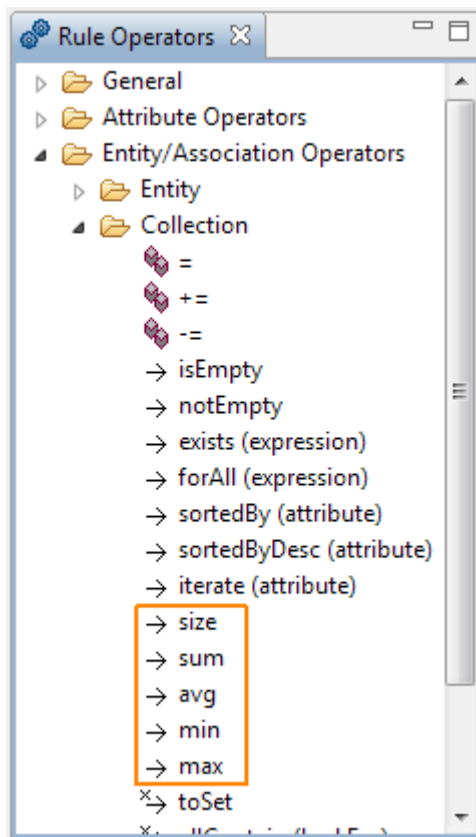
When these aggregations are applied through the Enterprise Data Connector in a Rulesheet set to **Extend to Database**, the performance effect against large tables can be minimized by performing non-conditional actions that force the calculations onto the database. For an example of this, see Optimize aggregations that extend to database on page 238

TestYourself questions for Collections

Note: Try this test, and then go to TestYourself answers for Collections on page 338 to see how you did.

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 can 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
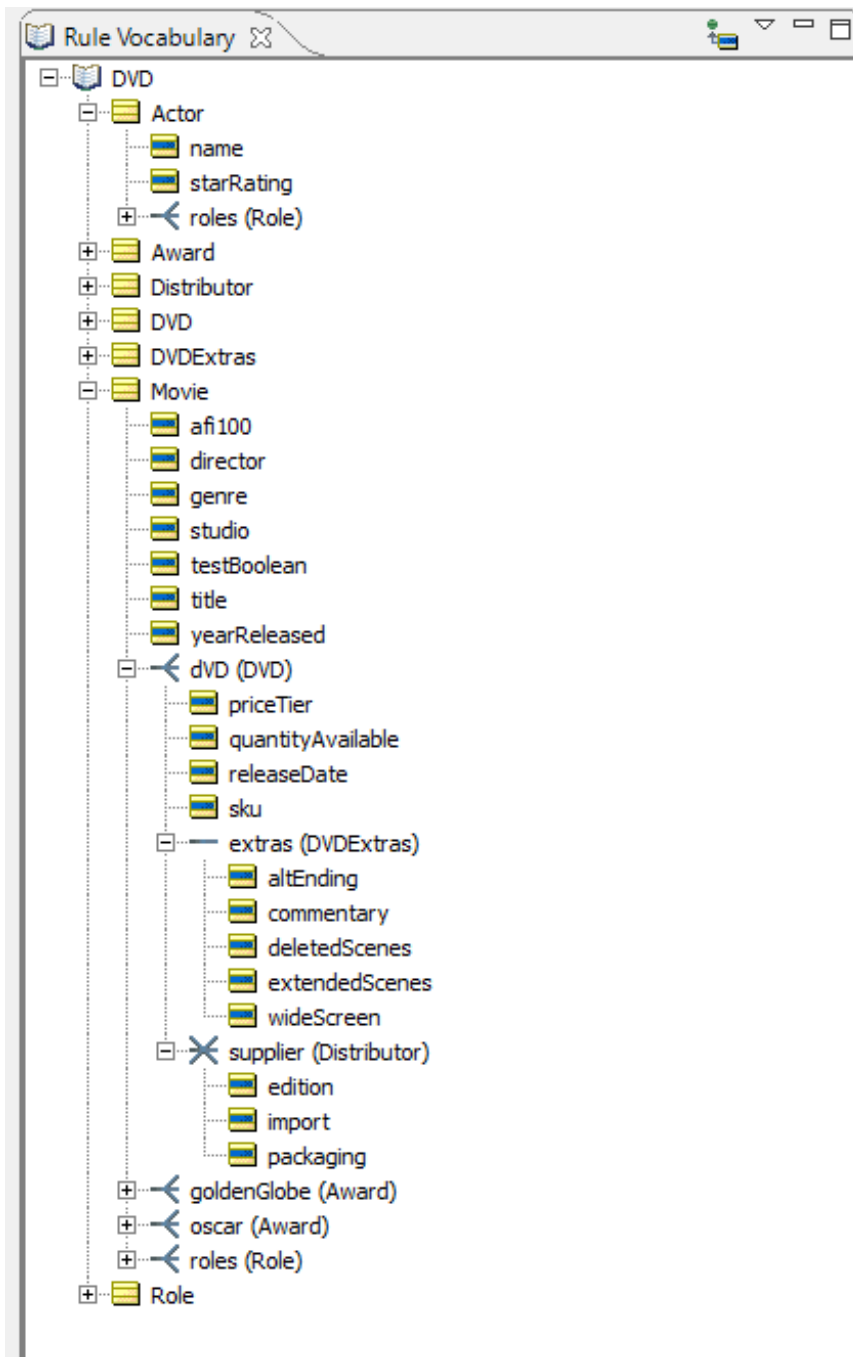
16. Write expressions for each of the following phrases:

    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 are at least 100 copies available of a movie

      g. If there are less than 2 copies available of a movie

      h. If the DVD can be obtained from more than 1 supplier

17. Which entities could be grandchildren of Movie?

18. Which entites could be children of Role?

19. Describe the difference between `->forAll` and `->exists` operators.

20. Describe the difference between `->notEmpty` and `->isEmpty` operators.

21. Why are aliases required to represent collections?

6

Rules containing calculations and equations

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

Terminology that will be used throughout this section

In the simple expression `A = B`, `A` is the *left-hand side* (LHS) of the expression, and `B` is 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 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`?

However, 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 assigned to `A`. As with other actions, you can activate or deactivate this action for any column in the decision table (numbered columns in the Rulesheet) 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 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 changed in the rule; that is, setting a `true` Boolean attribute to the value of `true` does not modify the element.

You can have reliable behavior when you use following workaround. To guarantee a modification in the data, you need to guarantee that the rules 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
```

For details, see the following topics:

- Operator precedence and order of evaluation

- Data type compatibility and casting

- Supported uses of calculation expressions

- Unsupported uses of calculation expressions

- TestYourself questions for Rules containing calculations and equations

Operator precedence and order of evaluation

Operator precedence is the order in which Corticon Studio evaluates multiple operators in an equation. Operator precedence 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 does not change the result. When expressed as `2*(3+4)`, however, the result is `14`.

The precedence of operators affects the grouping and evaluation of expressions. Expressions with higher-precedence operators are evaluated first. When 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	(*expression*, *expression*)	Logical: AND	(>5.5,<10)
	(*expression* or *expression*)	Logical: OR	(<5.5 or >10)

Note: Even though 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. The 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.

Data type 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 does not make sense unless both refer to the same kinds of data. How does one compare `orange` (a String) to `July 4, 2014 12:00:00` (a DateTime)? 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, the generic Vocabulary from the *Rule Language Guide* will be used because the generic attribute names indicate their data types:

Figure 114: Generic Vocabulary used in the Rule Language Guide
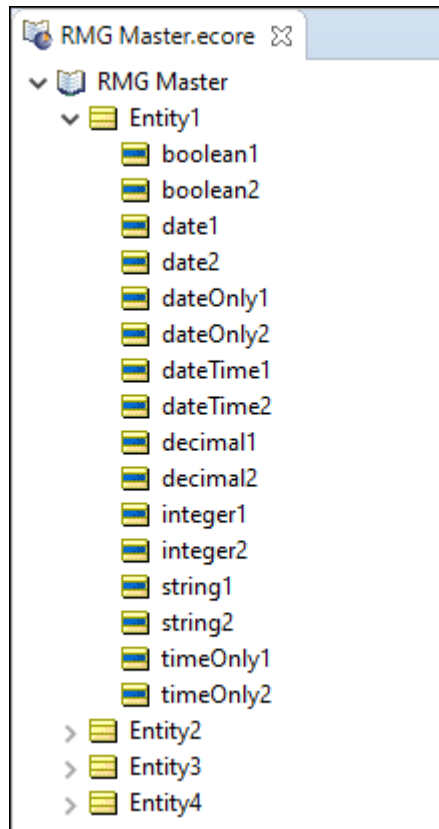
The following figure shows a set of Action rows that illustrate the importance of data type compatibility in assignment expressions:

Figure 115: Data type mismatches in assignment expressions

	ActionsNotMatching.ers ⊠		
	Conditions	0	1
a			
b			
	Actions	‹	
	Post Message(s)		
A	Entity1.boolean1 = Entity1.boolean2	☐	
B	Entity1.dateTime1 = Entity1.string1		
C	Entity1.string1 = Entity1.dateTime1		
D	Entity1.decimal1 = Entity1.integer1	☐	
E	Entity1.boolean1 = Entity1.decimal1		
	Overrides		

📄 Rule Statements 🗒 Problems ⊠	
3 errors, 0 warnings, 0 others	
Description	Resource
⌄ ⊗ Errors (7 items)	
⊗ Data type mismatch: Expecting type [Boolean], Actual type [Decimal].	ActionsNotMatching.ers
⊗ Data type mismatch: Expecting type [DateTime], Actual type [String].	ActionsNotMatching.ers
⊗ Data type mismatch: Expecting type [String], Actual type [DateTime].	ActionsNotMatching.ers

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 are 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 Corticon's general rule regarding data type compatibility: Decimals can 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.

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 116: Datatype mismatches in comparision expressions

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

a—This comparison expression is valid because the data types of the LHS and RHS sides of the equation are compatible. They are 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 Corticon's general rule regarding data type compatibility: Decimals can be 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 **d**, this also illustrates the exception to Corticon's general rule regarding data type compatibility: Decimals can be compared to Integer values. Unlike an assignment, however, whether the Integer and Decimal types occupy the LHS or RHS of a comparison is unimportant.

Data type 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. The previous examples were simple, and the data types of the LHS or the RHS of an equation correspond to the data types of those single attributes. But, the data type to which an expression resolves could be more complicated.

Figure 117: Examples of expression datatypes

DataTypeOfExpression.ers ⋈		
Conditions		**0**
a		
b		
Actions		‹
Post Message(s)		
A	e1.integer1 = e1.dateTime1.dayOfWeek	☑
B	e1.integer2 = e1.string1.size	☑
C	e1.boolean1 = e2 -> isEmpty	☑
D	e1.boolean2 = e2 -> exists(dateTime1 = today)	☑
E	e1.decimal1 = e2.integer1 -> sum	☑
	Overrides	

Let's examine each assignment to understand what is happening:

A—The RHS of this equation resolves to an Integer data type because the `.dayOfWeek` operator "extracts" the day of the week from a DateTime value (in this case, the value held by attribute `date1`) and returns it as an Integer between 1 and 7. Because 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. Because 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 is not. Because 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 ensures 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 Decision Service. Because this is a critical function, much effort was put into the Parser's accuracy and efficiency. But rule modelers should understand that the Parser is not perfect, and cannot anticipate all possible combinations of the rule language. It is still possible to "slip one past" the Parser. Here is an example:

Figure 118: LHS and RHS resolve to integers

	Conditions	0
a		
b		
	Actions	<
	Post Message(s)	
A	Entity1.integer1 = (Entity1.integer2 * 2) + 1	☑
B		
	Overrides	

In the preceding figure, there is 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 119: Will the RHS still resolve to an integer?

	Conditions	0
a		
b		
	Actions	<
	Post Message(s)	
A	Entity1.integer1 = ((Entity1.integer2 * 2) + 1) / 2	☑
B		
	Overrides	

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 returns an Integer, but an *odd* Integer! When an odd Integer is divided 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 Decision Service 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.

Looking at this rule in isolation, it is not difficult 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, iut is a best practice to ensure the LHS of numeric calculations has a Decimal data type so no data is inadvertently lost through truncation.

Manipulating data types 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: Special casting operators

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

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

Figure 120: Using casting operators
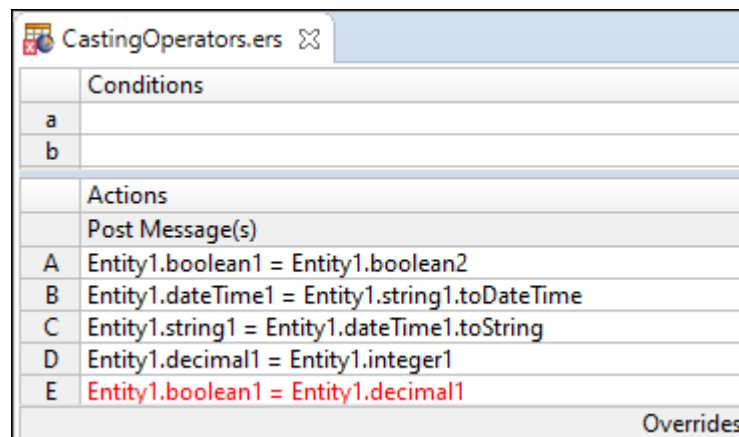
Casting operators were used in actions rules B and C to make the data types of the LHS and RHS match. Notice, however, that no casting operator exists to cast a Decimal into a Boolean data type for action E, hence the error.

Supported uses of calculation expressions

You can do comparisons and assignments in a few different ways:

-

-

-

-

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

Figure 121: Basic Tutorial Vocabulary Extended
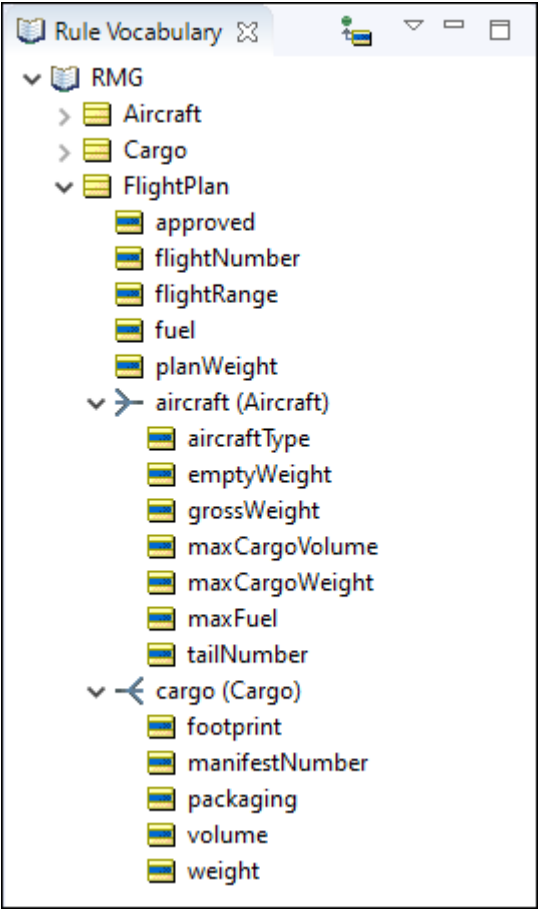
The new attributes are described in the following table:

Table 7: New attributes added to the Basic Tutorial Vocabulary

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

Calculation as a comparison in a precondition

In the following figure, a numeric calculation is used as a comparison in the filters section of the Rulesheet:
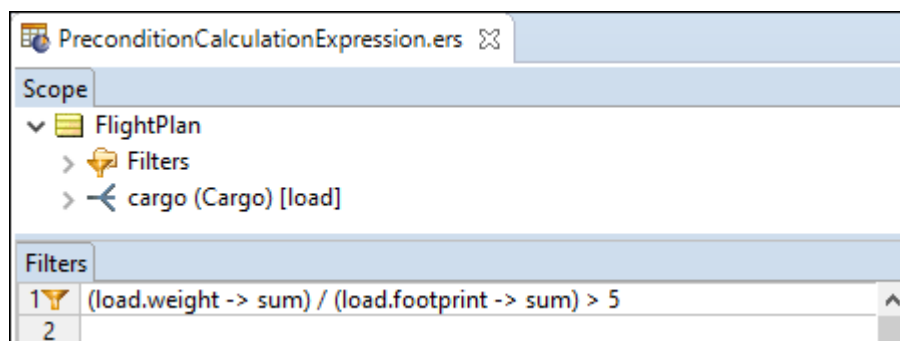
The LHS of the expression calculates the average pressure exerted by the total cargo load on the floor of the aircraft (sum of the cargo weights divided by the sum of the cargo containers' footprints). This result is compared to the RHS, which is the literal value 5. You might expect to see this type 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

The example shown in the following 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.

Figure 122: A calculation in a nonconditional expression
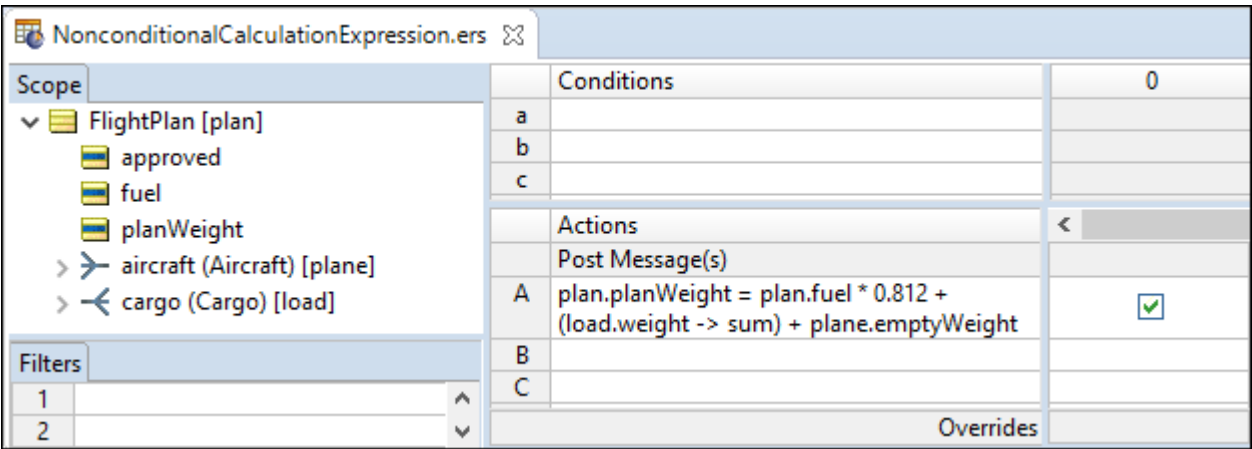
The portion that converts a fuel load measured in liters—the unit of measure that airlines purchase and load fuel—into a weight measured in kilograms, the unit of measure used for the weight of the cargo as well as the aircraft and crew:

```
plan.fuel * 0.812
```

Note that this conversion is conservative because Jet A1 fuel expands as it warms so this figure is 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 flight plan. The final sum of the fuel, cargo, and aircraft weights is assigned to the flight plan's `planWeight`. Note that parentheses are not required. The calculation will produce the same result without them. The parentheses were added to improve clarity.

Calculation as a comparison in a condition

After `planWeight` is derived by the nonconditional calculation in the following figure, it can immediately be used 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 123: `planWeight` Derived and used in same Rulesheet

In Condition row a, `planWeight` is compared to the aircraft's `grossWeight` to make sure that 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, can be used anywhere to represent the data produced by the calculation. It is also much simpler 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 following figure places the calculation in the LHS of the Conditional comparison to derive `planWeight` and compare it to `grossWeight` all in the same expression.

Figure 124: Calculation in a conditional expression

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

Calculation as an assignment in an action

The following figure shows two rules that each make an assignment to `maxFuel`, depending on the type of aircraft:

Figure 125: A calculation in an action expression

	Conditions		1	2
a	plane.aircraftType		'747'	'DC-10'
b				
	Actions	◁		
	Post Message(s)			
A	plane.maxFuel = plane.grossWeight - plane.maxCargoWeight - plane.emptyWeight		☑	☐
B	plane.maxFuel = 1000000		☐	☑
	Overrides			

Scope: FlightPlan [plan]; aircraft (Aircraft) [plane]; cargo (Cargo) [load]. Filters: 1, 2.

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 assigned the literal value `100000`.

Unsupported uses of calculation expressions

Some calculation expressions you might want to try do not provide expected or reliable results.

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 is a dangerous habit to get into because more complex equations generally do not work. It is best to express equations as shown in the previous sections.

Figure 126: Calculation in a Values Cell and Column

	Conditions	1
a	plan.planWeight	plane.emptyWeight + plan.fuel + load.weight
b		
c		
d		
	Actions ◁	
	Post Message(s)	
A		
B		
	Overrides	

Scope: FlightPlan [plan]; fuel; planWeight; aircraft (Aircraft) [plane]; cargo (Cargo) [load]. Filters: 1, 2.

Calculations in rule statements—Even though 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 braces { . . } 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, then the message will be displayed exactly as shown; it will not calculate a result of any kind.

Figure 127: Calculation in a Rule Statement

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

Figure 128: Embedding a calculation in a rule statement

However, even if the syntax does not turn red, you should not 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.

TestYourself questions for Rules containing calculations and equations

Note: Try this test, and then go to to correct yourself.

1. What are the two possible meanings of the equals operator =? In which sections of the Rulesheet is each of these meanings applicable?

2. What is the result of each of the following equations?

   a. $10 + 20 / 5 - 4$ ___

   b. $2 * 4 + 5$ ___

   c. $10 / 2 * 6 - 8$ ___

   d. $2 ** 3 * (1 + 2)$ ___

   e. $-5 * 2 + 5 * 2$ ___

3. Is the following assignments expression valid? Why or why not? `Entity1.integer1 = Entity1.decimal1`

4. What is the data type of each of the following expressions based on the scope shown in the following figure?
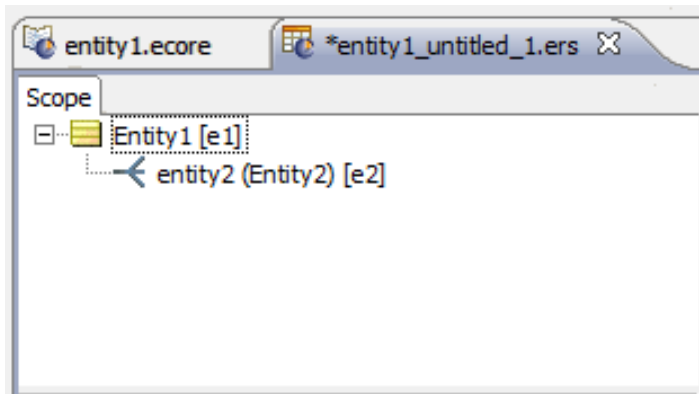
- e1.dateTime1.year ___
- e1.string1.toUpper ___
- e2 -> forAll (integer1 = 10) ___
- e2.decimal1 -> avg ___
- e1.boolean1 ___
- e1.decimal1 > e1.decimal2 ___
- e2.string2.contains('abc') ___

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:

Scope			Conditions		0	1	2
□ Movie [film]		a	aDVD.quantityAvailable -> sum / aDVD->size		-	<= 50000	> 50000
⊞ Filters		b					
yearReleased		c					
□ dVD (DVD) [aDVD]		d					
quantityAvailable		e					
		f					
		g					
		h					

Filters			Actions		◄		
1 ▼ today.year - film.yearReleased > 10			Post Message(s)			✉	✉
2		A					
3		B					
4		C					
5		D					
6		E					
7			Overrides				

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   ____

7

Rule dependency in chaining and looping

This section explores how Corticon determines the sequencing of rules, and looping, which involves controls you can set over the revisiting, re-evaluating, and possible re-firing of rules.

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 on the first.

For details, see the following topics:

- Forward chaining

- Rulesheet processing modes of looping

- Looping controls in Corticon Studio

- Looping examples

- How to use conditions as a processing threshold

- TestYourself questions for Rule dependency chaining and looping

Forward chaining

The first step in learning to use looping is to understand how it differs from the normal inferencing behavior of executing rules, whether executed by Corticon Studio or Corticon Server. When a Ruleflow is compiled into a Decision Service, a *dependency network* for the rules is automatically generated. Corticon uses this network to determine the order in which rules fire at run time. For example, in the following simple rules, 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 that 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 is 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 the preceding example, 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). You 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 occurs too. Looping and its enablement are discussed next.

Rulesheet processing modes of looping

Occasionally, you need 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. This behavior is called *advanced inferencing*, and to enable it in Rulesheet execution, you must set Rulesheet processing mode to **Advanced Inferencing** by selecting **Rulesheet > Processing Mode > Advanced Inferencing** from the Studio menubar, as shown:

Figure 129: Selecting Advanced Inferencing processing mode for a Rulesheet
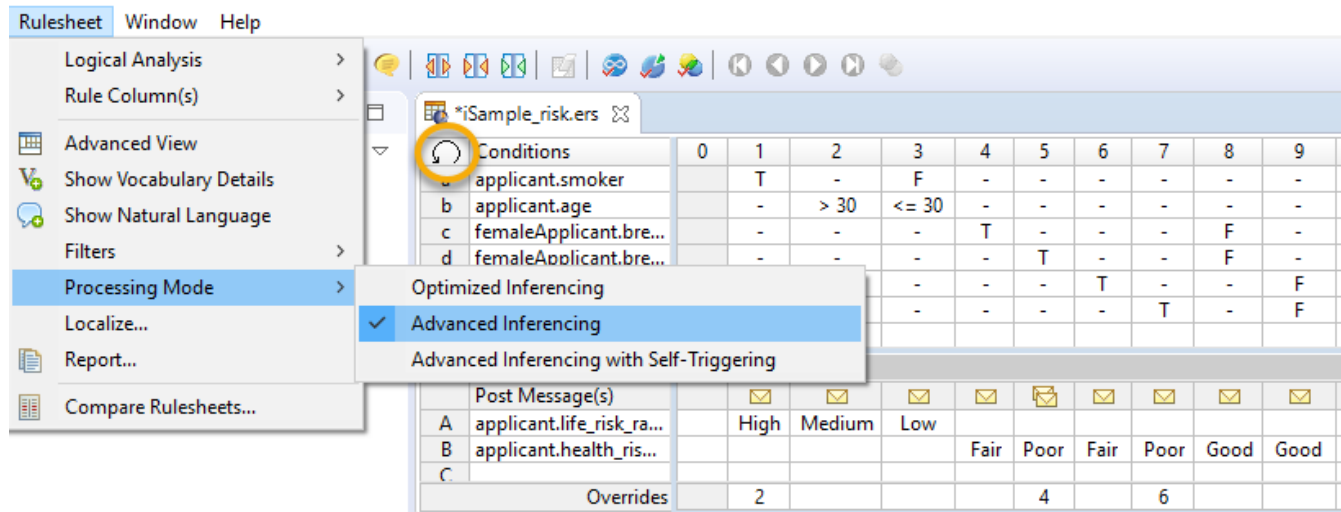
We emphasize it here with an orange highlight to the immediate left of the **Conditions** header.

If the rule engine is permitted to loop through the rules, 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 fireirclsetting the value to B. The second 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 fourth rule fires. Up to this point, the rule engine is exhibiting standard, optimized inferencing behavior.

Here is the new part: the value (state) changed since the second rule last fired, so the rule engine will re-evaluate the condition, and, finding it satisfied, will fire the second rule again, advancing the value to C. The third 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 130: 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 is the key to understanding looping: when a looping processing mode is enabled, rules are continually re-evaluated and re-fired in a sequence determined by their dependency network as long as the data state changed since their last firing. Once the data state no longer changes, looping ceases.

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 Loop iterations example, 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 is not 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 topic.

Trivial loops

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

To enable the self-triggering mode of looping, select **Rulesheet > Processing Modes > Advanced Inferencing with Self-Triggering** from the Corticon Studio menubar, as shown:

Figure 131: Selecting Advanced Inferencing with Self-Triggering processing mode for a Rulesheet

	Conditions	0	1	2	3	4	5	6	7	8	9
a	applicant.smoker		T	-	F	-	-	-	-	-	-
b	applicant.age		-	> 30	<= 30	-	-	-	-	-	-
c	femaleApplicant.bre...		-	-	-	T	-	-	-	F	-
d	femaleApplicant.bre...		-	-	-	-	T	-	-	F	-
			-	-	-	-	-	T	-	-	F
			-	-	-	-	-	-	T	-	F

	Post Message(s)	✉	✉	✉	✉	✉	✉	✉	✉	✉
A	applicant.life_risk_ra...	High	Medium	Low						
B	applicant.health_ris...				Fair	Poor	Fair	Poor	Good	Good
C										
	Overrides	2			4		6			

Menu items shown:
Rulesheet | Window | Help
- Logical Analysis >
- Rule Column(s) >
- Advanced View
- Show Vocabulary Details
- Show Natural Language
- Filters >
- Processing Mode >
  - Optimized Inferencing
  - Advanced Inferencing
  - ✓ Advanced Inferencing with Self-Triggering
- Localize...
- Report...
- Compare Rulesheets...

*iSample_risk.ers

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

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

Figure 132: Example of an infinite single-rule loop

InfiniteSingleLoop.ers

	Conditions	1
a	Cargo.weight >= 0	T
b		

	Actions	<
	Post Message(s)	
A	Cargo.weight += 1	☑
B		
	Overrides	
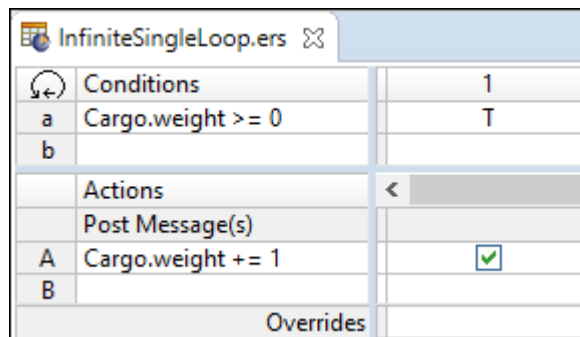
When `Cargo.weight` has a value equal to or greater than `0`, then rule 1 fires and the value of `Cargo.weight` is incremented by `1`. Data state has now *changed*—in other words, the value of at least one of the attibutes has changed. In this case, it is the value of `Cargo.weight`.

Because rule 1 executing that *caused* the data state change, and because self-triggering is enabled, the same rule 1 will be re-evaluated. Now, if the value of `Cargo.weight` satisfies the rule initially, it 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 133: Example of an infinite loop created by a self-triggering rule

SingleLoopAsNonconditional.ers		0
Conditions		
a		
b		
Actions	<	
Post Message(s)		
A	Cargo.weight += 1	☑
B		
Overrides		

This figure is also a good example of why it might be appropriate to disable self-triggering processing. You 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 you intentionally prevented this rule from iterating, even though self-triggering is enabled. This rule will execute only once, regardless of the loop processing mode.

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

Figure 134: Example of a finite single-rule loop

FiniteSingleRuleLoop.ers		1
Conditions		
a	Cargo.weight	0..20
b		
Actions	<	
Post Message(s)		
A	Cargo.weight += 1	☑
B		
Overrides		

In the preceding figure, 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 is 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.

Multi-rule loops

As the name suggests, multi-rule loops exist when two 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 is an example of a multi-rule logical loop:

Figure 135: Example of a finite multi-rule loop

In the figure, rule 2 is dependent upon rule 1, and rule 1 is dependent upon rule 2. Rule 3 was also added, which does not participate in the 1—2 loop, but generates a `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. Now you will see how they behave. In **Ruletest for the multi-rule Rulesheet**, you see a simple Ruletest.

Figure 136: Ruletest for the multi-rule Rulesheet

`Cargo.weight` has a starting value to get the loop going. According to the condition in rule 1, this value must be between 1 and 10 (inclusive).

Figure 137: Ruletest for the multi-rule Rulesheet

Severity	Message	Entity
Info	If weight is between 0 and 10 [1] then volume is twice weight [=2]	Cargo[1]
Warning	As long as volume is greater than 0, increment weight by 1 [now2]	Cargo[1]
Info	If weight is between 0 and 10 [2] then volume is twice weight [=4]	Cargo[1]
Warning	As long as volume is greater than 0, increment weight by 1 [now3]	Cargo[1]
Info	If weight is between 0 and 10 [3] then volume is twice weight [=6]	Cargo[1]
Warning	As long as volume is greater than 0, increment weight by 1 [now4]	Cargo[1]
Info	If weight is between 0 and 10 [4] then volume is twice weight [=8]	Cargo[1]
Warning	As long as volume is greater than 0, increment weight by 1 [now5]	Cargo[1]
Info	If weight is between 0 and 10 [5] then volume is twice weight [=10]	Cargo[1]
Warning	As long as volume is greater than 0, increment weight by 1 [now6]	Cargo[1]
Info	If weight is between 0 and 10 [6] then volume is twice weight [=12]	Cargo[1]
Warning	As long as volume is greater than 0, increment weight by 1 [now7]	Cargo[1]
Info	If weight is between 0 and 10 [7] then volume is twice weight [=14]	Cargo[1]
Warning	As long as volume is greater than 0, increment weight by 1 [now8]	Cargo[1]
Info	If weight is between 0 and 10 [8] then volume is twice weight [=16]	Cargo[1]
Warning	As long as volume is greater than 0, increment weight by 1 [now9]	Cargo[1]
Info	If weight is between 0 and 10 [9] then volume is twice weight [=18]	Cargo[1]
Warning	As long as volume is greater than 0, increment weight by 1 [now10]	Cargo[1]
Info	If weight is between 0 and 10 [10] then volume is twice weight [=20]	Cargo[1]
Warning	As long as volume is greater than 0, increment weight by 1 [now11]	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 135: Example of a finite multi-rule loop on page 177) 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 you have only seen simple examples so far, looping rules can get much more complicated. Sometimes, rules have mutual dependencies by accident—you did not 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 were already evaluated.) You must manually enable your preferred loop processing mode to cause the loops to execute. This is the strongest, most foolproof mechanism for preventing unexpected looping behavior: simply keep loop processing disabled.

How to identify loops
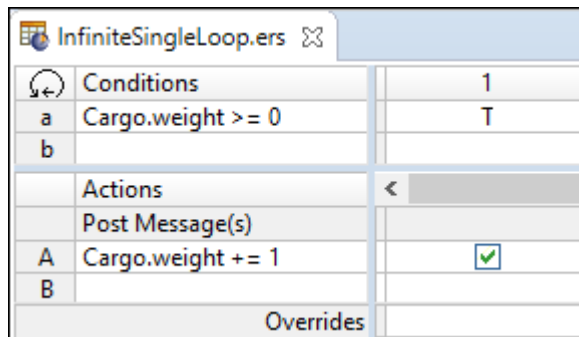
Assuming that you have not intentionally incorporated looping logic in your Rulesheet, you need a way to discover if unintentional loops occur in your rules.

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, a few of the same examples will be used.

Figure 138: Example of an infinite single-rule loop

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

Figure 139: Checking for logical loops in a Rulesheet
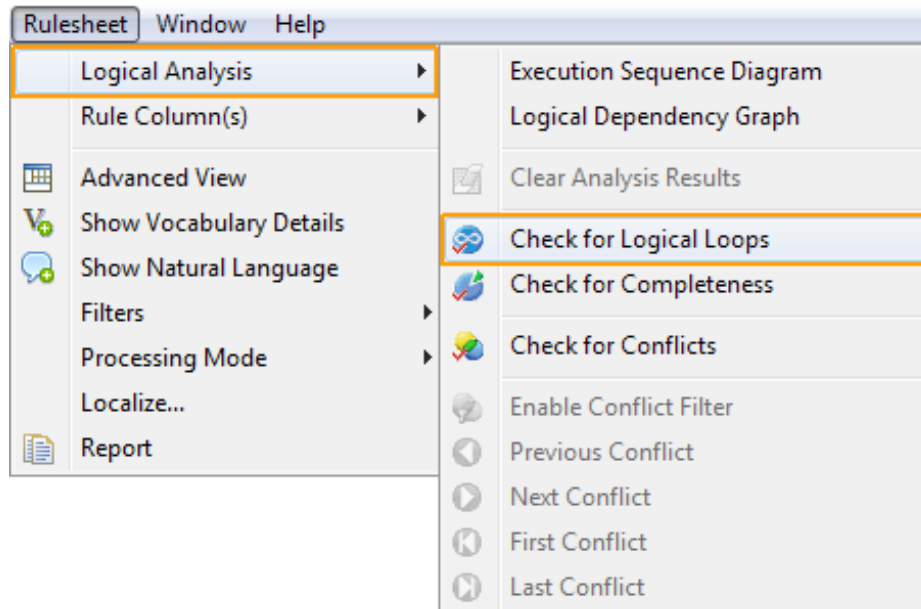
Figure 140: A single-rule loop detected by the Check for Logical Loops tool
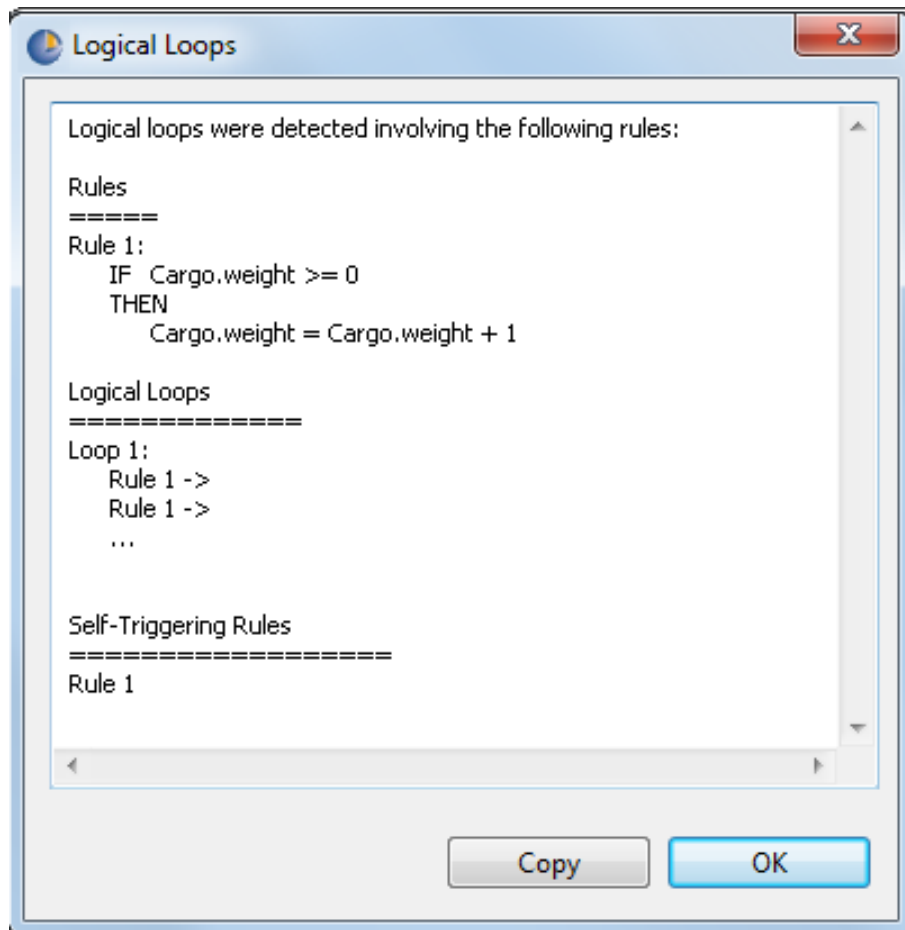
The Check for Logical Loops tool first lists rules where mutual dependencies exist. Then, it lists the distinct, independent loops in which those rules participate, and finally it lists where self-triggering rules exist (if any). In this simple single-rule loop example, only one rule contains a mutual 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 your rules *have* loops, and gives you an opportunity to remove or modify the offending logic.

How to remove loops

If the Check for Logical Loops tool detects loops, you 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. When 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`. When `Cargo.weight` reaches `21`, the rule's condition will no longer be satisfied and the loop terminates.

- If some loops are good and some are not, then remove the inter-dependencies in the unwanted loops and ensure that the selected loops are not infinite.

How to terminate infinite loops

By definition, infinite loops will not terminate by themselves. Therefore, Corticon provides a safety valve 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. After the number of loops exceeds the `maxloops` 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 following figure shows a `Violation` message:

Figure 141: 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, set the value that determines what constitutes an endless loop. For Decision Services that have Rulesheets with a **Processing mode** that allows looping , it is important to limit the loop count and prevent endless loops.

In the `brms.properties` file, add the following property with your preferred maximum number of iterations allowed for any loop:

```
com.corticon.reactor.rulebuilder.maxloops=100
```

Looping examples

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

Determine the next working day when given a date

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 is 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 142: Sample Vocabulary for holiday rules
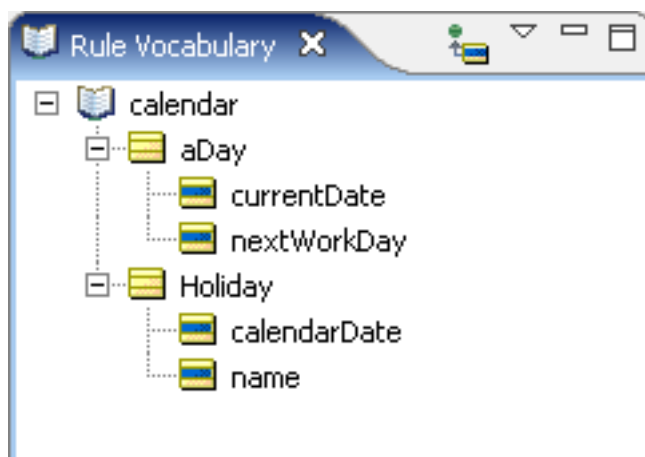
Next, the rules are implemented in the Rulesheet shown in the following figure:

Figure 143: Sample Rulesheet for determining next workday

To step through this Rulesheet:

1. Notice that the **Scope** section is not used. A very simple Vocabulary is used with short entity names and no associations, so aliases are not necessary. Furthermore, none of the rules use collection operations, so aliases representing collections are not required either.

2. The first rule executed is the nonconditional equation (in column 0) setting `nextWorkDay` equal to `currentDate` plus one day.

3. Rule 1 (in column 1) checks to see if the DateTime of the `nextWorkDay` matches any of the holidays defined in one or more `Holiday` entities. If it does, then the Action row B increments `nextWorkDay` by one day and posts a warning message.

4. Rule 3 checks to see if the `nextWorkDay` falls on a Sunday. Notice that this rule uses the `.dayOfWeek` operator, which is described in full detail in the *Rule Language Guide*. If the day of the week is Sunday (in other words, `.dayOfWeek` returns a value of 1), then the Action increments `nextWorkDay` by <u>one</u> day and posts a warning message.

5. Rule 4 checks to see if the `nextWorkDay` falls on a Saturday. If the day of the week is Saturday (in other words, `.dayOfWeek` returns a value of 7), then the Action row C increments `nextWorkDay` by <u>two</u> days and posts a warning message. By incrementing 2 days, an extra iteration is skipped because we know Sunday is also a non-workday.

Do not forget to check for conflicts: they exist in this Rulesheet. Assume that a holiday never falls on a weekend.

Note: Resolution of the conflicts is straightforward, so that is not addressed in detail here. One conflict – that between rules 1 and 4 - is left unresolved because the assumption is 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 144: Holiday rules with ambiguities resolved by overrides

	Conditions	0	1	2	3	4	5
a	aDay.nextWorkDay = Holiday.calendarDate	-	T	F	-	-	-
b	aDay.nextWorkDay.dayOfWeek	-	-	-	1	7	other
c							

	Actions						
	Post Message(s)	✉	✉	✉	✉	✉	✉
A	aDay.nextWorkDay = aDay.currentDate.addDays(1)	☑					
B	aDay.nextWorkDay = aDay.nextWorkDay.addDays(1)		☑		☑		
C	aDay.nextWorkDay = aDay.nextWorkDay.addDays(2)					☑	
D							
	Overrides		5		2	2	

Rule Statements ⊠   Rule Messages

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

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

Figure 145: Results of Logical Loop Check

```
Logical Loops                                    [ X ]

Logical loops were detected involving the following rules:

Rules
=====
Rule 1:
     IF   aDay.nextWorkDay = Holiday.calendarDate
     THEN
          aDay.nextWorkDay = aDay.nextWorkDay.addDays ( 1 )

Rule 3:
     IF   aDay.nextWorkDay.dayOfWeek = 1
     THEN
          aDay.nextWorkDay = aDay.nextWorkDay.addDays ( 1 )

Rule 4:
     IF   aDay.nextWorkDay.dayOfWeek = 7
     THEN
          aDay.nextWorkDay = aDay.nextWorkDay.addDays ( 2 )

Logical Loops
=============
Loop 1:
     Rule 1 ->
     Rule 3 ->
     Rule 4 ->
     Rule 1 ->
     ...

Loop 2:
     Rule 2 ->
     Rule 5 ->
     Rule 2 ->
     ...

Self-Triggering Rules
==================
Rule 1
Rule 3
Rule 4

              [ Copy ]    [ OK ]
```

This window first identifies which rules are involved in loops. The window also outlines the specific attribute interactions that create the loops.

Now that you understand the looping logic in your Rulesheet, create a Ruletest to verify that the loops operate as intended and produce the correct business results.

Figure 146: Ruletest for holiday rules
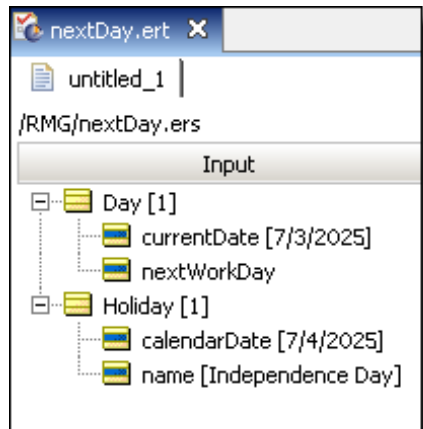
Given that July 4th, 2025 falls on a Friday, you expect `nextWorkDay` to contain a final value of July 7th, 2025, a Monday, when the loops terminate. When the Ruletest runs, you see the following:

Figure 147: Ruletest output for holiday rules

As you can see, the result is a three-day weekend!

Remove duplicated children in an association

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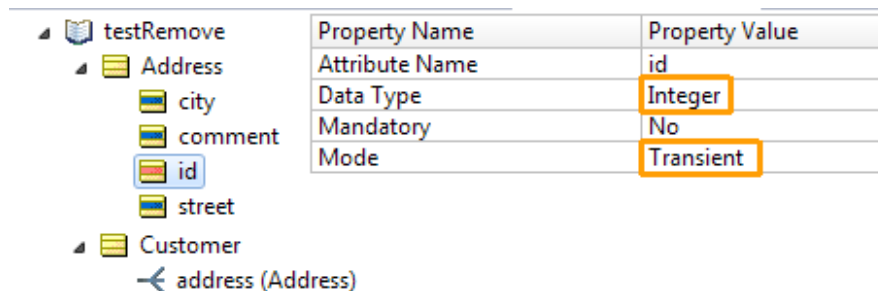
For a `Customer->Address` association (one-to-many), each address must be unique.

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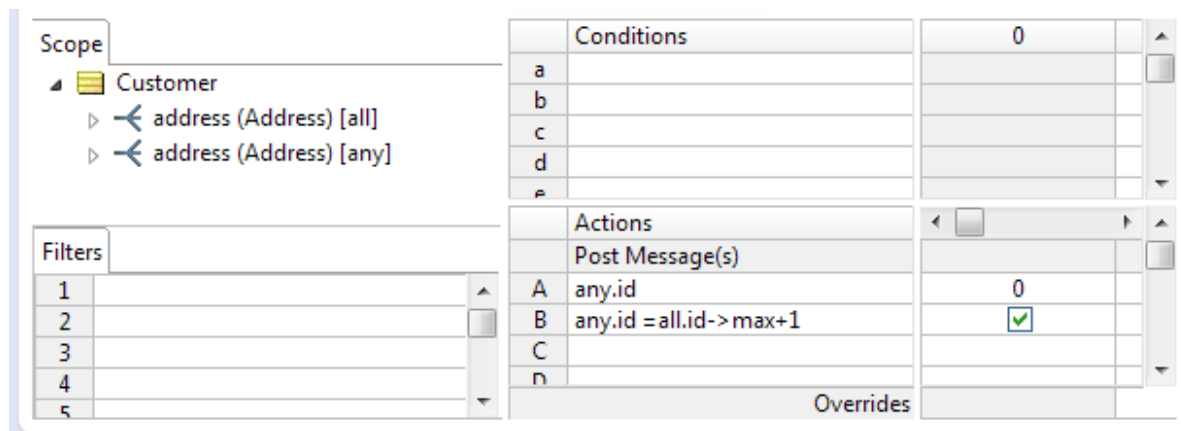
To ensure that the filtering process is controlled, you need unique identifier attribute values to distinguish the instances. If the address already has an attribute that is a unique identifier, then you could use that in the filter; otherwise, you need to create a transient, integer attribute, `id`,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

Using the created identifier attribute, 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:

After each address has a unique identity, the second Rulesheet does 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:

Scope		Conditions	1	2
▲ 🔲 Customer	a			
▷ 🔻 Filters	b	a1.street = a2.street	T	
▷ ◄ address (Address) [a1]	c	a1.city = a2.city	T	
▷ ◄ address (Address) [a2]	d			
	e			
	f			
	g			
	h			
	.			

Filters		Actions		
		Post Message(s)		
1 🔻 a1.id < a2.id	A	a2.remove	✔	
2	B			
3	C			
4	D			
5	E			
6	F			
7	G			
8	..			
		Overrides		
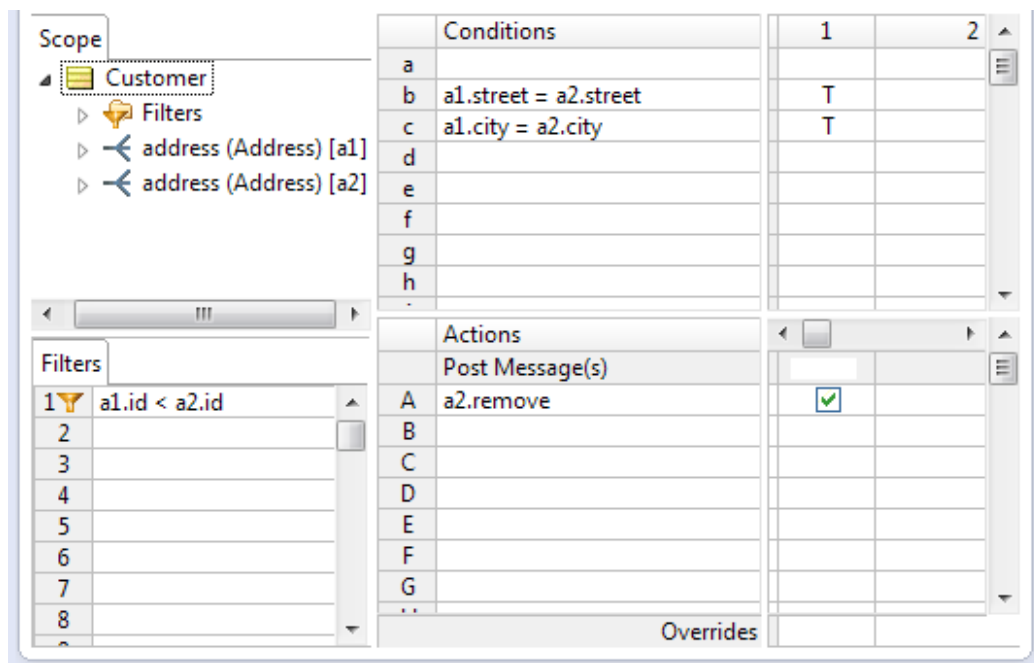
A Ruleflow puts the two Rulesheets into sequence, as shown:

testUnique → testRemove

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

Input	Output
▲ 🔲 Customer [1]	▲ 🔲 Customer [1]
▲ ◇— address (Address) [1]	▲ ◇— address (Address) [3]
🔲 city [city1]	🔲 city [city2]
🔲 street [street1]	🔲 id [4]
▲ ◇— address (Address) [2]	🔲 street [street2]
🔲 city [city1]	▲ ◇— address (Address) [4]
🔲 street [street1]	🔲 city [city1]
▲ ◇— address (Address) [3]	🔲 id [1]
🔲 city [city2]	🔲 street [street1]
🔲 street [street2]	
▲ ◇— address (Address) [4]	
🔲 city [city1]	
🔲 street [street1]	

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

Note: Rule Statements were not requested for this process. Because the duplicates are being removed during the execution of the rule, each removed address was dropped from memory, and no longer has a meaningful reference when the 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:

Scope		Conditions	1	2
▲ ▣ Customer	a			
▷ 🏺 Filters	b	a1.street = a2.street	T	
▷ ◁ address (Address) [a1]	c	a1.city = a2.city	T	
▷ ◁ address (Address) [a2]	d			
	e			
	f			
	g			
	h			

Filters		Actions	◀ ☐ ▶
		Post Message(s)	
1 🏺 a1.id < a2.id	A	a2.remove	
2	B	a2.comment	'Duplicate'
3	C		
4	D		
5	E		
6	F		
7	G		
8		Overrides	

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

Input	Output
⊟ ▣ Customer [1]	⊟ ▣ **Customer [1]**
⊟ ◇ address (Address) [1]	⊟ ◇ **address (Address) [1]**
city [city1]	city [city1]
street [street1]	**comment [Duplicate]**
⊟ ◇ address (Address) [2]	**id [3]**
city [city1]	street [street1]
street [street1]	⊟ ◇ **address (Address) [2]**
⊟ ◇ address (Address) [3]	city [city1]
city [city2]	**comment [Duplicate]**
street [street2]	**id [4]**
⊟ ◇ address (Address) [4]	street [street1]
city [city1]	⊟ ◇ **address (Address) [3]**
street [street1]	city [city2]
	id [2]
	street [street2]
	⊟ ◇ **address (Address) [4]**
	city [city1]
	id [1]
	street [street1]

> **Note:** Again, Rule Statements were not used. There are 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 because 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 were flagged as `'Duplicate'`.

How to use conditions as a processing threshold

Looping, which involves revisiting, re-evaluating, and possible re-firing rules, and requires you to enable one of the looping modes, must be distinguished from another behavior that may appear to be similar.

You probably noticed Corticon's inherent ability to process multiple test scenarios at once. For example, a rule written using the Vocabulary term `Cargo.weight` is evaluated (and potentially fired) for every instance of `Cargo` encountered during execution. If a Ruletest contains four `Cargo` entities, then the rule engine tests the rule's conditions with each of them. If any of the `Cargo` entities satisfy the rule's conditions, then the rule fires. This could mean that the rule fires once, twice, or up to four times, depending on the actual data values of each `Cargo`. From the prior discussion of scope, a rule will evaluates *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 is nothing that you need to do to enable it. Note that this behavior is different from the modes of looping 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 four `Cargo` entities.

The advantage of this natural iteration is that you do not need to force it. The rule engine automatically processes all data that shares the same scope as the rule. If the Ruletest contains four Cargos, the rule will be evaluated four times. If the Ruletest contains 4000 Cargos, the rule is evaluated 4000 times. You do not write the rule differently in Corticon Studio.

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

Here is a simple example.

Figure 148: Rulesheet and Ruletest, no threshold condition, CaPT disabled
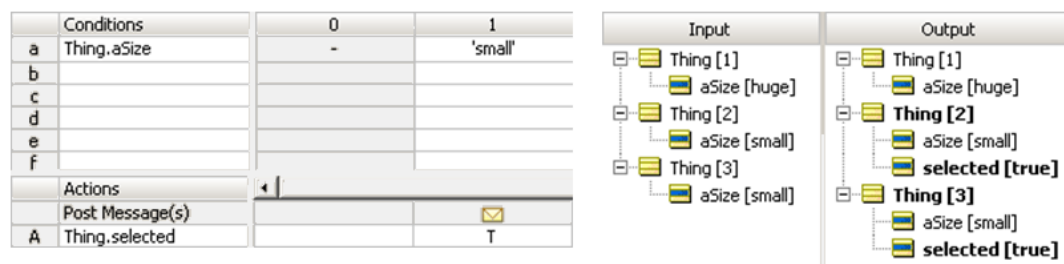
In the preceding example, no threshold condition, CaPT disabled, you see a simple rule that sets `thing.selected = true` for all `thing.aSize = 'small'`. Notice in the adjacent Ruletest, that each small `Thing` is selected. `Thing[2]` and `Thing[3]` are both small, so they are both selected by the rule. The rule evaluated all three Things, but finding only two that satisfy the rule's condition, only fires twice. This iteration happened automatically.

What if you 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 accomplishes this goal.

Figure 149: Rulesheet and Ruletest, threshold condition added, CaPT disabled

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

You might 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 you see in the adjacent Ruletest, this is not the case. The reason is that `Thing[3]` is an 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 you to interrupt processing of the binding. You activate this option by selecting the rule column involved, and then from the Corticon Studio menu bar, choose **Rulesheet > Rule Columns(s) > Use Condition as Processing Threshold**.

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

Figure 150: Rulesheet and Ruletest, threshold condition added, CaPT enabled

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

TestYourself questions for Rule dependency chaining and looping

Note: Try this test, and then go to TestYourself answers for Rule dependency and inferencing on page 340 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.

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 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?
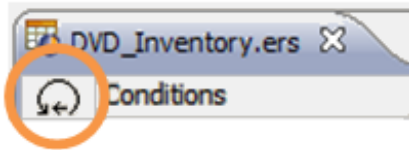
10. This icon indicates which type of inferencing is enabled for this Rulesheet?
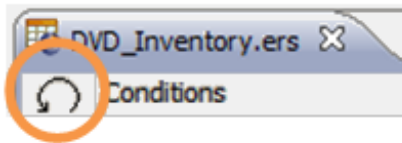
11. A _____ determines the sequence of rule execution and is generated when a Rulesheet is _____.

8

Filters and preconditions

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

Filters can be set to be Database filters on page 204 when its entity is defined to persist in a datastore and the entity is set to extend to database.

Any conditional expression entered in the **Filters** window of a Rulesheet is referred to as a *filter*, regardless of its strict mode of behavior. This will help you differentiate the expression from its specific behaviors.

For details, see the following topics:

- What is a filter
- What is a precondition
- How to use collection operators in a filter
- Filters that use OR
- TestYourself questions for 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.

Data that satisfies a Filter expression is referred as surviving the Filter. Data that does not survive the filter is filtered out, and then 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 151: Aliases declared
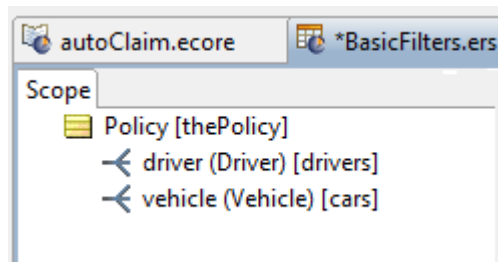
The **Scope** window in this figure defines aliases for a root-level `Policy` entity, a collection of `Driver` entities related to that `Policy`, and a collection of `Vehicle` entities related to that `Policy`, named `thePolicy`, `drivers`, and `cars`, in that order.

To start with, write a simple Filter and observe its default behavior. In the following simple scenario, the Filter expression reduces the set of data acted upon by the nonconditional rule (column 0), which in this case posts the rule statement as a message.

Figure 152: Rulesheet to illustrate basic filter behavior
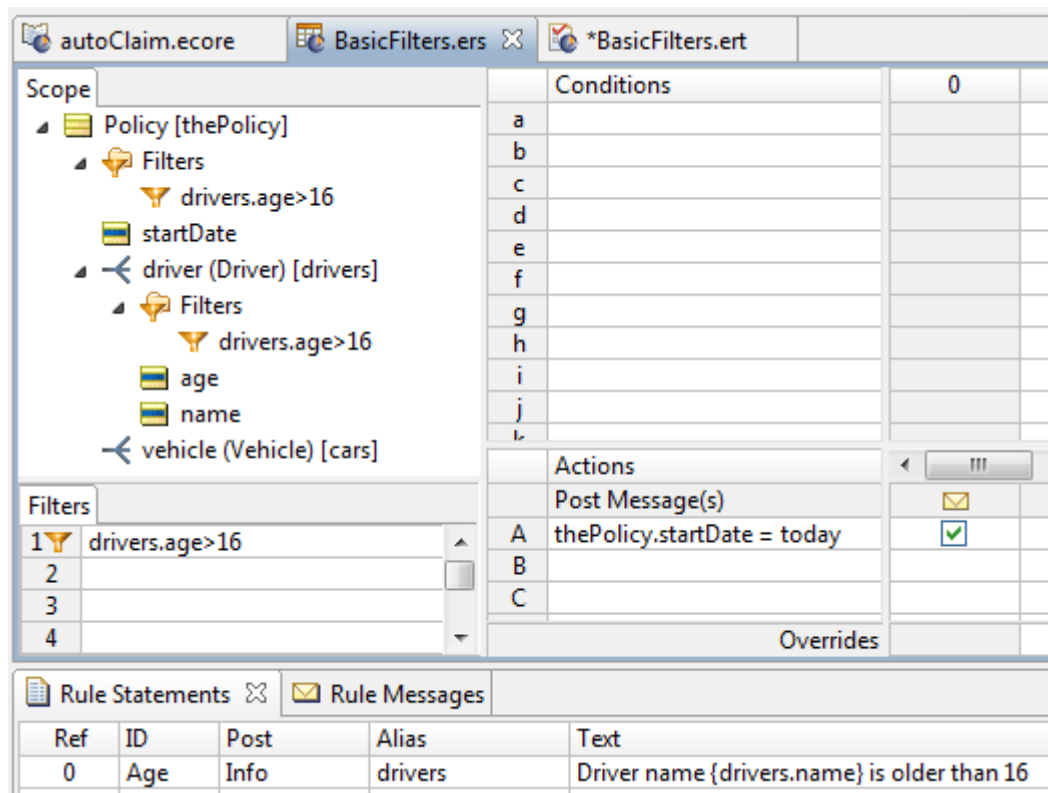
The result is not unexpected: for every element in the collection (every `Driver`) whose `age` attribute is greater than `16`, you see a posted message in the Ruletest, as shown:

Figure 153: Ruletest to test filter behavior

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

Full filters

By default, each filter you write acts as a *full* filter. This means not only will the data not satisfying the Filter expression 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 154: Ruletest for Full Filter

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

Why would you want a Filter to behave this way? Perhaps because, if these are the only drivers seeking a policy, then 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 an easy way to specify and control *which* `Policy` entities are affected.

Disabling a Full Filter

When testing, you might want to remove one filter. Instead of deleting the filter, you can *disable* it by right-clicking the rule and then choosing **Disable**, as shown:
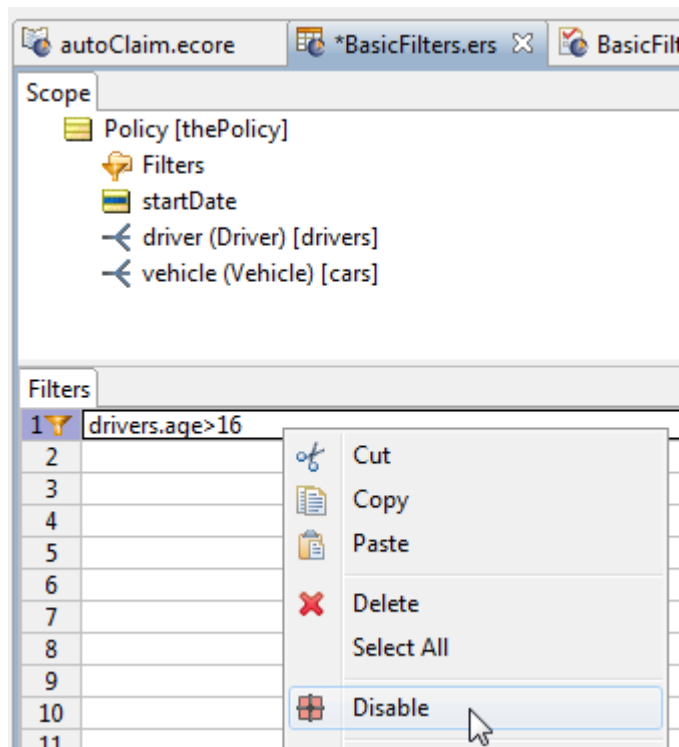
After the filter is disabled, all applications of the filter are rendered in gray, as shown:
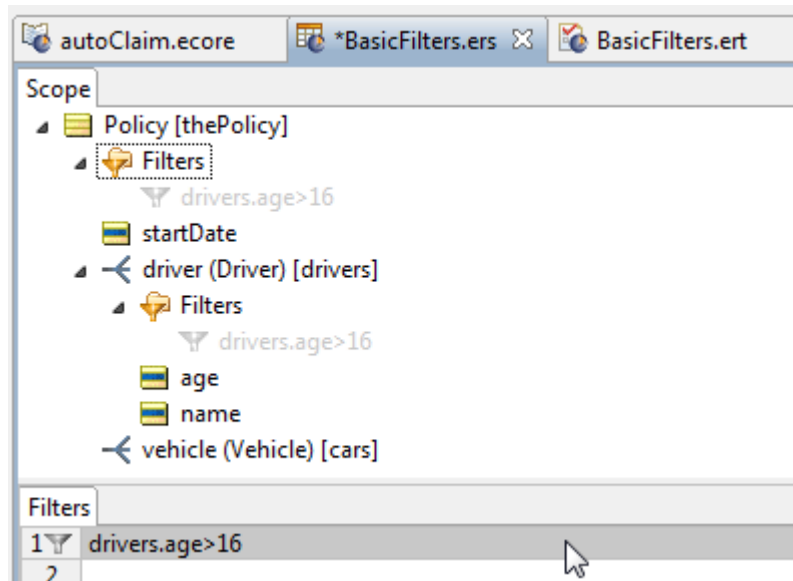
A disabled full filter is really no filter at all. You can perform the corresponding action to again **Enable** the filter.

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, you 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 155: Selecting to limit a filter
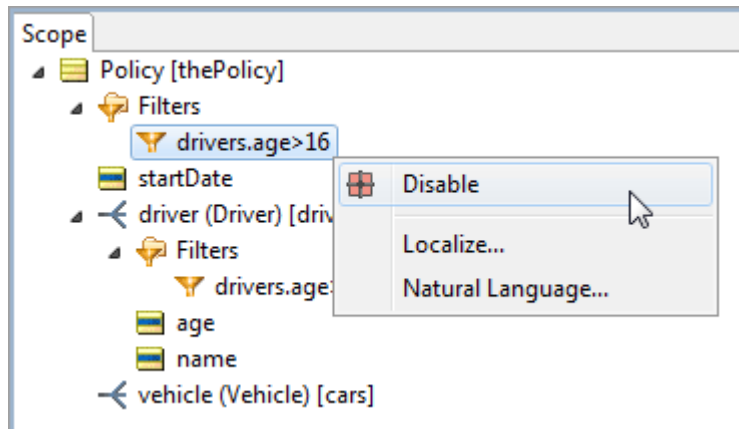
This causes that specific filter position to no longer apply, indicated in gray:

Figure 156: Limiting filter set
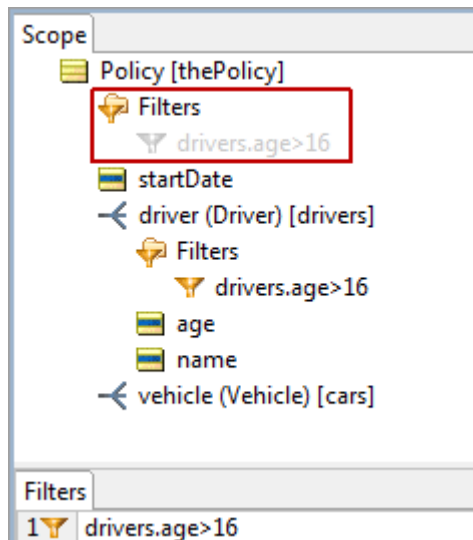
Notice that the filter is still enabled, and that it will still be applied at the `Driver` level. The filter was *limited*.

Use case for limiting filters

The preceding example was basic. Let's explore some more complex examples 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 two objectives: one to process customers, and the second to process orders.

If you define a filter that tests for a GOLD status on an order, there can be four logical iterations of how the filter could be applied to the rules.

```
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, a more complex set of limiting filters is discussed.

Example of limiting filters

Consider the following Rulesheet Scope of a Vocabulary:

Figure 157: Scope in a Rulesheet that will be filtered
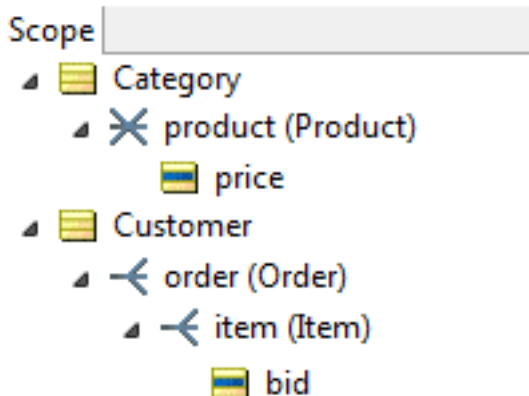
Consider the filter to be applied to data:

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

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

Figure 158: Definition of a filter
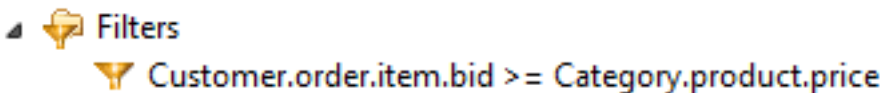
The resulting filter application applies at several levels, as shown:

Figure 159: Application of the filter to the Scope's tree structure

```
Scope
  ⊿ ▦ Category
     ⊿ 🝑 Filters
           ▽ Customer.order.item.bid >= Category.product.price
     ⊿ ✖ product (Product)
        ⊿ 🝑 Filters
              ▽ Customer.order.item.bid >= Category.product.price
           ▦ price
  ⊿ ▦ Customer
     ⊿ 🝑 Filters
           ▽ Customer.order.item.bid >= Category.product.price
     ⊿ ⊰ order (Order)
        ⊿ 🝑 Filters
              ▽ Customer.order.item.bid >= Category.product.price
        ⊿ ⊰ item (Item)
           ⊿ 🝑 Filters
                 ▽ Customer.order.item.bid >= Category.product.price
              ▦ bid
```
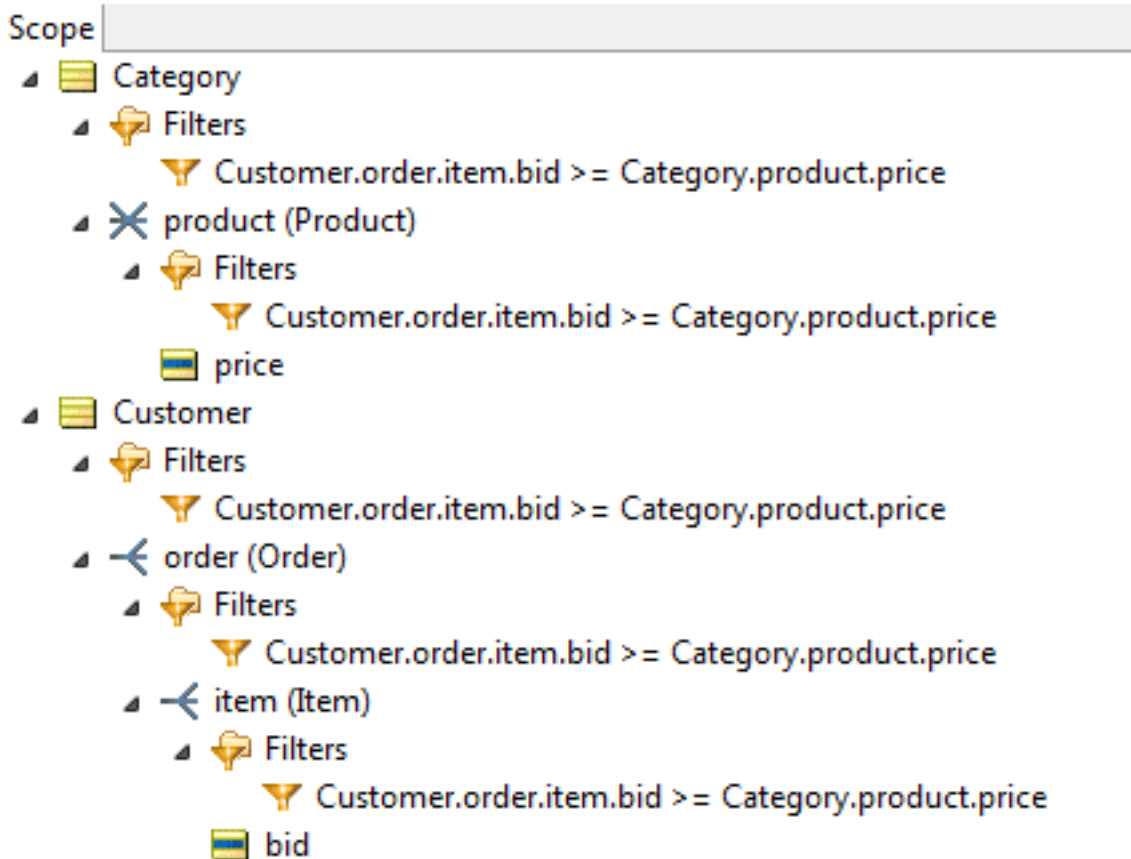
A Ruletest Testsheet might be created as follows:

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[1] 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:

- If 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.

- If the filter is applied to `Customer.order`, then the surviving tuples are {`Customer[1]`, `Order[1]`} and {`Customer[1]`,`Order[2]`}. If **not** applied, then there is no effect (because there was no `Order` child of `Customer[1]` that did not survive the filter).

- If 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 the result: {`Customer[1]`,`Order[1]`}, { `Customer[1]`,`Order[2]`}, {`Customer[2]`,`Order[3]`}.

- If 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 are {Customer[1],Order[1],Item[1]}, {Customer[1],Order[1],Item[2]}, {Customer[1],Order[2],Item[3]}.

- If the filter is applied to Category, then the surviving entities are Category[1], Category[2]. If **not** applied, then Category[1], Category[2], Category[3].

- If the filter is applied to the Category.product level, then the surviving tuples are 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 a Vocabulary has elected to have an EDC Datasource, setting an entity's **Datastore Persistent** property to **Yes** declares that the entity will map to a table in the Datasource. A database cylinder decorates the icons of the entity and its attributes, as shown:

After the property is set, right-clicking an Entity's alias in a Rulesheet's **Scope** section shows the menu command to **Extend to Database**, as shown:
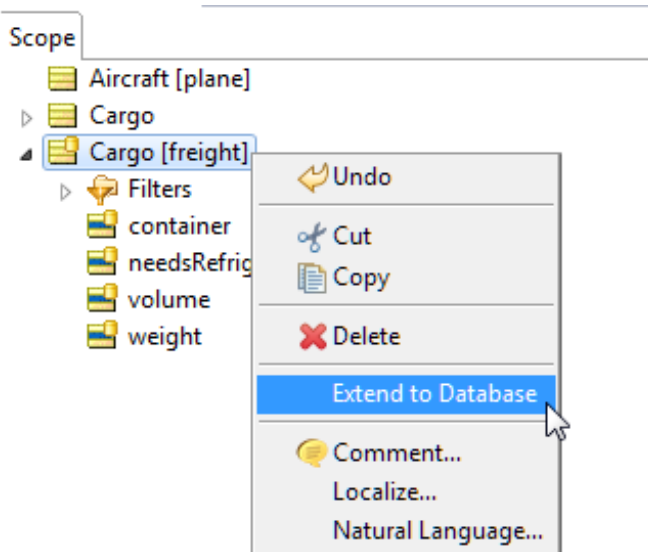
Then, you can define filters and set them each as a **Database Filter**, as shown:
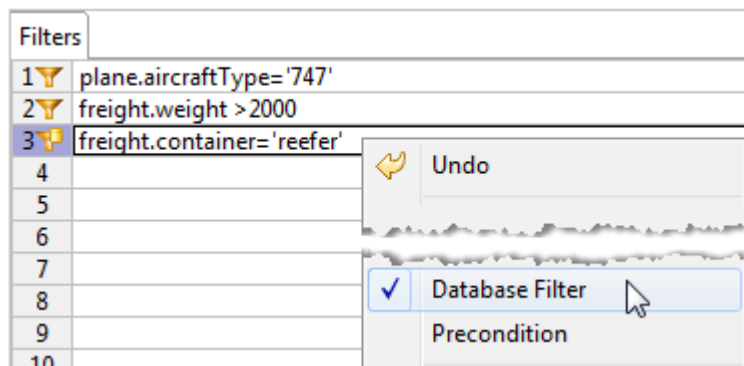
When checked, the filter becomes a *database query* that retrieves data from the connected database, and then adds the retrieved data to working memory.

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 239 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:
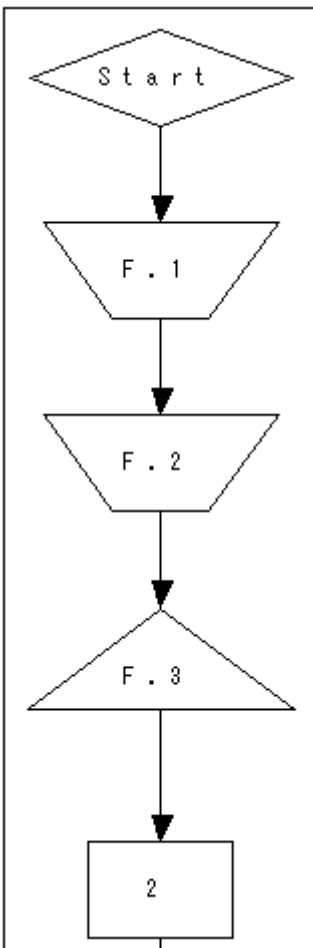
In this example, **F.1**, the database query, is displayed within a triangle, while **F.2**, the local filter, is displayed within an inverted trapezoid (a quadrilateral with parallel horizontal bases and legs that converge downward).

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. If you do so, then 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 are 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 *optional* behavior, then you will be in good shape later. Also, it may be helpful to think of the precondition behavior, if enabled, as 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.

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 you 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 survive, and at least one must survive for the parent `Policy` also to survive.

Figure 160: Input Rulesheet for Precondition

But, how does this affect the `Claim` in nonconditional row A (rule column 0)? `Claim`, as a root-level entity, is safely *outside of the scope* of the filter, and therefore unaffected by it. Nothing the filter does (or does not do) has any effect on what happens in Action row A—the two logical expressions are 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 161: Rulesheet for an action unaffected by a filter

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

Using the same example, right-click on **Filters** row 1 and select **Precondition**.

Figure 162: Selecting precondition behavior from the filter menu
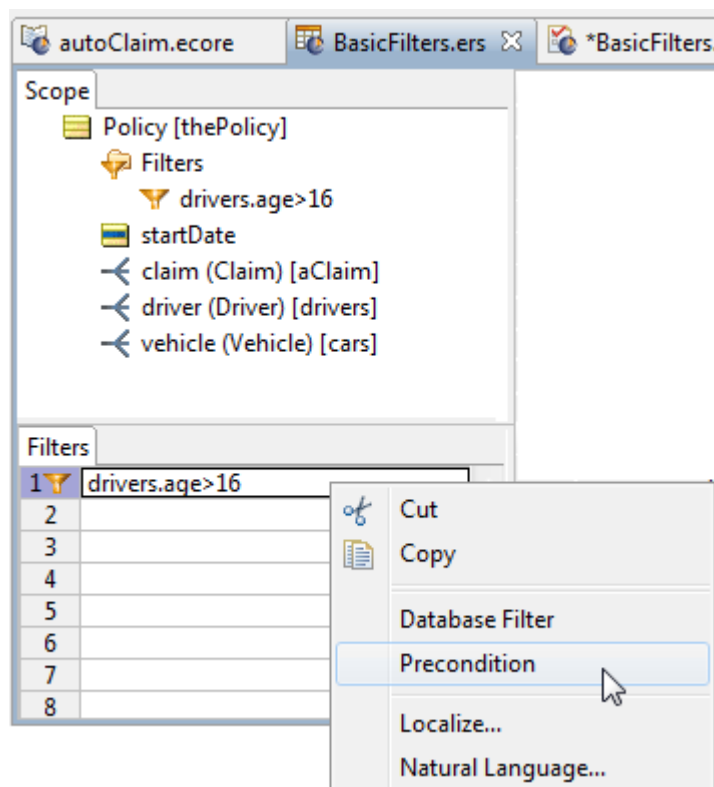
Note that the two options, **Precondition** and **Limiting Filter**, are mutually exclusive: turning one on turns the other off. A filter cannot be both a precondition **and** a limiting filter because at least one piece of data **always** survives a limiting filter, so a precondition never stops execution.

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 163: A Filter in Precondition Mode
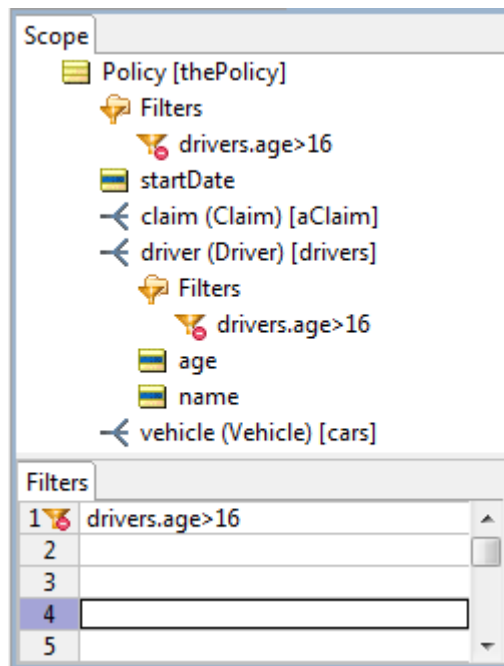
As described before, the precondition behavior of the filter causes Rulesheet execution to stop whenever no data survives the filter. So, in the original case where `Policy` and `Claim` were unassociated, a filter in Precondition mode accomplishes the business intent without artificially changing the Vocabulary or underlying data model, as shown:

Figure 164: Rulesheet with a filter in Precondition mode
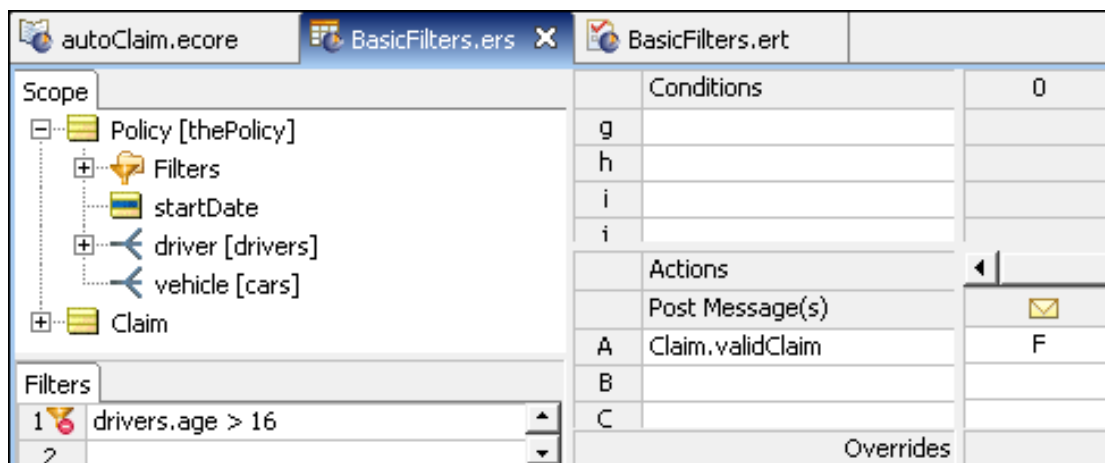
A final proof is provided in the following figure:

Figure 165: Testsheet for a filter in Precondition mode
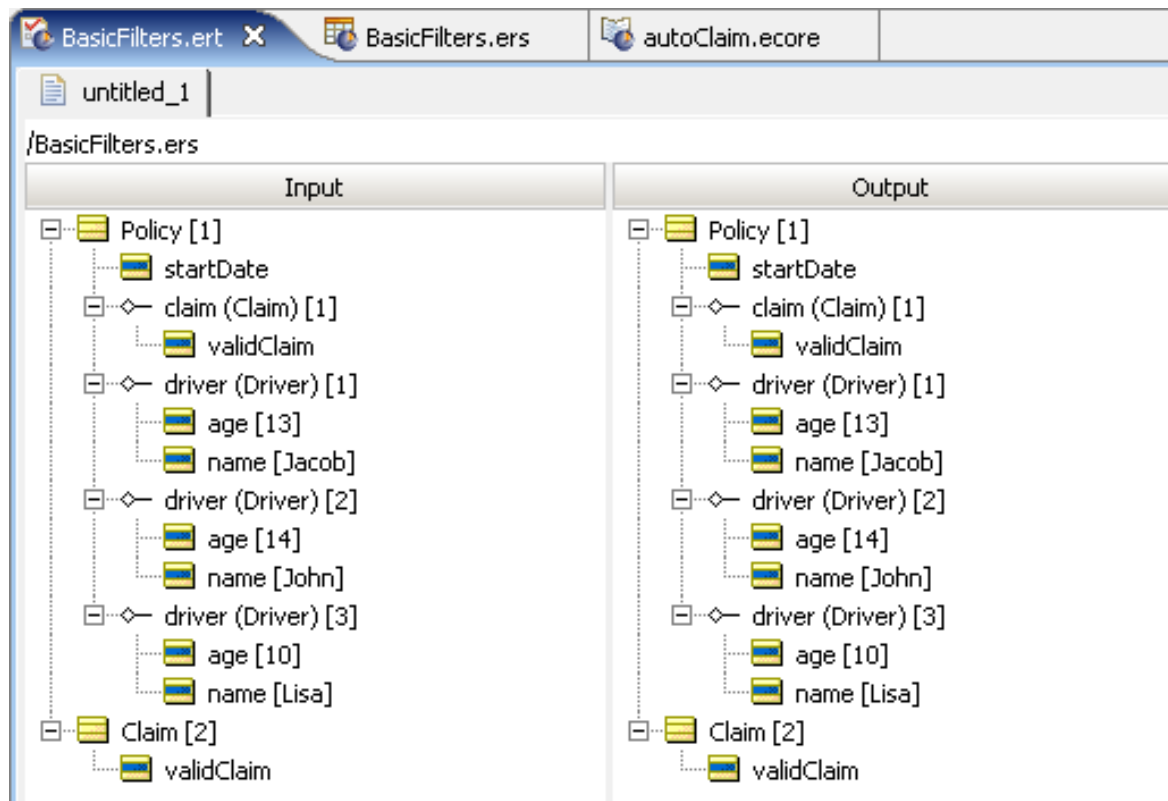
Summary of filter and preconditions behaviors

- A filter reduces the available data for other rules in the Rulesheet to use. Filters show as gray text rather than black. 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 are explicit: data either survives the filter, and allows Rulesheet execution to continue, or no data survives and the Rulesheet execution stops.

- 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 enables 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 fire, the rules engine may have still needed to work hard to come to that conclusion. And, hard work requires time, even for a high-performance rules engine like Corticon.

A Filter expression acting only as a filter never stops Rulesheet execution; it 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 still evaluates this empty set of data against the available remaining rules. Of course, no rules fire, but the evaluation process occurs and 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. No additional evaluations are performed by the rules engine. That Rulesheet is done, and the rules 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 evaluated were the expression in filter-only mode (the default mode).

How to use collection operators in a filter

In the following examples, all filter expressions use their default filter-only behavior. As detailed in the Rule Writing Techniques topics, the logic expressed by the following three Rulesheets provides the same result:

Figure 166: A Condition/Action rule column with 2 Conditional rows

Figure 167: Rulesheet with one condition row moved to filters row

Figure 168: Rulesheet with filter and condition rows swapped

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

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

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

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

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

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

Location matters

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 169: Collection operator in Condition row

Figure 170: Collection operator in Filter row

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 171: Ruletest with three skydivers

What happened here? Because filters are always applied first, the Rulesheet initially 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 the 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.

Now modify the Ruletest and rerun the rules:

Figure 172: Ruletest with four skydivers
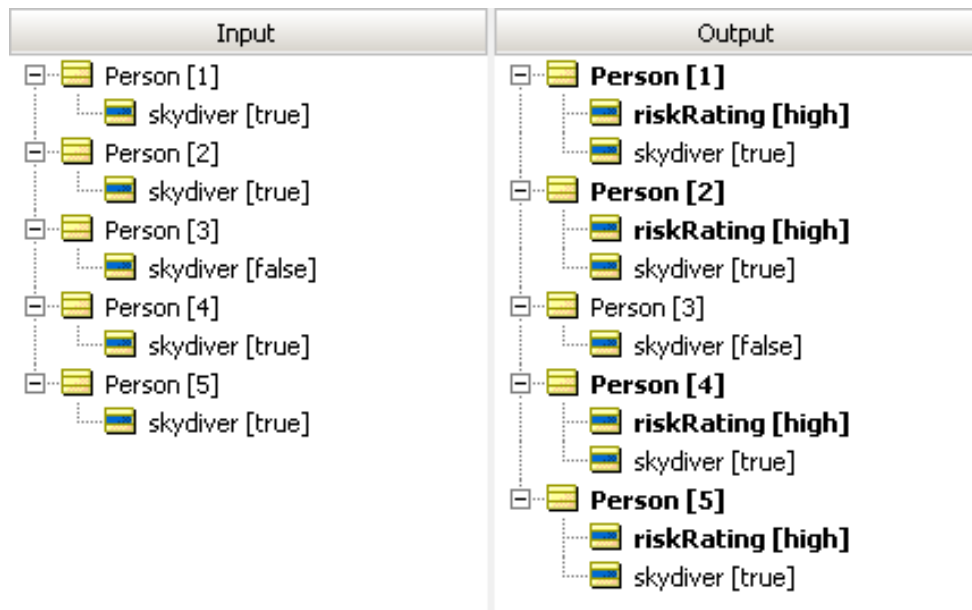
It is clear from this run that the rules fired correctly, and assigned a `riskRating` of `high` to all skydivers for a collection containing more than three skydivers.

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

Figure 173: Ruletest2 with three skydivers
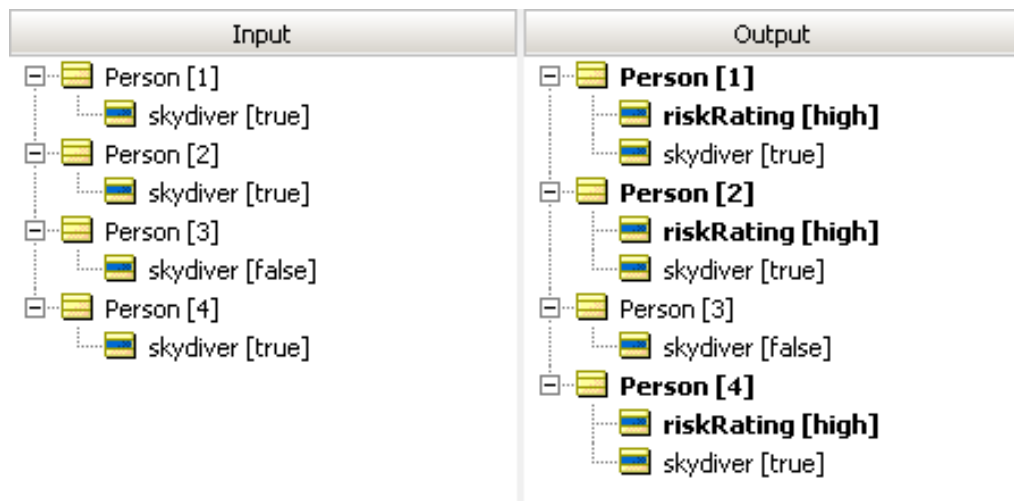
What happened this time? Because filters apply first, the `->size` operator counted the number of elements in the `person` collection, regardless of who skydives and who does not. Here, the filter allows any collection – *and the whole collection* – of more than three persons to pass through to the **Conditions** 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 the Ruletest included only three skydivers, the collection contains four persons, and, therefore, passes the Preconditional filter. Any `skydiver` in the collection has its `riskRating` assigned a value of `high`.

It is important to point out that the Rulesheets in **Collection Operator in Condition row** and **Collection Operator in Filter row** implement two different business rules. When the Rulesheets were built, the plain-language business rule statements violated the 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 is subtle but important. In the first rule statement, the test is for skydivers within groups that contain more than three *skydivers*. In the second, the test is for skydivers within groups of more than three *people*.

Multiple filters on collections

A slightly more complicated example will be constructed by adding a third conditional expression to the rule.

Figure 174: Rulesheet with two conditions

RulesheetWithTwoConditions.ers ✕				
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 175: Rulesheet with two filters

RulesheetWithTwoPreconditions ✕				
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, the Rulesheets differ only in the location of a conditional expression. In the first rulesheet, the gender test is modeled in the second conditional row, whereas in the other rulesheet (Rulesheet with two filters), it is implemented in the second filter row. Does this difference have an effect on rule execution? Build a Ruletest and use it to test the Rulesheet in **Rulesheet with two conditions** first.

Figure 176: Ruletest for Rulesheet with two conditions
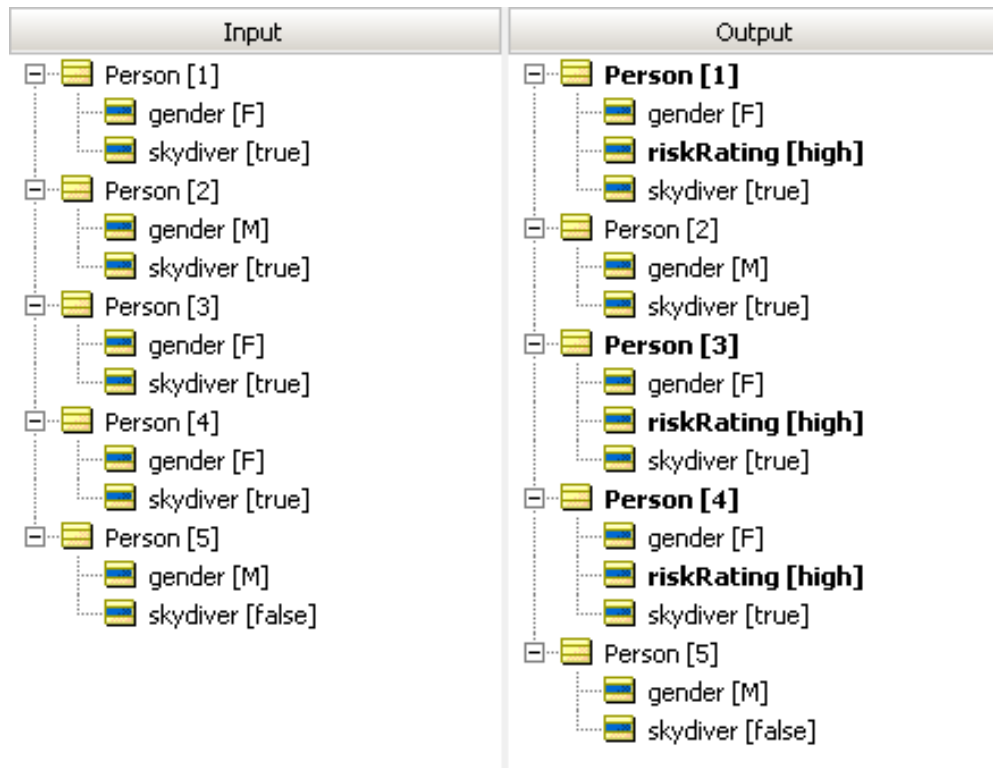
As you see in this figure, the combination of a condition that uses a collection operator (the size test) with another condition that does not (the gender test) produces an interesting result. What appears to have happened is that, for a collection of more than three skydivers, all females in that group were assigned a `riskRating` of `high`. Step-by-step, here is what the rules engine did:

1. The filter screened the collection of persons (represented by the alias `person`) for `skydivers`.

2. If there are more than three surviving elements in `person` (that is, `skydivers`), then all females in the filtered collection are assigned a `riskRating` value of `high`. It may be helpful to think of the rules engine checking to make sure there are more than three surviving elements, then reviewing those whose gender is female, and assigning `riskRating` one element at a time.

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

It is 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* on rule execution.

Now that you understand the results in the **Ruletest for Rulesheet with 2 Conditions**, look at what our second Rulesheet produces.

Figure 177: Ruletest for Rulesheet with two filters
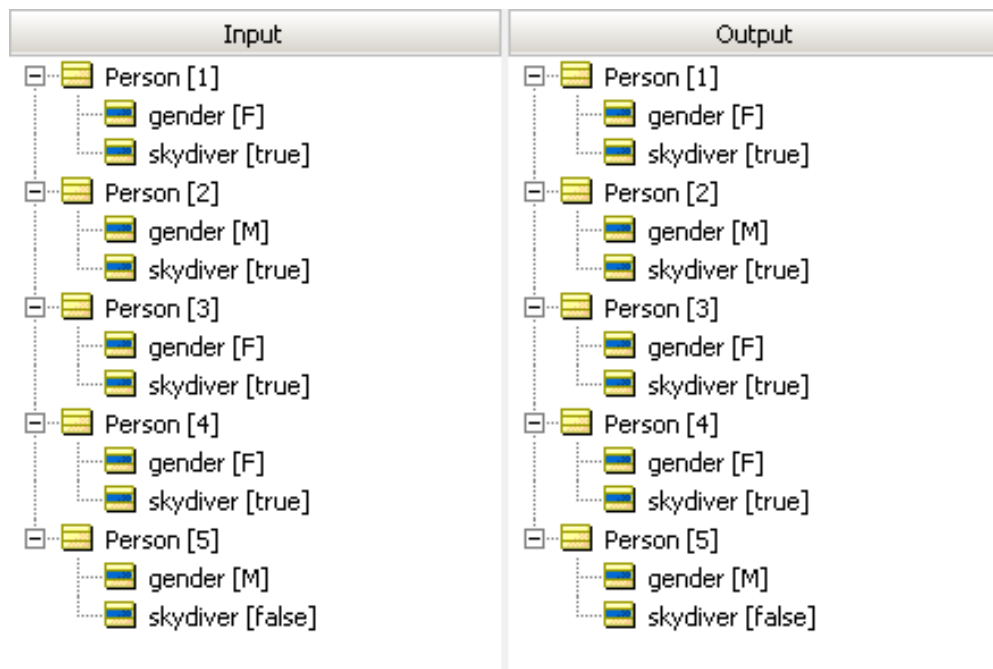
This time, no `riskRating` assignments were made to any element of collection `person`. Why? Because multiple filters are logically **AND**'ed together, forming a compound filter. In order to survive the compound filter, elements of collection `person` must be both skydivers **AND** female. Elements 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 three remaining, then all surviving elements are assigned a `riskRating` value of `high`. Rephrased, the 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

To confirm that you understand how the rules engine executes this Rulesheet, modify the Ruletest and rerun:

Figure 178: Ruletest with risk ratings
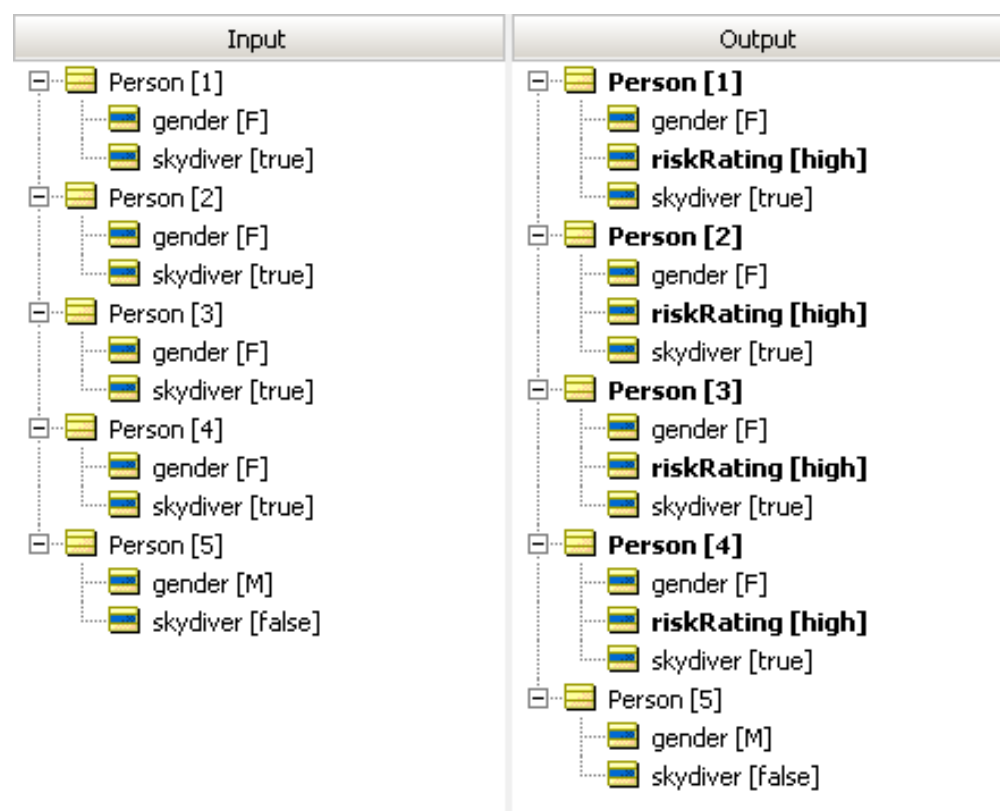
That Ruletest includes four female skydivers, so, if you understand our rules correctly, you expect all four to pass through the compound filter, and then satisfy the size test in the conditions. This test should result in all four surviving elements receiving a `riskRating` of `high`. That test confirms that the expectation 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.

TestYourself questions for Filters and preconditions

Note: Try this test, and then go to 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 two 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 following Filter expression has which behaviors?

limiting filter	full filter	precondition	noncondition

12. The following Filter expression has which behaviors?

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 can be active at the same time?

    a. Full filter and precondition

    b. Limiting filter and precondition

    c. Limiting and full filter

    d. Precondition can only act alone

15. For the sample data in the following figure, determine which data survives the filter for each question. Enter the entity number (the number in brackets) for each survivor in the appropriate column. Assume the collection `Movie` has the alias `movies`; `Movie.dvd` has the alias `dvds`; and `Movie.oscar` has alias `oscars`. None behave as preconditions.

```
untitled_1
┌──────────────────────────────┐
│                              │
├──────────────────────────────┤
│ 📁 DVD                        │
│ ⊟ ▦ Movie [1]                │
│    ├─ ▬ director [Welles]    │
│    ├─ ▬ genre [Drama]        │
│    ├─ ▬ studio [RKO]         │
│    ├─ ▬ title [Citizen Kane] │
│    ├─ ▬ yearReleased [1941]  │
│    ⊟◇─ dVD [1]               │
│    ├─ ▬ priceTier [high]     │
│    ├─ ▬ quantityAvailable [100] │
│    └─ ▬ releaseDate [9/25/2001] │
│    ⊟◇─ oscar [1]             │
│    ├─ ▬ category [Best Actor] │
│    ├─ ▬ nomination [true]    │
│    └─ ▬ win [false]          │
│    ⊟◇─ oscar [2]             │
│    ├─ ▬ category [Best Screenplay] │
│    ├─ ▬ nomination [true]    │
│    └─ ▬ win [true]           │
│    ⊟◇─ oscar [3]             │
│    ├─ ▬ category [Best Picture] │
│    ├─ ▬ nomination [true]    │
│    └─ ▬ win [false]          │
│    ⊟◇─ oscar [4]             │
│    ├─ ▬ category [Best Sound] │
│    ├─ ▬ nomination [true]    │
│    └─ ▬ win [false]          │
│    ⊟◇─ oscar [5]             │
│    ├─ ▬ category [Best Director] │
│    ├─ ▬ nomination [true]    │
│    └─ ▬ win [false]          │
└──────────────────────────────┘
```

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

9

How to recognize and model parameterized rules

Patterns emerge in rules that show that there are limits and constraints that you have to handle.

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
- How to populate an AccountRestriction table from a sample user interface
- TestYourself questions for Recognizing and modeling parameterized rules

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

During development, **patterns** can emerge in the way business rules define relationships between Vocabulary terms. For example, in the sample FlightPlan application, a recurring pattern might be that all aircraft have limits placed on their maximum takeoff weights. You 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 (`200000` or `150000` kilograms) are different in each rule, the basic form of the rule is the same. In fact, you 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:

Aircraft type X	Maximum cargo weight Y
747	200000
DC-10	150000

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, you could build these two rules as a pair of Rulesheets, each with a Filter expression that filters data by `aircraftType`.

Figure 179: Non-parameterized rule
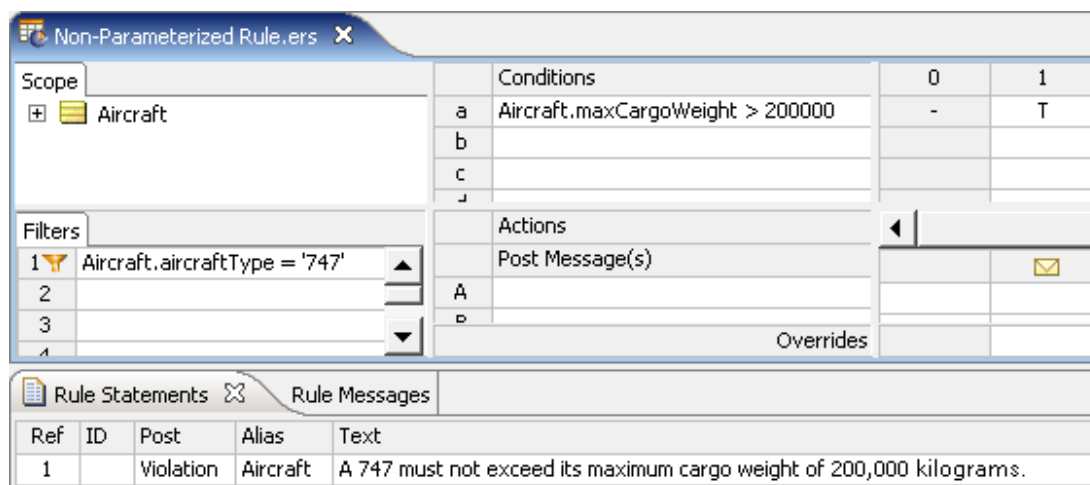
But, there is a simpler and more efficient way of writing these two rules that leverages the concept of parameterization. The following figure illustrates how this is accomplished:

Figure 180: Parameterized rules
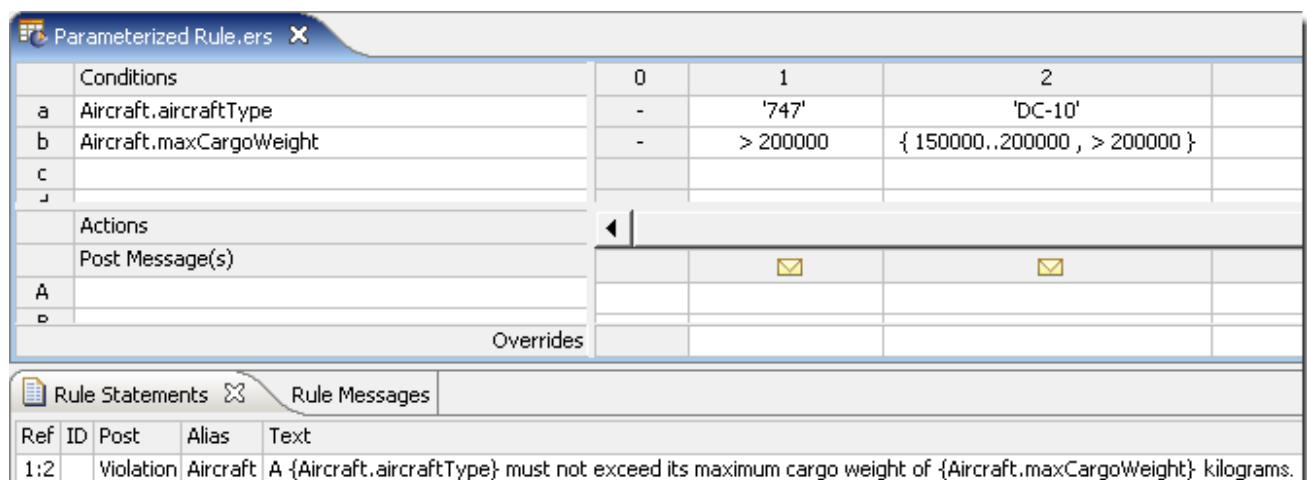
Notice how both rules are modeled on the same Rulesheet. This makes it easier to organize rules that share a common pattern and maintain them over time. If the air cargo company decides to add new aircraft types to its fleet in the future, then the new aircraft types can be added as additional columns.

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 `{..}` braces brackets enclosing the term. See the *Rule Language Guide* for more details about 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 topic 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. You 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 you to define a generic business rule:

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

You can also write the rule as a constraint:

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

Because there is not a method for accommodating many similar rules as a single, generalized case, you 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.

How to populate 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. A user can 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 only as an example of a parameterized rule maintenance application. It is not provided as part of the Corticon Studio installation.

Figure 181: 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 the 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. The example used `issuer.name` and `industry.name`.

4. After 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** removes 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 run time.

    CAUTION!

    Whenever you decide to maintain rule parameters outside of Corticon Studio, you risk introducing ambiguities or conflicts into your Rulesheet. The Conflict Checker may not help you discover these problems because some of the rule data is not shown in Corticon Studio. Always try to design your parameter maintenance forms and interfaces to prevent ambiguities from being introduced.

TestYourself questions for Recognizing and modeling parameterized rules

Note: Try this test, and then go to 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 is 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 $75,000 but less than $100,000 of product each year.

    - Silver customers buy more than $50,000 but less than $75,000 of product each year.

    - Bronze customers buy between $25,000 but $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?

10

How to write rules to access external data

Corticon provides three mechanisms that let you interface your rules with databases and other data sources:

- **Enterprise Data Connector** (EDC) — This technique provides access to a single database from a project Vocabulary. You map your Vocabulary to the database and rely on Corticon to retrieve data when needed. EDC makes data access simple and is a great option when small amounts of data are needed or performance is not paramount. EDC is tightly integrated with rule models, so the functions described in this chapter are how you effectively create queries to the database.

- **Advanced Data Connectors** (ADC) — This technique provides control over the SQL queries used to retrieve or update data in a database. Using ADC requires more database knowledge but provides benefits such as optimized query performance when retrieving large amounts of data, and the ability to map a Vocabulary to multiple data sources. ADC is recommended when performing batch rule processing.

- **REST Datasource** (REST service) — This read-only technique provides secure access to REST services to retrieve data for use in your decision services. Queries—either preset or specified by data in your payload—limit the results brought into the server's memory, which are then filtered to get the data needed to enrich the rule in process.

For additional information, see the *Data Integration topics.*

Overview

Corticon lets you define mappings to a Datasource so that rules can access (query) a database directly, and then retrieve what it needs during execution, thus enriching the information available to the rules, and then writing data to the database when appropriate.

This capability is transparent to the rule modelers so that they are only concerned with getting the rules right, and do not have to get into SQL syntax to interface with an EDC Datasource.

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

While you could start learning how to use any of these Datasources, it is a good idea to start with *"Getting Started with EDC" in the Data Integration Guide.*

For details, see the following topics:

- A scope refresher

- Quick steps for setting up the Cargo sample

- Enable database access for rules using root-level entities

- Precondition and filters as query filters

- Insert new records in a middle table

- Integrate EDC Datasource data into rule output

- TestYourself questions for how to write rules to access external data

A scope refresher

The concept of *scope* is key to rule design and execution. Scope in a Rulesheet 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.

You 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`, and `Aircraft.flightPlan`, or any other role using `FlightPlan` that may exist in our Vocabulary.

When you 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 you write rules using root-level `FlightPlan`, then the Studio processes all `FlightPlans` in the Input Ruletest during rule execution.

Quick steps for setting up the Cargo sample

For the examples that use Cargo, here are the steps to set up the sample data in Corticon Studio and your preferred database:

1. In your database administrative tool, create a database named `Cargo`.

2. In Corticon Studio:

   a. Import the `Cargo` sample.

   b. Create the EDC Datasource, and then define and test its connection to the `Cargo` database.

   c. On each of the three entities, set **Datastore Persistent** to `Yes`, and choose its appropriate **Entity Identity**:

      - **Aircraft**: `tailNumber`

      - **Cargo**: `manifestNumber`

- **FlightPlan**: `flightNumber`

    d. On the **EDC Datasource** tab, click **Create/Update Schema**

3. Copy the contents of the project's `Cargo_data.sql` file to your database administrative tool's editor, and then click **Execute**.

Enable database access for rules using root-level entities

Once interfaced with EDC, the amount of test data is no longer limited to that contained in a single Input Ruletest. It is limited by the sizes in the connected database. Rules using root-level `FlightPlan` (or any other root-level entity) forces 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 you do not accidentally force the Server to perform extremely large and time-consuming data retrievals from the database unless you explicitly require it.

Because database access for rules using root-level terms is disabled by default, you need to know how to enable it for those circumstances when you do want it. This is called *extending* a root-level entity to the database. To illustrate, a simple rule based on the Cargo project's Vocabulary is used, as follows:

1. In Corticon Studio, create a new Rulesheet in the Cargo project, and open its advanced view.

2. Drag from the **Vocabulary** into the **Scope** as shown, including adding `Cargo.weight` to the `FlightPlan` association as shown.

3. Add the aliases in the **Scope** as shown.

4. Write the rule condition and its values in columns 1 and 2.

5. Add the rule statement as shown.

6. Save the Rulesheet as `CargoLoad.ers`.

Figure 182: CargoLoad Rulesheet

The Rulesheet shown adds up (sums) the *collection* (see Collections on page 115) of `Cargo` weights associated with a `FlightPlan` (`load.weight`) and compares this to the `maxCargoWeight` of the root-level `Aircraft`. The intention is to perform this comparison for every available `Aircraft`, so the root-level `Aircraft` in our Conditional expression was used. Any `Aircraft` whose `maxCargoWeight` is inadequate is identified with a posted `Violation` message.

Test 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 that was created against the `Cargo.ecore` named `CargoLoad.ert`. Its input data is as shown:

Figure 183: Sample Input Ruletest
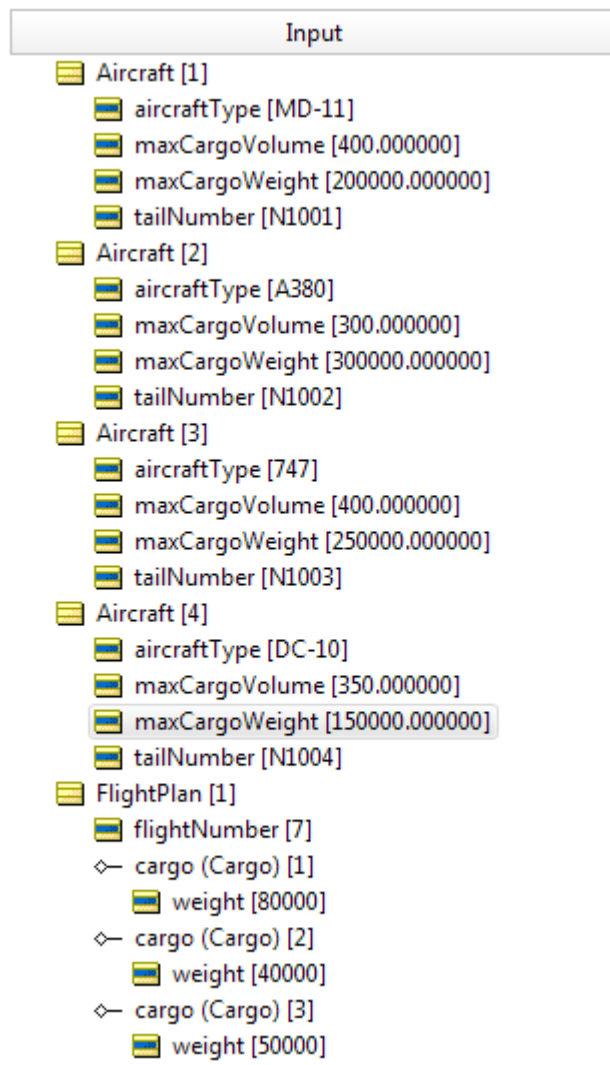
Looking at this Input Ruletest, there is a single `FlightPlan` with its collection of `Cargo`. This collection is what is represented with the alias `load` in our Rulesheet's **Scope** section. Each `Cargo` has a `weight` value entered.

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 is known about rule scope, you 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. Because `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 184: Ruletest Violation Message

Severity	Message	Entity
Violation	The [N1004] must not be assigned to flightplan [7] because the assigned cargo weighs too much.	Aircraft[4]

So far, this behavior is exactly what is expected from rules: they process data of the same scope.

Save the `CargoLoad.ert` Ruletest.

Test the Rulesheet with database access enabled

First, you need to update the database in the EDC tutorial to prepare for the features that will be demonstrated. The Ruletest, `CargoLoad.ert`, has the aircraft data including the primary key. Copy the Ruletest, drop those unwanted inputs, and then update the database column `tailNumber`. That edit actually extends the tutorial's data with one added row that is cargo info we want in this topic.

Note: The procedure for connecting and mapping a Vocabulary to an external database, and setting an Input Ruletest to access that database in **Read OnlyRead/Update** modes is discussed in the topic *"How data from an EDC Datasource integrates into rule output"* section of the *Data Integration Guide.*

To load the aircraft data:

1. In the Project Explorer, copy and paste the `CargoLoad.ert` file. Name the copy `AircraftLoader.ert`.

2. Open `AircraftLoader.ert`.

3. In the Input area, click `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, you need to add some heavy cargo to one of the flight plans. Here are 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 then execute it.

Alternative approach: Using a Ruletest to load a database

You could instead 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 185: Using a Ruletest to add Cargo rows to the connected external database
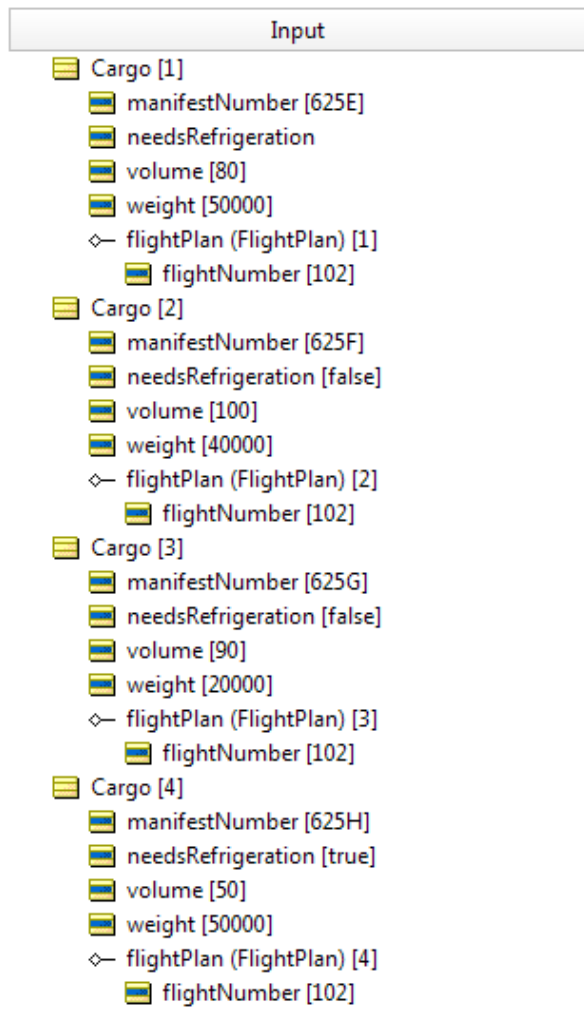
Setting up the test

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

Figure 186: Cargo Table from Database

	manifestNum...	container	needsRefrigera...	volume	weight
▶	625A	NULL	NULL	10	1000
	625B	NULL	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
＊	NULL	NULL	NULL	NULL	NULL

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

Figure 187: Ruletest of FlightPlan seed data

Input	Output
▤ FlightPlan [1]	▤ FlightPlan [1]
▭ flightNumber [102]	▭ 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 no `Aircraft` records were retrieved, therefore the rule's comparison couldn't be made, so the rule couldn't fire. This behavior was expected because that database access for root-level terms is disabled by default.

Now 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	▶
	None
	✔ Read Only
	Read/Update
	✔ Return All Entity Instances
	Return Incoming/New Entity Instances Only
	Enable Cache
	✔ Disable Cache
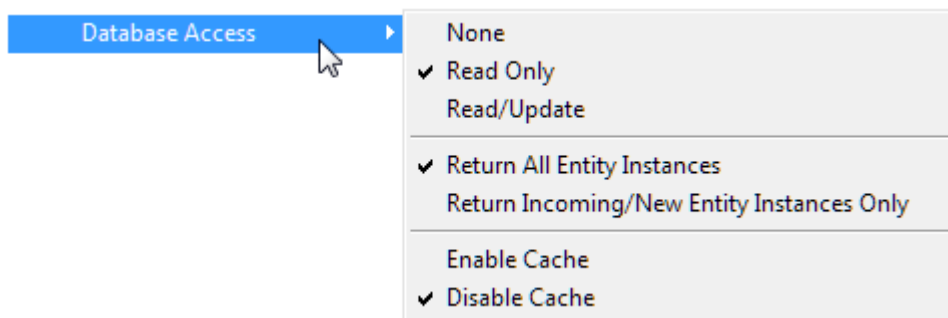
When you run the test again, the output is the same as the input and there are no messages.

Extend to Database

Now you will set the Rulesheet to **Extend to Database**, and then see how it affects the test. On the `CargoLoad.ers` Rulesheet, right-click `Aircraft` in the **Scope** area, and then select **Extend to Database**, as shown:
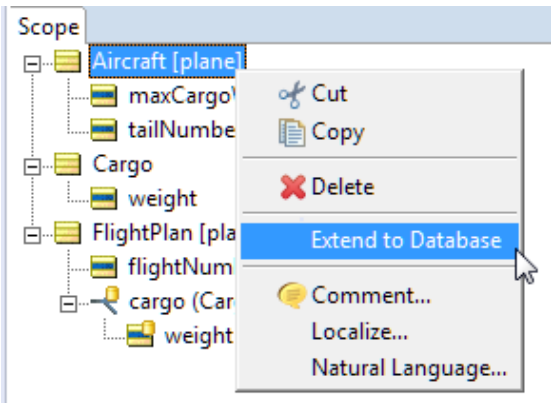
Save your Rulesheet to ensure that the changes take effect. Now, retest the same Input Ruletest shown in . The results are as follows:

Figure 188: Results Ruletest showing a successful Extend-to-Database retrieval
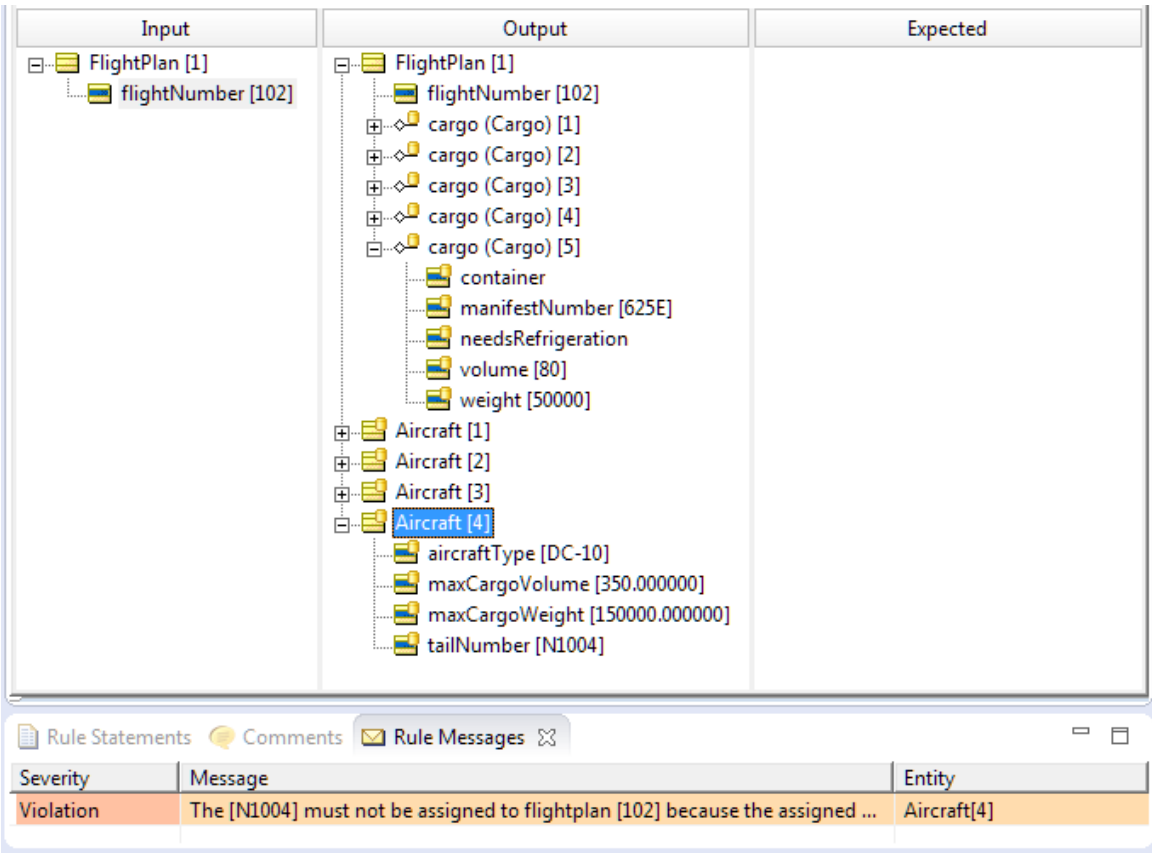
These results are much different! Corticon successfully retrieved all `Aircraft` records, performed the summation of all the cargo in the given flight plan, and identified an `Aircraft` record that fails the test. Given this set of sample data, it is the `Aircraft` with `tailNumber N1004` that receives the `Violation` message.

Returning all instances can be overwhelming

While this rich relational data retrieval is good to see, there are only four planes and five packages in the flight plan. What if there are 1,000 planes and hundreds of thousands of packages every day? That amount of data would be overwhelming. So what you 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:
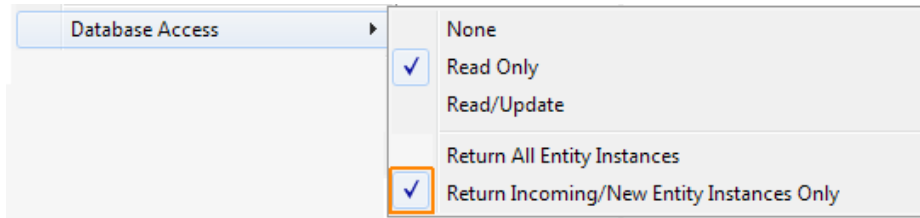
The data that returns is taken only from those entities that were:

- Directly used in the rules.

- Present in the request message.

- Generated by the rules (if any).

Note: This option can be set in a 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. You see a Violation message as to which plane cannot be assigned that flight plan.
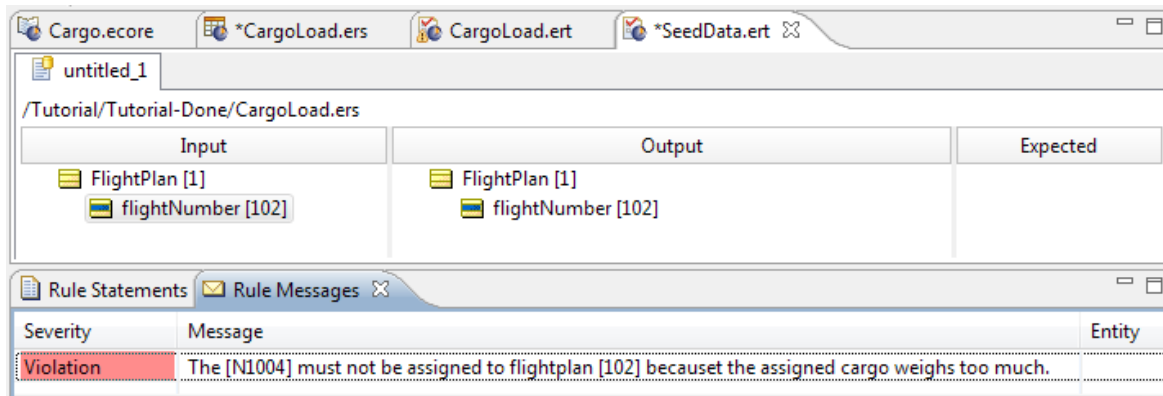
That result is concise, providing the information you wanted from this test.

Optimize aggregations that extend to database

This 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 copies all the relevant cargo records into Corticon's memory to perform its `sum`, and then evaluates whether total weight is greater than the plane's capacity. Because you chose to extend 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, the `load` attribute was created, and then its properties were set so that the data type is `Integer`(the same as the `weight` data it will aggregate), and its mode is to 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:

Progress Corticon: Rule Modeling: Version 6.2

This nonconditional rule 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 EDC database access on page 148 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 you often want or need to qualify those queries to further constrain the data returned to the 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, review the Filters and Preconditions topics 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. After the results set enters 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 you 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 that was extended to the database.

2. If it contains any attributes of Boolean data type. Boolean data types 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 the next topic)

5. If it references more than one alias not extended to the database.

Operators supported in query filters

Query filters are Corticon Rule Language expressions that 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 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 following table:

Table 8: Operators supported by query filters

Operator Name	Operator Syntax	Data types Supported
Add	+	Decimal, Integer
Subtract	–	Decimal, Integer
Multiply	*	Decimal, Integer
Divide	/	Decimal, Integer
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
Character Count	`.size`	String
Convert to Upper Case	`.toUpper`	String
Convert to Lower Case	`.toLower`	String
Substring	`.substring`	String
Equal To (comparison)	`.equals`	String
Collection is Empty	`->isEmpty`	Collection
Collection is not Empty	`->notEmpty`	Collection
Size of Collection	`->size`	Collection
Sum	`->sum`	Collection

Operator Name	Operator Syntax	Data types Supported
Average	`->avg`	Collection
Minimum	`->min`	Collection
Maximum	`->max`	Collection
Exists	`->exists`	-

Note: The Collection operators listed 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.

How to use 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`

The compound filter 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`

Using a compound filter results in the query:

```
Select * from Order o,Item I where o.bid > i.price and o.status = "VALID" and
i.qty > 0
```

Insert new records in a middle table

In relational databases, many-to-many relationships are modeled using a "middle" table (also known as an intersection table). Assume two tables named **A** and **B**, and they have a many-to-many relationship. A third or middle table named **AB** has a many-to-1 relationship with both **A** and **B**.

A many-to-many association between two entities in the Vocabulary can be mapped to such a middle table. Therefore, table **AB** does not need to correspond to a specific entity in the Vocabulary. However, should the middle table contain additional business fields, then it must have a corresponding entity in the Vocabulary. In such a situation, attempting to create a new record/row in table **AB** using rules can cause limitations depending on:

- The cardinalities of the associations between **AB** and **A**, and **AB** and **B**

- The identity strategy used for **A**, **B**, and **AB**

The following table highlights known limitations for combinations of entity identity (Application or Datastore) and association directionality (**bi**directional or **uni**directional):

Application Identity			Datastore Identity (using 'identity' strategy)		
Both Uni	Both Bi	One Uni/One Bi	Both Uni	Both Bi	One Uni/One Bi
OK	NO	NO	OK	OK	OK

The *one uni / one bi* configuration should be avoided when Application Identity is used.

Integrate EDC Datasource data into rule output

EDC introduces a new dimension to rule execution. When EDC is not used, data management during Decision Service execution is relatively straightforward: incoming data contained in the request payload is modified by rules, and the resulting updated state for all objects is returned in the response.

However, when EDC is used, data management becomes more complicated. Data in the database needs to synchronize with the data in the request payload and the data produced by Decision Service execution.

This functionality is discussed in detail in the topic *"How data from an EDC Datasource integrates into rule output" in the Data Integration Guide.*

Using several scenarios, that section describes the algorithms used by Corticon Server to perform this synchronization in a variety of read-only and read-write cases. All scenarios use the familiar `Cargo.ecore`, as set up and verified in Quick steps for setting up the Cargo sample on page 230.

TestYourself questions for how to write rules to access external data

Note: Try this test, and then go to TestYourself answers for Logical analysis and optimization on page 343 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 you know about that entity's database access setting?

4. True or False. Only root-level entities can be extended to a database.

5. Where can I learn more about accessing external data?

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 Rulesheets?

7. Are there any exceptions to the general rule you defined in the preceding question?

11

Logical analysis and optimization

A strength of Corticon's toolset is the ability to perform extensive tests and analysis of your rules using traditional methods as well as within Studio. You can evaluate the completeness of rule coverage, conflicts between rules, and looping in rules. You can even test the subtleties of rule executions with expected results. You are offered techniques to compress and optimize your rules.

For details, see the following topics:

- Test, validate, and optimize your rules

- Traditional methods of analyzing logic

- Validate and test Rulesheets in Corticon Studio

- Test rule scenarios in the Ruletest Expected panel

- How to optimize Rulesheets

- Precise location of problem markers in editors

- TestYourself questions for Logical analysis and optimization

Test, validate, and optimize your rules

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 an *actual* decision operation to an *expected* operation using data scenarios or test cases. The Ruletest provides the capability to build test cases using real data, which can then be submitted as input to a set of rules for evaluation. The actual output produced by the rules is then compared to the expected output from those rules. If the actual output matches the expected output, then you may have *some* degree of confidence that the decision is performing properly. Why only *some* confidence and not *complete* confidence is addressed in this set of topics.

For complete details about settings and analysis for scenario testing, see

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 the 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 189: 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					
	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	

To test these rules, create a new scenario in a Ruletest, as shown:
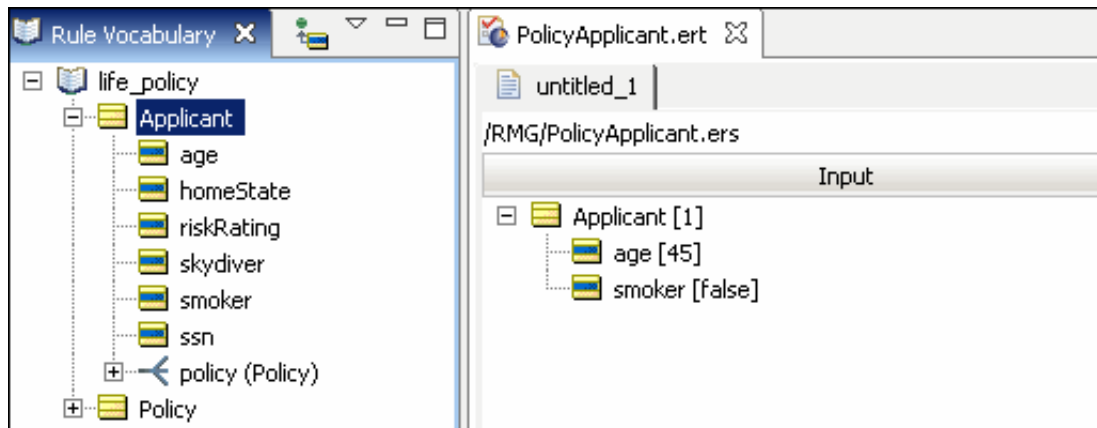
In this scenario, a single example of `Person`, a non-smoker aged 45 is created. Based on the rules just created, the expectation is that the Condition in Rule 1 will be satisfied *(People aged 55 or younger…)* and that the person's `riskRating` will be assigned the value of `low`. To confirm the expectations, run the Ruletest:

Figure 190: Ruletest
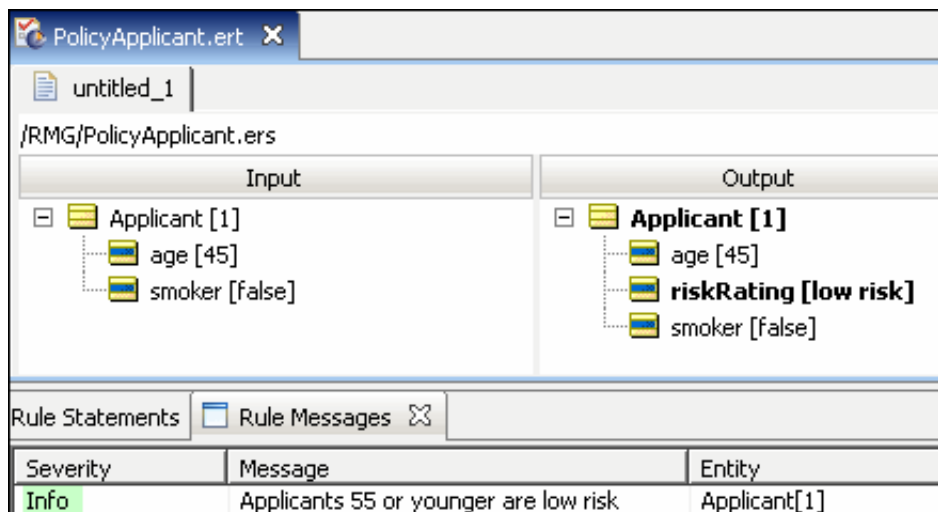
As you can see in the figure, the expectations are confirmed: Rule 1 fires and `riskRating` is assigned the value of `low`. Furthermore, the `.post` command displays the appropriate rule statement. Based on this single scenario, can we say conclusively that these rules will operate properly for other possible scenarios; that is, for all instances of `Person`? How do we answer this critical question?

Traditional methods 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 191: Flowchart with two rules
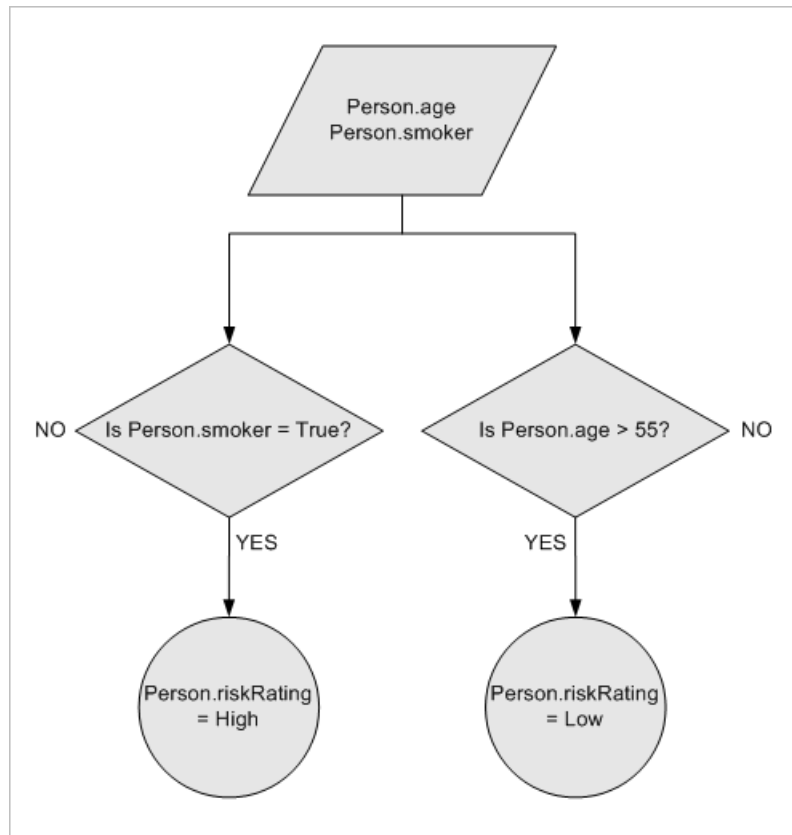
Upon closer examination, the flowchart reveals two problems with our rules: what happens if `Person.age>55` or if `Person.smoker=false`? The rules built in Simple rules for profiling insurance policy applicants do not handle these two cases. But, there is also a third, subtler problem here: what happens if **both** conditions are satisfied, specifically when `Person.age<=55` **and** `Person.smoker=true`? When `Person.age<=55`, `Person.riskRating` should be given the value of `low`. But, when `Person.smoker=true`, `Person.riskRating` should be given the value of `high`.

There is 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 turns out to be an incorrect graphical representation of the rules, because the decision flow does not truly follow two parallel and independent paths. Let's try a different flowchart:

Figure 192: Flowchart with two dependent rules
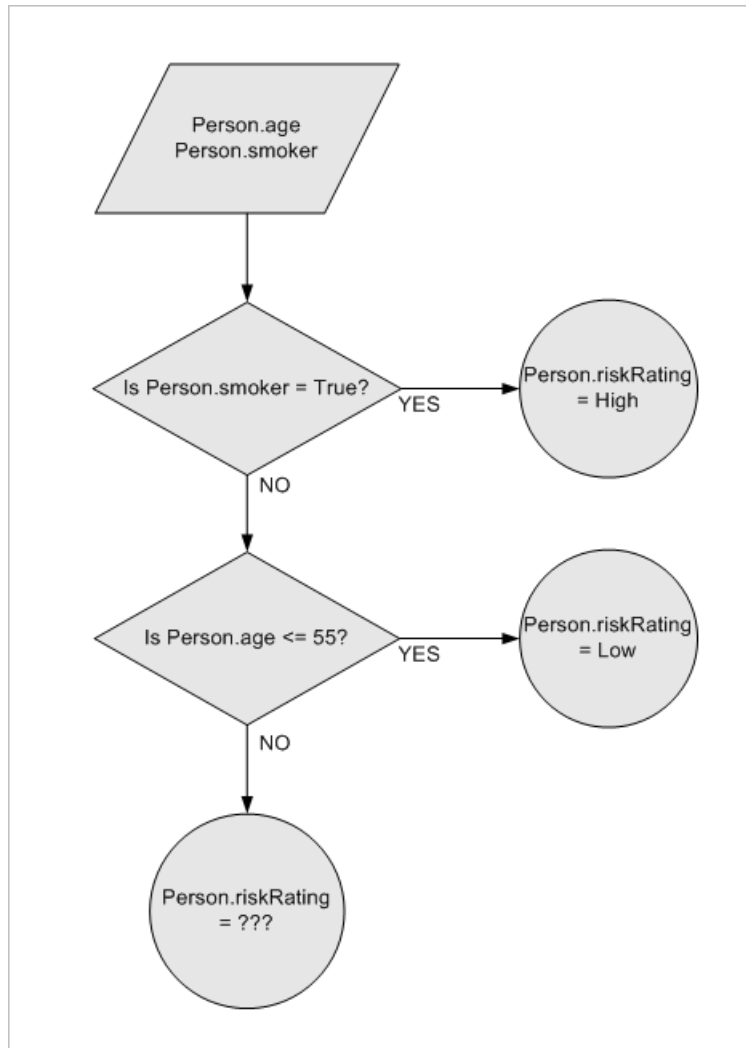
In this flowchart, an interdependence between the two rules was acknowledged, and they were arranged accordingly. However, a few questions still exist. For example, why is the smoker rule *before* the age rule? By doing so the smoker rule has an implicit priority over the age rule because any smoker is immediately 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?

This is a problem of **logical conflict**, or **ambiguity**, because it is not clear from the two rules, as they were 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. This is a problem of **logical completeness** and it must be taken into consideration, no matter which flowchart is used.

The point 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 suites

The use of a test suite is another common method for testing rules (or any kind of business logic). 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 table with only a few test cases, determine the expected outcomes, and then run the tests and compare the results. To ensure that the rules execute properly for all cases that might be encountered in a "real-life" production system, create a set of cases that includes **all** such possibilities.

In a simple example of two rules, this is a relatively straightforward task:

Table 9: All combinations of conditions in table form

Condition	Smoker (smoker = true)	Non-Smoker (smoker = false)
Age <= 55		
Age > 55		

In this table, there is a matrix that uses 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, the Cross Product (also known as the *direct product* or *cross product*) of the two Values sets is created, which means that every member of one set is paired with every member of the other set. Because 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. 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. Recall that rule 1 deals exclusively with the age of the applicant and is not affected 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 193: 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, **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 194: Rule 2 expected outcome

condition	Smoker (smoker = true)	Non-Smoker (smoker = false)
Age <= 55	**low, high**	**low**
Age > 55	**high**	

The table format illustrates that a complete set of test data should contain four distinct cases (each cell corresponds to a case). Rearranging, the test cases and expected results can be summarized as follows:

Figure 195: 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 that were 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 lost in the implementation. As before, you cannot tell by the way the two rules are expressed. Logical conflict reveals itself again.

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 again.

Before you can deal with the logical problems discovered here, let's build a Ruletest in Studio that includes all four test cases in the preceding figure.

Figure 196: Inputs and outputs of the four test cases
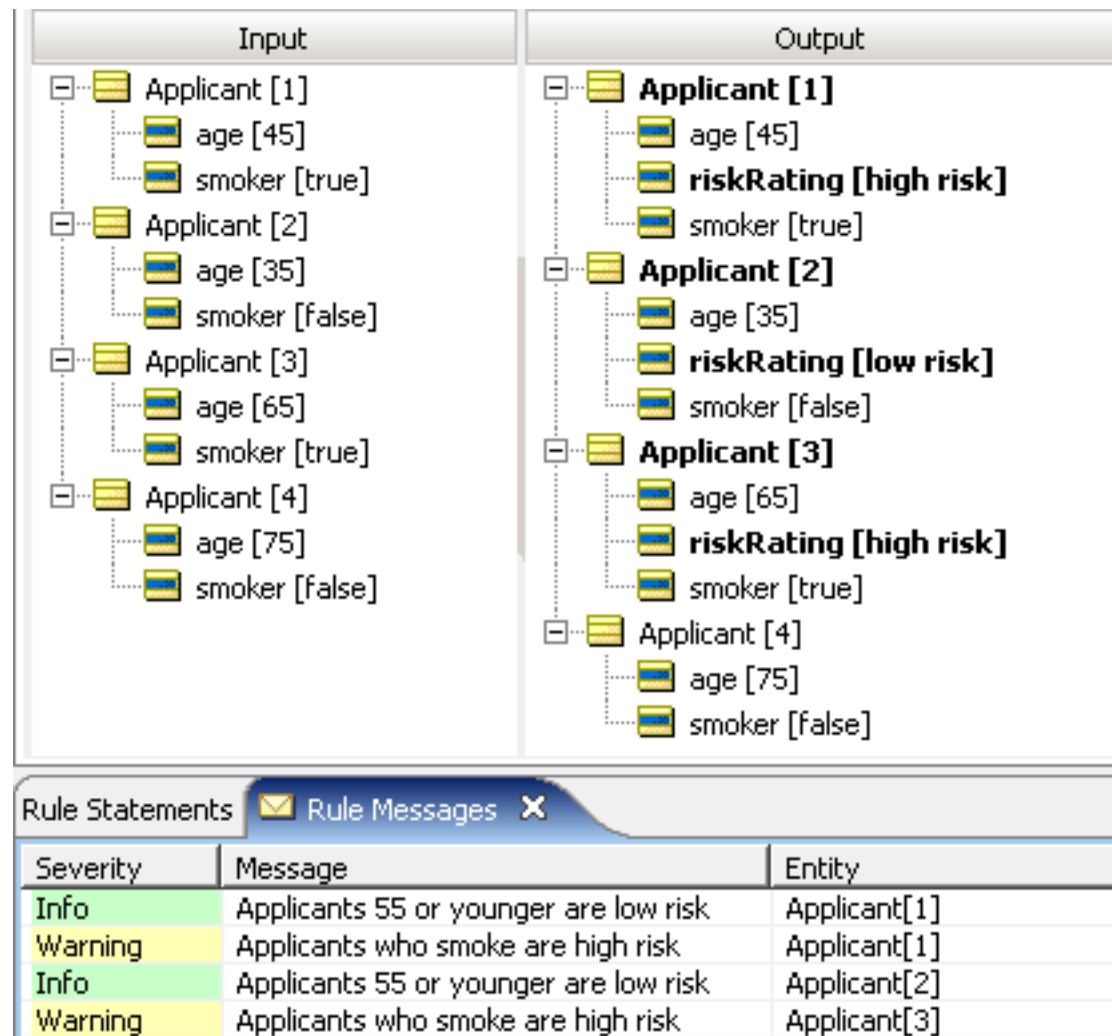
Let's look at the test case results in the figure above. Are they consistent with your expectations? With a minor exception in case #1, the answer is yes. In case #1, `riskRating` was assigned the value of `high`. But, also notice the rule statements posted: case #1 produced two messages which indicate that both the age rule and the smoker rule fired as expected. But, because `riskRating` can hold only one value, the system non-deterministically assigned it the value of `high`.

So, if using test cases works, what is wrong with using it as part of your Analysis methodology? Let's look at the assumptions and simplifications made in the previous example:

1. There are 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 because 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 the 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 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.

Validate and test Rulesheets in Corticon Studio

Now, having demonstrated how to test rules with real cases (as performed in Inputs and outputs of the four 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.

How to expand rules

Look at this table:

Figure 197: Rule 1 expected outcome

condition	Smoker (smoker = true)	Non-Smoker (smoker = false)
Age <= 55	low	low
Age > 55		

Then look at this Rulesheet:

Figure 198: 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				
	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

Rule 1 (the age rule) is a combination of two *subrules*; an age value was specified 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 subrules 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
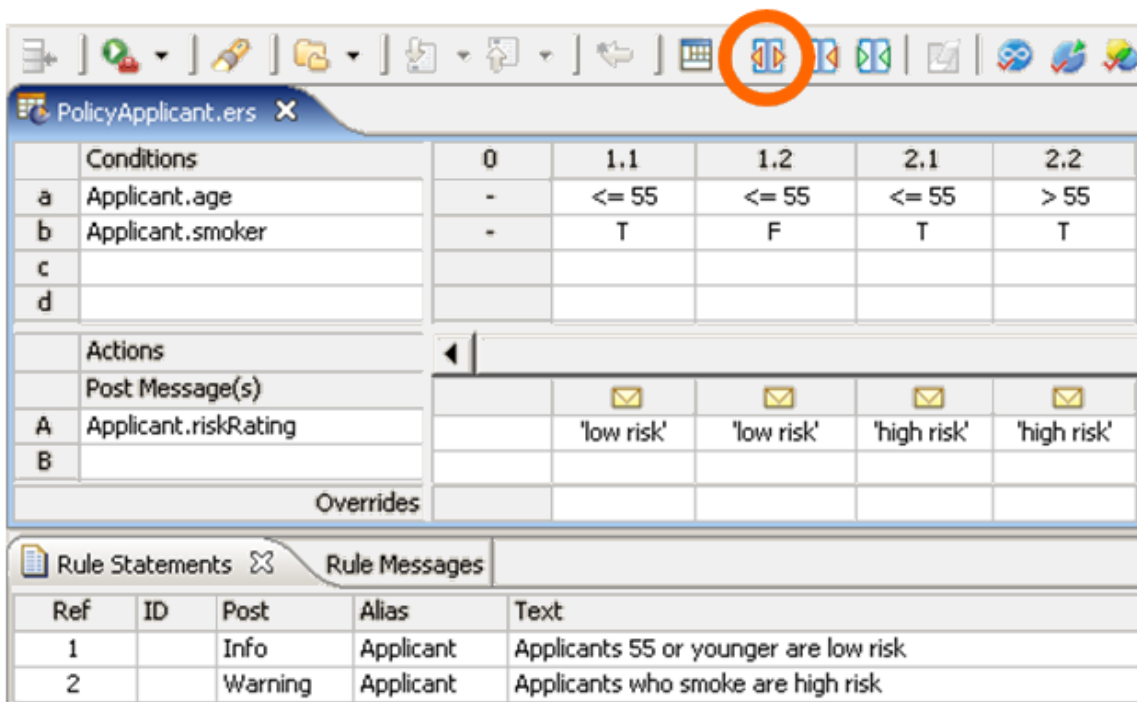
Corticon Studio makes it easy to view subrules for any or all columns in a Rulesheet. By clicking the **Expand Rules** ⏸ button on the toolbar, or double-clicking the column header, Corticon Studio displays subrules for any selected column. If no columns are selected, then all subrules for all columns are shown. Subrules are labeled using decimal numbers: rule 1 below has two subrules labeled 1.1 and 1.2. Subrules 1.1 and 1.2 are equivalent to the upper left and upper right cells in Rule 1 Expected Outcome.

Figure 199: Expanding rules to reveal components

As pointed out before, the outcome is the same for each subrule. Because of this, the subrules can be summarized as the general rules shown in column 1 of Simple Rules for Profiling Insurance Policy Applicants. The two subrules collapse into the rules shown in column 1. The dash character 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 subrules, one for each value in the complete value set determined for that Condition row.

The conflict checker
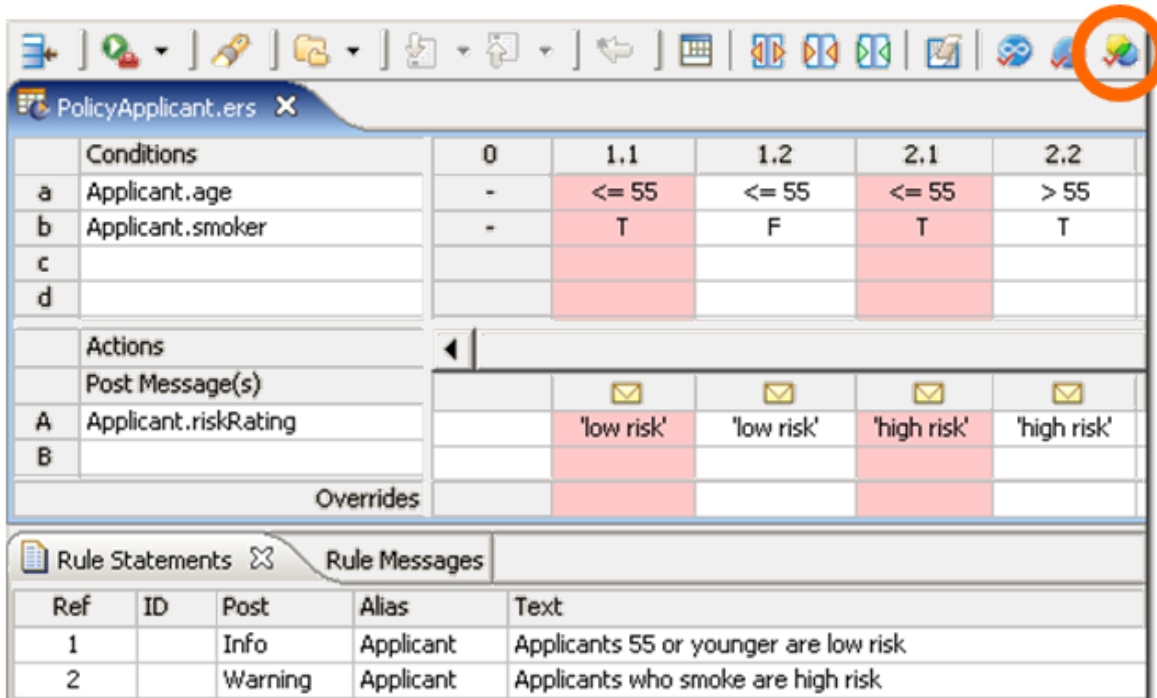
With the two rules expanded into four subrules, most of the Cross Product is displayed. Click the **Check for Conflicts** button in the toolbar.

Figure 200: Conflict revealed by the Conflict Checker

Note: The mechanics of confict checks are described in the *Tutorial: Basic Rule Modeling* topic *"Analyze rules.*

Note: Refresher about 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, the intent is to correlate the results of the automatic conflict check with the problems we identified first with the flowchart method, and then later with test cases. Subrules 1.1 and 2.1, the subrules highlighted in pink and yellow in Figure 200: Conflict revealed by the Conflict Checker on page 255, 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 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 Actions for one or both rules. You could change the Action in subrule 1.1 to match 2.1 or vice versa. Or, you could introduce a new Action, say `riskRating = medium`, as the Action for both 1.1 and 2.1. If either method is used, then the result will be that the Conditions and Actions of subrule 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 How to optimize Rulesheets on page 276 topics.

2. Use an **Override**. Think of an override as an exception. To override one rule with another means to instruct the rules engine 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 two dependent Rules. At the time, it was unclear which decision should execute first. No priority was declared in the rules. But, it made a big difference how our flowchart was constructed and what results it generated. To use an override here, select the number of the subrule *to be overridden* from the drop-down box at the bottom of the column of the *overriding* subrule, as shown circled in the following figure. This is expressed as "subrule 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 and **then** fire rule 1.1." It means "fire rule 2.1 and **do not** fire rule 1.1".

Figure 201: Override entered to resolve conflict
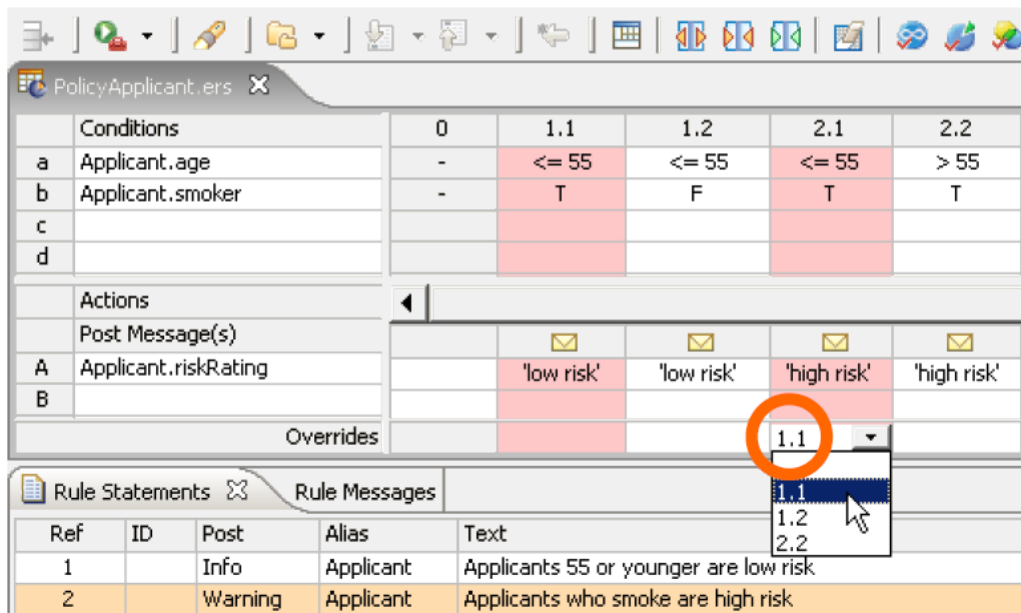
An override is essentially another business rule, which should to be expressed somewhere in the *Rule Statements* section of the Rulesheet. To express this override in plain English, the rule writer might choose to modify the rule statement for the *overridden* rule:

This modification successfully expresses the effect of the override.

If you are ever in doubt as to whether you have successfully resolved a conflict, click the **Check for Conflicts** button again. The affected subrules should not highlight as you step through any remaining ambiguities. If all ambiguities were resolved, then you see the following window:

Figure 202: 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 recorded; if so, then the rule is overridden, and processing of the rule with those entities stopped. If the override context is empty, then any execution of the overriding rule will stop all executions of the overridden rule.

Use 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. 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	◄	▥	
	Post Message(s)			
A	Cargo.volume		200	150
B				
	Overrides			

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

Input	Output
▲ ▦ Cargo [1]	▲ ▦ **Cargo [1]**
▬ container	▬ container
▬ volume [100]	▬ **volume [150]**
▬ weight	▬ weight
▲ ▦ Cargo [2]	▲ ▦ **Cargo [2]**
▬ container	▬ container
▬ volume [200]	▬ **volume [150]**
▬ weight	▬ weight

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	◄	▥	
	Post Message(s)			
A	Cargo.volume		200	150
B				
	Overrides		2 ▼	

... the rules produce the preferred output:

Input	Output
▲ ▦ Cargo [1]	▲ ▦ **Cargo [1]**
▬ container	▬ container
▬ volume [100]	▬ **volume [200]**
▬ weight	▬ weight
▲ ▦ Cargo [2]	▲ ▦ **Cargo [2]**
▬ container	▬ container
▬ volume [200]	▬ **volume [150]**
▬ weight	▬ weight

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

The completeness checker

When rules are expanded, check for completeness by correlating results with the previous manual methods of logical analysis.

Note: The mechanics of completeness checks are described in the *Tutorial: Basic Rule Modeling* topic *"Analyze rules.*

Clicking the **Check for Completeness** button, the message window is displayed:

Figure 203: Completeness Check message window

Corticon Studio

i Completeness check has added 1 missing scenarios which have been automatically compressed in to 1 non-overlapping columns.

OK
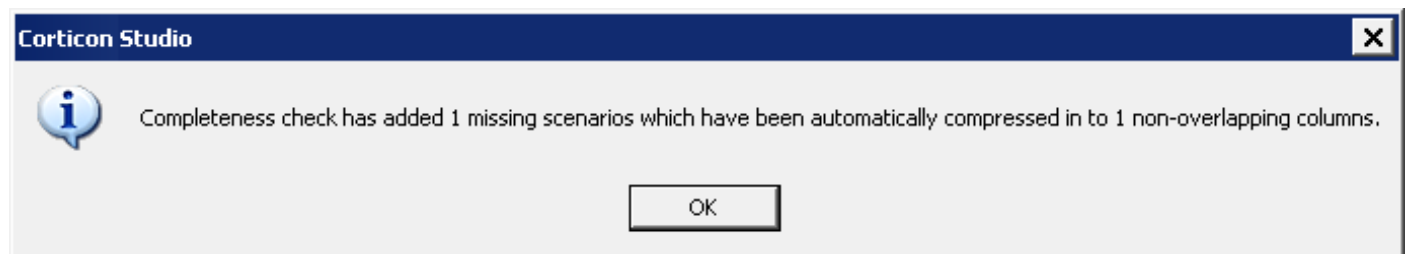
After clicking **OK** to dismiss the message window, notice that the Completeness Check produced a new column (3), shaded in green:

Figure 204: New rule added by completeness check

PolicyApplicant.ers			0	1.1	1.2	2.1	2.2	
	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				1.1		

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

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 the missing rule! To do this, the Completeness Checker employs an algorithm that 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, then 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 determine 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 lists you select from will grow and change.

In the drop-down list, you will see the list of values you 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 but only shows you the elements that you manually inserted.

This automatically generated complete value list feeds 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.

Automatically compress the new columns

An important aspect of the Completeness Checker's operation is the automatic compression it performs on the resulting set of missing Conditions. As you can 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 Compress Tool introduced in *"How to optimize Rulesheets" in the Corticon.js Rule Modeling Guide*. The optimized columns produced by the Completeness Check contain *no redundant subrules* (that is what non-overlapping means), whereas the Compression Tool intentionally injects redundant subrules 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 subrules being assigned different Actions).

Handle 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 205: Example prior to completeness check

Now, click on the Completeness Checker:

Figure 206: Example after completeness check

Notice that columns 3-4 were 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, it contains component or subrules. By double-clicking column 3's header, its subrules are revealed:

Figure 207: Non-female subrules 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 (that is, male) patients. And if male patients are evaluated, then the scenarios described by subrules 3.1 and 3.3—those scenarios containing pregnant males—are unnecessary. While these combinations may be members of the Cross Product, they are not combinations that can occur in real life. If other rules in an application prevent combinations like this from occurring, then subrules 3.1 and 3.3 may also be unnecessary. On the other hand, if no other rules catch this faulty combination earlier, then you 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.

Renumber rules

Assume that subrules 3.1 and 3.3 are impossible, and so they can be ignored. However, if you decide to keep subrules 3.2 and 3.4 and assign Actions to them. For this example, violation messages will be posted.

However, when you try to enter Rule Statements for subrules 3.2 and 3.4, you will discover that Rule Statements can only be entered for general rules (whole-numbered columns), not subrules. To convert column 3, with its four subrules, into four whole-numbered general rules, select **Rulesheet >Rule Column(s)>Renumber Rules** from the **Studio** menubar.

Figure 208: 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								

Rule Statements ⊠ / Rule Messages

Ref	ID	Post	Alias	Text
1		Info	Patient	Pregnant patients age 40 and younger are assigned a risk factor of normal risk
2		Warning	Patient	Pregnant patients older than 40 are assigned a risk factor of elevated risk

Now that the columns are renumbered, Rule Statements can 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 209: 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								

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
{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

Let the expansion tool work for you with 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, a simple Vocabulary with which to implement these rules was built. Because each cell in the table represents a single rule, the Rulesheet contains 25 columns (the Cross Product equals 5x5 or 25).

Figure 210: 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 211: Rulesheet with Conditions automatically populated

Now, use the Expansion Tool to expand column 1 into 25 non-overlapping columns. You now see the 25 subrules of column 1 (only the first ten sub-rules are shown in the following figure due to page width limitations in this document):

Figure 212: Rule 1 expanded to show sub-rules

Each subrule represents a single cell in the original table. Now, select the appropriate value of `shipCharge` in the **Actions** section of each subrule as shown:

Figure 213: 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, select **Rulesheet >Rule Column(s)>Renumber Rules** to *renumber* the subrules to arrive at the final Rulesheet with 25 general rules, each of which can now be assigned a Rule Statement.

Figure 214: 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												

For more about this example, see the section *"How to optimize Rulesheets"*.

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 *"Increase Corticon Studio 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.

Test 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 they reconcile 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 in either of two ways:

1. Create expected data from test output:

    a. Create or import a request into a Ruletest.

    b. Run the test against an appropriate Rulesheet or Ruleflow.

    c. Choose the menu command **Ruletest** > **Testsheet** > **Data** > **Output** > **Copy to Expected**, or click in the Corticon Studio toolbar.

2. Create expected data directly from the Vocabulary:

    a. 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.

    b. Enter expected values for the **Input** attributes as well as the attributes that will be added in the **Output** panel.

Note: See the topics in

How to navigate 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. After you set the panels 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 identified six differences:

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

Review test results when using the Expected panel

The following topics present a variety of test results.

Output results match expected exactly

In the following example, 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.

Different values output than expected

In the following example, one difference was identified. The expected value of `Cargo[2]` packaging value is `standard`, but the Ruletest produced an actual value of `oversize`. Because the **Output** does not match the **Expected** data, the text is colored red.

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.

Fewer values output than expected

In the following example, `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.

More values output than expected

In the following example, `Cargo[3]` was added in the **Intput**, 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.

All Expected panel 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.

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:

Set 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`, then 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:

When the test runs, it is valid.

But, when the input gets a value and the output still has no value (or a different value), the test fails.

Clicking the expected attribute, you can choose **Ignore Validation**.
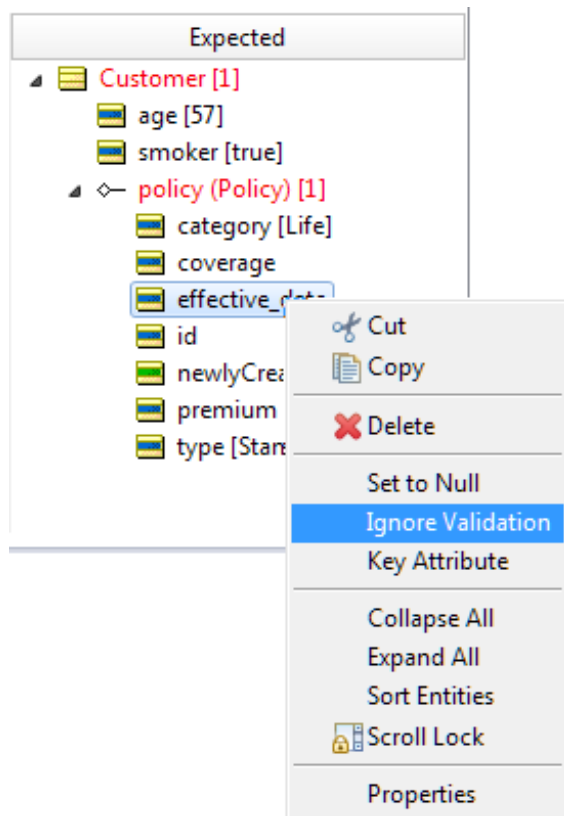
An attribute that will be ignored is greyed out.

Running the same test, the test passes.

Input	Output	Expected
▲ 🗒 Customer [1]	▲ 🗒 **Customer [1]**	▲ 🗒 Customer [1]
🟦 age [57]	🟦 age [57]	🟦 age [57]
🟦 smoker [true]	🟦 smoker [true]	🟦 smoker [true]
▲ ◇— policy (Policy) [1]	▲ ◇— **policy (Policy) [1]**	▲ ◇— policy (Policy) [1]
🟦 category [Life]	🟦 category [Life]	🟦 category [Life]
🟦 coverage	🟦 coverage	🟦 coverage
🟦 effective_date [7/4/2014]	🟦 effective_date [7/4/2014]	🟦 *effective_date*
🟦 id	🟦 id	🟦 id
🟩 newlyCreated [true]	🟩 newlyCreated [true]	🟩 newlyCreated [true]
🟦 premium [0]	🟦 **premium [2220.000000]**	🟦 premium [2220.000000]
🟦 type [Standard]	🟦 type [Standard]	🟦 type [Standard]

The setting can revert by selecting the attribute and then choosing **Enable Validation**.
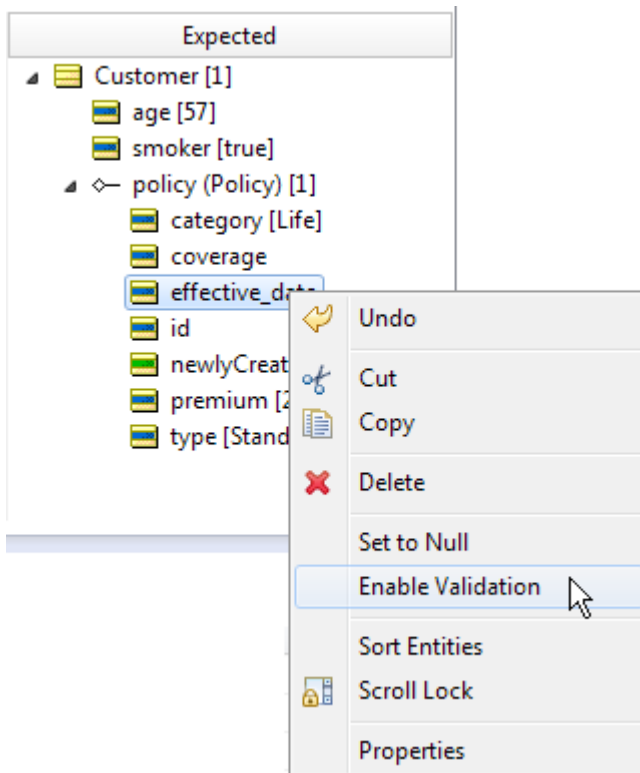
Use key attributes to improve difference detection in Ruletests

The execution of Ruletests can, in some cases, erroneously detect differences between the Output and Expected results. This typically occurs in Rulesheets that 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 the attribute in the Expected panel, and then choose **Key Attribute**, as shown:
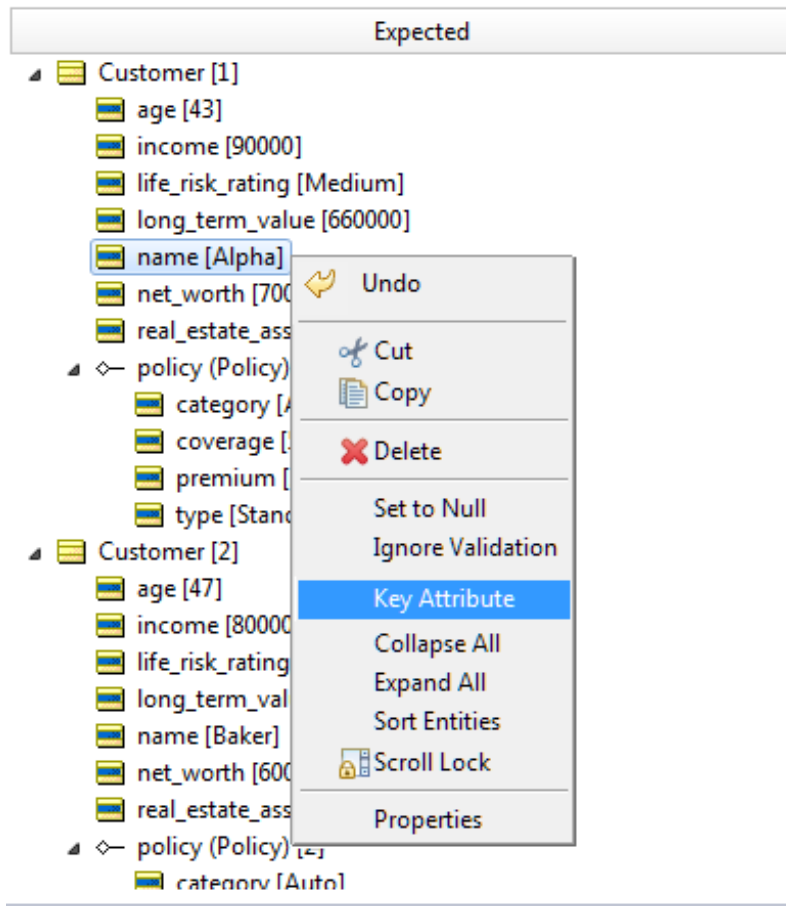
Key attributes are shown in italic in the current entity as well as all other corresponding entities in the **Expected** panel, as shown:
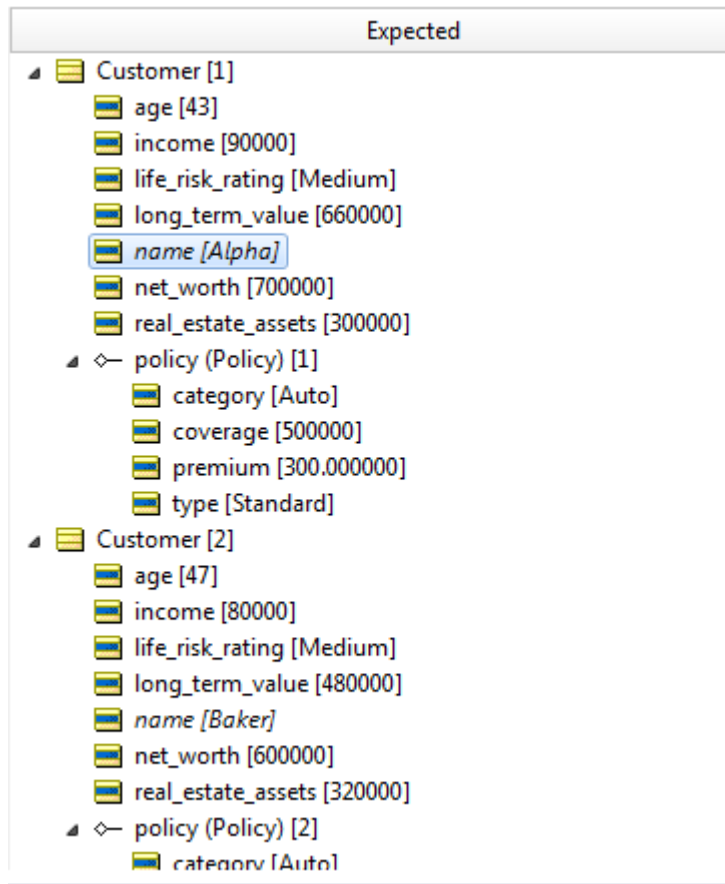
To remove a key attribute, right-click on the attribute again in the **Expected** panel, and then choose **Key Attribute** to clear the setting.

Setting multiple key attributes attempts to match the full set.

Set 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, because that action strips whitespaces, and often reveals apparent mismatches immediately.

Numerical equivalence

When comparing expected results with output results during the validation stage of testing, two values that have a different number of trailing zeros to the right of the decimal place will validate correctly. However, you should avoid introducing rounding errors and inconsistent use of big decimal data types because they can produce unintended differences during comparisons.

How to optimize Rulesheets

The tools that evaluate completeness and that perform compression can be reviewed to ensure that the decision service executes them efficiently .

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 subrule duplication**. The Compress Tool searches a Rulesheet for duplicate columns (including subrules that may not be visible unless the rule columns are expanded), and deletes 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 215: New Rule (#4) added

	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 four rules use only two Conditions and take just two 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 216: Redundancy problem exposed

	Conditions	0	1.1	1.2	2.1	2.2	3	4
a	Applicant.age	-	<= 55	<= 55	<= 55	> 55	> 55	<= 55
b	Applicant.smoker	-	T	F	T	T	F	F
c								
d								
	Actions							
	Post Message(s)		✉	✉	✉	✉		✉
A	Applicant.riskRating		'low risk'	'low risk'	'high risk'	'high risk'	'medium risk'	'low risk'
B								
	Overrides				1.1			

Rule Statements ⬛ **Rule Messages**

Ref	ID	Post	Alias	Text
1		Info	Applicant	Applicants 55 or younger are low risk
2		Warning	Applicant	Applicants who smoke are high risk
3		Info	Applicant	Applicants 55 or older who do not smoke are medium risk
4		Info	Applicant	Applicants 55 or younger who do not smoke are low risk

Clicking the **Compress Tool** ⬛ has the effect shown in the following figure:

Figure 217: Rulesheet after compression

	Conditions	0	1	2	3
a	Applicant.age	-	<= 55	-	> 55
b	Applicant.smoker	-	-	T	F
c					
d					
	Actions				
	Post Message(s)		✉		✉
A	Applicant.riskRating		'low risk'	'high risk'	'medium risk'
B					
	Overrides				1.1

Rule Statements ⬛ **Rule Messages**

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, notice that column #4 disappeared. More accurately, the Compress Tool determined that column 4 was a duplicate of one of the subrules in column 1 (1.2) and removed it.

Compression does not, however, alter the *text* of the rule statement; that task is left to the rule writer.

It is important to note that the compression does not alter the Rulesheet's logic; it simply affects the way the rules **appear** in the Rulesheet: the number of columns, Values sets in the columns, and such. Compression also streamlines rule execution by ensuring that no rules are processed more than necessary.

2. **Combining Values sets to simplify and shorten Rulesheets**. In the Shipping charge example, the Compress Tool combined Rulesheet columns wherever possible by creating Values sets in Condition cells. For example, rule 6 in the figure **Compressed Shipping Charge Rulesheet**is the combination of rule 6 and 8 from Rulesheet with Renumbered Rules.

Figure 218: Compressed shipping charge Rulesheet

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 decision service will execute this new rule 6 faster than the previous rule 6 and 8 together.

How to produce characteristic Rulesheet patterns

Because Corticon Studio is a visual environment, patterns often appear in the Rulesheet that provide insight into the decision logic. After rule writers recognize and understand what these patterns mean, they 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 modeling techniques, you might implement rule 1 as:

Figure 219: Implementing the 747 rule

	Conditions	0	1	2
a	Aircraft.maxCargoVolume		> 300	
b	Aircraft.maxCargoWeight		> 200000	
c	Aircraft.tailNumber		'N123UA'	
d				
e				
f				
	Actions	‹		
	Post Message(s)		✉	
K	Aircraft.aircraftType		'747'	
L				
M				

Now let's have the Completeness Checker populate any missing columns:

Figure 220: Remaining columns produced by the Completeness Checker

	Conditions	2	3	4	5
a	Aircraft.maxCargoVolume	> 300	> 300	{<= 300, null}	
b	Aircraft.maxCargoWeight	> 200000	{<= 200000, null}	-	
c	Aircraft.tailNumber	not 'N123UA'	-	-	
d					
	Actions	‹			
	Post Message(s)				
K	Aircraft.aircraftType				
L					

✉ Rule Messages

	Text
t	An aircraft with a maximur
t	A Boeing 747 can transpor

Progress Corticon Studio ✕

ⓘ Completeness check has added 17 missing scenarios which have been automatically compressed in to 3 non-overlapping columns.

OK

Click **Expand** to fill out the Rulesheet so you can examine the 17 cross-product subrules:

Figure 221: Underlying subrules produced by the Completeness Checker

0	1	2	3.1	3.2	3.3	3.4	4.1	4.2	4.3
	> 300	> 300	> 300	> 300	> 300	> 300	<= 300	<= 300	<= 300
	> 200000	> 200000	<= 200000	<= 200000	null	null	<= 200000	<= 200000	> 200000
	'N123UA'	not 'N123UA'	'N123UA'	not 'N123UA'	'N123UA'	not 'N123UA'	'N123UA'	not 'N123UA'	'N123UA'

The 17 new columns (counting both rules and subrules) include an optimization that combined `<>` `'N312UA'` and `null` into `not` `'N312UA'`. So, the number of combinations is 3*3*2 = 18. Subtracting the rule in column 1, 17 new columns were added.

Now, click **Compress** ⧓.

There are now just 4 rules. Fill in the Actions for the new columns, DC-10, as shown:

Figure 222: Missing Rules with Actions assigned

	Conditions	0	1	2	3	4
a	Aircraft.maxCargoVolume		> 300	-	-	{<= 300, null}
b	Aircraft.maxCargoWeight		> 200000	-	{<= 200000, null}	-
c	Aircraft.tailNumber		'N123UA'	not 'N123UA'	-	-
d						
e						
f						

	Actions	<				
	Post Message(s)		✉			
K	Aircraft.aircraftType		'747'	'DC-10'	'DC-10'	'DC-10'
L						
M						
N						
O						
	Overrides					

Because the added rules are non-overlapping, you can be sure they won't introduce any ambiguities into the Rulesheet.

To be sure, click the **Conflict Checker** 🌐.

Figure 223: Proof that no new conflicts were introduced by the Completeness Check

	Conditions	0	1	2	3	4
a	Aircraft.maxCargoVolume		> 300	-	-	{<= 300, null}
b	Aircraft.maxCargoWeight		> 200000	-	{<= 200000, null}	-
c	Aircraft.tailNumber		'N123UA'	not 'N123UA'	-	-
d						
e						
f						

Progress Corticon Studio ✕

ⓘ No conflicts detected

OK

	Actions	<				
	Post Message(s)		✉			
K	Aircraft.aircraftType		'747'			
L						
M						
N						
O						
	Overrides					
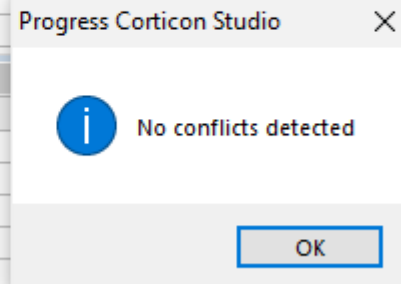
This pattern tells you that the only case where the aircraft type is a 747 is when max cargo volume is greater than 300 **AND** max cargo weight is greater than 200,000 **AND** tail number is N123UA. This rule is expressed in column 1. In all other cases, specifically where max cargo volume is 300 or 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.

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 subrule redundancy

Compressing the example in the preceding topic into a recognizable pattern, however, has an interesting side effect: it introduced more subrules than were initially present. To see this, click **Expand** to compress the Rulesheet as shown:

Figure 224: Expanding Rules following compression

0	1	2.1			3.1			4.1
-	> 300	<= 300			<= 300			<= 300
-	> 200000	<= 200000			<= 200000			<= 200000
-	'N123UA'	<> 'N123UA'			<> 'N123UA'			<> 'N123UA'
	✉							
	'747'	'DC-10'			'DC-10'			'DC-10'

You may be surprised to see a total of 54 subrules (columns) displayed (in the preceding figure) instead of the 26 prior to compression. Look closely at the 54 columns, and you will see several instances of subrule redundancy. Of the 18 sub-rules within the original columns 2, 3 and 4, almost half are redundant (for example, subrules 2.1, 3.1 and 4.1, shown in the preceding figure, 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.

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 subrules 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 Corticon rules engine. Smaller Rulesheets (lower column count) result in faster 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 the rules engine, only the max cargo weight Condition is considered. The other two Conditions are ignored 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) executes 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 each annotated error line in the **Problems** view to open the corresponding file in its editor, and then bring the specific location into view and give it 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:
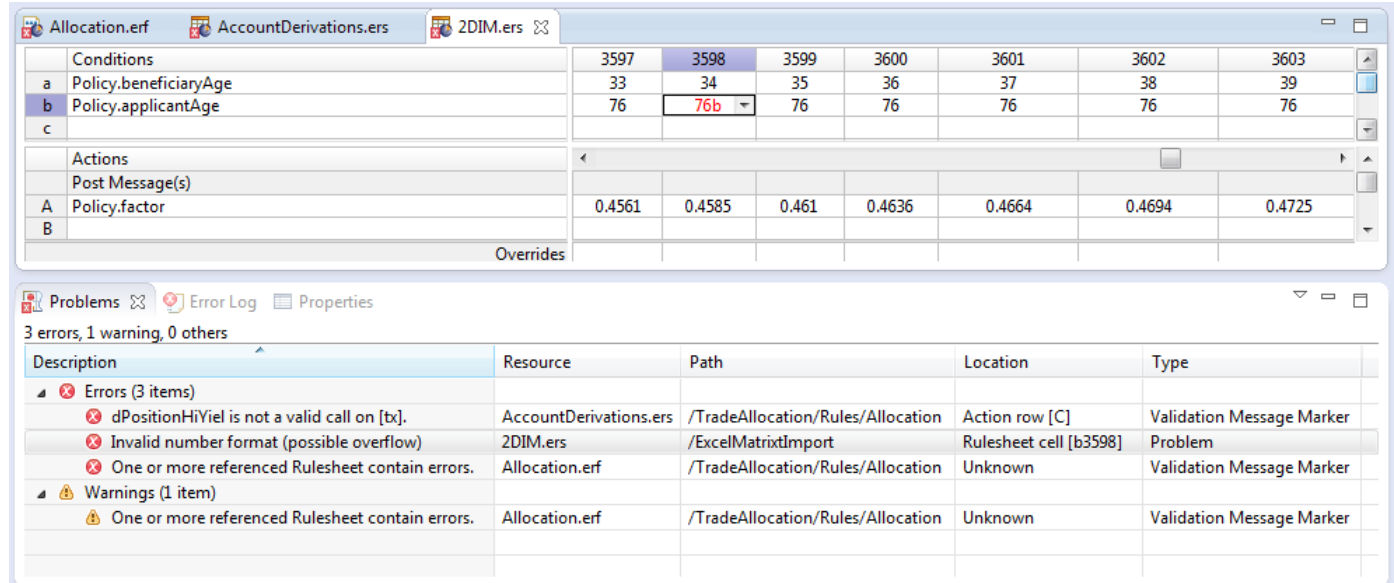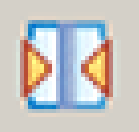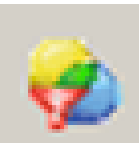
This functionality applies to Vocabularies, Rulesheets, Ruleflows, and Ruletests.

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.

TestYourself questions for Logical analysis and optimization

Note: Try this test, and then go to 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 (that is, for every column)?

18. If you do not want to assign actions for every column, what can be done to or 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 is 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 subrules, what will Corticon Studio force you to do before saving the changes?

23. What does it mean for one rule to subsume another?

12

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 compiled into Decision Services.

For details, see the following topics:

- How to use a Ruleflow in another Ruleflow

- Conditional branching in Ruleflows

- How to generate Ruleflow dependency graphs

- Ruleflow versions and effective dates

- TestYourself questions for Ruleflow versions and effective dates

How to use 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:
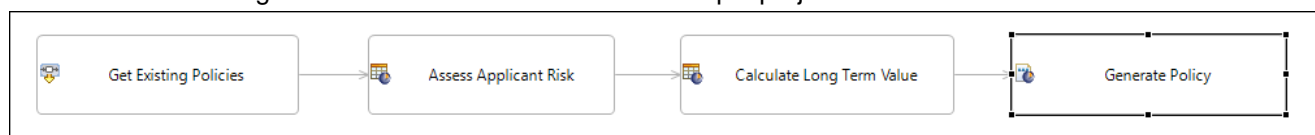
The Ruleflow editor shows the `iSample_policy_pricing.erf` canvas with four objects in sequence. The first three apply the risk assessment rules and the other object is altogether separate Ruleflow file, as you can see in the object's properties:
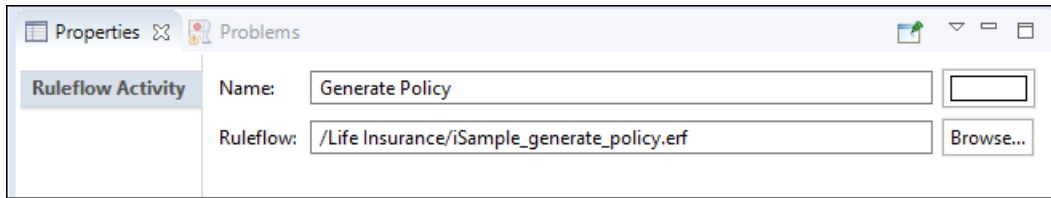
Figure 225: A Ruleflow overrides settings on an embedded Ruleflow

A Ruleflow file's **Properties** provide settings for versioning and effective date stamping of the Decision Service that will be created. (See the topic Ruleflow versions and effective dates on page 309 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.

You can change the name of the Ruleflow on the canvas 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.

The referenced Ruleflow, `iSample_generate_policy.erf`, contains four Rulesheets, as shown:
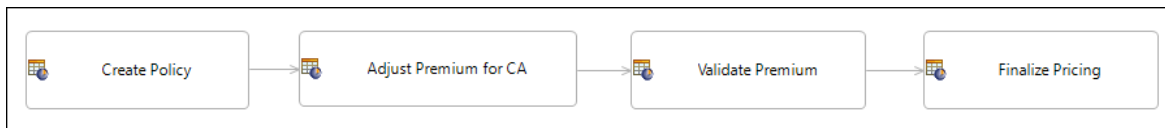
With these two Ruleflows, each can be updated and tested independently, and -- as long as you ensure that the Vocabulary stays consistent -- separate teams can collaborate on developing risk rules and policy rules. That makes it easy to *reuse* either of these Ruleflows. For example, if policy pricing varies in different markets, then you can create a new Ruleflow that brings in the same risk assessment rules to provide the data to process against a modified policy pricing Ruleflow for the other market.

The parent Ruleflow provides its own settings for versioning and effective date stamping of the Decision Service that will be created, as illustrated:
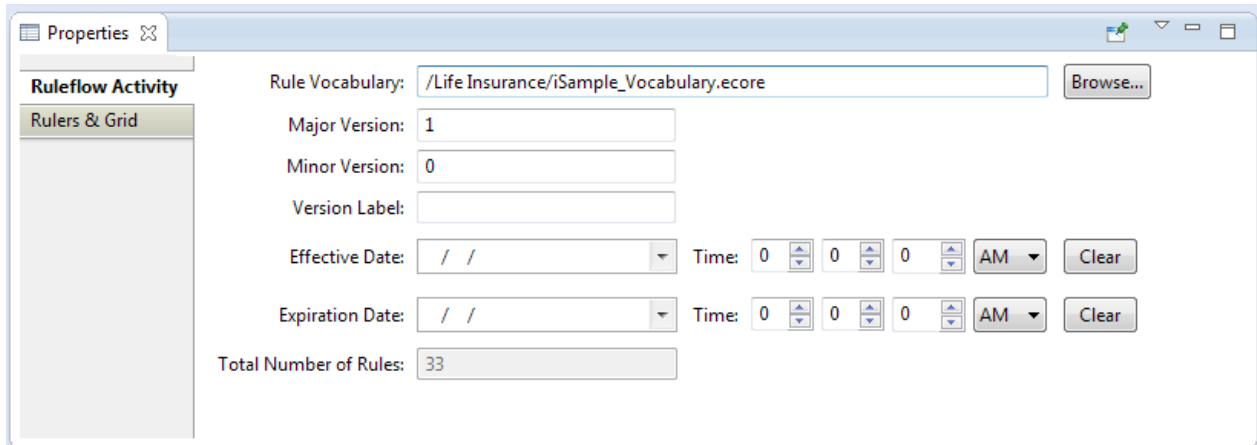
Note: Deploying Ruleflows within a Ruleflow: When this Ruleflow is deployed, the generated Decision Service includes the content of both Ruleflows. However, when either of the included Ruleflows changes, Ruleflows that include one of them are not automatically updated: each must be redeployed to include 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 that 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 represent a progression because the applicant's state is going to trigger only one of these Rulesheets to fire its rules:
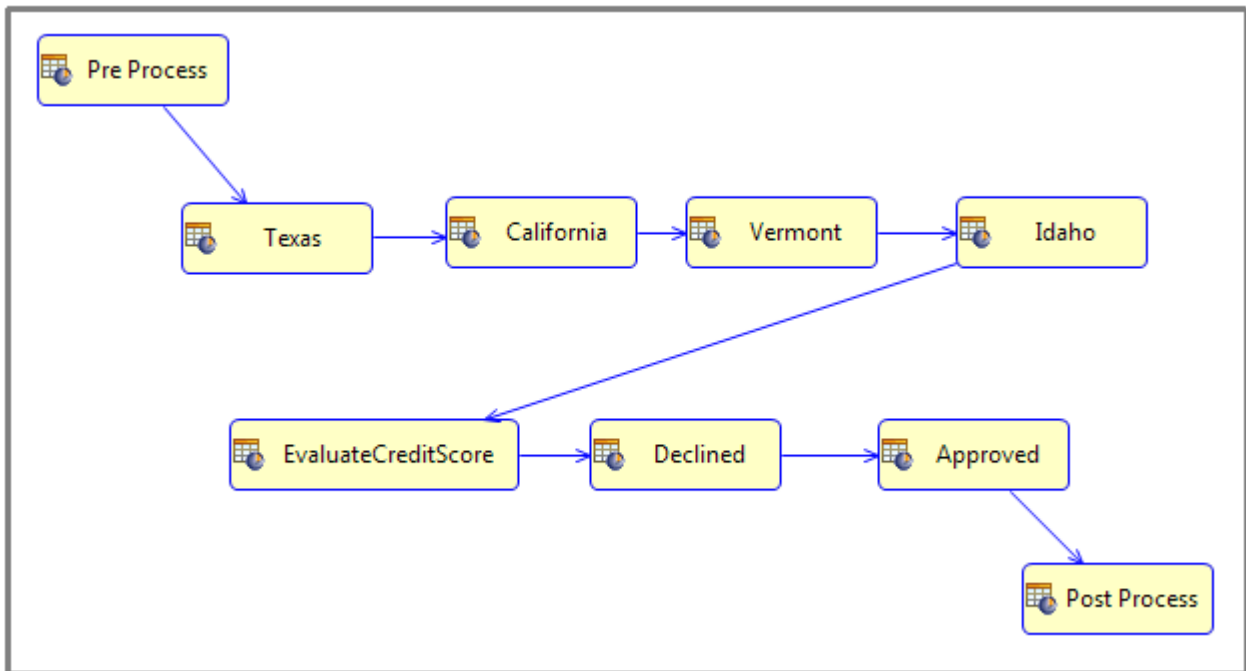
Looking at this Ruleflow, the real flow is somewhat hidden. If the Rulesheets for Texas, California, Vermont, and Idaho each had a precondition such that only matching states were processed, then they represent a set of mutually exclusive options, not the linear flow depicted in the Ruleflow. You will see how to create a branch in a Ruleflow like this:
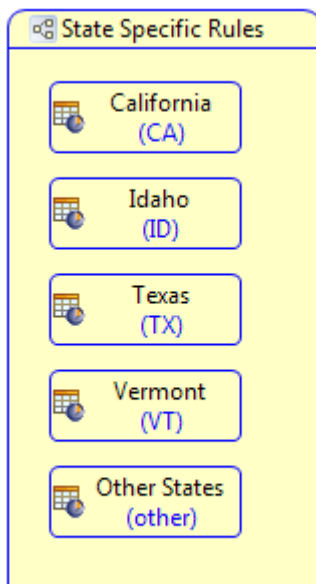
And then bring that Ruleflow into another Ruleflow where you will also create a branch for the Declined and Approved Rulesheets that also might have needed to use preconditions. The completed Ruleflow looks like this:
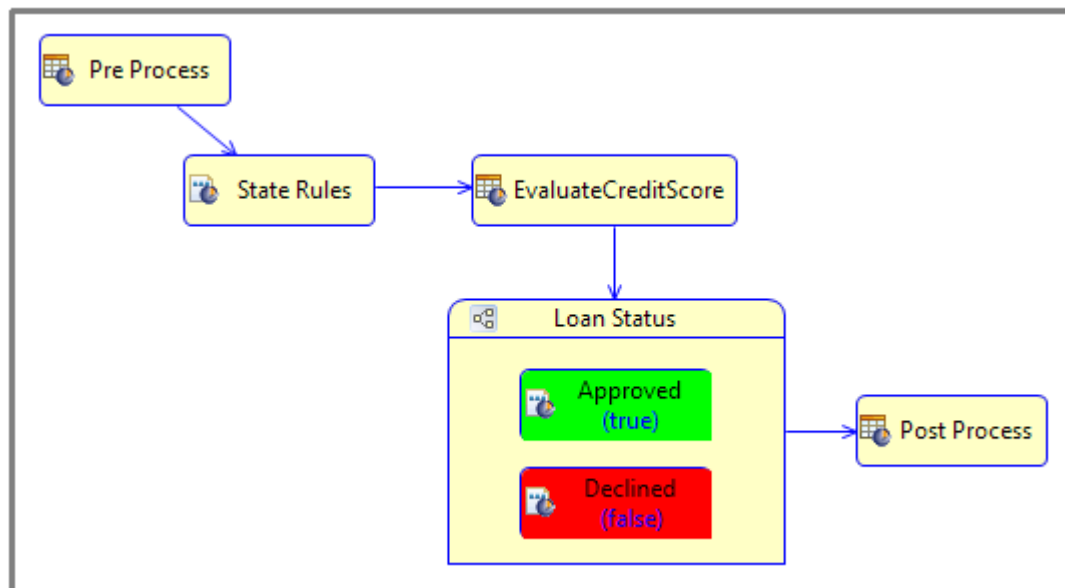
A branch node can be Rulesheet, Ruleflow, Service Call Out, Subflow, or another Branch container.

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 the 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 the general concepts of branching. First, let's review enumerations and Booleans because 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 data types 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 drop-down lists when you are defining Ruletests. Notice that the Boolean data type is not listed as it is implicitly an enumeration.

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

Boolean
Decimal
DateTime
Date
Integer
String
 States
Time

Every attribute that is an enumerated data type or a Boolean is available for branching. For more information, see Enumerations on page 35.

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 in a Ruleflow that contains 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 Browse... . 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 specific, the attribute preferred is `Applicant.mortgage.approved`. Click on that attribute as shown:
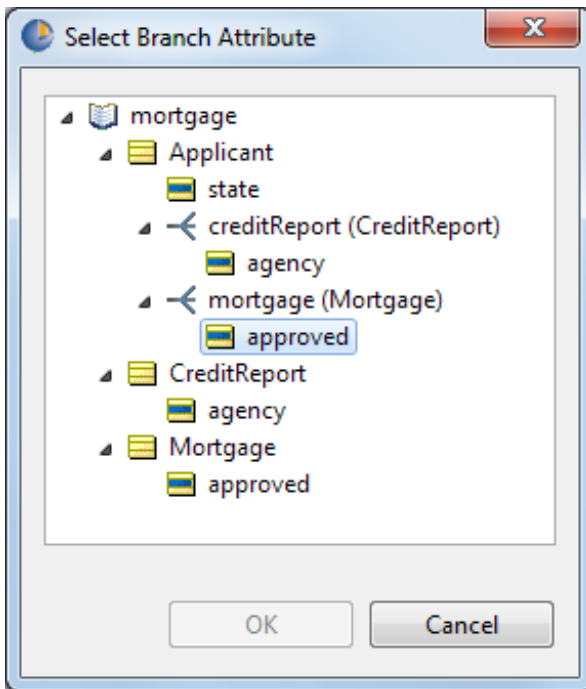
5. Click **OK**.

6. You can define the Boolean branches in a few ways:

   - Click the **Value** drop-down list, as shown:
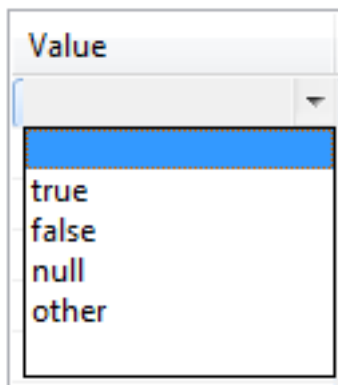
   Notice that there four choices for a Boolean. The null value is offered because the attribute is not set as Mandatory so `null` is allowable. The other value is demonstrated below.

   - 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:
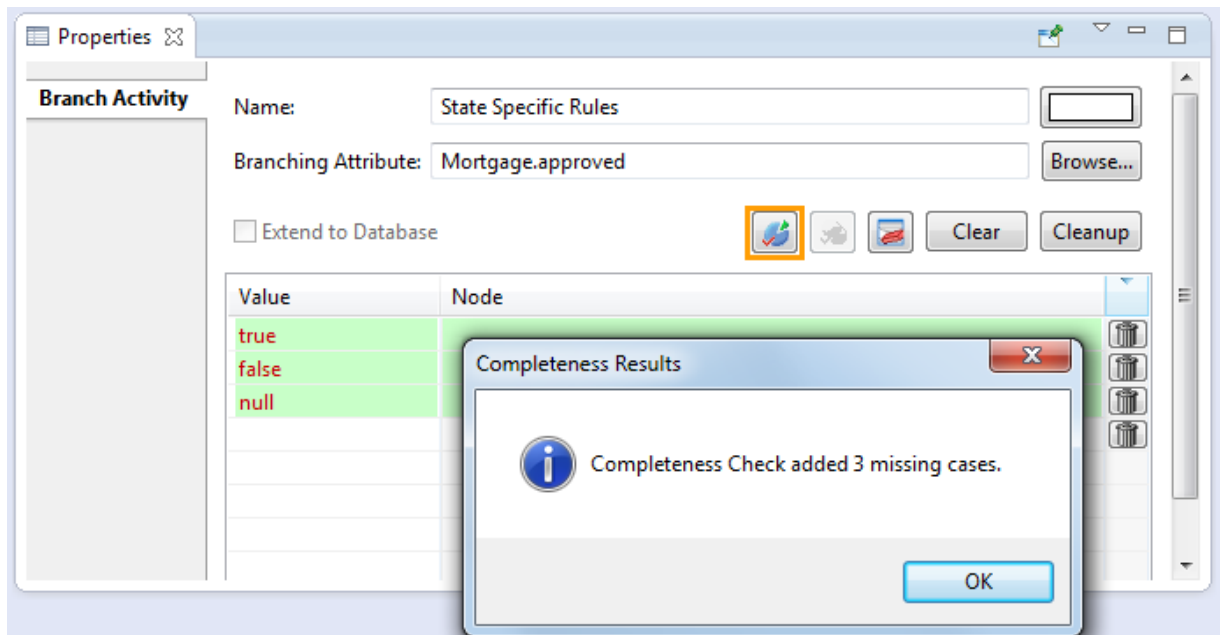
Notice that it does not add `other` to the list. If you set `true` and `other` as shown above, clicking **Check for completeness** would have nothing to add because `other` implies completeness. You can clear green highlights by clicking the **Clear analysis results** button.

The values listed are in red until we each one is bound 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 a Value line and then use the drop-down list 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, the required branches for this rule flow are defined.

8. Connect the incoming and outgoing connections to the branch to complete the flow on the canvas.
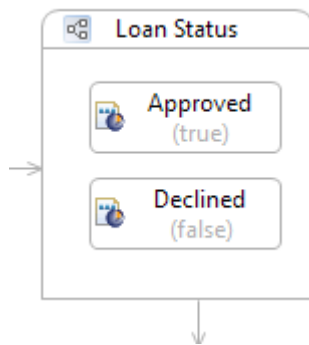
Multiple values can direct to the same target node, as shown in these colorized examples, where all the 'not true' possibilities are assigned to the **Declined** node:
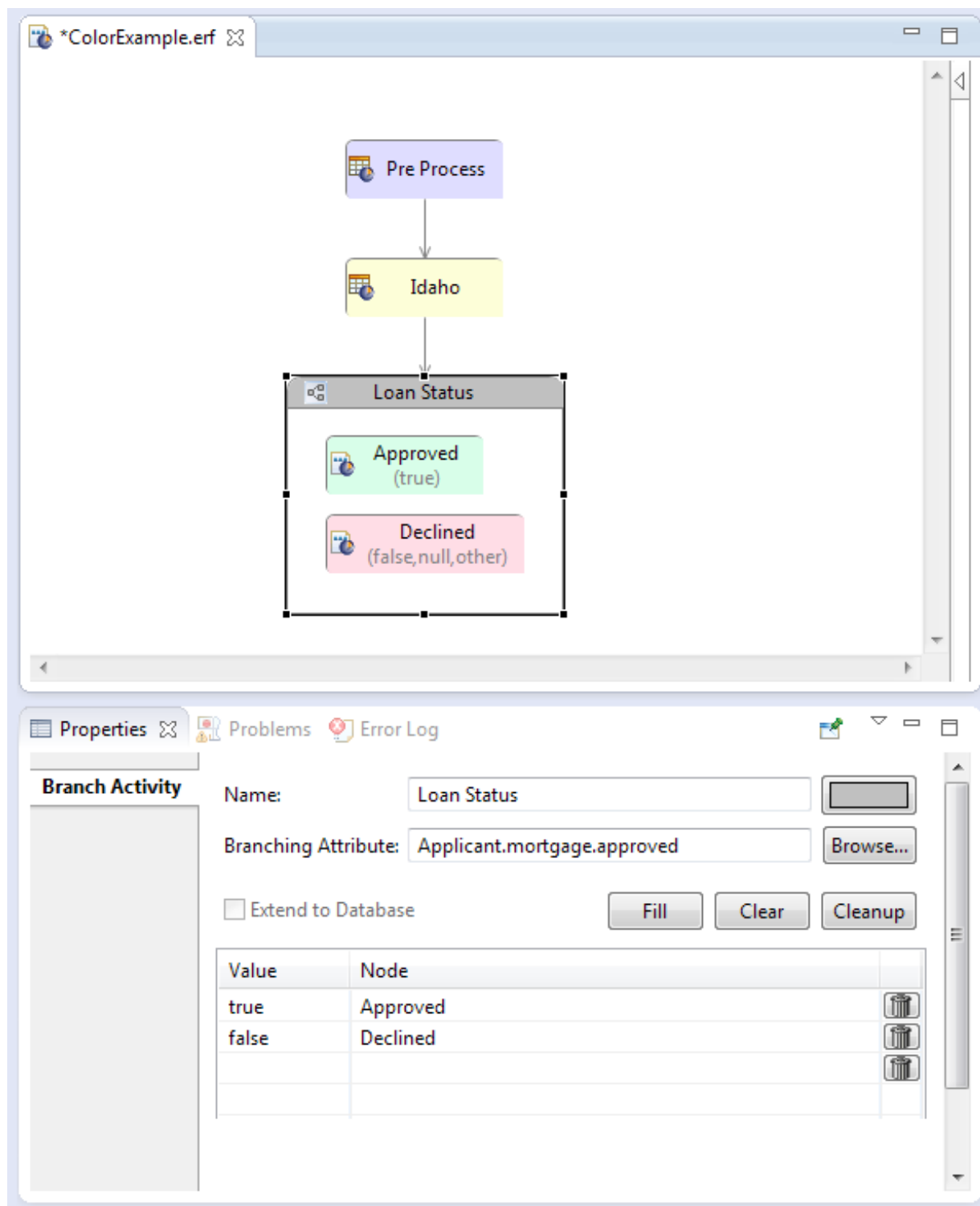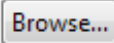
That completes the creation of this Boolean-based branch.

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 Browse... . 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:

Data Type Name	Base Data Type	Enumeration	Constraint...		Label	Value	
States	String	Yes			AK	'AK'	
Agency	String	Yes			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'	

Vocabulary tree:
- mortgage
  - Applicant
    - address
    - city
    - name
    - state
    - creditReport (CreditReport)
    - mortgage (Mortgage)
  - CreditReport
    - agency
    - score
  - Mortgage
    - amount
    - approved
    - rate

Note: See Enumerations on page 35 for information about 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:
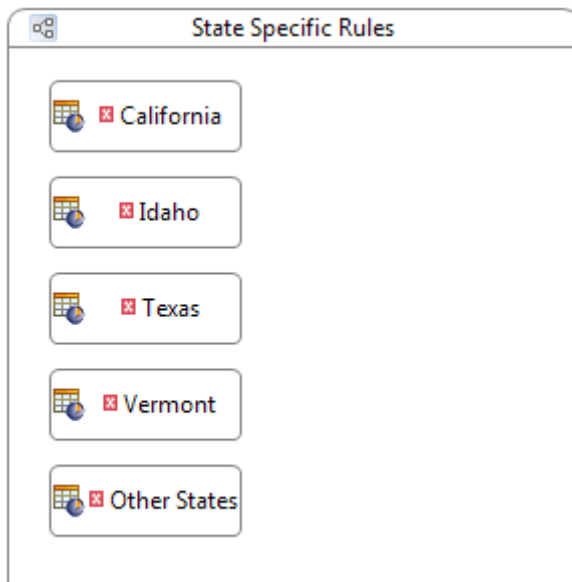
6. On the canvas, click the branch to open its **Properties** tab. You can define the enumeration branches in a few ways:

- Click the **Value** drop-down list. 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:
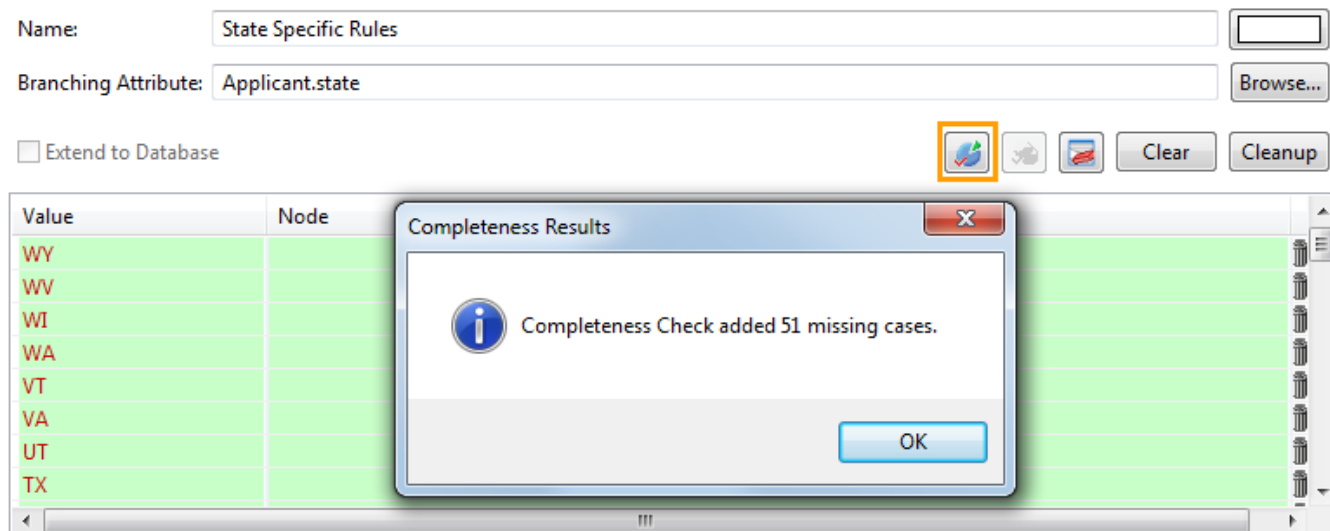
Notice that it does not add `other` to the list. If you set `true` and `other` as shown above, clicking **Check for completeness** would have nothing to add because `other` implies completeness. You can clear green highlights by clicking the **Clear analysis results** button.

The values listed are in red until each one is bound to a node.

7. Click a state value, then use the drop-down list 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 drop-down 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 the down arrow and choose `other`.
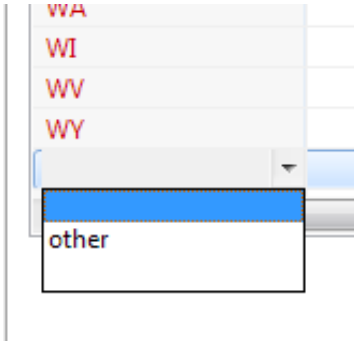
   Assign the `Other States` Rulesheet to that value.

9. After all the nodes are assigned to values, click **Cleanup** to clear all the unassigned values, as shown:
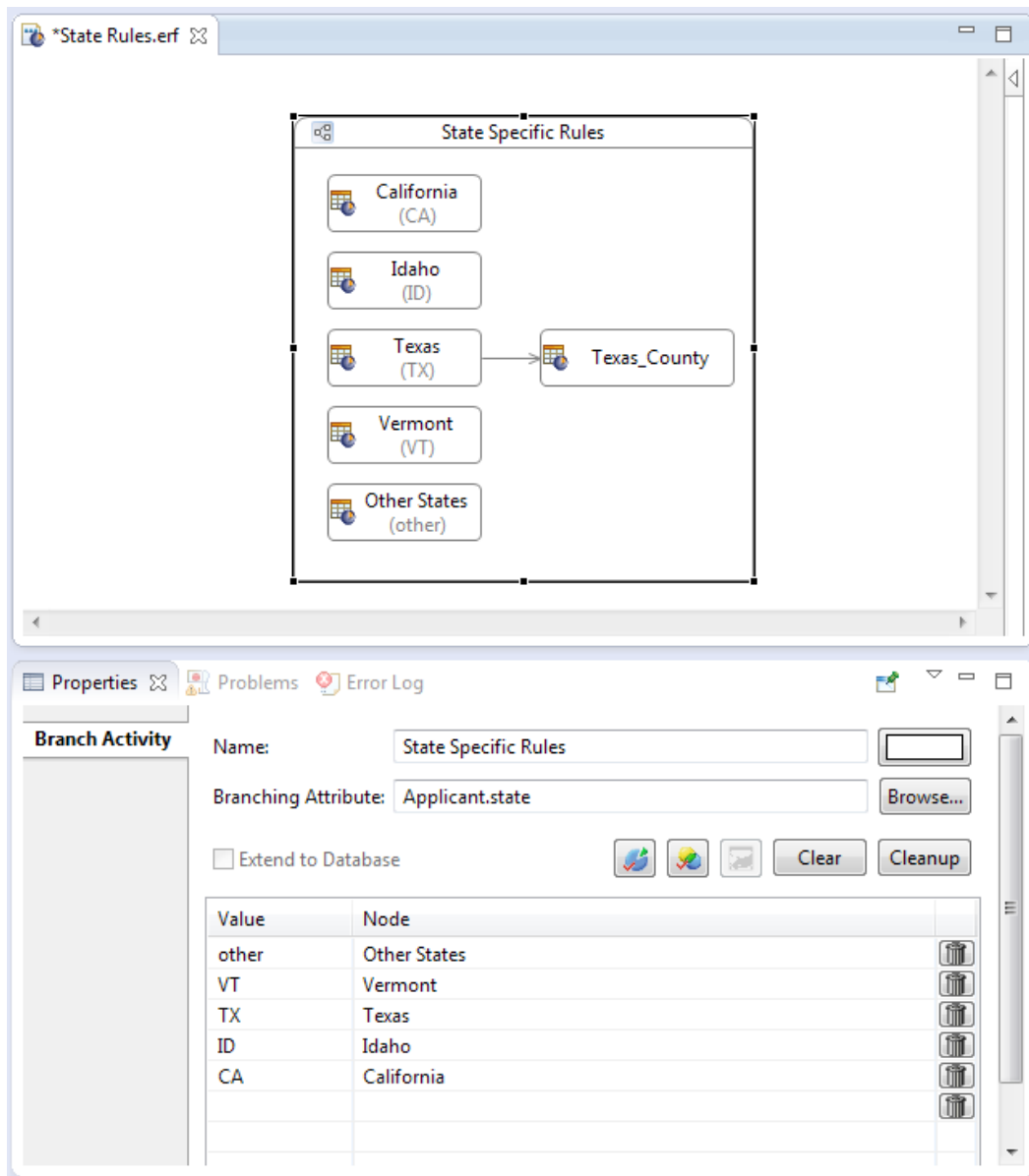
The unassigned values that were removed will all be handled by the `other` value's node. If you click **Check for Completeness** now, you get that the branch is complete.

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 are 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:
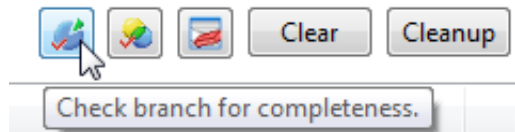
This adds all missing values as branch targets.

When branching by a Boolean attribute, three values are added, as shown:
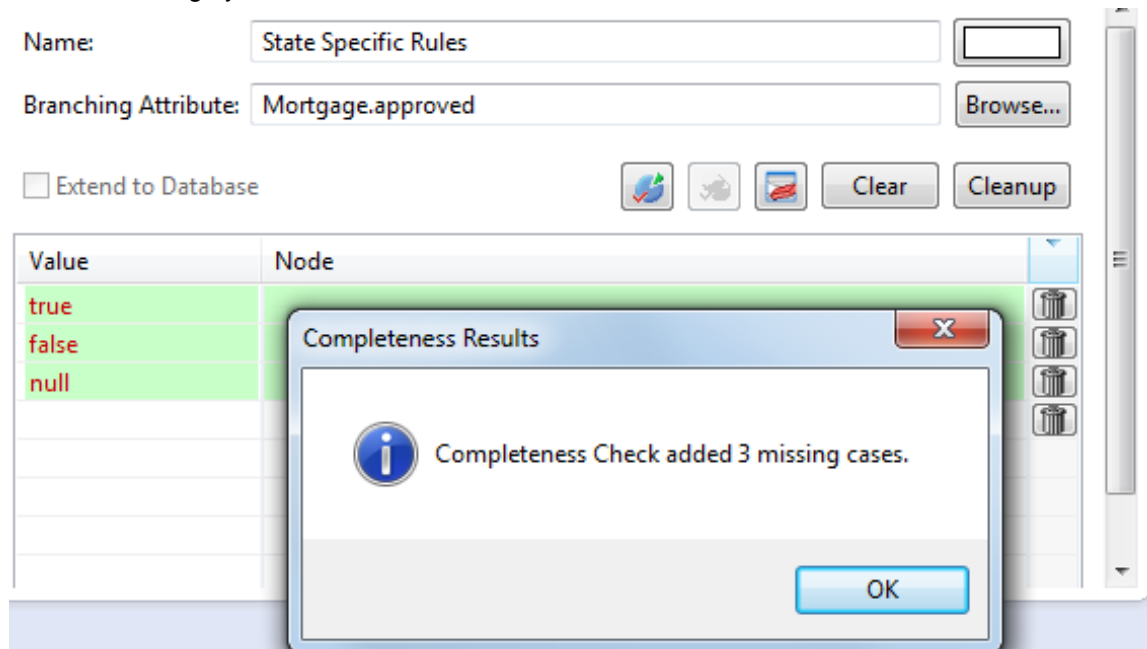
When branching by an enumerated Custom Data Type attribute, each label in the enumeration is added, as illustrated:
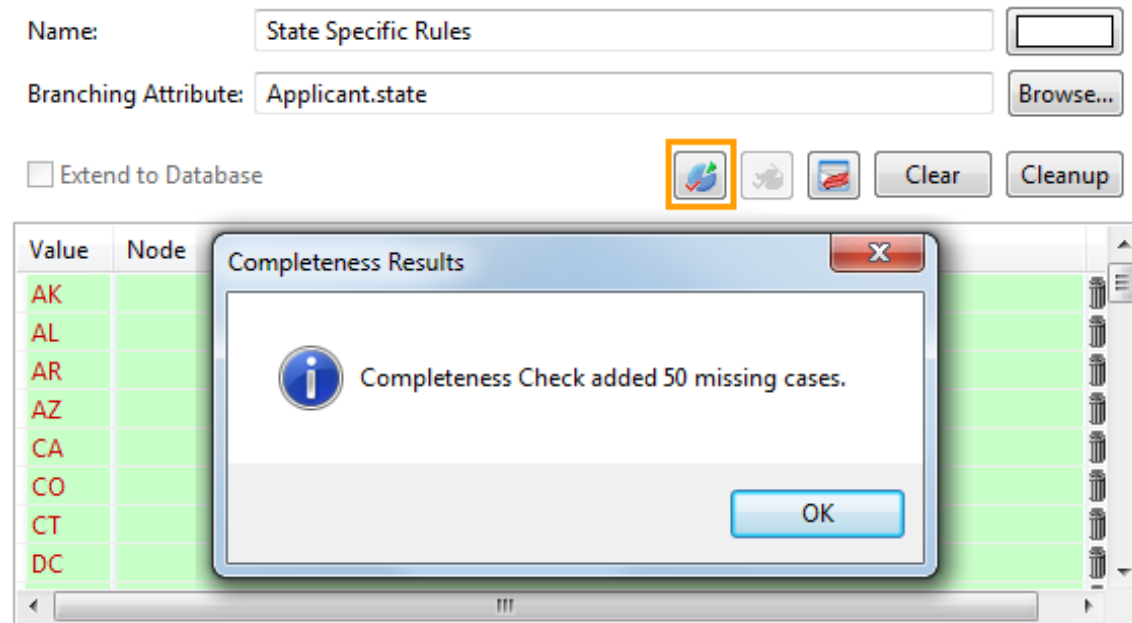
If the completeness check adds additional branch values, these will be highlighted in green. Clicking **Clear analysis results** removes color highlighting:
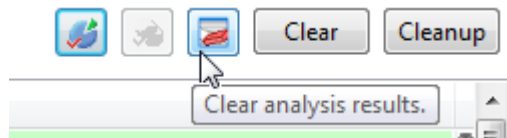
Assign nodes in the branch to appropriate listed value values. When you are done, click **Cleanup** to remove any branch values which do not have corresponding branch nodes. Unless you specify the keyword `other` as a branch value and assign it a branch node, your branch would be incomplete; you have 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 shown:

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 activities are executed in the enumeration order as defined in the Vocabulary. Branch activities are not processed concurrently, they are executed sequentially.

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 does 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 are 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`, then 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`, then the rules will execute on every `Customer.policy`, not just the branched ones. Because the branch was on `Policy`, rules that reference `Policy` only execute on the branched ones.

How to generate 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:

Ruleflow	Window	Help
💡 Comment...		
📑 Properties		
📇 Dependency Graph...		
📄 Report...		
📄 Service Contract...		
Export WSDL...		
Export XSD...		
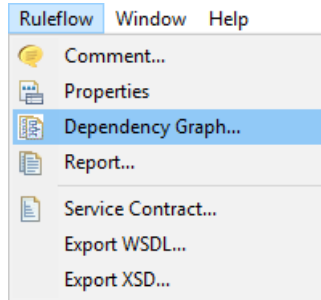
The **Generate Dependency Graph** dialog box opens:

Generate Dependency Graph

Graph Options

Choose options for generating the graph.

Graph Type: Attribute Dependency Graph ▼ Attributes...

Output Folder: C:/_6.0/work_dir/Studio/Graphs Browse...

☑ Graph only selected ruleflow nodes

Finish Cancel
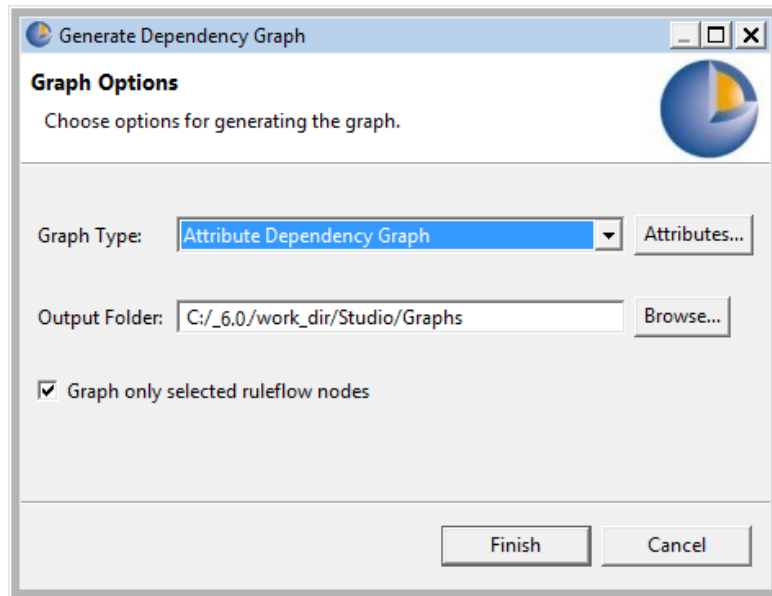
Choose the type of graph you want and the output folder. You can focus the analysis on just nodes that you selected before opening the dialog, or all nodes on the Ruleflow canvas.

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:
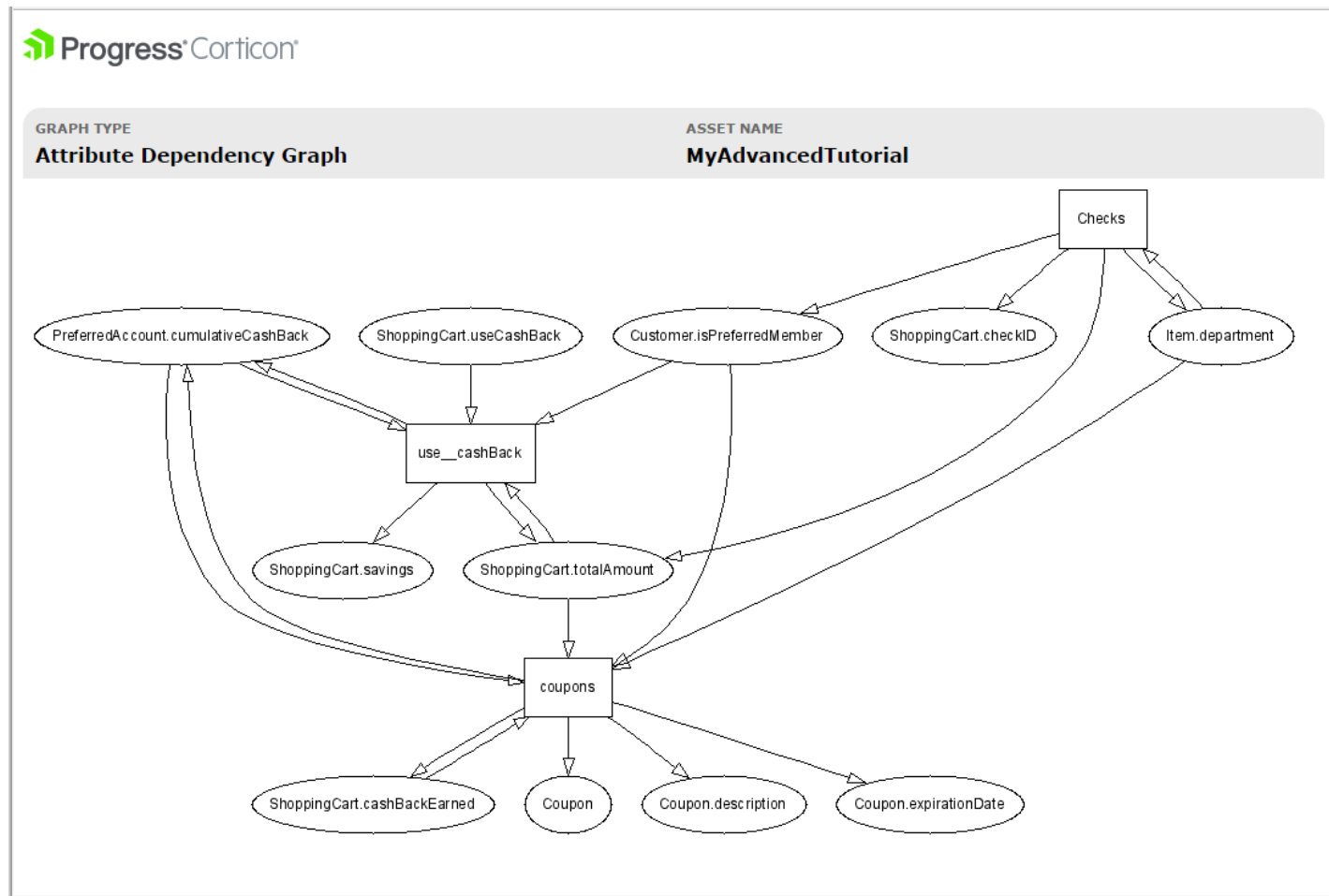
For large projects, graphs with all the attributes and dependencies can be difficult to work with. You can specify that only selected attributes are to be analyzed. Click **Attributes** to open the **Attribute Selector** dialog box, as shown:
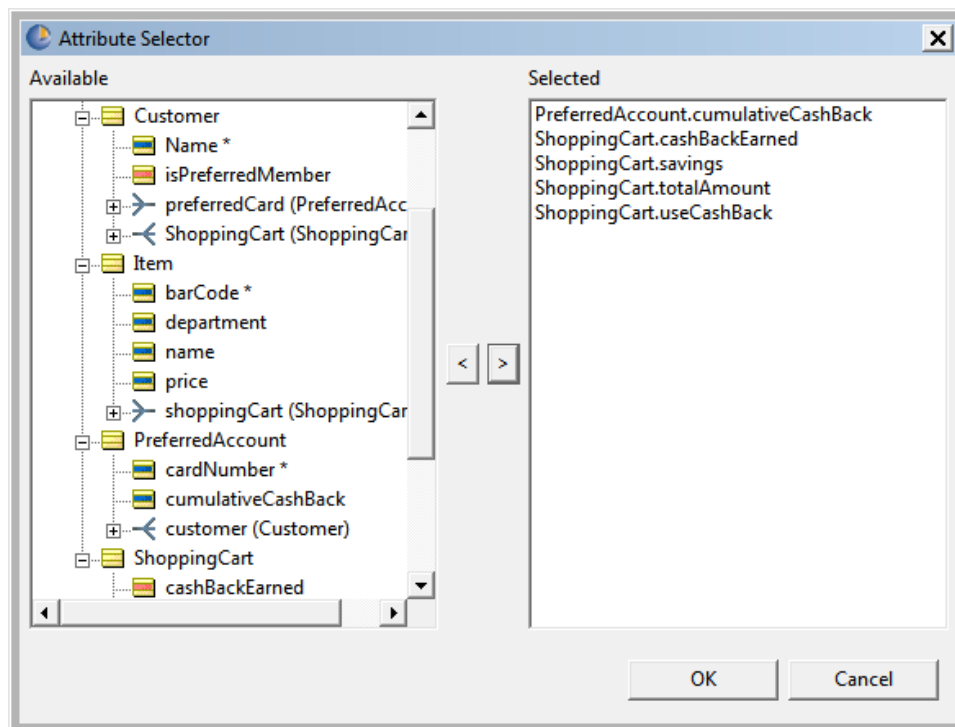
In this illustration, five attributes were selected. Clicking **OK** returns to the graph options. Clicking **Finish** generates the graph.

The graph opens in your default browser, as shown:
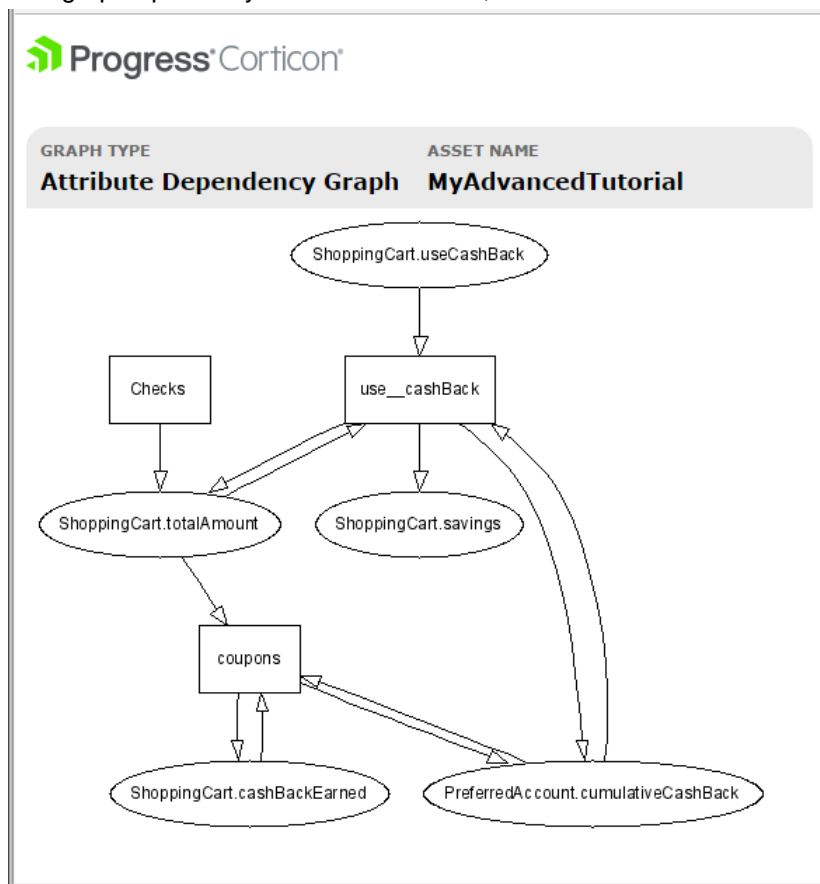
The graph image and its supporting files are saved in the output folder.

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:
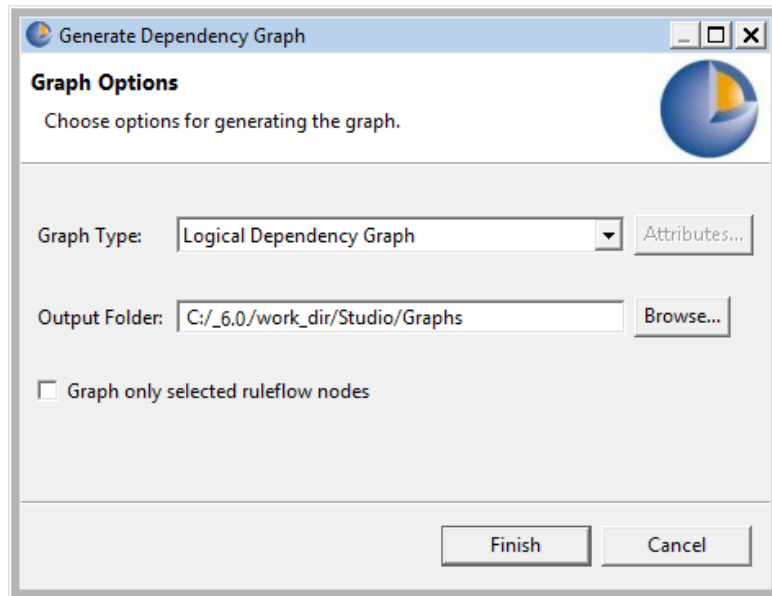
You can set the output folder to your preference and if Ruleflow nodes were selected before opening the dialog box, the analysis is limited to those nodes. The option to specify attributes is not relevant and not available.

Clicking **Finish** generates the graph. The following figure is the logical dependency graph for Rulesheets in the advanced tutorial's Ruleflow:
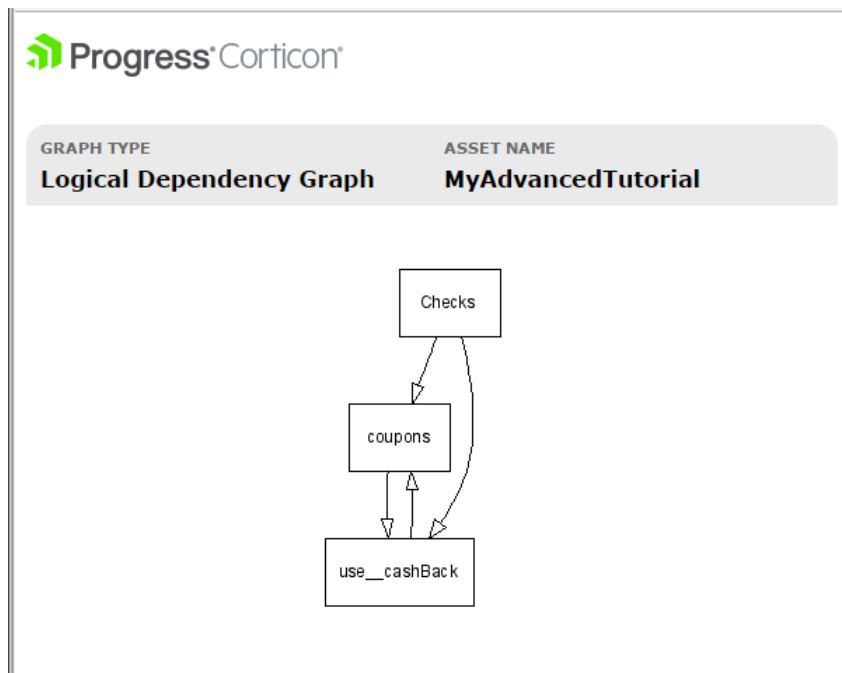
The graph image and its supporting files are saved in the output folder.

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 versions and effective dates

You can apply versioning and effective dates so that you can focus requests, and delimit availability of a Decision Service

Ruleflow version

Major and minor version numbers for Ruleflows are optional. With the Ruleflow open in its editor, select the menu command **Ruleflow > Properties**, and then enter the **Major Version** and **Minor Version** as integer values, as shown:

Figure 226: Assigning a version number to a Ruleflow

When you use different version numbers to describe identically named Ruleflows, the Corticon Server keeps each Decision Service 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 Decision Service 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 Deployment Guide*.

Version label

You can add a text descriptor to the Ruleflow by typing in the **Version Label** field. The description stays with the Ruleflow file, and is packaged in any Decision Services created from the Ruleflow. In the Web Console, every deployed instance of the Decision Service lists the **Version Label** on its details page.

Major and minor versions

Major and Minor version designations are arbitrary and can be adapted to fit the version naming conventions used in different environments. As an example, Ruleflow minor versions can be incremented whenever a component Rulesheet is modified. Major Ruleflow versions can 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 about how to invoke a Ruleflow by version number, see the topic *"Decision Service versioning and effective dating" in the Deployment Guide*.

Effective and expiration dates

Effective and expiration dateTimes are optional for Ruleflows and can be assigned singly or in pairs. When you 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 described in the *Deployment Guide*.

Effective and expiration dates can be assigned using the same window as for the version numbers. Clicking on the **Effective Date** or **Expiration Date** drop-down displays a calendar and clock interface, as shown:

Figure 227: Setting Effective and Expiration Dates
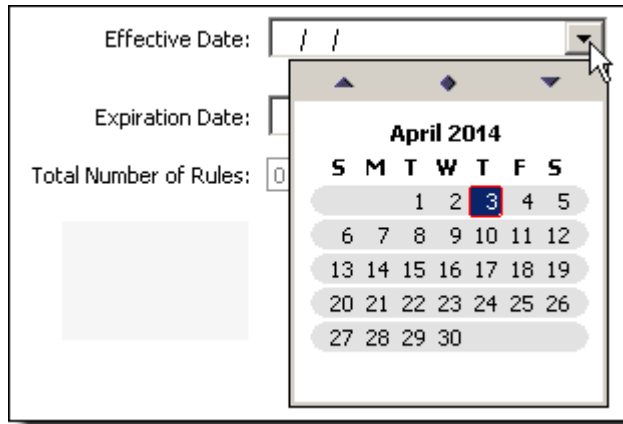
Setting a specific target date for a Ruletest

When you execute a Ruletest against a corresponding Decision Service that is deployed and running on a Corticon 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 shown:

5. Click **OK**. The dialog box closes. The details of the deployed Decision Service and its overrides are displayed at the top of the Testsheet:

```
📄 untitled_1
http://localhost:8850/axis?name=Cargo,major version=1,effective target date=01/29/16 12:00:01 AM
```
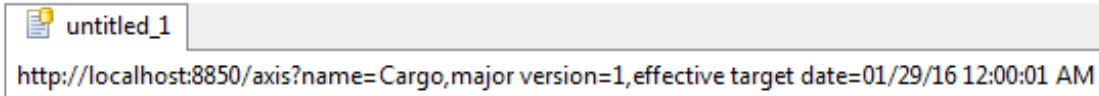
6. Run the Ruletest.

The test executes against the specified Decision Service on the selected server using the overrides you entered.

TestYourself questions for Ruleflow versions and effective dates

Note: Try this test, and then go to TestYourself answers for Ruleflow versioning and effective dating on page 344 to correct yourself.

1. True or False. If a Ruleflow has an Effective date, then it must also have an Expiration date.

2. True or False. If a Ruleflow has an Expiration date, then 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 can be raised or lowered.

6. True or False. Ruleflow Effective and Expiration dates are mandatory.

13

Troubleshooting Corticon Studio problems

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 operations. 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 with or invocation of Corticon Server.

The following methodology will 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, and 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 because, 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

- Use Corticon Studio to reproduce the behavior

- Studio license expiration

- How to compare and report on Rulesheet differences

- TestYourself questions for Troubleshooting rulesheets and ruleflows

Where did the problem occur

Regardless of the environment the error or problem occurred in, always 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 a test or production deployment environment, then obtain a copy of the Ruleflow (`.erf` file) and open it, its constituent Rulesheets (`.ers` files), and its Vocabulary (`.ecore` file) in Studio.

Use 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 Ruletest known to be good not only verifies that 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, 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.

Observe constraint violations or severe errors

When you run a Ruletest in Studio, it might produce error messages. 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.

Constraint violation

A constraint violation indicates that values in the test's attributes are not within numeric constraint ranges or not included in enumerated lists that were set in the Vocabulary's Custom Data Types. A constraint violation might look like this:

Figure 228: A Constraint violation in a Ruletest

In the example, the constraint is shown, and its violation is marked on the attribute and its entity in the Input column:

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 the enforcement of such violations through a setting in the `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>
  ...
  </Messages>
</CorticonResponse>
```

See How to relax enforcement of Custom Data Types on page 46 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, then you reveal several additional execution properties that can be helpful in support efforts, but they are otherwise not meaningful to users.

Severe errors

Some errors indicate problems with how the rules engine is handling the Decision Service: Null Pointer Exception, Reactor Exception, Fatal Exception. These error conditions are important to resolve as soon as possible.

Immediately capture and save any advanced information in the alert, and then copy and save the logs. You might want to try closing Corticon Studio and running the Ruletest again. If it reliably fails with a severe error, package the current project and logs, and then contact support. If you followed the best practice of retaining offline backups of the project as well as saving your work, you might be able to resume with most-recent backup in a different project workspace.

Note: Next step in troubleshooting—If you did not encounter constraint violations or severe errors, any other problems are within your rules. Proceed to Analyzing Test Results. To work around a problem in rules, you can identify the expression syntax that produces it, and then try 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.

Analyze Ruletest results

This section assumes:

- Your Ruletest produced none of the previously mentioned errors, 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.
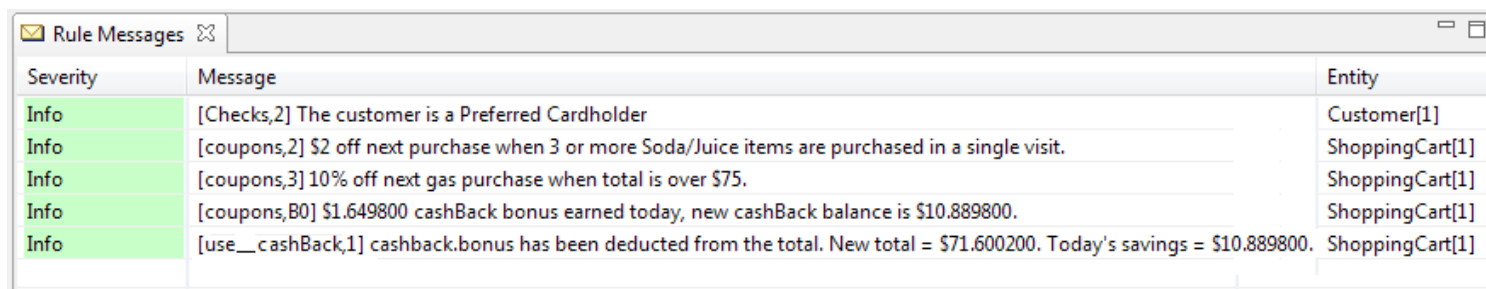
  The Corticon Server log captures errors and exceptions caused by certain rule and request errors. These log messages are detailed in the *Using Corticon Server logs section of the Server Guide*.

- If not, the problem is with the rules themselves. Continue in this section.

Trace 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 which 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 229: 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, add a line to the `brms.properties` as:

```
com.corticon.reactor.rulestatement.metadata=true
```

Note: See *"Server properties and settings" in the Server Guide* for more information.

Identify the breakpoint

To understand why your rules are producing incorrect results, it is 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. After you 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 displays those messages in the order that 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 the header of those columns, clicking 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 prevents the rule from firing, becoming the cause of another failure!

Disable/Enable

Disabling and then re-enabling individual Condition/Action rows, entire rule columns, Filter rows, and even whole Rulesheets is a powerful way to isolate problems:

* **Rulesheet elements** - Right-click active Condition or Action row headers, column headers, or Filter row headers to display a pop-up menu containing enable/disable options. Disabled rows and columns will be shaded in gray on the Rulesheet.

Figure 230: Rulesheet with Rule Column 2 disabled.

* **Ruleflow objects** - Select objects on a Ruleflow canvas, and then click the **Disable/Enable** toolbar button

 to toggle the disabled objects to dark gray. Redo the action to re-enable the object.

Figure 231: Ruleflow with coupons object disabled
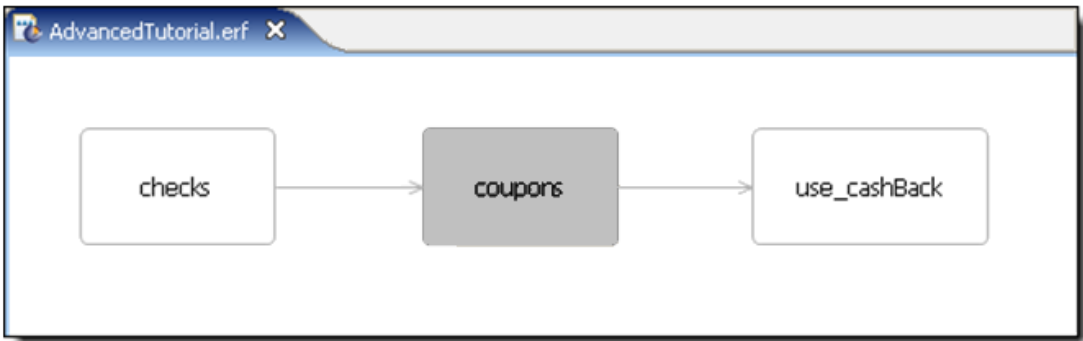
Be sure to save these changes before running a Ruletest to ensure the changes take effect.

Disable and re-enable Rulesheet elements and Ruleflow objects 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 indicates that the rule is telling the rule engine to do something it cannot 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 is not defined at the 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

After the breakpoint rule is isolated, it is often helpful to copy the relevant logic into another Rulesheet for more focused testing. See the *Rule Language Guide* to ensure you have expressed your rules correctly. Be sure to review the usage restrictions for the operators 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 the product version used, and the set of Corticon files (`.ecore`, `.ers`, `.erf`, and `.ert`) that will enable us to reproduce the problem.

Partial rule firing

A Condition/Action rule column might partially fire, meaning Action A is executed but Action B 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, then the rule *does* fire partially. All the Aircraft values are computed, but the weight is not computed from the null value of the attribute.

How to initialize 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 – they are assigned the result of the calculation. For example, when you calculate 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 was 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.

How to handle 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:
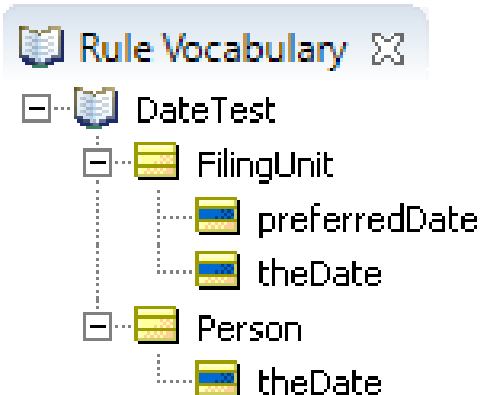
Here are two scenarios:

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 that happens to be null. If the date is null, then it is used in the subsequent Rulesheets in their Conditions section. However, because the value is null, a warning is 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

Rule Statements ⊠   ✉ Rule Messages   💬 Natural Language   ▦ Properties   🗐 History

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 prefered 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:

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

Rule Statements ⊠   ✉ Rule Messages   💬 Natural Language   ▦ Properties   🗐 History

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 prefered 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.

Studio license expiration

If your license indicates that it has expired, contact your Progress Corticon representative to obtain an updated license file. Corticon Studio alerts the user at startup, and then limits functionality:

Figure 232: License expiration alert at Studio startup
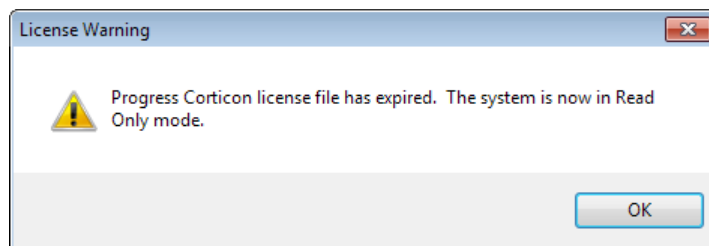
How to compare and report on Rulesheet differences

When the execution of your rules is not producing the expected results and your not sure what changed, Corticon Studio provides difference reports to help identify changes. Two versions of a Rulesheet can have modest changes, yet it can be difficult to see all the differences during a visual inspection of the two Rulesheets. Reporting about differences between Rulesheets provides help in debugging mistaken rule changes, and inconsistent rule definitions, for example:

- **Diagnosing a Ruletest failure**: When a Ruletest fails because of changes in newer Rulesheets, you can use Rulesheet difference reports to determine what changed, and then make changes to a Rulesheet to fix bad rules, or to indicate changes to make to your Ruletest expected results.

- **Resolving merge conflicts**: When using a source control system such as git, you may encounter situations where you want to commit a Rulesheet that someone else has changed, and discover a merge conflict. Using Rulesheet difference analysis and reports, you can see what changed and decide how to manually merge the differences so you can commit your changes.

To compare two versions of a Rulesheet:

1. Right-click within a Rulesheet, and then choose the menu command **Compare Rulesheets**.

2. The **Compare Rulesheets** dialog box opens, as shown:

Rulesheet 1: Tutorial/Tutorial-Done/Cargo.ers

Rulesheet 2: C:\workspace\Tutorial\Tutorial-Done\CargoReefer.ers   Browse

Report Type: DetailedRulesheetDifferences

Report Style: Corticon Blue

Output Folder: C:/work_dir/Studio/Reports   Browse

Finish   Cancel
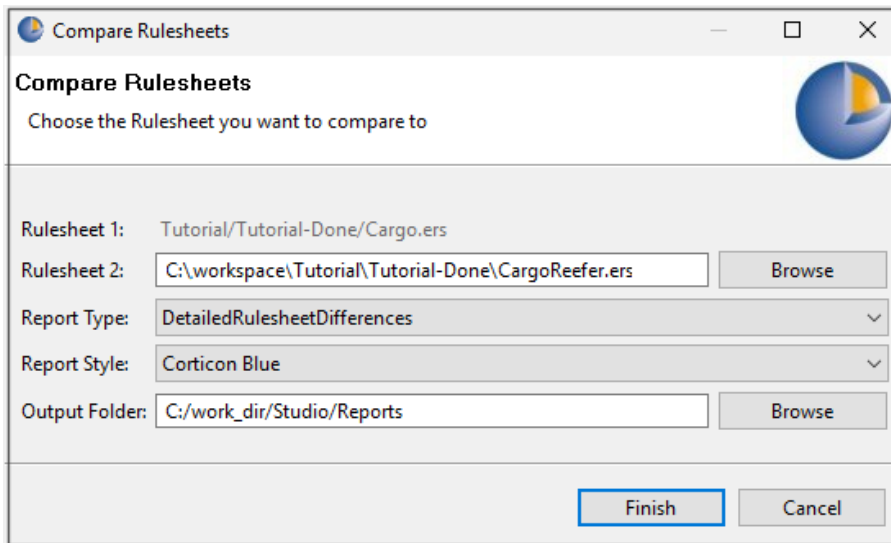
Rulesheet 1 is the Rulesheet currently in the editor.

3. Locate **Rulesheet 2**, a variation of Rulesheet 1, typically produced earlier in development or by another developer.

4. Choose a preferred **Report Type**.

5. Choose a preferred **Report Style**: The CSS stylesheet to use for the report. The basic stylesheets are **Corticon Blue** and **Corticon Green**.

6. Choose a preferred **Output Folder**: 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.

7. Click **Finish**.

Customized difference reports

Advanced users might want to create alternative report types and styles:

- 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 drop-down list for the asset type.

- 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** drop-down list.

Reading a differences report

The Rulesheet difference report evaluates what's changed -- additions, deletions, and modifications as well as items set as disabled. Presentation differences—colors, fonts, natural language, and widths—between the Rulesheets are ignored.

A report lists all the data in both Rulesheets. Items that are the same in both Rulesheets are not highlighted while those that are different are highlighted. The reason could be because the item changed. These need to be researched to see if they pair with an item on the other Rulesheet that has a variation of the item in that location.

Examples of how differences are reported

The following examples use the basic tutorial's Cargo Rulesheet as the Rulesheet to which variations are compared:

*Cargo.ers

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

	Actions					
	Post Message(s)		✉	✉	✉	
A	Cargo.container		standard	oversize	heavyweight	
B						
	Overrides			1		

Example: Extra condition

*Cargo.ers

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

	Actions					
	Post Message(s)		✉	✉	✉	✉
A	Cargo.container		standard	oversize	heavyweight	reefer
B						
	Overrides			{1, 4}		{1, 3}

Conditions a and b are matched; however, Rulesheet 2 has an extra Condition, c.

Conditions

Rulesheet1	Rulesheet2
a. Cargo.weight	a. Cargo.weight
b. Cargo.volume	b. Cargo.volume
	c. Cargo.needsRefrigeration

Example:One match that is in sequence and one that is out of sequence

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

	Actions						
	Post Message(s)		☑	☑	☑		
A	Cargo.container		standard	oversize	heavyweight		
	Overrides			1			

There are a few differences illustrated in this example:

- In-sequence match: Condition c in Rulesheet 1 matches condition b in Rulesheet 2.

- Out-of-sequence match: Condition d in Rulesheet 1 is marked as different because Condition a in Rulesheet 2 is out of sequence, and is marked as different.

- Extra: Condition: c in Rulesheet 2 is extra, and therefore different.

- Empty Condition Rows: Rulesheet1 has two empty Condition rows a and b are highlighted.

Conditions

Rulesheet1	Rulesheet2
a.	
b.	
c. Cargo.volume	b. Cargo.volume
d. Cargo.weight	
	a. Cargo.weight
	c. Cargo.needsRefrigeration

Example: A Condition is disabled

When the *state* of the condition is different, the conditions are matched, but marked as different, as shown. Condition c is disabled in Rulesheet 1; it is highlighted but matched.

TestYourself questions for Troubleshooting rulesheets and ruleflows

Note: Try this test, and then go to to correct yourself.

1. reproduce the behavior in a Corticon Studio as when executed on _____.Troubleshooting is based on the principle that Rulesheets behave the same way when tested in

2. Troubleshooting is based on the principle that Rulesheets behave theThe 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. Rulesheet is _____.

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 Nonconditional row (that is, an Action row in Column 0)

    f. a rule column

    g. a Rulesheet

    h. a Ruleflow

A

Studio properties and settings

Corticon Studio provides properties that specify property names and default values of user-configurable behaviors.

The settings file `brms.properties` is installed at the root of `[CORTICON_WORK_DIR]` for each Corticon Studio installation. If you install Corticon Studio and Corticon Server on one machine and accept the default colocating paths, one `brms.properties` file is installed to be shared by Studio and Server:
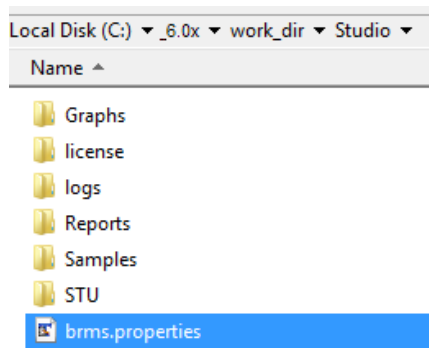
About the brms.properties file

- It is good practice to back up the file before you start to make changes.

- When installed separately, the Studio and Server `brms.properties` files are identical.

- If you delete the file, it does not get re-created if you restart. However, because these are overrides to default properties, there is no loss of features or functionality when the file is not present.

- In the absence of a `brms.properties` file, you can simply list the property settings in a text file, and then save it to its proper location as `brms.properties`.

- An update of the installation preserves a modified `brms.properties` file, and adds the default file if none is present.

Enabling settings listed in the default brms.properties file

The file lists properties that users commonly want to change. Each group of properties provides descriptive comments and the commented default name=value pair.

To specify a preferred value for a listed property, edit the file, remove the# tag from the beginning of a property's line, and then add your preferred value after the equals sign (=). For example, to express a preference for decimal values displayed and rounded to two places instead of the six places preset for this property, locate the line:

```
#decimalscale=6
```

Change the line to:

```
decimalscale=2
```

Add unlisted settings to brms.properties file

Some locations in the documentation tell you about other property settings that you might want to add to the settings file. Or, you might be directed by technical support or your Progress representative to add or change settings to provide certain behaviors or functions.

For example, to change the interval of diagnostic readings from five minutes to two minutes, add the following line to the `brms.properties` file—it does not matter where in the file as long as it is on a separate line:

```
com.corticon.server.DiagnosticWaitTime=120000
```

If you add the same property more than once in the settings file, the last instance takes precedence.

Save and apply the revised Studio property settings

When your changes are complete, you can choose to save the settings file with its default name and location, but you could save a copy with a useful name, such as `debuggingLogSettingsbrms.properties`.

In Studio, you can save multiple settings files, and then use Studio's **Preferences** to specify the **Override Properties File** for the `brms.properties` file to use, as shown:
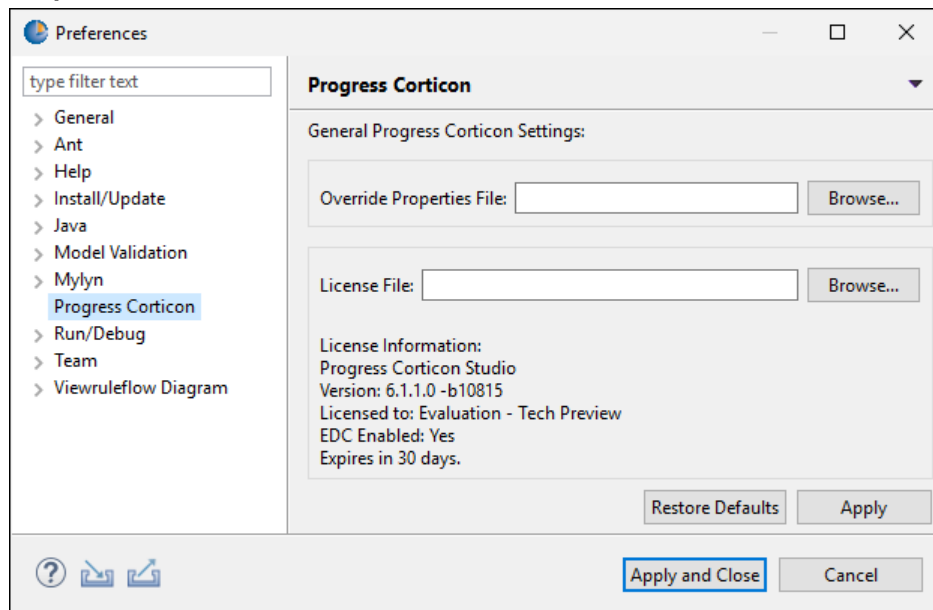
Note: The overrides and license specified are stored in the Studio Workspace. If you change the Workspace, then those overrides or defaults take effect.

For the revised settings to take effect, save the edited file, and then restart the Corticon Studio.

Note: Property settings you list in your `brms.properties` file *replace* the 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 is the only mask that Corticon uses.

The following properties are settings you can apply to your Corticon Studio installation by adding the properties and appropriate values as lines in its `brms.properties` file, and then restarting Studio.

GENERAL

Decimal scale sets the default precision for Decimal values. All Decimal values are rounded to the specified number of decimal places. The default value is `6`. For example, `4.6059556` is rounded, displayed, and returned as `4.605957`. In the `brms.properties` file, set the Studio Test's decimal scale:

```
decimalscale=2
```

When the Decimal scale is set to `2`, the rounded value is `4.61`.

To property sets the default character encoding for objects, such as Vocabulary, Rulesheet, and Ruletest XML files. Examples: UTF-8, UTF-16, ISO-8859-1, US-ASCII. Default value is UTF-8.

```
com.corticon.encoding.standard=UTF-8
```

RULESHEETS

Specifies the number of rows that are added to the end of a Rulesheet section when **Rulesheet** > **Add Rows to End** is selected from the Corticon Studio menubar or popup menu. Default is `10`.

```
com.corticon.designer.corticon.insertrowstoend=10
```

Specifies the number of columns that are added to the end of a Rulesheet section when **Rulesheet** > **Add Columns to End** is selected from the Corticon Studio menubar or popup menu. Default is `10`.

```
com.corticon.designer.corticon.insertcolumnstoend=10
```

When there are any null attributes on the right hand side of a clone assignment expression, the assignment does not occur because it will not override the cloned value. If the null result is preferred, add this property set to `true` so that the null checks are removed. Be aware that using this setting on a Studio machine should be applied on any other machine that will work on a related project, and that Decision Services created when the property is `true` have the setting embedded in the Decision Service. Default is `false`.

```
com.corticon.reactor.rulebuilder.DisableNullCheckingOnClone=false
```

RULETESTS

Specifies how the Rule Messages are displayed in the Tester after execution based on the data in the columns. The options are `ExecutionOrder`, `Severity`, and `Entity`. Default value is `ExecutionOrder`.

```
com.corticon.tester.result.messages.sorting=ExecutionOrder
```

Option to specify how many variable substitutions could be applied to an ADC PreparedStatement. The restriction on how many PreparedStatement variables is controlled by the Database Driver. Different Databases have different maximums.

Default value is 1000

```
com.corticon.server.adc.preparedstatements.maxvariables=1000
```

Specifies whether String attribute values should be trimmed in the Tester Expected tree. When set to `false`, suppresses trimming of leading and trailing whitespaces.

Default value is `true`.

```
com.corticon.tester.trimstringvalues=true
```

By default, Corticon Studio uses the Corticon Server's REST API to run ruletests against a remote server. You can change this to use the SOAP API by setting the following property. Default value is JSON:

```
com.corticon.tester.ccserver.execute.format=XML
```

When using the SOAP API and testing against an IIS server you also need to set this property for Corticon Studio (Default value is JAVA):

```
com.corticon.studio.client.soap.clienttype=IIS
```

Sets the Studio Test's XML messaging style: `Hier` (hierarchical), `Flat`, or `Autodetect` Default value is `Hier`.

```
com.corticon.designer.tester.xmlmessagingstyle=Hier
```

RULEFLOWS: Packaging

Corticon Studio uses the following properties when compiling assets into a Decision Service through the "Package and Deploy" wizard. (Corticon Server utilities also use these properties when compiling a Decision Service.)

Compile option: This property lets you configure memory settings for compiling the Rule Assets into an EDS file.

Default value is `-Xms256m -Xmx1g`

```
com.corticon.ccserver.compile.memorysettings=-Xms256m -Xmx1g
```

Compile option: Add the Rule Asset's Report to the compiled EDS file. By having the Report inside the EDS file, any user can get the report for a deployed Decision Service through an in-process or a SOAP call to the Corticon Server. Including the Report in the EDS file will increase the EDS file significantly.

Default value is false

```
com.corticon.server.compile.add.report=true
```

Compile option: Add the Rule Asset's WSDL to the compiled EDS file. By having the WSDL inside the EDS file, any user can get the WSDL for a deployed Decision Service through an in-process or a SOAP call to the Corticon Server. Including the WSDL in the EDS file will increase the EDS file significantly.

Default value is false

```
com.corticon.server.compile.add.wsdl=true
```

Note: In prior releases, the default action was to automatically produce the WSDL and reports to add to the EDS. Given the techniques to produce WSDL and reports without having them in the EDS, the option to suppress the WSDL and reports in packaging unless explicitly requested, results in smaller packages and better compilation performance.

GRAPHIC VISUALIZER

Sets the font type and size used by the Graphic Visualizer. Default values are `Helvetica-Narrow.ttc` and `9`, respectively.

```
com.corticon.crml.CrmlGraphVisualizer.fontname=Helvetica-Narrow.ttc
com.corticon.crml.CrmlGraphVisualizer.fontname.ja=msgothic.ttc
com.corticon.crml.CrmlGraphVisualizer.fontsize=9
```

B

Answers to TestYourself questions

Check out your results from the tests at the end of each section.

For details, see the following topics:

- TestYourself answers for Building the vocabulary
- TestYourself answers for Rule scope and context
- TestYourself answers for Rule writing techniques and logical equivalents
- TestYourself answers for Collections
- TestYourself answers for Rules containing calculations and equations
- TestYourself answers for Rule dependency and inferencing
- TestYourself answers for Filters and preconditions
- TestYourself answers for Recognizing and modeling parameterized rules
- TestYourself answers for Writing rules to access external data
- TestYourself answers for Logical analysis and optimization
- TestYourself answers for Ruleflow versioning and effective dating
- TestYourself answers for Troubleshooting rulesheets

TestYourself answers for 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.
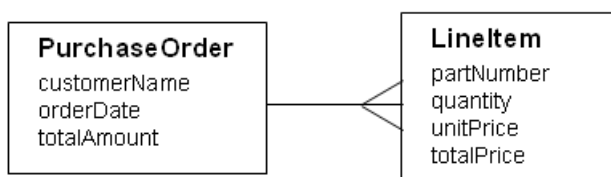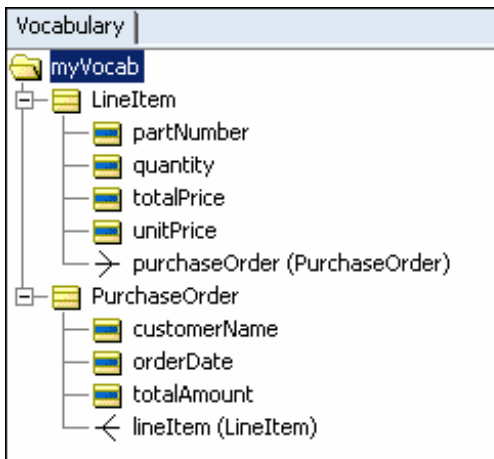
22.

23. Design vocabulary, identify terms, separate terms, assemble and relate terms, diagram vocabulary model in Studio

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 drop-down lists based on the enumerated values.

35. Allow you to re-use entities by bundling or creating a subset within the Vocabulary. (Technically equivalent to packages in Java or namespaces in XML.)

36. True

37. True

38. All entities have native attributes, but `Bicyle` = 100% native. The others have 1 native attribute each and 3 inherited. Entities with inherited attributes are `MountainBike`, `RoadBike`, `TandemBike`

39. `cadence`, `gear`, or `speed`

40. True

TestYourself answers for Rule scope and context

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, you 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 role names 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. You 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. You 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` (that is, the cross product of the DVDs and the Movies 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.

TestYourself answers for 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 **<u>scope</u>**

2. An expression that evaluates to a `True` or `False` value is called a **<u>Boolean</u>** expression.

3. True

4. False. The requirement for complete Values sets only applies to Condition rows.

5. The special term **<u>other</u>** 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 because 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:
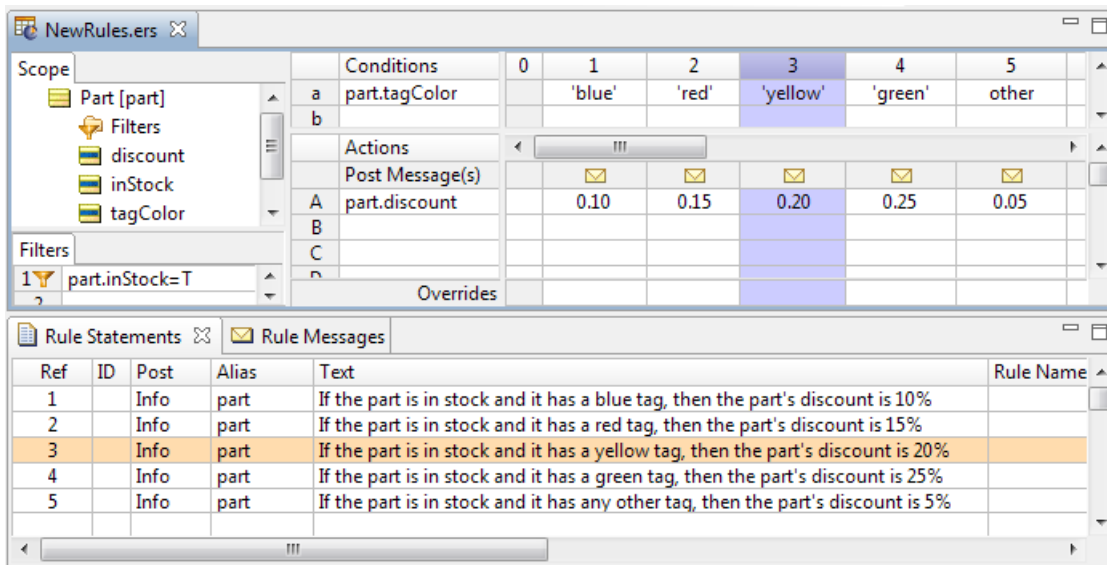
13. True

14. False. Nonconditional rules are governed by preconditions on the same Rulesheet only if they share the same scope as the preconditions.

TestYourself answers for Collections

Show me this set of test questions.

1. Children of a Parent entity are also known as **elements** of a collection.

2. False. A collection can contain root-level entities.

3. True

4. True

5. Refer to the Rule Language Guide for a full list and description of all collection operators.

6. Rule Language Guide

7. Order total is equal to the sum of the line item prices on the order.

8. Items

9. one-to-many (1->*)

10. It is not an acceptable replacement because the use of any collection operator requires that the collection be represented by an alias.

11. Set the navigability of the association between `Order` and `LineItem` to Order->lineItem. In other words, make the association one-directional from `Order` to `LineItem`.

12. Optional, convenient

13. A collection alias is not required in this case because no collection operator is being applied to the collection.

14. `->forAll`

15. `->exists`

16. a. `aroles ->size > 3` *where aroles is an alias for Actor.roles*

  b. `mdvd ->isEmpty` *where mdvd is an alias for Movie.dVD*

  c. `mdextras ->exists(deletedScenes=T)` *where mdextras is an alias for Movie.dVD.extras*

  d. `mgglobes ->exists(win=T)` *where mgglobes is an alias for Movie.goldenGlobe*

  e. `mroles ->size > 15` *where mroles is an alias for Movie.roles*

  f. `mdvd.quantityAvailable ->sum >= 100` *where mdvd is an alias for Movie.dVD*

  g. `mdvd.quantityAvailable ->sum < 2` *where mdvd is an alias for Movie.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, use a unique alias to refer to each collection.

TestYourself answers for 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 because 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 cannot anticipate all possible combinations of the rule language.

8. This Filter tests if the difference between the current year and the year a movie was released is more than 10 years.

9. This Condition tests if the total quantity of DVDs available divided by the number of DVD versions of a movie is less than or equal to 50,000 or greater than 50,000. This same calculation could be performed by using the `->avg` operator.

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

TestYourself answers for Rule dependency and inferencing

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*.

TestYourself answers for 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. none

    o. Movie 1; DVD 1; Oscars 1, 2, 3, 4, 5

TestYourself answers for 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..10001], >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 can be maintained in the values sets of an individual Rulesheet, which is easy to perform, but makes reuse more difficult. They can be maintained as Custom Data Types (Enumerated) in the Vocabulary, which makes reuse easier.

TestYourself answers for 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 *Data Integration 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 Rulesheet is database-enabled, then these root-level terms may need to be extended to the database.

TestYourself answers for 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 can exist in Rulesheets when 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 making the Actions match for both rules, or 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 always fires **instead of** the rule being overridden when the Conditions they share are satisfied.

8. False. The Completeness Checker auto-completes the Condition's value set prior to inserting missing rules. This ensures that the Rulesheet, post-application of the Completeness Check, is truly complete.

9. The Completeness Checker detects Rulesheet incompleteness caused by an incomplete values set because it automatically completes the value set 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 subrules 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 subrules 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. It is a good practice to disable columns added by the Completeness Check that you determine need no Actions.

18. They can be disabled, deleted, or left as-is with no Actions (but being left as-is is not recommended because it will cause activity that 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 not contain all possible combinations of data necessary to verify Rulesheet completeness. In short, the databases 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.

TestYourself answers for 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 can only be raised, not lowered.

6. False. Ruleflow Effective and Expiration dates are optional.

TestYourself answers for 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 Corticon Server.

2. The first step in troubleshooting a suspected rule problem is to reproduce the behavior in a Corticon Studio Ruletest.

3. In the integration with Corticon Server.

4. All of them!

5. The specific rule where execution behavior begins acting abnormally is called the **<u>breakpoint</u>**.

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 **<u>disabled</u>**.

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