Progress®

Corticon.js
Rule Modeling

Progress® Corticon®

Copyright

© 2021 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

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

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries.

Analytics360, AppServer, BusinessEdge, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

Last updated with new content: Corticon.js 1.2

Updated: 2021/06/04

Table of Contents

Introduction to Corticon.js rule modeling 9

Create the Vocabulary.....11

Generate a Vocabulary.....12

Use JSON to generate a vocabulary.....12

Use JSON Schema to generate a vocabulary.....17

Build a Vocabulary by hand 23

Extend a Vocabulary.....27

Enumerations.....27

Domains.....35

Rule scope and context.....37

Rule scope.....44

Aliases.....47

Scope and perspectives in the vocabulary tree.....48

How to use roles.....50

Rule writing techniques.....57

How to work with rules and filters in natural language.....57

Filters versus conditions.....60

Qualify rules with ranges and lists.....61

Ranges and lists in conditions and filters.....62

Ranges and value sets in condition cells.....63

How to use standard Boolean constructions.....76

How to embed attributes in posted rule statements.....76

How to include apostrophes in strings.....78

How to initialize null attributes.....78

How to handle nulls in compare operations.....78

Collections.....81

How Corticon Studio handles collections.....81

How to visualize collections.....82

A basic collection operator.....83

How to filter collections.....84

How to use aliases to represent collections.....84

Special collection operators.....92

Universal quantifier....93
Existential quantifier....95
Another example using the existential quantifier....99

Rules containing calculations and equations....107

Operator precedence and order of evaluation....108
Data type compatibility and casting....110
Data type of an expression....113
Defeating the parser....114
Manipulating JS datatypes with casting operators....115
Supported uses of calculation expressions....115
Calculation as a comparison in a precondition....117
Calculation as an assignment in a noncondition....118
Calculation as a comparison in a condition....118
Calculation as an assignment in an action....120
Unsupported uses of calculation expressions....120

Rule dependency in chaining....123

Filters....125

Full filters....127
Limiting filters....129
Filters that use OR....134
What is a precondition 134
Summary of filter and preconditions behaviors....138
Performance implications of the precondition behavior....138
How to use collection operators in a filter....140
Location matters....142
Multiple filters on collections....144

How to recognize and model parameterized rules....149

Parameterized rule where a specific attribute is a variable or parameter within a general business rule....149
Parameterized rule where a specific business rule is a parameter within a generic business rule....151

Logical analysis and optimization....153

Test, validate, and optimize your rules....153
Scenario testing....154
Rulesheet analysis and optimization....154
Traditional methods of analyzing logic....155
Flowcharts....156

Test suites....158
Validate and test Rulesheets in Corticon Studio....161
How to expand rules....161
The conflict checker....163
The completeness checker....167
Logical loop detection....173
Test rule scenarios in the Ruletest Expected panel....174
How to navigate in Ruletest Expected comparison results....174
Review test results when using the Expected panel....174
Techniques that refine rule testing....178
How to optimize Rulesheets....183
The compress tool....183
How to produce characteristic Rulesheet patterns....186
Compression creates subrule redundancy....189
Effect of compression on Decision Service performance....189
Precise location of problem markers in editors....190

Advanced Ruleflow techniques and tools....191

How to use a Ruleflow in another Ruleflow....191
How to generate Ruleflow dependency graphs....192

Troubleshooting Corticon.js Studio problems....197

Where did the problem occur....199
Use Corticon Studio to reproduce the behavior....199
Analyze Ruletest results....199
Trace rule execution....200
Identify the breakpoint....200
At the breakpoint....201
How to compare and report on Rulesheet differences....203

Appendix A: Customize Corticon.js Studio....207

1

Introduction to Corticon.js rule modeling

Corticon enables business users and domain experts to define rules for automating critical business decisions. Corticon has long been a leader in decision automation for Java and .NET applications. Corticon.js provides a new deployment option, JavaScript. With Corticon.js you can define rules and package them into fully self-contained JavaScript bundles which can be deployed to any compatible JavaScript platform. Example usages include:

- Rules deployed as serverless functions on AWS Lambda or Azure Functions
- Rules integrated into a cloud work flow such as AWS Step Functions, Google Cloud Functions, and Azure Flow
- Rules run on your own backend as part of your Node.js platform
- Rules bundled in a mobile app with Xamarin, React, Vue or other toolkit
- Rules executed in a browser as part of a web application

This guide introduces you to Corticon.js Studio for defining business rules and packaging them for deployment. Like Corticon, Corticon.js provides an easy to use spreadsheet metaphor for defining business rules as well as tools to define your rule vocabulary, rule flows and to test them. Corticon frees you from dependence on your IT department writing code for your automated business decisions.

The topics here are supported by guides to the rule modeling language and a quick reference for the Corticon.js Studio user interface. If you are new to Corticon, you will benefit from the Basic and Advanced Rule Modeling tutorials on the Corticon Information Hub.

2

Create the Vocabulary

Rule projects are based a vocabulary so you must create one, either by building one by hand, or by generating one. This section describes the concepts and purposes of a Corticon.js Vocabulary. You see how to create a Vocabulary from general business concepts and relationships.

For the rule modeler, the Vocabulary terms represent business objects, people, or other items. These could be customers, mortgage applications, purchase orders or any other thing that can automate decisions. Throughout the documentation, these things are referred to as *entities*, and properties or characteristics of these things are referred to as *attributes*.

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 be included in the Vocabulary.

For details, see the following topics:

- Generate a Vocabulary
- Build a Vocabulary by hand
- Extend a Vocabulary

Generate a Vocabulary

Overview

Corticon.js makes it easy to start your rule projects by letting you generate the Vocabulary directly from the JSON that your rules will process. This technique accelerates development, so that you can quickly get started writing rules, and ensures your vocabulary matches the JSON payloads that will be passed as input to your rules when deployed.

To generate a vocabulary, select a JSON file that is representative of the range of objects and fields (entities and attributes) that could be passed to your rules when deployed.

You need not be concerned if your JSON data model changes. Corticon.js lets you easily update your vocabulary by reimporting JSON, or by editing your Vocabulary by hand.

Note: JSON or JSON schema as a source?—JSON schema is more common when working with industry-standard data models, and has benefits because it more fully describes a data model, but JSON schema is not widely used. In most projects, all you will have is JSON, in which case, try to have JSON that represents all the entities and attributes that might occur in rule requests and output.

This topic has the following sections:

- Use JSON to generate a vocabulary on page 12
- Use JSON Schema to generate a vocabulary on page 17

Use JSON to generate a vocabulary

Create a Vocabulary from a JSON payload

Suppose you are writing rules for a B2B e-commerce application that will determine what, if any, discounts should be applied to an order. An order contains contact information about the customer, their partnership status ('elite' or 'standard') and the items in the order. Your rules will examine this information to determine a discount rate for the order in line with the promotions being offered by your company. For example, 'elite' customers might get 15% off on orders over $10,000.

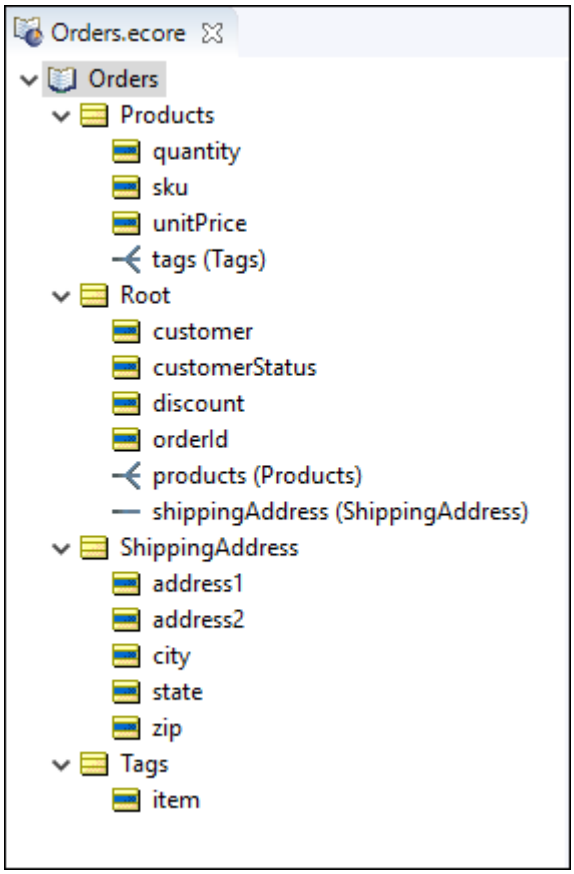
Working with IT, you've been supplied this sample JSON file representing an order. JSON in this format is used by other components of your e-commerce application:

```
{
    "orderId": 494748,
    "customer": "Acme Industries",
    "customerStatus": "elite",
     "shippingAddress": {
        "address1": "1234 Industrial Lane",
        "address2": null,
        "city": "Boston",
        "state": "MA",
        "zip": "01234"
    },
    "products": [
        {
            "sku": "XYZ-BB-43",
            "unitPrice": 2300.00,
            "quantity": 2,
            "tags": [
                "industrial",
                "compressor"
            ]
        }
    ],
    "discount": 0.0
}
```

To populate a Vocabulary from a JSON payload:

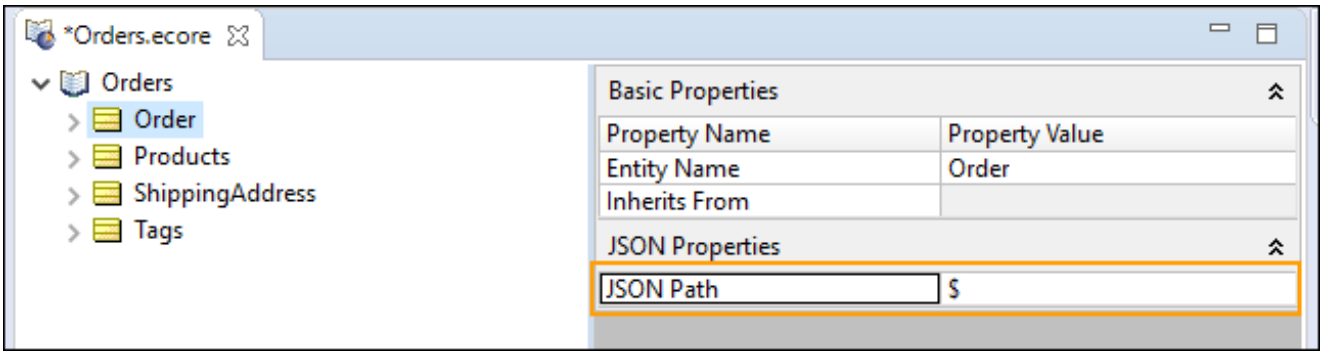
1. Copy the preceding JSON and then save in a temporary file.
2. In Corticon.js Studio, create a new Rule Project named `OnlineRetail`.
3. In the project, create a Vocabulary named `Orders`.
4. Click in the Vocabulary edit window, and then select **Vocabulary > Populate Vocabulary from JSON**.
5. Choose the temporary file with the JSON you saved, and then click **Open**.

The Vocabulary that the JSON generates is the following:

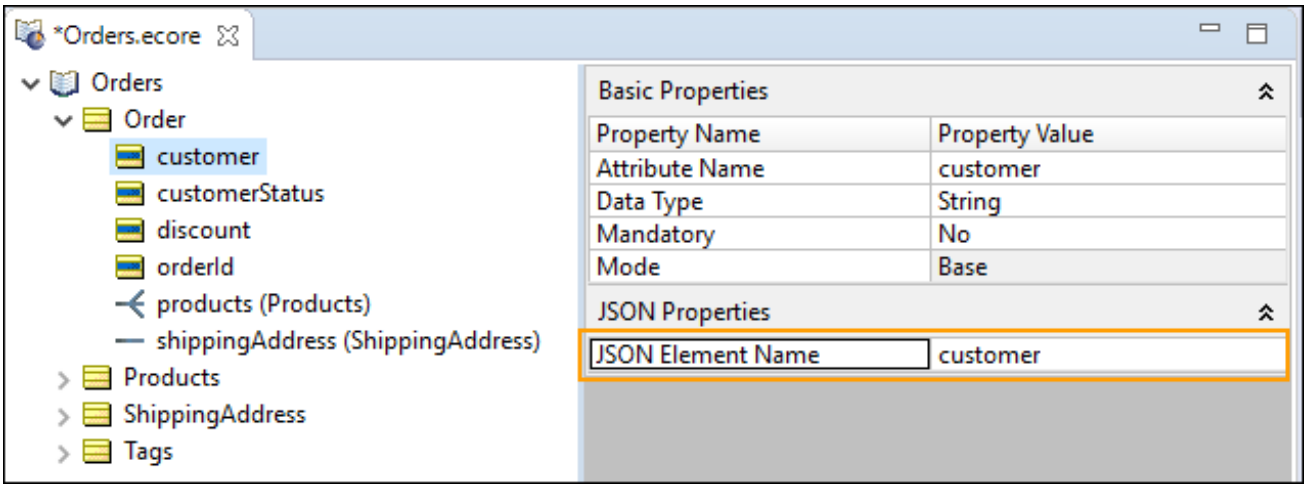


Let's take a closer look at the Vocabulary:

- **Root entity**—The JSON source has an object definition at root, indicated by the JSON starting with initial brace. You know this root entity is an order. Corticon does not know that, so it named the top-level entity `Root`. After vocabulary generation completes you can refactor the root entity name to `Order`:

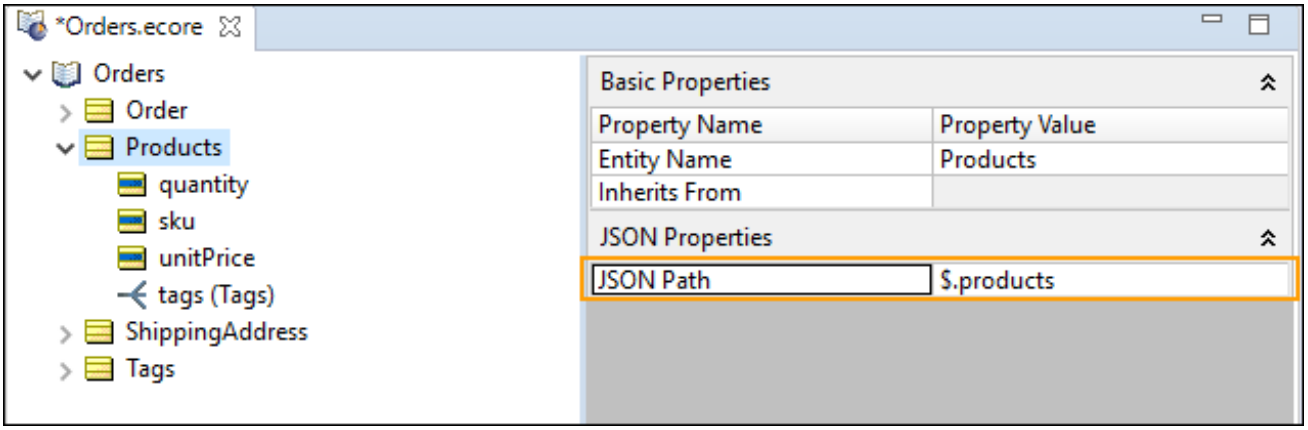


- **Attributes**—Each attribute takes the **JSON Element Name** that was in the source JSON. The root entity has five attributes that are added as attributes of `Root`. You can manually revise the data type as appropriate. This is the incoming payload identifier that will map to its Vocabulary attribute name:

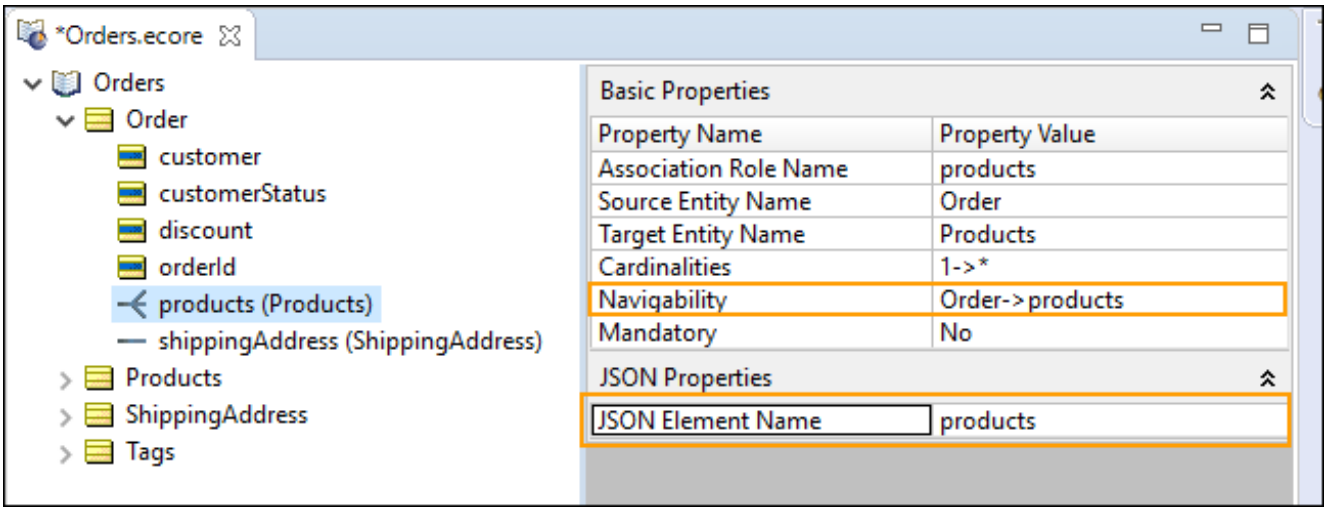


Note: If an attribute has a null value in the source JSON, the data type String is assumed.

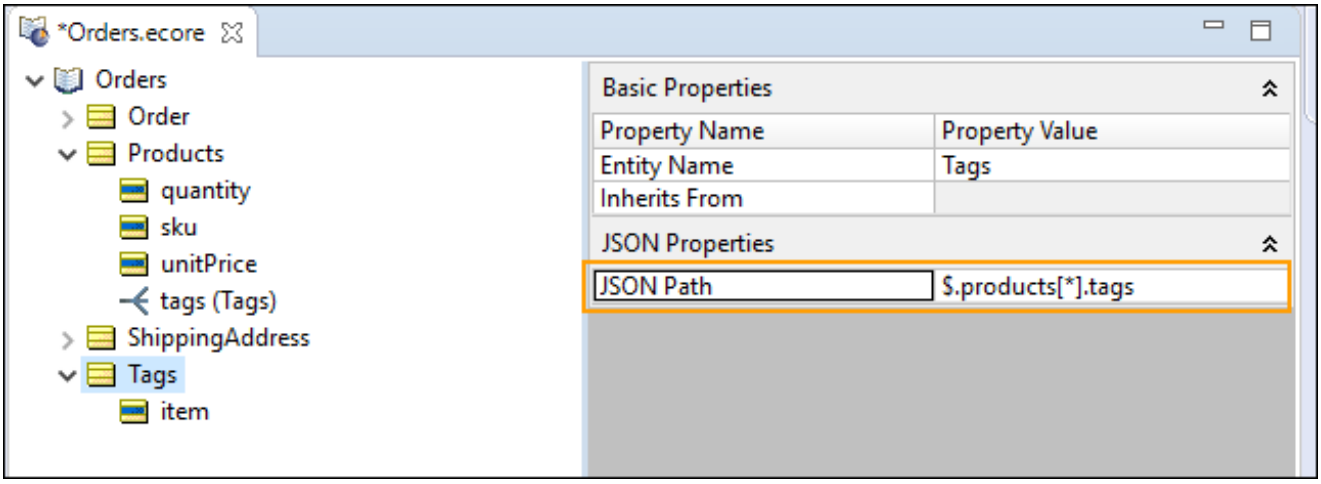
- **Non-root entities**—Other entities take the name in the source JSON, and specify their **JSON Path** as relative to the root:



- **Associations**: Corticon added the `Products` entity, and then added an association from `Root (Order)` to `products`:



- **Scalar arrays**—A scalar array is handled as an association from the entity with its own identifying Entity. The JSON Array's relationship shows that `products` is relative to root ($) and one or more `tags` are related to `products`:



Update a vocabulary from a JSON payload

Suppose your Sales department wants to enhance the discount program to provide an additional discount to government agencies and whether an order is marked for expedited handling. In support of this IT has provided an updated sample JSON the includes the new information.

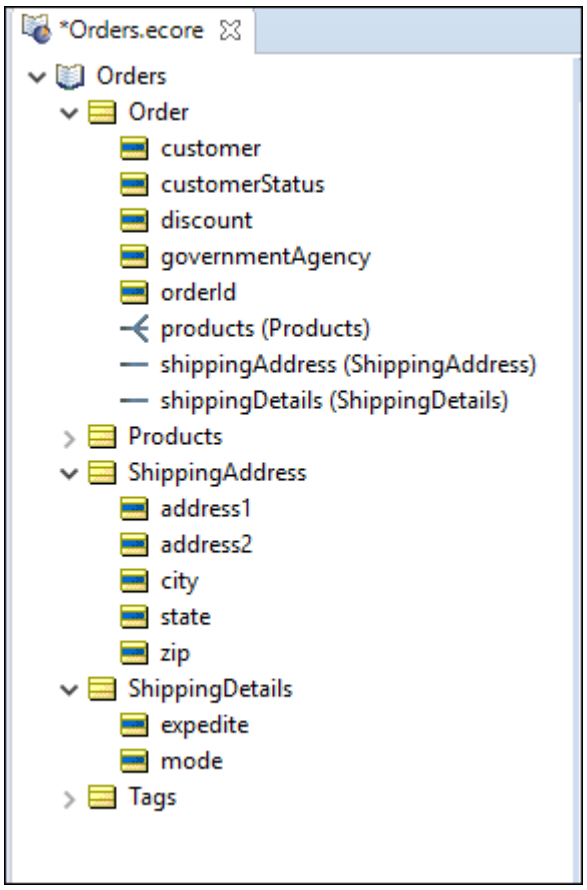
An update generates new entities, attributes, and associations. The existing entities, attributes, and associations are not revised by regenerating over the existing Vocabulary. If you want one element to be regenerated, delete it before you perform the update. You could even delete the vocabulary entirely, and then start fresh. The original sample payload adds a requirement for `Billing Address` to the `sampleCustomer` Vocabulary.

```
{
    "orderId": 494748,
    "customer": "Acme Industries",
    "customerStatus": "elite",
    "governmentAgency": false,

    "shippingAddress": {
        "address1": "1234 Industrial Lane",
        "address2": null,
        "city": "Boston",
        "state": "MA",
        "zip": "01234"
    },
    "shippingDetails": {
        "expedite": true,
        "mode": "ground"
    },

    "products": [
        {
            "sku": "XYZ-BB-43",
            "unitPrice": 2300.00,
            "quantity": 2,
            "tags": [
                "industrial",
                "compressor"
            ]
        }
    ],
    "discount": 0.0
}
```

When you *regenerate* your vocabulary from this JSON, it will add new entities, attributes and associations to your vocabulary for the new items in the JSON. The Vocabulary shows the added entity, attributes, and association:



Note: If you rename or refactor entity or attribute names, an update from the same source will generate duplicate entities and attributes for the ones you renamed in the Vocabulary. You will need to delete the duplicates.

Use JSON Schema to generate a vocabulary

Create a Vocabulary from a JSON schema

Suppose your company belongs to an industry consortium that has defined a standard format for JSON messages for communication between suppliers and customers. The consortium may opt to define a JSON schema for the JSON. JSON schema providers a greater ability to define valid content for JSON payloads.

The use of JSON schema is in the early days of being adopted. JSON Schema is primarily used when different organizations need a formal definition of an agreed upon data model. Using JSON schema has advantages for vocabulary generation such as options for defining enumerated values and for transcribing comments into the Vocabulary. Be careful: Some schemas are very large and have more than you need. You may want to cut the schema down to just what you need before generating the vocabulary.

Note: Corticon uses JSON Schema Draft-07 to infer the patterns in the given source—whether a JSON payload file or parsing a JSON schema file—to make its best effort to set up the entire Vocabulary complete with associations. You might be using a different draft. As the specification gets more refined, improvement are added to the schema.

- Sample JSON Schema on page 19
- To populate a Vocabulary from a JSON schema on page 20
- How Corticon generates a vocabulary from JSON on page 20
- How descriptions in your schema are handled on page 21
- How references in your schema are handled on page 22
- How enumerations in your schema are handled on page 21
- How to extend type definitions in your schema on page 22

Sample JSON Schema

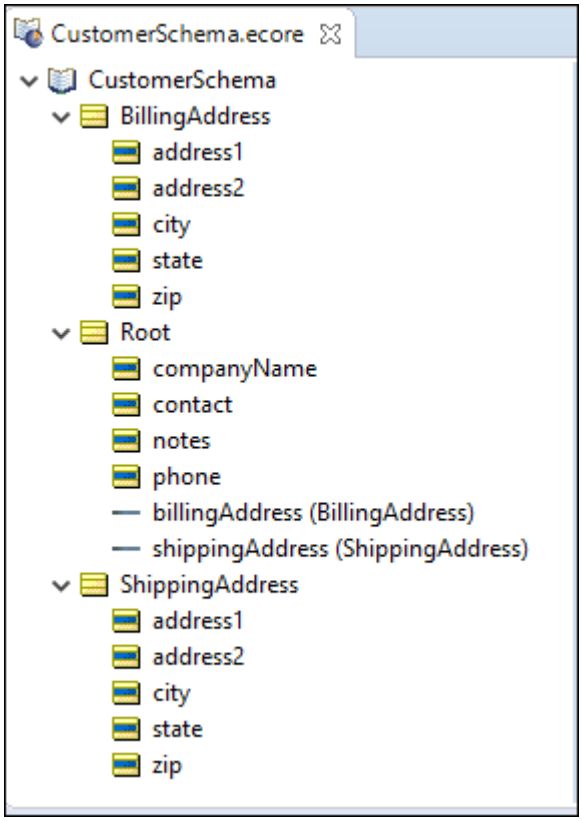
The following code is an example of a JSON schema:

```
{
   "$schema": "http://json-schema.org/draft-07/schema#",
   "type": "object",
   "properties": {
      "BillingAddress": {
         "description": "Address to where a Customer's invoice must go",
         "type": "object",
         "properties": {
            "Zip": {
               "type": "string"
            },
            "State": {
               "type": "string"
            },
            "Address2": {
               "type": "string"
            },
            "Address1": {
               "type": "string"
            },
            "City": {
               "type": "string"
            }
         }
      },
      "CompanyName": {
         "type": "string"
      },
      "Phone": {
         "type": "string"
      },
      "ShippingAddress": {
         "description": "Address to where a Customer's product must go",
         "type": "object",
         "properties": {
            "Zip": {
               "type": "string"
            },
            "State": {
               "type": "string"
            },
            "Address2": {
               "type": "string"
            },
            "Address1": {
               "type": "string"
            },
            "City": {
               "type": "string"
            }
         }
      },
      "Notes": {
         "type": "string"
      },
      "Contact": {
         "type": "string"
      }
   }
 }
```

To populate a Vocabulary from a JSON schema

1. Copy the preceding JSON and then save in a temporary file.
2. In Corticon.js Studio, create a new Rule Project named `CustomerSchema`.
3. In the project, create a Vocabulary named `CustomerSchema`.
4. Click in the Vocabulary edit window, and then select **Vocabulary > Populate Vocabulary from JSON**.
5. Select the sample file `CustomerSchema.json`, and then click **Open**.

The Vocabulary that the JSON schema generates is the following:



How Corticon generates a vocabulary from JSON

To generate a vocabulary from a JSON schema document, Corticon examines the contents of the document to identify the entities in the document, their attributes, and their associations. Where data types are not defined with JSON, Corticon infers the data type of attributes based on the values present.

The process of inferring the schema is essentially as follows:

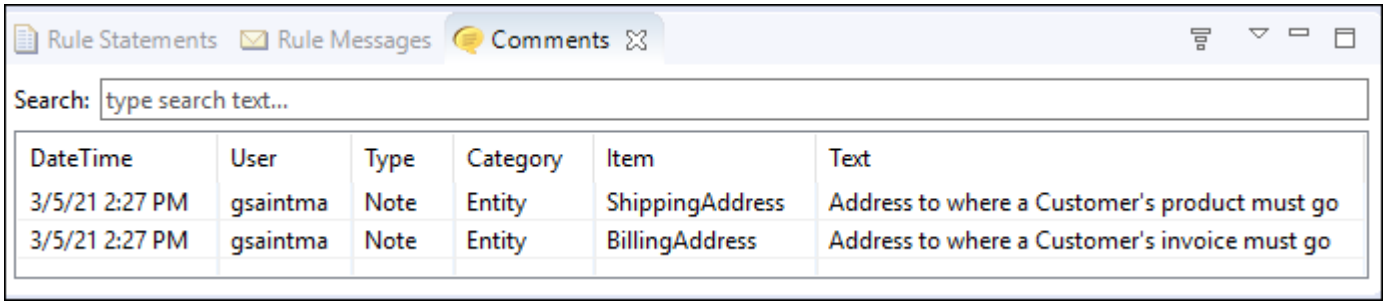
- **Entities**: Entity names follow Corticon naming conventions and uppercase the first character of the entity name.
  - The entity `Root` entity always generated.
  - If an existing entity has already been mapped to a JSON object, use that entity.
  - If no entity is found, then create a new entity, and set the entity name to the object name.
- **Attributes**: For each attribute in an Entity:
  - If an entity has no attributes, assign it one string attribute with the name `item`
  - Create a new attribute (no duplicate names including case) with attribute name in the Entity

- **Data type**
  - For a JSON schema where a data type is specified, use that data type.
  - For a JSON instance:
    - For a number that can be successfully converted to a relevant Java Date, set its data type as `DateTime`.
    - For a number with a decimal point, set its data type as `Decimal`.
    - For a number without a decimal point, set its data type as `Integer`.
    - For a string that is an ISO 8601 value, set its data type as `DateTime`, else it is a `String`.
    - For an attribute with a data type of null, it is a `String`.
    - For an empty array, it is a `String` array.

- **Associations**: Association role names are auto-assigned.
  - Arrays are specified as a one-to-many with its corresponding parent entity.
  - Associations are not be bidirectional.
  - Both ends are not mandatory.

How descriptions in your schema are handled

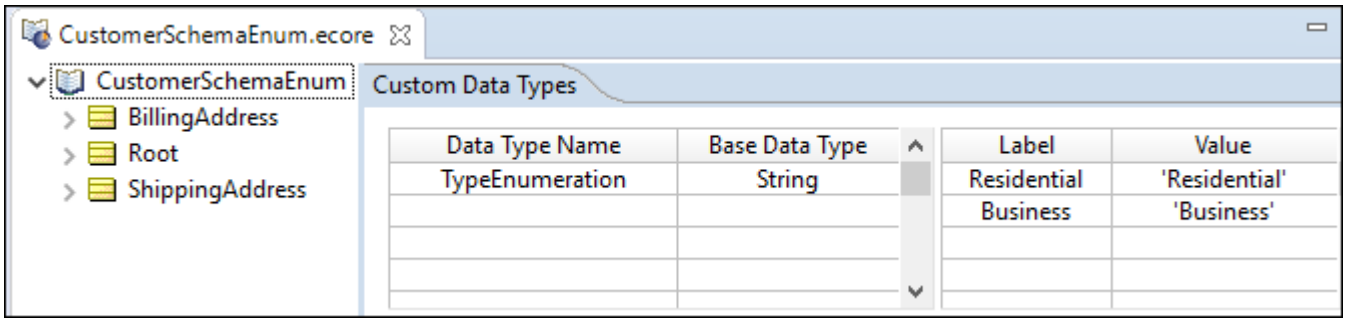
The JSON Schema specification has description attributes that can be used to document your data structure. The Vocabulary Generator puts the `description` fields in the schema into the Vocabulary's **Comments** tab, as shown:



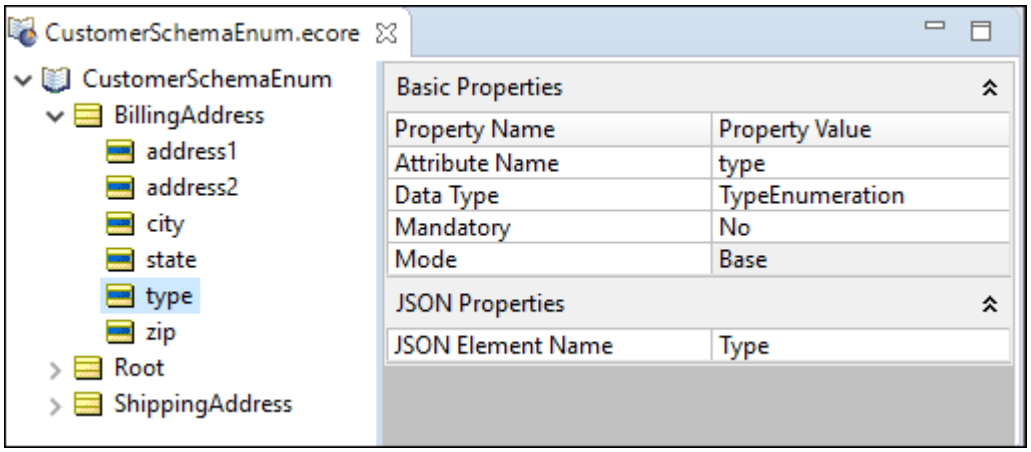
How enumerations in your schema are handled

The JSON Schema specification might have enumerations. When the Vocabulary Generator sees an `enum` tag, it creates a Custom Data Type of that enumeration and use that as the attributes data type.

When a schema with an `enum` populates the Vocabulary, it generates a custom data type:



The `type` attribute is still `String`, but its `Data Type` is now the custom data type `TypeEnumeration`, as shown:



How references in your schema are handled

The JSON Schema specification provides for the use of `$ref` attributes to have a single definition of an object that can then be incorporated elsewhere in the schema. An example is an `address` object defined once and included as part of `customer` and `supplier` objects in the schema.

When Corticon generates a vocabulary from JSON schema, associations will be added from the referring entity to the target entity. In the example, the generated vocabulary would contain `Customer`, `Supplier`, and `Address` entities. Corticon then adds associations from both `Customer` and `Supplier` entities to the `Address` entity.

How to extend type definitions in your schema

The JSON Schema specification allows you to specify different validation rules through the use of `oneOf`, `anyOf`, or `allOf` tags. For the most part, these tags do not effect vocabulary generation except when used to extend a type definition. In the following example, the `Type` enumeration was added to the `address` definition because it is needed for `ShippingAddress`. However, it is not needed for other types of addresses, so does it make sense to include it, optionally, in all addresses? This is where the `allOf` tag comes in handy. You can use it to extend the address type only for the `ShippingAddress`. A schema fragment that uses `allOf` is shown:

```
...
      "ShippingAddress": {
         "description": "Address to where a Customer's product must go",
         "allOf": [
            { "$ref": "#/definitions/address" },
            { "properties":
               { "type":
                  {
                  "title": "Address Type Enumeration",
                 "description": "Specifies if the address is a Business or Residence",

                  "enum": [ "residential", "business" ]
```

Note: Get the complete extend sample.

The difference in the vocabulary generated by this schema and the previous one is that the type attribute will only be in the `ShippingAddress` entity and not the `BillingAddress` entity.

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.js Studio.

Follow these steps to create a Corticon.js Vocabulary:

- Step 1: Design the Vocabulary on page 23
- Step 2: Identify the terms on page 24
- Step 3: Separate the generic terms from the specific on page 24
- Step 4: Assemble and relate the terms on page 24
- Step 5: Diagram the Vocabulary on page 25
- Step 6 Modeling the Vocabulary in Corticon.js Studio on page 26

Step 1: Design the Vocabulary

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. Step 1: Design the Vocabulary on page 23 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

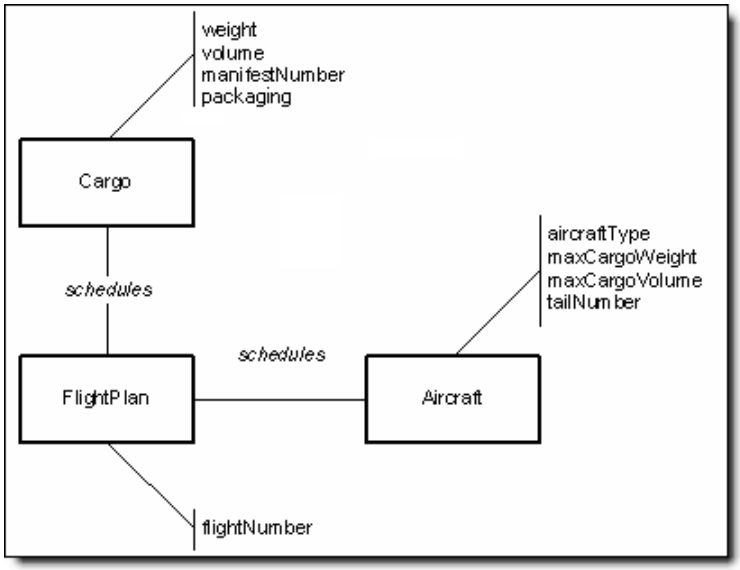
Attributes: weight, volume, manifest number, packaging

Entity: FlightPlan

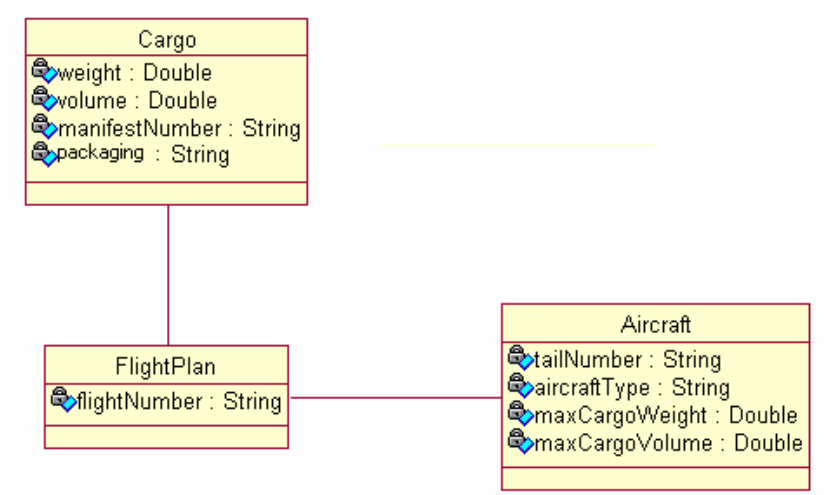
Attributes: flight number

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 Modeling the Vocabulary in Corticon.js Studio

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

In Corticon.js, choose **New > Rule Project** to create a Rule Project. Click that Rule Project, and then choose **New > Vocabulary**. Then create the entities, attributes, and associations that were defined in the diagram.

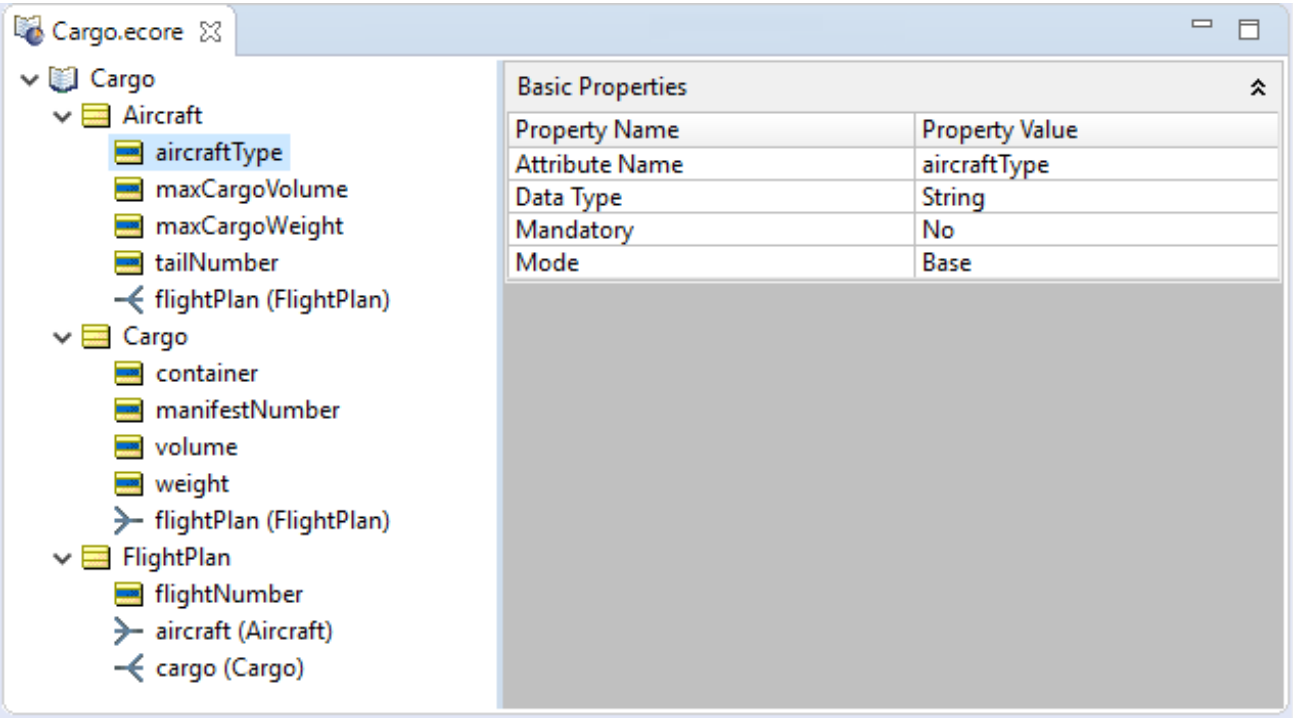
Note: See *"Vocabularies" in the Quick Reference Guide* for complete details on building a Vocabulary.

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

- **Attributes**: Each attribute must have a data type that is: **String**, **Boolean**, **DateTime**, **Integer** or **Decimal.**
- **Associations**:
  - **Role names**: Relationships between entities have role names that are assigned when you build associations in the Vocabulary Editor. A default role name mimics the entity name, with the first letter in lowercase.
  - **One-way directionality**: Associations between entities are directional (one way).
  - **Cardinality**: Associations have cardinality, which indicates how many instances of a given entity can be associated with another entity.

The Vocabulary must contain all of the entities and attributes needed to build rules, a process that is typically iterative, with Vocabulary changes made as the rules are built, refined, and tested.

Figure 1: Vocabulary Window in Corticon.js Studio



-->

Extend a Vocabulary

When creating a vocabulary, you can define enumerations to represent valid choices of values and use domains to segment your vocabulary into logical namespaces.

Enumerations

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

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

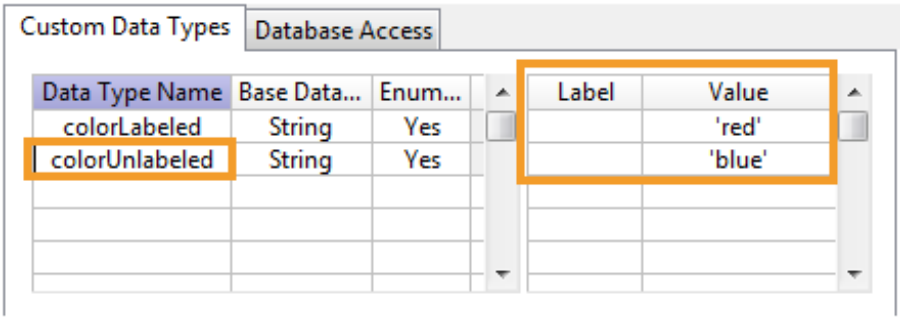
How enumeration labels and values behave

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

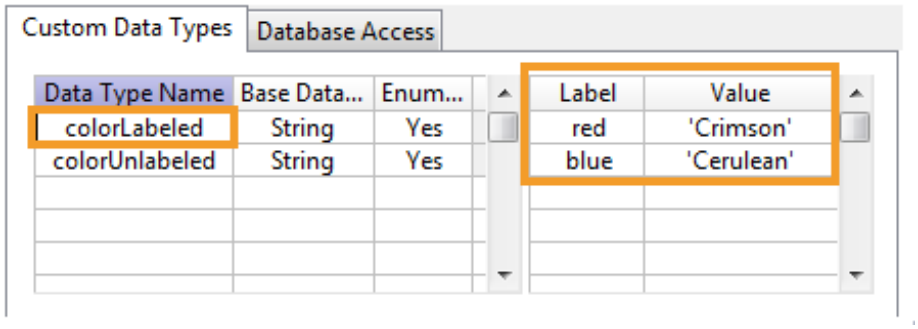
Note: It is important that you determine whether you want to use labels, because changing a set of enumerations later to add or remove the labels data will affect any Rulesheets and Ruletests that use that custom data type's enumerations as you can observe in this topic.

At the Vocabulary root, you created a String enumeration with only values. The base data type can be any Corticon.js 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.

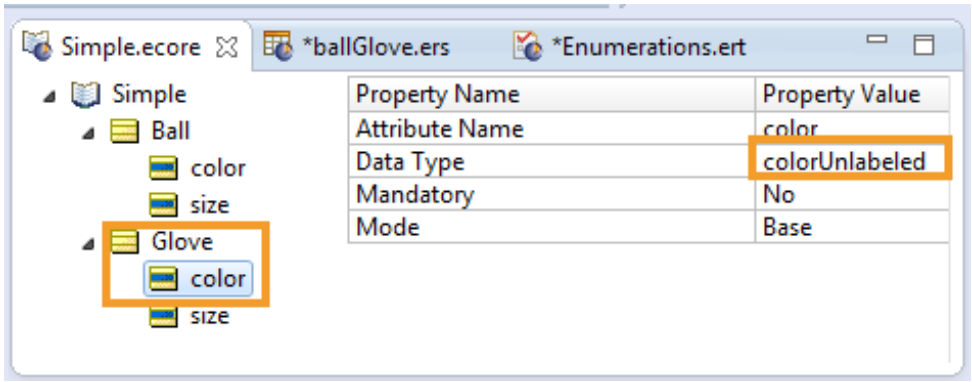
The following examples are String values. They can contain spaces and most other characters. It needs to be set off in plain single quotation marks. If you enter or paste text with the delimiters, they are added for you. Like this:



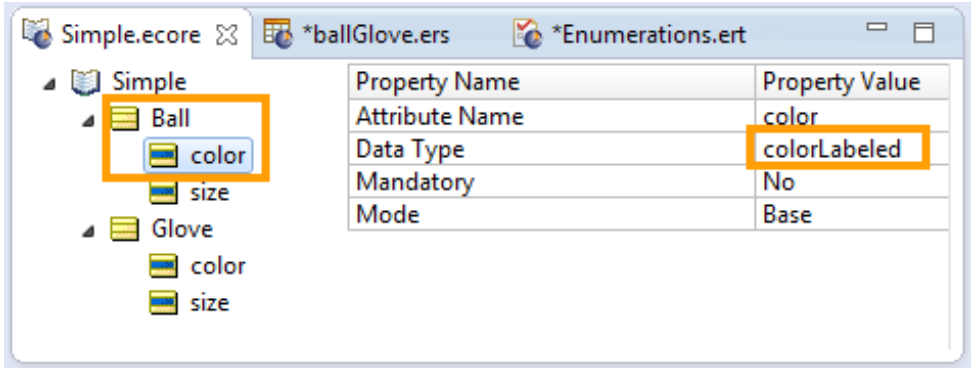
If you want to use labels, then the label is always a String of any alphanumeric characters but cannot contain spaces. Each must be unique and must have a corresponding value. Even when you use labels, the values must be unique.



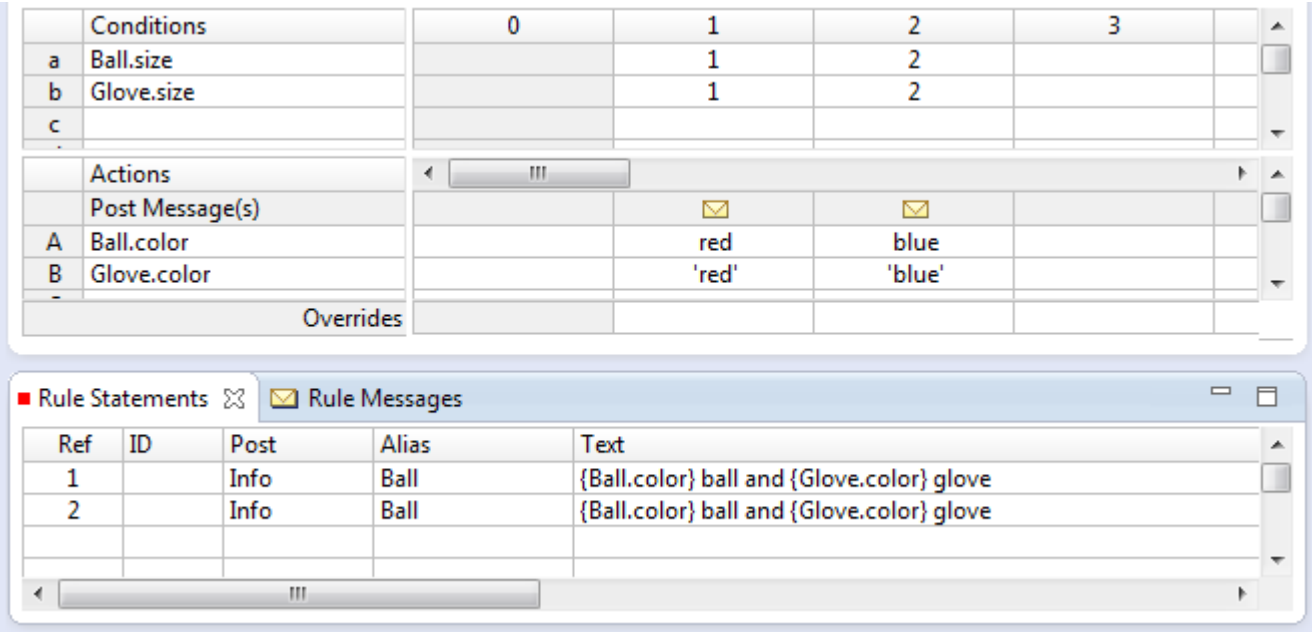
Set `Glove.color` to use the `colorUnlabeled` data type:



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

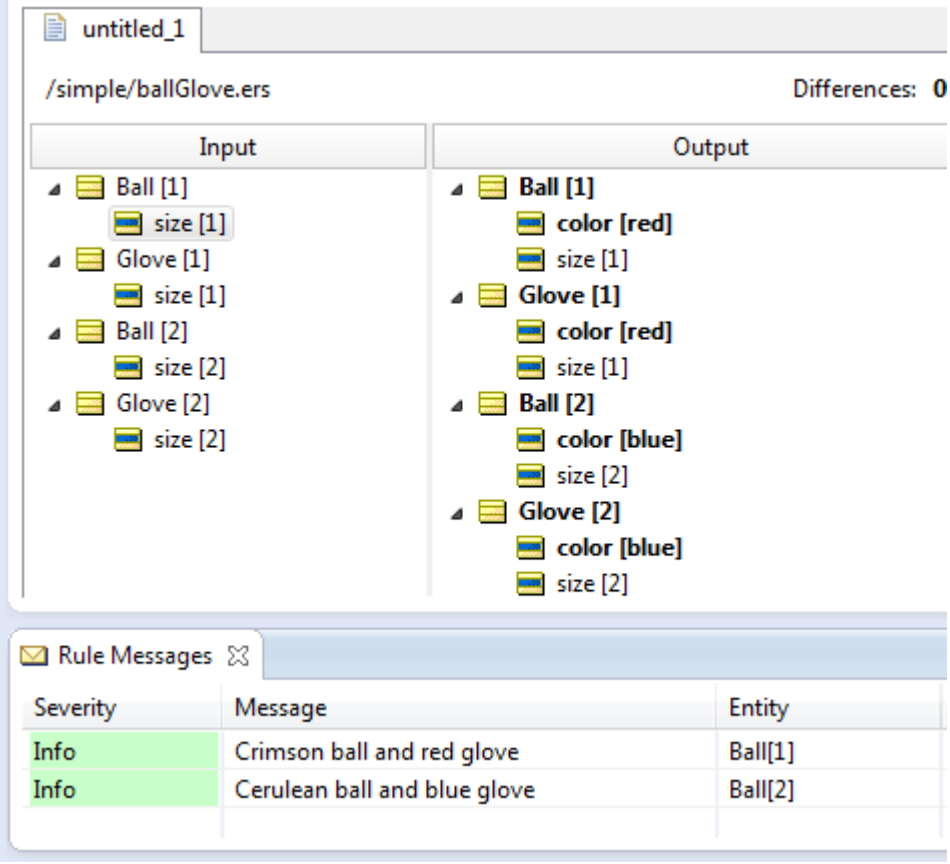


When you create a Rulesheet, the list offered at A1 contains the label (`Ball.color = red`) , while the list offered at B1 contains the value in qoutes (`Glove.color='red'`).

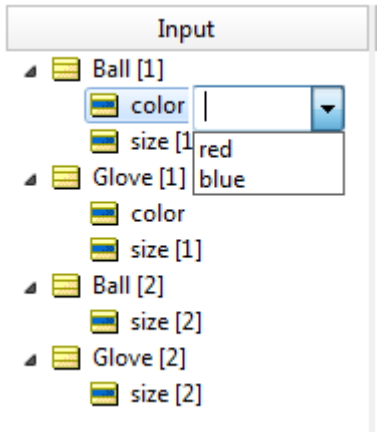


You add Rule Statements so that you can see how the labeled and unlabeled items are handled.

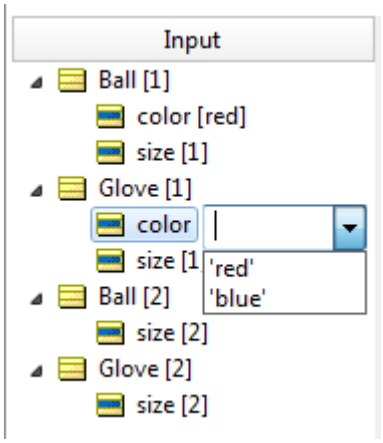
In a simple Ruletest, add some size tests to see what happens. As shown, the labels and values in the resulting **Output** are both unquoted. The **Rule Messages** tab displays the value when the label was in use and the value of the value-only enumeration.



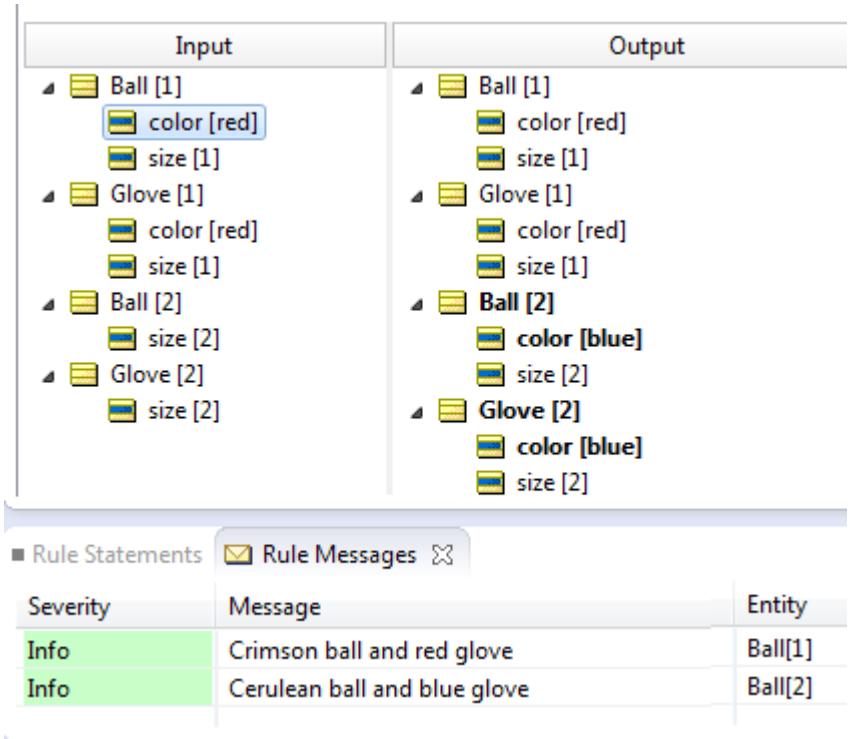
Entry of test values in the Ruletest list the label+value's label:



The value-only list has quoted values:



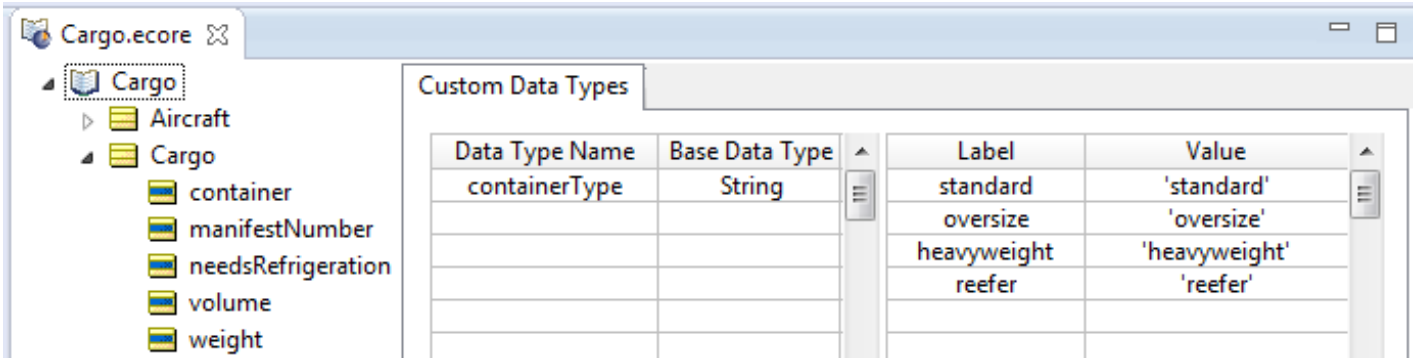
Both are reconciled to unquoted values in the displayed **Input** and **Output** columns:



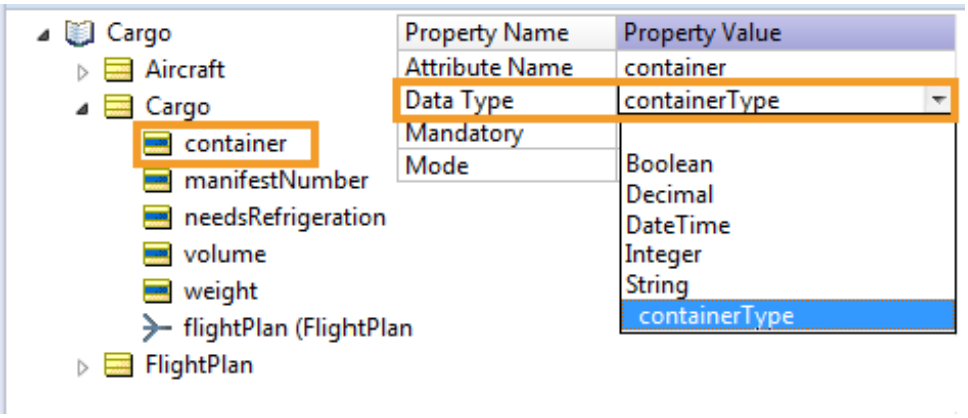
Note: It is important that you determine in each custom data type whether you want to use labels. Some enumerations can have labels while others do not. Changing a set of enumerations later, to add or remove the labels data, affects any Rulesheets and Ruletests that use that custom data type's enumerations 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, 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, base data types must be enclosed in single quote characters.

Here are some examples of enumerated custom data types:

Figure 2: Custom Data Type, example 1

Custom Data Types

Data Type Name	Base Data Type	Label	Value
containerType	String	standard	'standard'
PrimeNumbers	Integer	oversize	'oversize'
USHolidays2020	DateTime	heavyweight	'heavyweight'
ShirtSize	Integer	reefer	'reefer'
RiskProfile	Integer		
DevTeam	String		

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

Figure 3: Custom Data Type, example 2

Custom Data Types

Data Type Name	Base Data Type	Label	Value
containerType	String		2
PrimeNumbers	Integer		3
USHolidays2020	DateTime		5
ShirtSize	Integer		7
RiskProfile	Integer		11
DevTeam	String		13

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

Figure 4: Custom Data Type, example 3

Custom Data Types

Data Type Name	Base Data Type	Label	Value
containerType	String	NewYear	'1/1/2020 00:00:00'
PrimeNumbers	Integer	MemorialDay	'5/25/2020 00:00:00'
USHolidays2020	DateTime	IndependenceDay	'7/4/2020 00:00:00'
ShirtSize	Integer	LaborDay	'9/7/2020 00:00:00'
RiskProfile	Integer	ThanksgivingDay	'11/26/2020 00:00:00'
DevTeam	String	ChristmasDay	'12/25/2020 00:00:00'

`USHolidays2020` is a DateTime-based, enumerated custom data type with Label/Value pairs.

Figure 5: Custom Data Type, example 4

Custom Data Types

Data Type Name	Base Data Type	Label	Value
containerType	String	S	1
PrimeNumbers	Integer	M	2
USHolidays2020	DateTime	L	3
ShirtSize	Integer	XL	4
RiskProfile	Integer	XXL	5
DevTeam	String		

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

Figure 6: Custom Data Type, example 5

Custom Data Types

Data Type Name	Base Data Type	Label	Value
containerType	String	Low	1
PrimeNumbers	Integer	Medium	2
USHolidays2015	DateTime	High	3
ShirtSize	Integer	VeryHigh	4
RiskProfile	Integer		
DevTeam	String		

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

Figure 7: Custom Data Type, example 6

Custom Data Types

Data Type Name	Base Data Type	Label	Value
containerType	String		'John'
PrimeNumbers	Integer		'Jim'
USHolidays2020	DateTime		'Kendall'
ShirtSize	Integer		'Eric'
RiskProfile	Integer		'Cheryl'
DevTeam	String		'George'
			'Vidhi'
			'Suvasri'
			'Thierry'
			'Sravanthi'

Custom Data Types

Data Type Name	Base Data Type
containerType	String
PrimeNumbers	Integer
USHolidays2020	DateTime
ShirtSize	Integer
RiskProfile	Integer
DevTeam	String

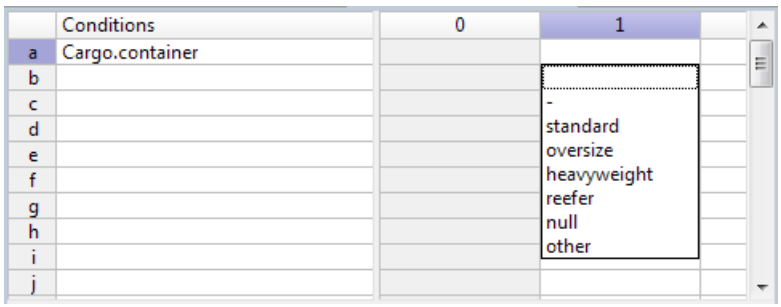
`DevTeam` is a String-based, enumerated custom data type with Value-only set members.

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

Use enumerated Custom Data Types in Rulesheets

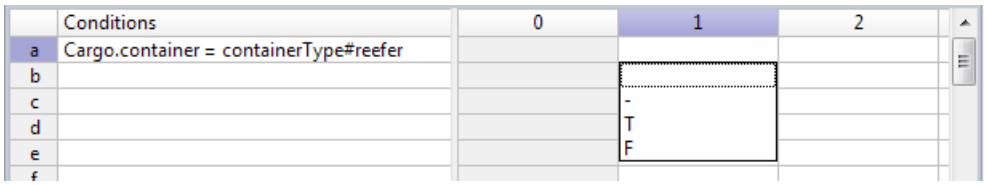
After an enumeration is defined and assigned to an attribute, its labels are displayed in selection drop-down lists in both Conditions and Actions expressions, as shown. If **Labels** are not available (because **Labels** are optional in an enumerated custom data type's definition), then **Values** are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

Figure 8: 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 9: Using # tag to test a custom data type

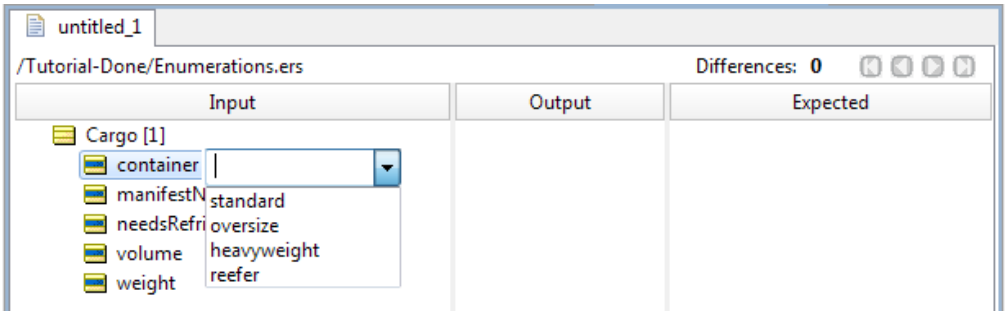


Note: Using a dot instead of a # tag works, but if there is custom data type with the same name as an entity, then the expression will be invalid.

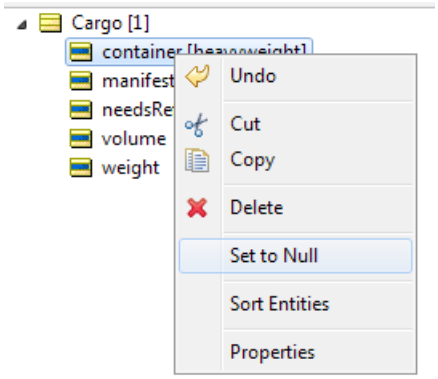
Use enumerated Custom Data Types in Ruletests

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

Figure 10: 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:



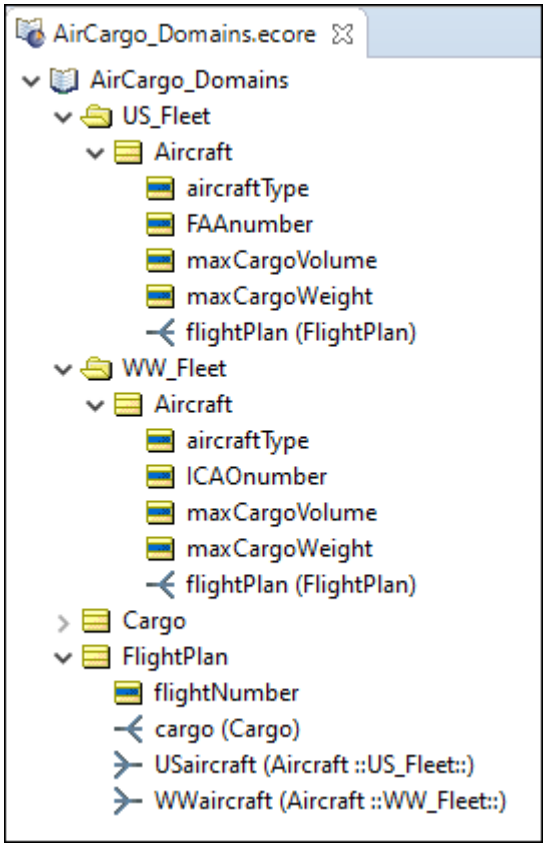
Domains

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 11: Using domains in the Vocabulary>

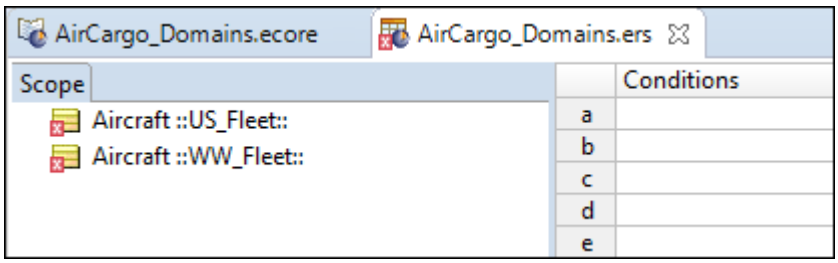


Notice that the entity `Aircraft` appears in each Domain, using the same spelling and containing slightly different attributes (`FAAnumber` vs. `ICAOnumber`). Notice, too, that the association role names from `FlightPlan` to `Aircraft` were named manually to ensure uniqueness: one is now `USaircraft`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

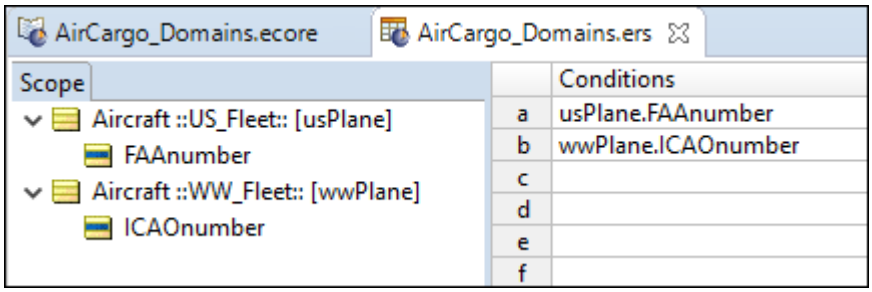
Figure 12: Non-unique Entity names prior to defining Aliases



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:

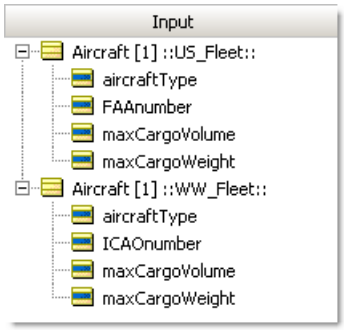
Figure 13: Non-unique Entity names after defining Aliases



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 14: Domains in a Ruletest



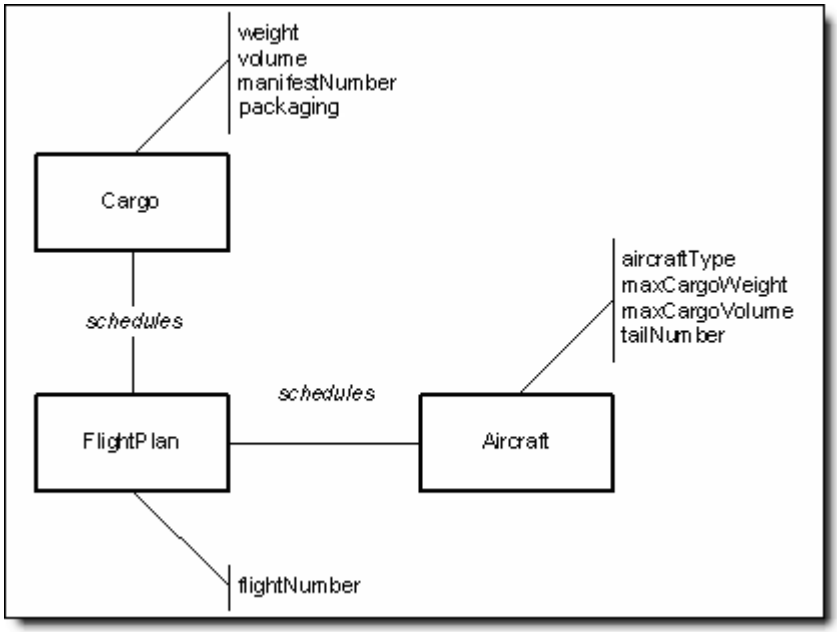
3

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 15: 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.

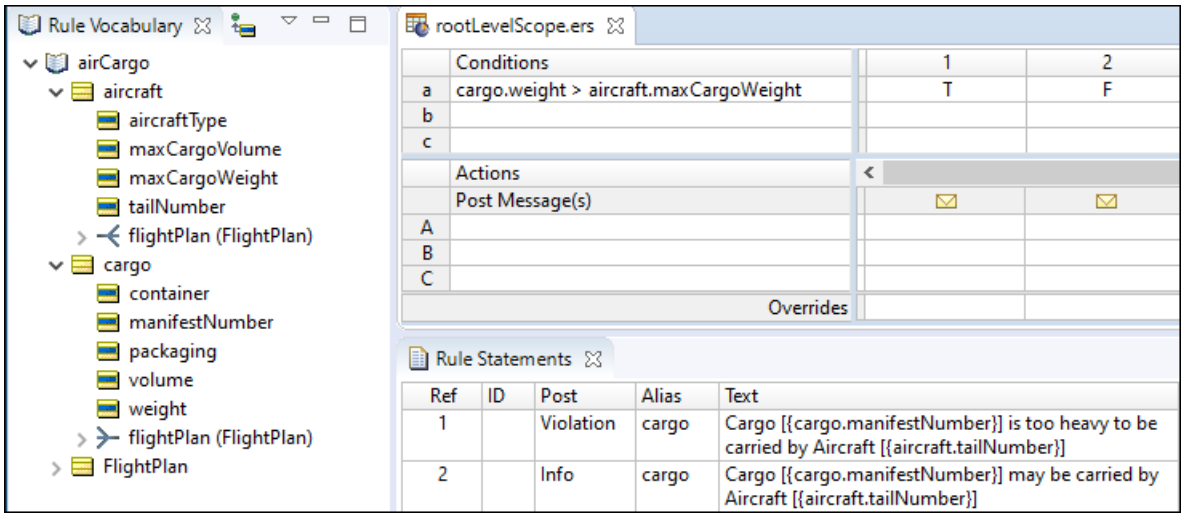
Also recall that the company wants to build a system that automatically checks flight plans to ensure no scheduling rules or guidelines are violated. One of the many business rules that need to be checked by this system is:

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.

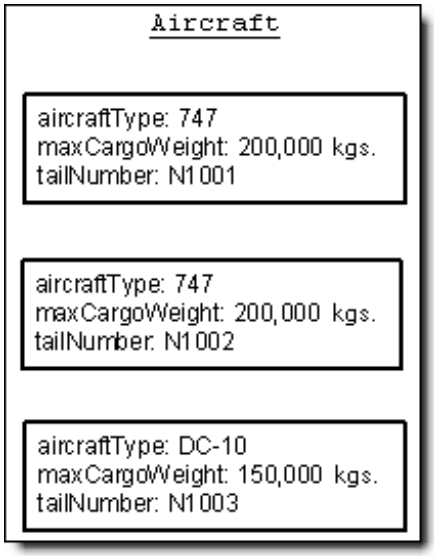
Figure 16: Expressing the rule using root-level Vocabulary terms



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

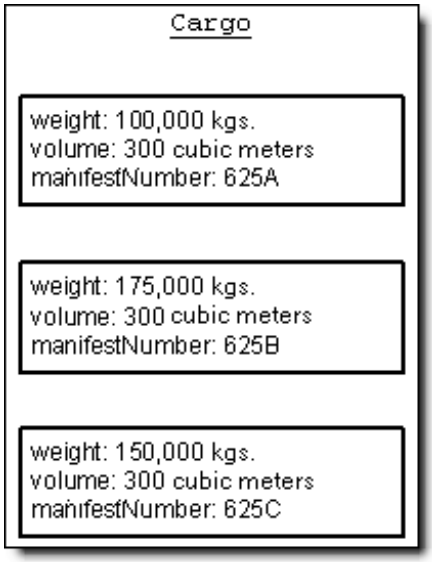
The company owns 3 Aircraft, 2 747s and a DC-10, each with different tail numbers. Each box represents a real-life example (or *instance*) of the `Aircraft` term from your Vocabulary.

Figure 17: The Cargo Company's 3 Aircraft



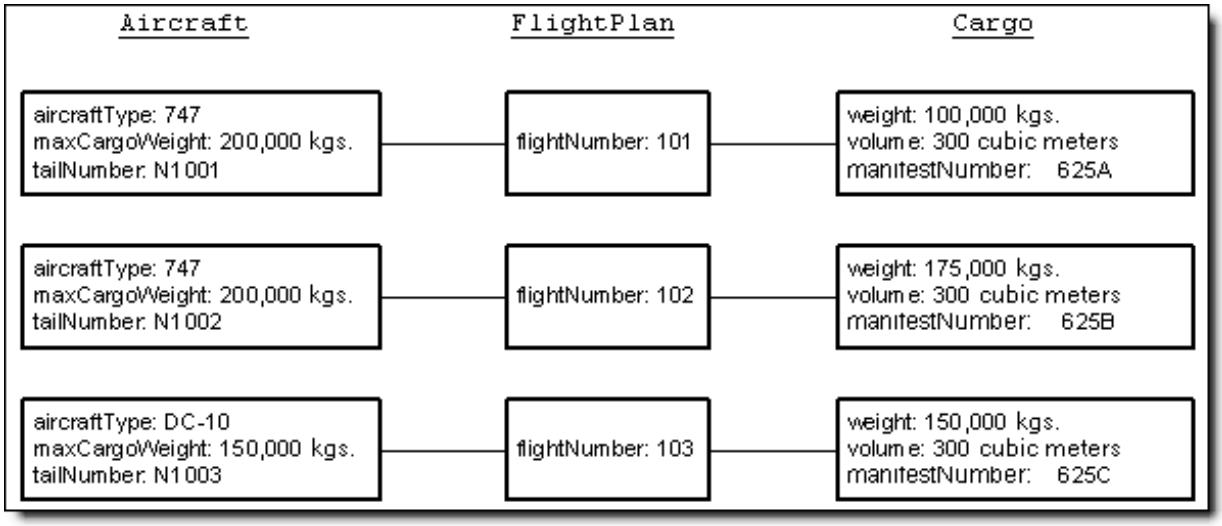
These aircraft give the company the ability to schedule 3 cargo shipments each night. Another business rule is implied:– “an `Aircraft` cannot be scheduled for more than one flight per night”. This rule is not addressed now because it is not relevant to the discussion}. On a given night, the cargo shipments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

Figure 18: The three cargo shipments for the night of June 25th



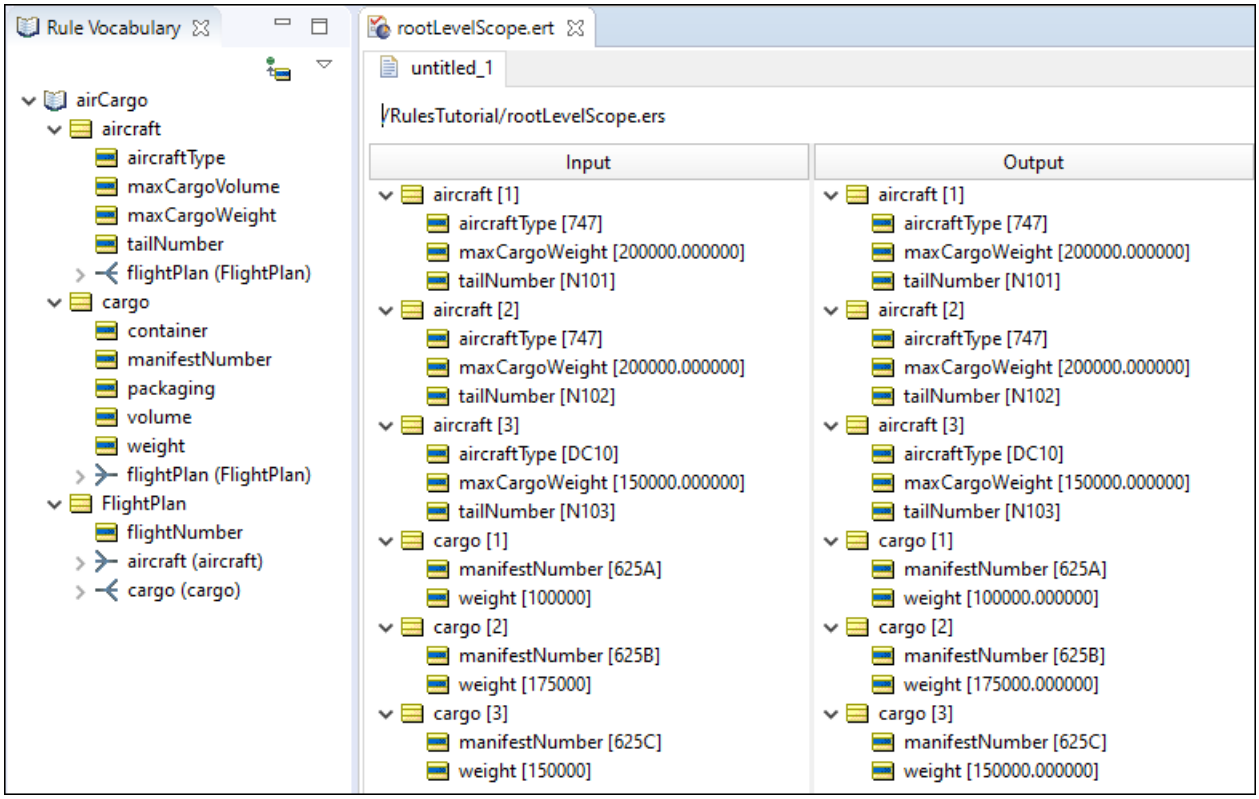
Finally, the sample business process manually matches specific aircraft and cargo shipments together as three flight plans, as shown below. This organization of data is consistent with the structure and constraints implicit in the Vocabulary.

Figure 19: The three FlightPlans with their related Aircraft and Cargo instances



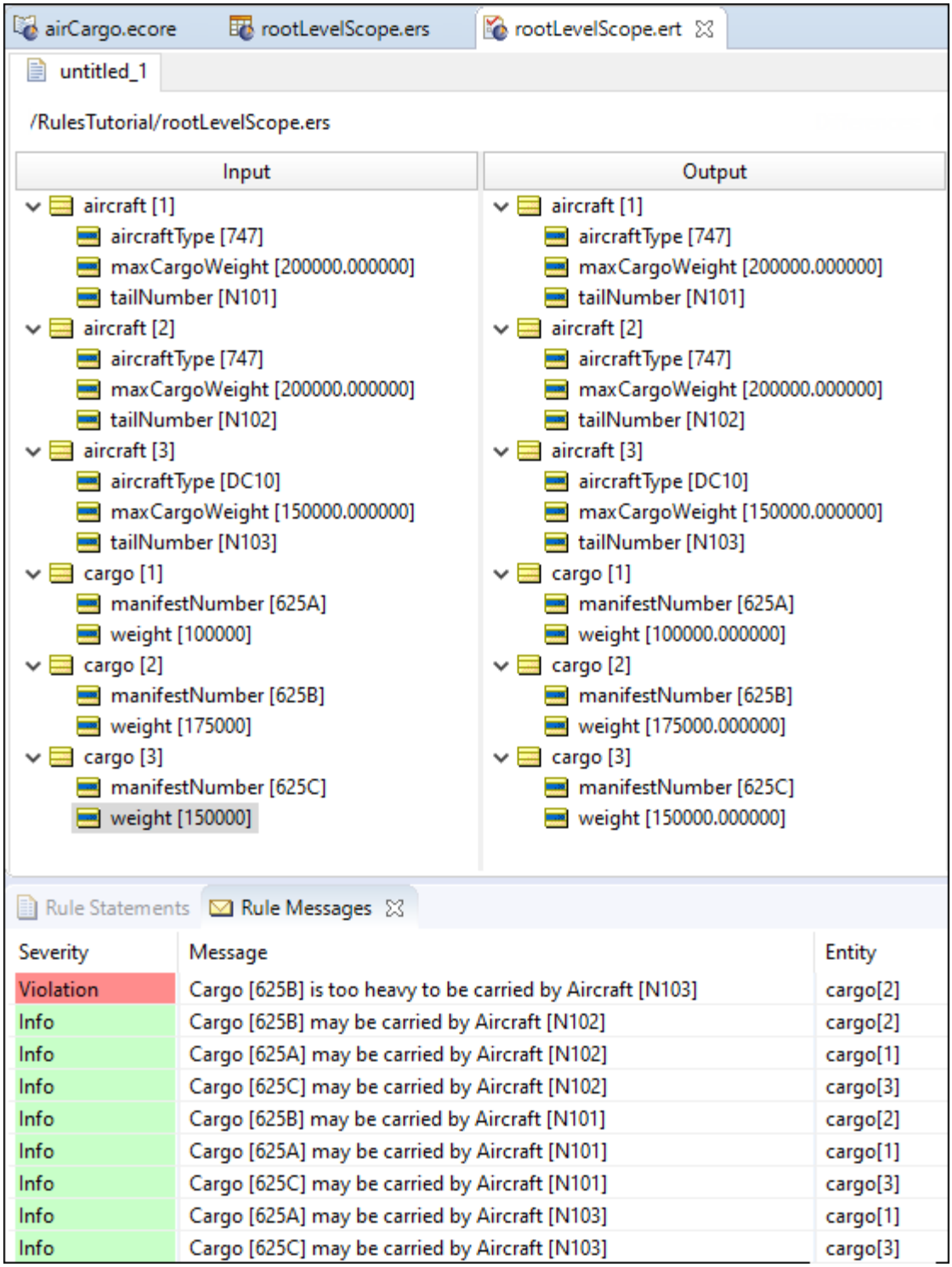
You can construct a Ruletest (in the following figure) so that the company's actual data is evaluated by the rule. Because the rule used root-level Vocabulary terms in its construction, you use root-level terms in the Ruletest:

Figure 20: Test the rule using root-level Vocabulary terms



Running the Ruletest:

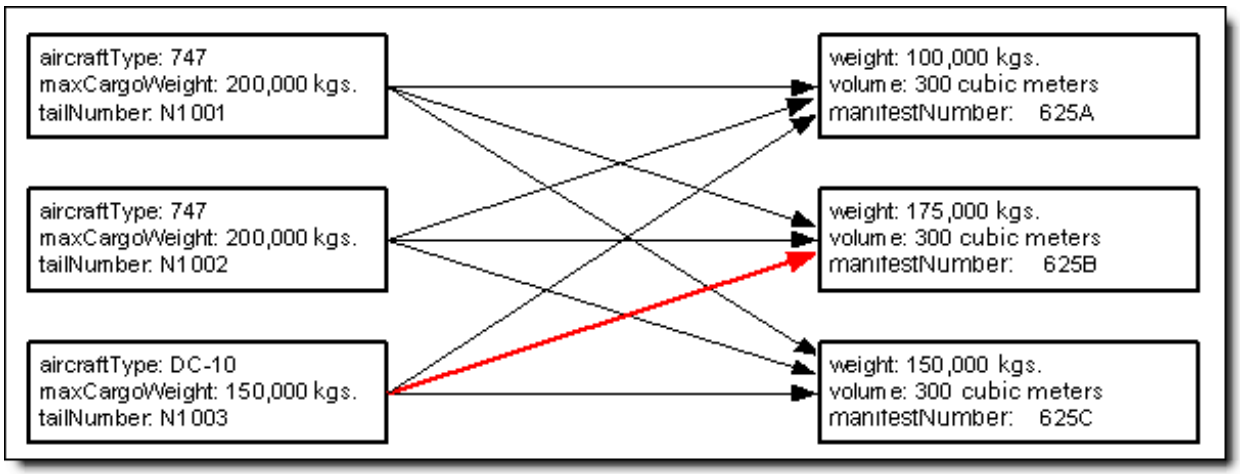
Figure 21: Results of the Ruletest



Note the messages returned by the Ruletest. Recall that the intent of the rule is to verify whether a given `Flightplan` is in violation by scheduling a `Cargo` shipment that is too heavy for the assigned `Aircraft`. You already know that there are only three `Flightplans`. And you also know, from examination of Figure 19: The three FlightPlans with their related Aircraft and Cargo instances on page 40, 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 22: All possible combinations of Aircraft and Cargo

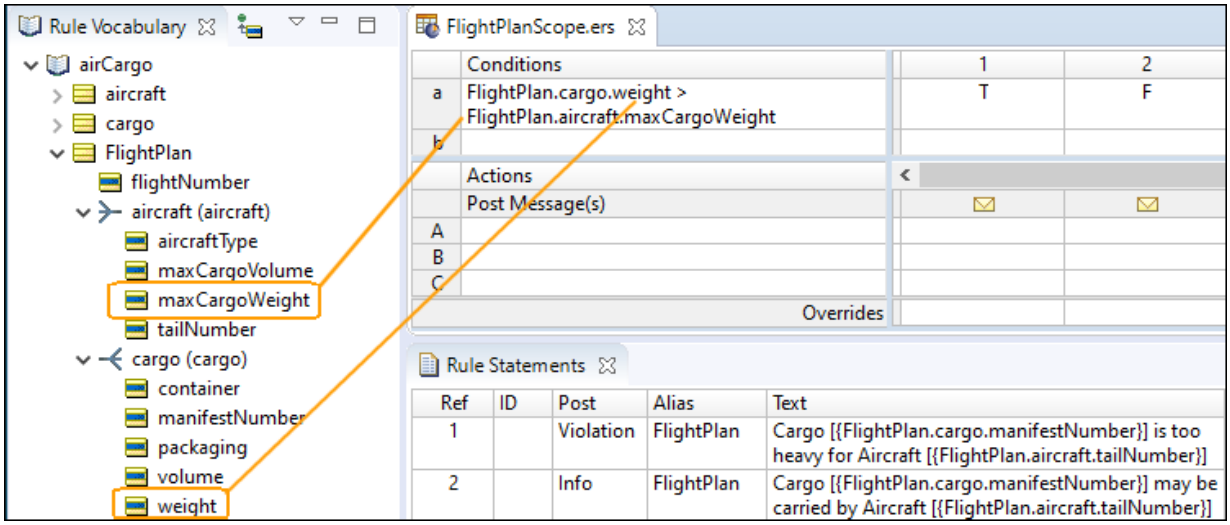


One pair, the combination of manifest `625B` and plane `N1003` (shown as the red arrow in the preceding figure), is illegal, because the plane, a `DC-10`, can only carry `150,000` kilograms, while the cargo weighs `175,000` kilograms. But, this pairing does not correspond to any of the three `FlightPlans` 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 16: Expressing the rule using root-level Vocabulary terms on page 38, 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 23: 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

Rule scope

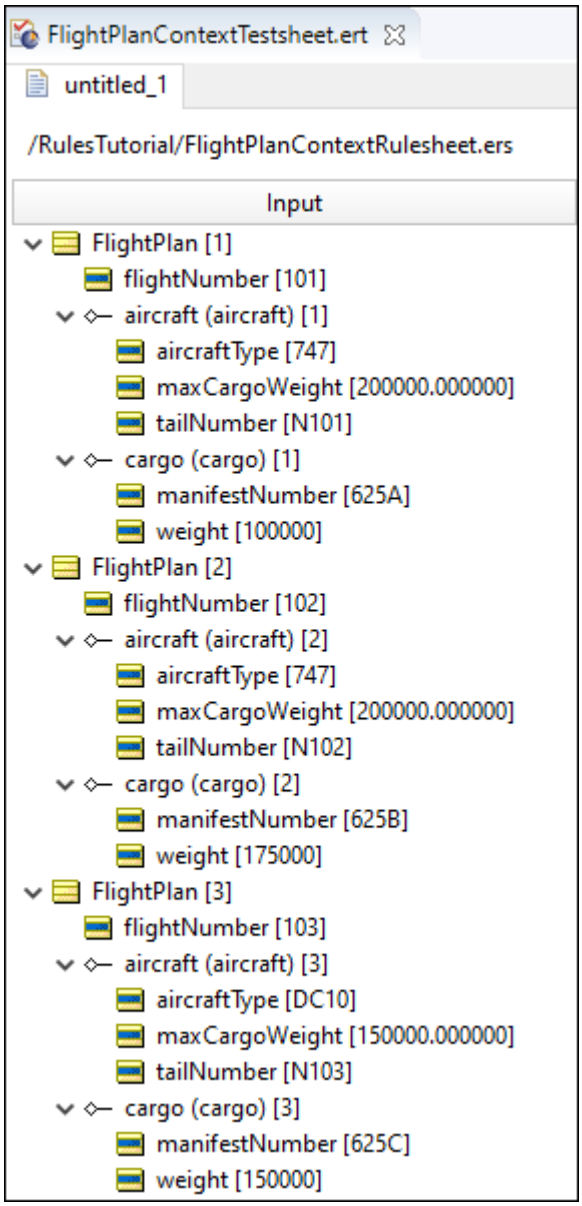
Because the rule evaluates both `Cargo` and `Aircraft` in the context of `FlightPlan`, the rule has *scope*, which means that *the rule evaluates only that data which matches the rule's scope*. This has an interesting effect on the way the rule is evaluated. When the rule is executed, its scope ensures that the rules evaluates only to 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.js 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 19: The three FlightPlans with their related Aircraft and Cargo instances on page 40 shows the possibilities. The rule evaluates these combinations and identifies any violations.

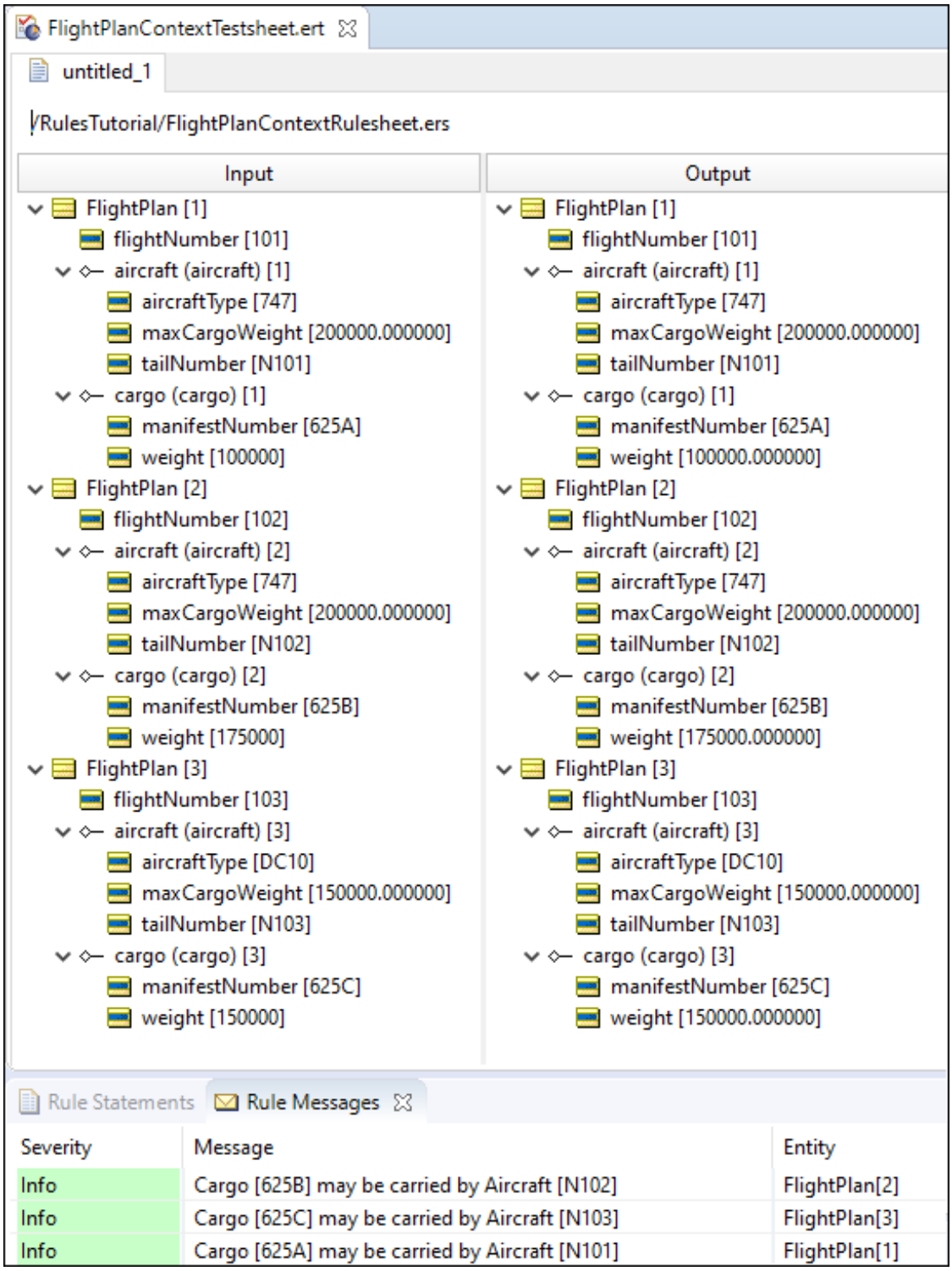
Figure 24: New Ruletest using flight plan as the rule scope



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 25: 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

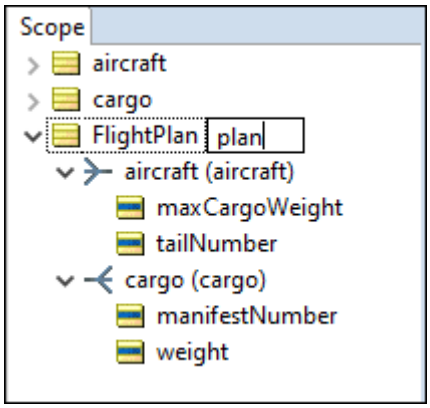
To clean up and simplify rule expression, Corticon.js Studio allows you to declare *aliases* in a Rulesheet. Using an alias to express scope results in a less cluttered Rulesheet.

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:

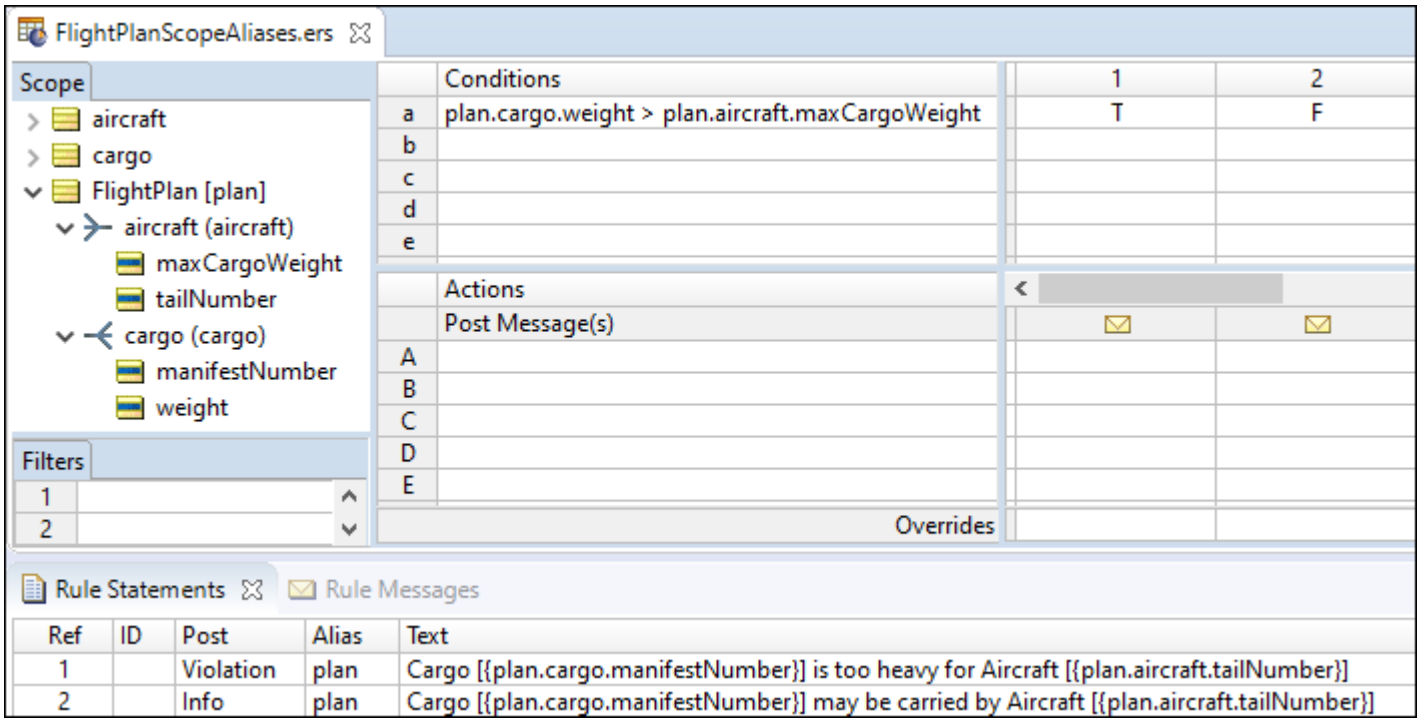
Figure 26: Defining an alias in the Scope window



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 27: 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`.

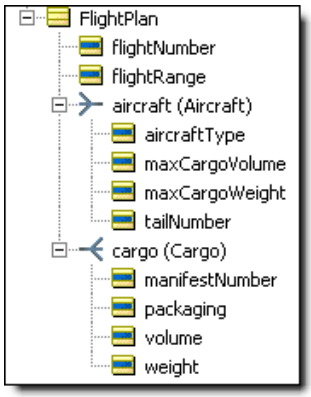
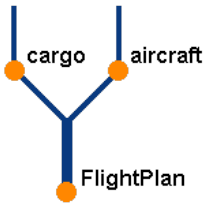
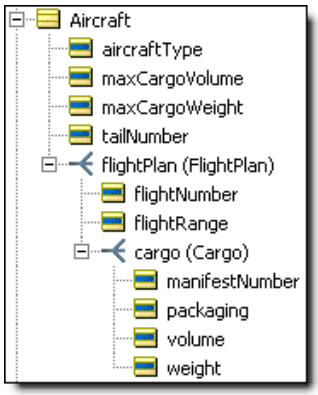
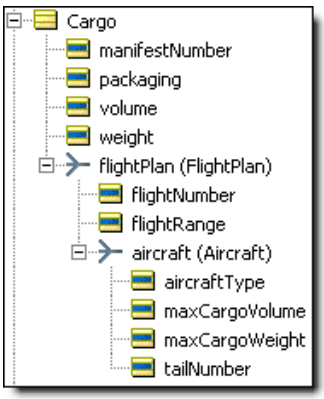
Aliases work in all sections of the Rulesheet, including the **Rule Statement** section. Modifying an alias name defined in the **Scope** section causes the name to update everywhere it is used in the Rulesheet.

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.js 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
	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.	
	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.	
	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.	

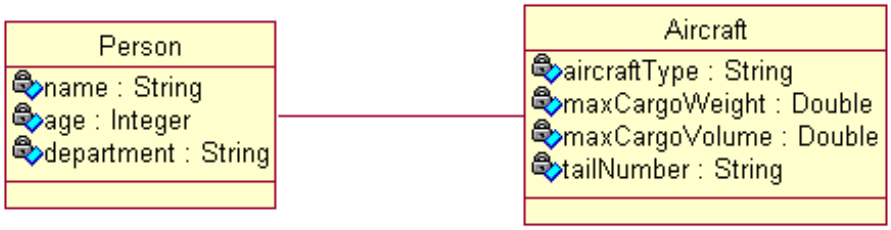
Scope can also be thought of as hierarchical, meaning that a rule written with scope of `Aircraft` applies to all root-level `Aircraft` data. And other rules using some piece (or branch) of the tree beginning with the root term `Aircraft`, including `Aircraft.flightPlan` and `Aircraft.flightPlan.cargo`, 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 28: UML Class Diagram without Roles



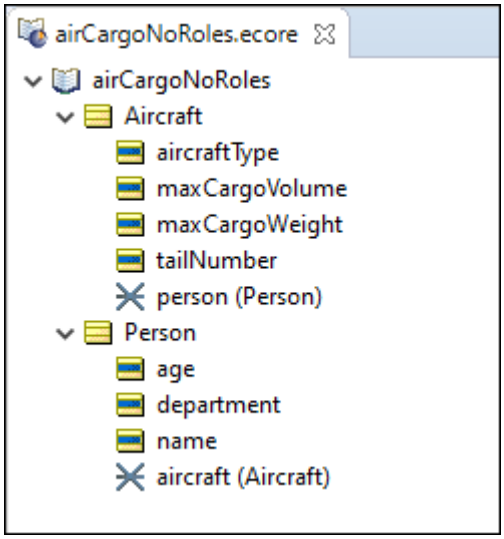
As shown in this class diagram, the entities `Person` and `Aircraft` are joined by an association. However, can this single association sufficiently represent multiple relationships between these entities? For example, a prior Fact Model might state that “a pilot flies an aircraft” and “a passenger rides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

Figure 29: Vocabulary without roles



However, there are several ways to modify this Vocabulary to allow you to implement these rules.

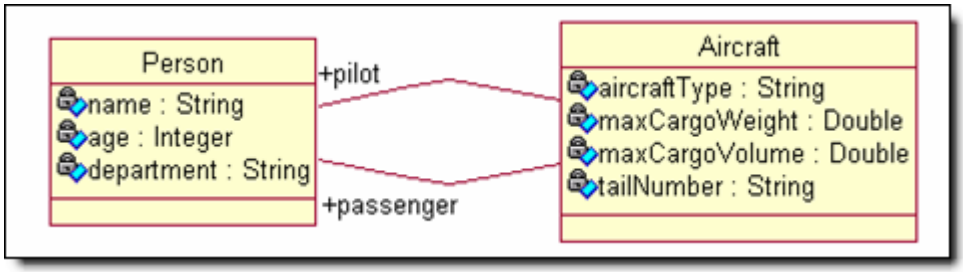
Add an attribute to Person

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

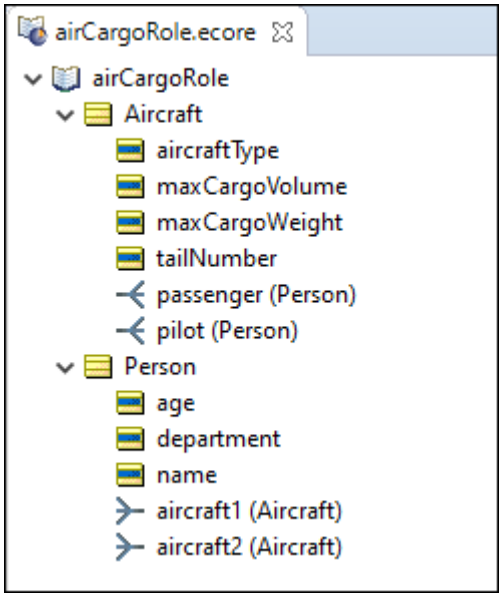
A role is a noun that labels one end of an association between two entities. For example, in our `Person-Aircraft` Vocabulary, the `Person` may have more than one role, or more than one kind of relationship, with `Aircraft`. An instance of `Person` may be a `pilot` or a `passenger`; each is a different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

Figure 30: UML class diagram with roles



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

Figure 31: Vocabulary with roles

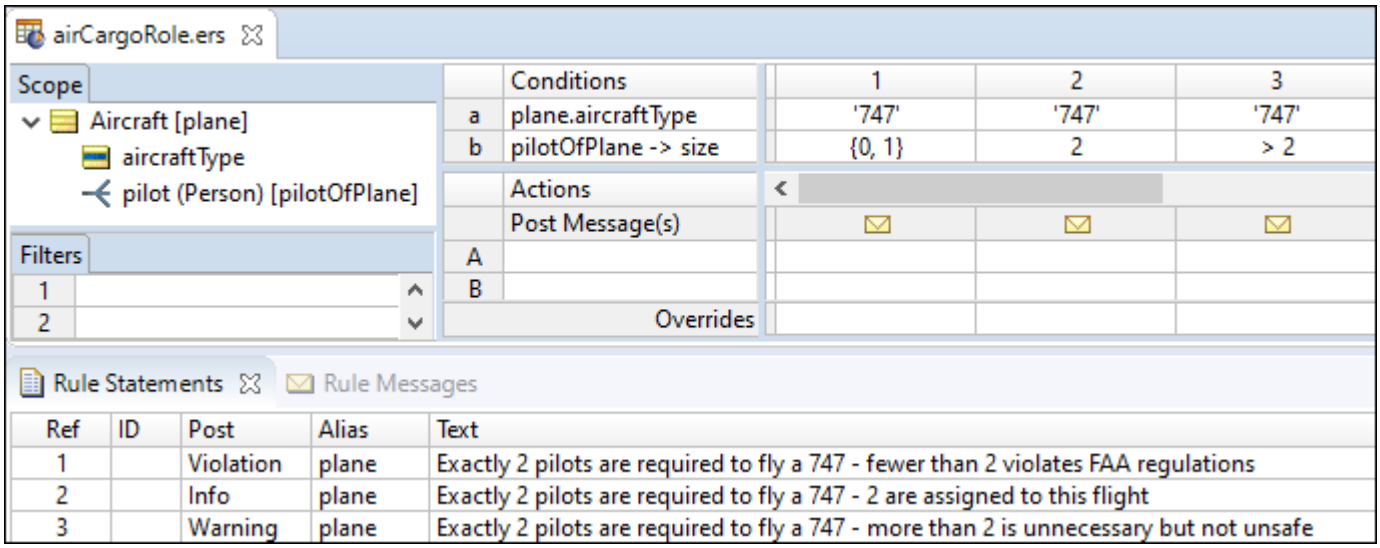


Notice the differences between the two preceding Vocabularies In **Vocabulary with Roles**, `Aircraft` contains 2 associations, one labeled `passenger` and the other `pilot`, even though both associations relate to the same `Person` entity. Also notice that the cardinalities of both `Aircraft`–`Person`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

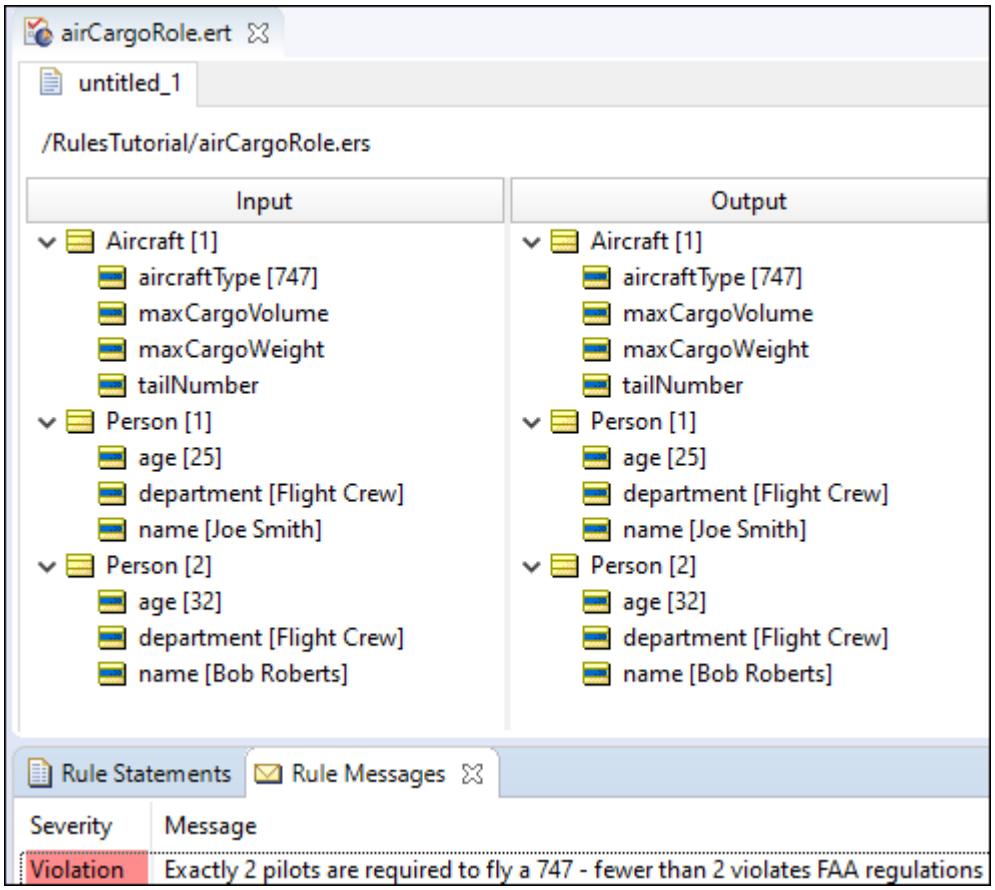
- Use of aliases for `Aircraft` and `Aircraft.pilot` (`plane` and `pilotOfPlane`, respectively). Aliases are just as useful for clarifying rule expressions as they are for shortening them.
- The rule **Conditions** evaluate data within the context of the `plane` and `pilotOfPlane` aliases, while the **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 81.

Figure 32: Rule #1 implemented using roles



To demonstrate how Corticon.js 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`.

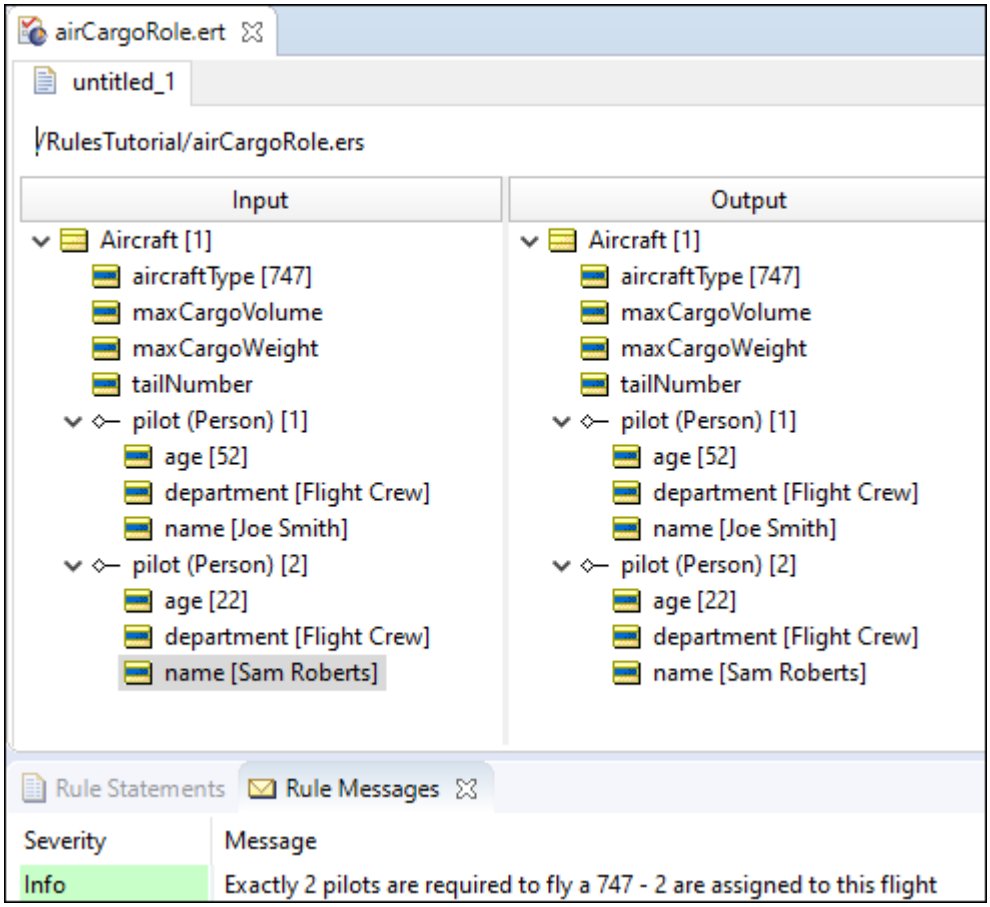
Figure 33: Ruletest with no Person entities in pilot role



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.

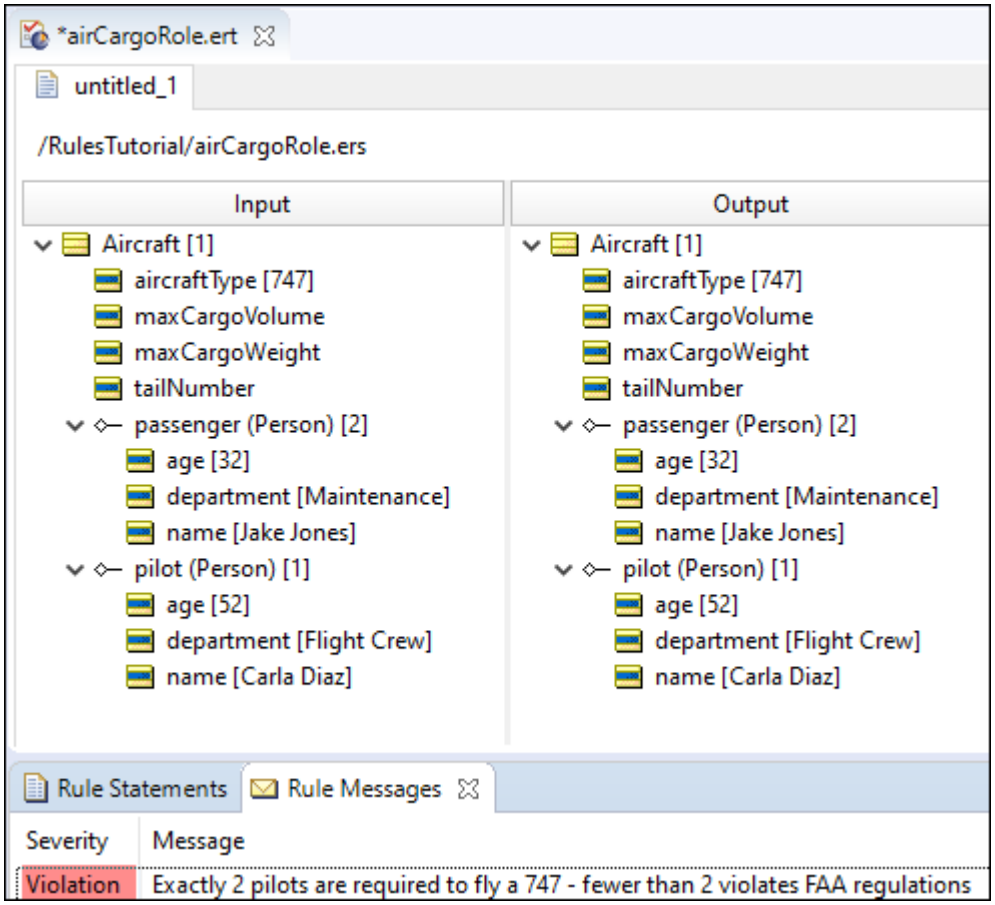
If you create a new Input Ruletest, then this time with both persons in the role of pilot, you see a different result, as shown:

Figure 34: Ruletest with both Person entities in role of pilot



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

Figure 35: 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.

4

Rule writing techniques

The Corticon.js 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
- How to initialize null attributes
- How to handle nulls in compare operations

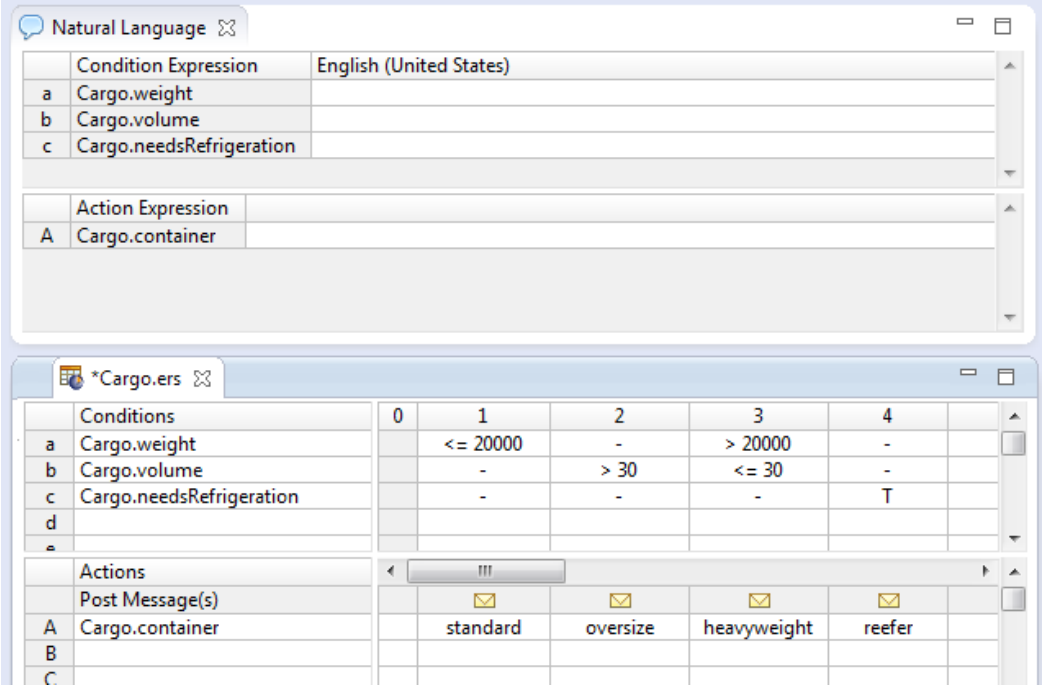
How to work with rules and filters in natural language

Progress Corticon.js 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:

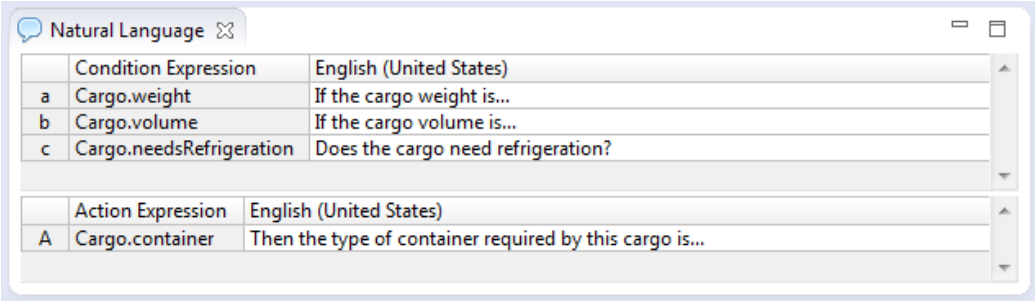
1. Right-click within a Rulesheet, and then choose **Natural Language**.

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



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

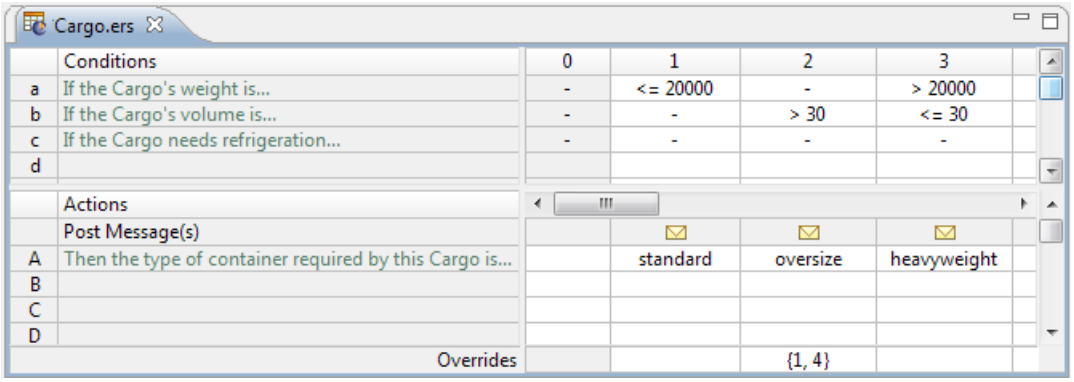
2. Enter plain language descriptive text for each condition and action, as shown:



   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.

3. 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:

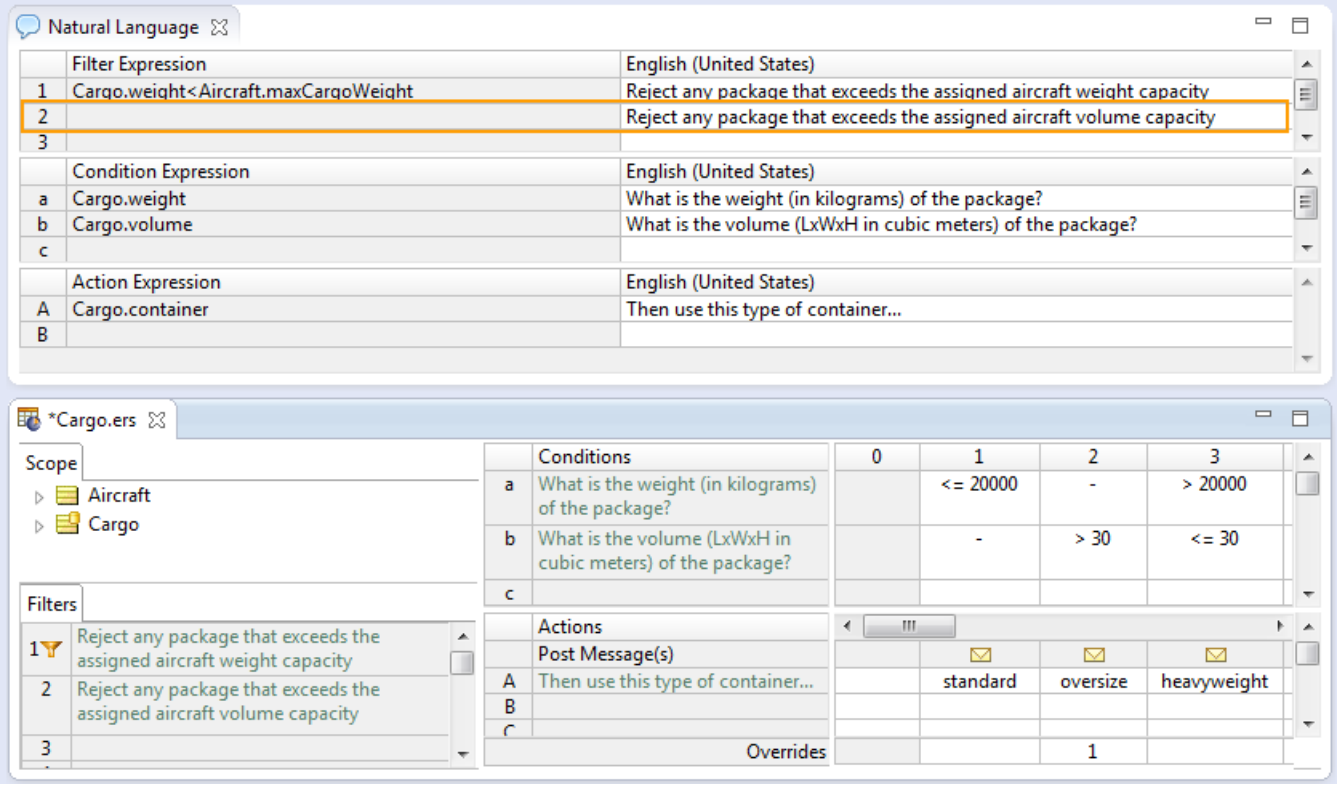


In Natural Language mode, the values in rule columns can be edited but the Condition and Action expressions are locked and cannot be edited.

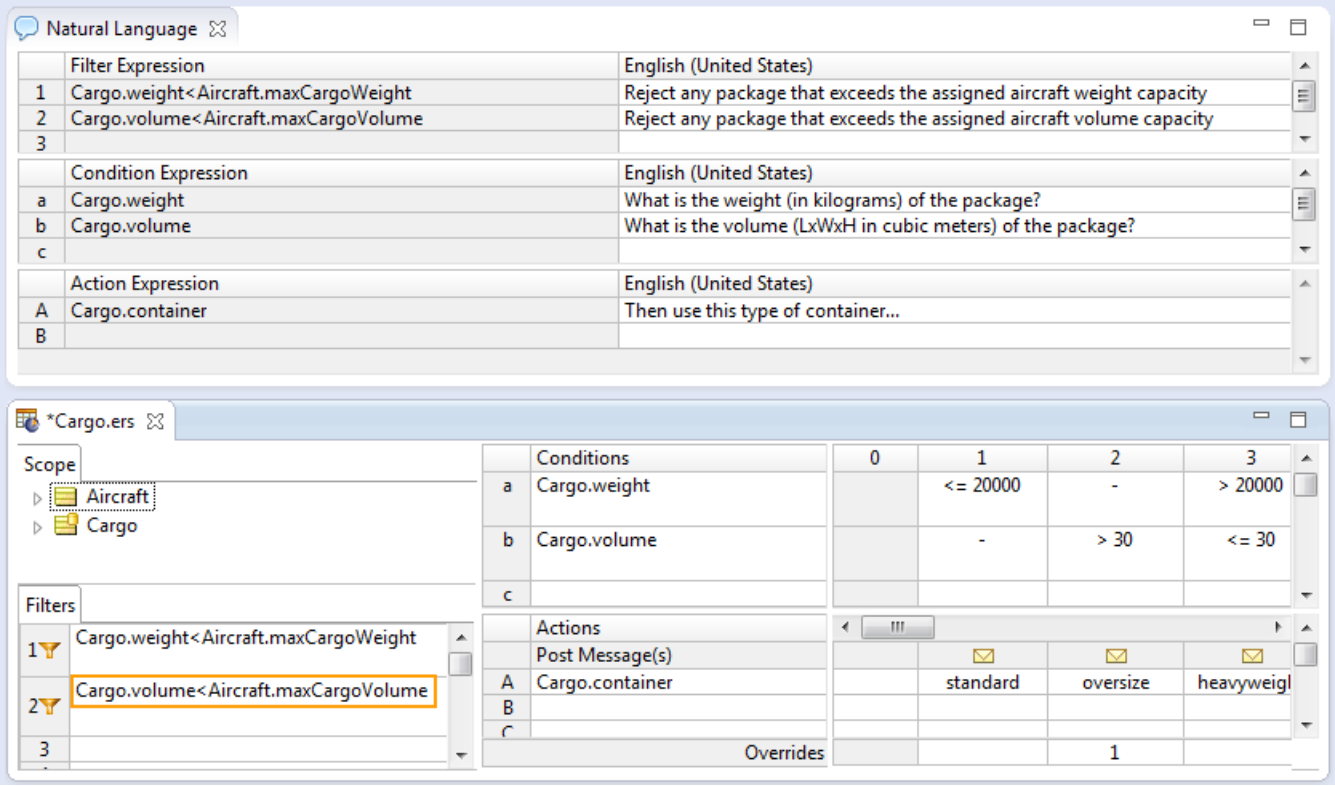
4. Save the Rulesheet to store its expressions as well as its natural language data.
5. You can revert to the actual, editable expressions by clicking the **Hide Natural Language** toolbar button , or the menu command **Rulesheet > Hide Natural Language**.
6. Close the **Natural Language** view by clicking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

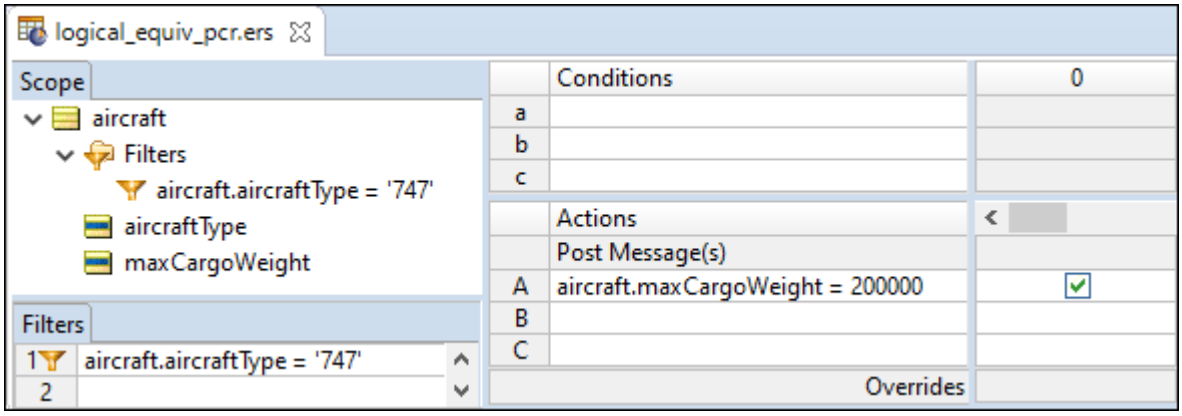


Filters versus conditions

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 36: Rulesheet using a filter and nonconditional rule

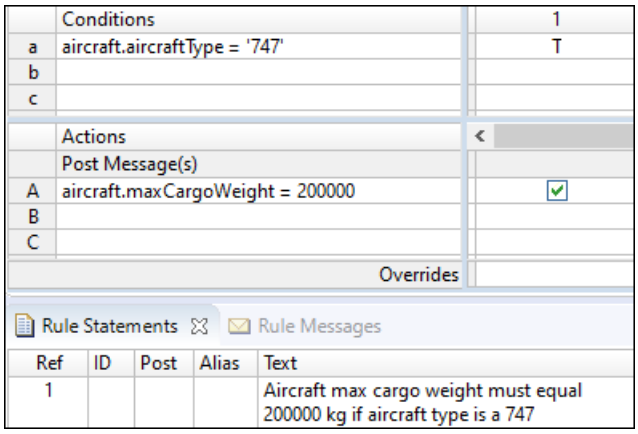


Here, the value of an aircraft's `maxCargoWeight` attribute is assigned by column 0 in the Conditions/Actions pane (what is sometimes called a *nonconditional* or *action-only* rule because it has no conditions). The filter acts as a master conditional expression because only aircraft that satisfy the 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 37: 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 183 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.

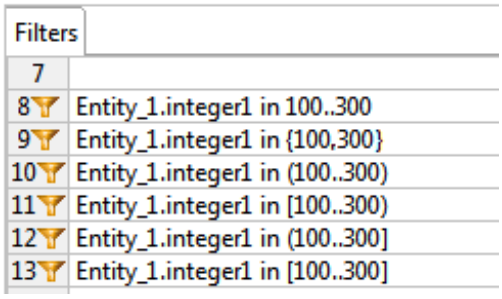
Value ranges in condition and filter expressions

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 38: 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]

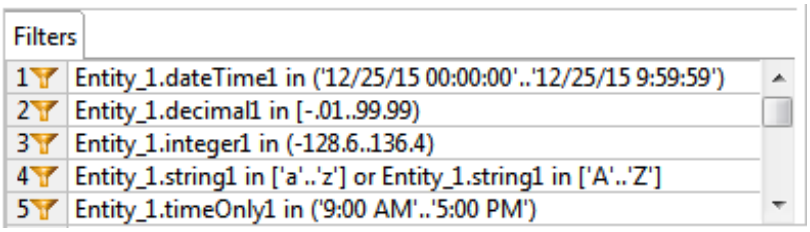
where:

- 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.js JavaScript data types can be used as Filter expressions.

Figure 39: Rulesheet filters showing the syntax of ranges for each Corticon JavaScript data type



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

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 40: Rulesheet Filters showing encapsulation of a list

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

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

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

ValueRanges.ers

	Conditions	1	2	3	4
a	FlightPlan.flightNumber	<= 100	101..200	201..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

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.js 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 42: Rulesheet using open-ended value ranges in condition cells

ValueRangesExclusiveInclusive.ers

	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 43: 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.js 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 62 for details about usage.

Boolean condition versus values set

The illustrations in the topic Filters versus conditions on page 60 that show **Rulesheet Using a Conditional Rule** and **Rulesheet Using a Conditional Rule** show simple Boolean conditions that evaluate to either `True` or `False`. The action related to this condition is either selected or not, on or off, meaning the value of `maxCargoWeight` is either assigned the value of `200,000` or it is not. (Action statements are activated by selecting the check box that automatically appears when the cell is clicked.) However, there is another way to express both Conditions and Actions using Values sets.

Figure 44: Rulesheet Illustrating use of Multiple values in the same Condition Row

OtherOperator.ers

	Conditions	1	2	3	4
a	aircraft.aircraftType	'DC-10'	'A340'	'747'	other
b					
	Actions				
	Post Message(s)				
A	aircraft.maxCargoWeight	100000	150000	200000	50000
B					
	Overrides				

Rule Statements | Rule Messages

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

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 would require 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 45: Exclusionary logic using Boolean condition, Pt. 1

ExclusionarySyntax.ers

	Conditions	0	1
a	aircraft.aircraftType <> '747'		T
b			
	Actions		
	Post Message(s)		
A	aircraft.maxCargoWeight = 100000		✓
B			
	Overrides		

Rule Statements | Rule Messages

Ref	ID	Post	Alias	Text
1		Info	aircraft	Aircraft max cargo weight must be 100000 when aircraft type is NOT a 747

Figure 46: Exclusionary logic using Boolean condition, Pt. 2

ExclusionarySyntax.ers

	Conditions	0	1
a	aircraft.aircraftType = '747'		F
b			
	Actions		
	Post Message(s)		
A	aircraft.maxCargoWeight = 100000		✓
B			
	Overrides		

Rule Statements | Rule Messages

Ref	ID	Post	Alias	Text
1		Info	aircraft	Aircraft max cargo weight must be 100000 when aircraft type is NOT a 747

Figure 47: Exclusionary logic using negated value

ExclusionarySyntax.ers

	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.

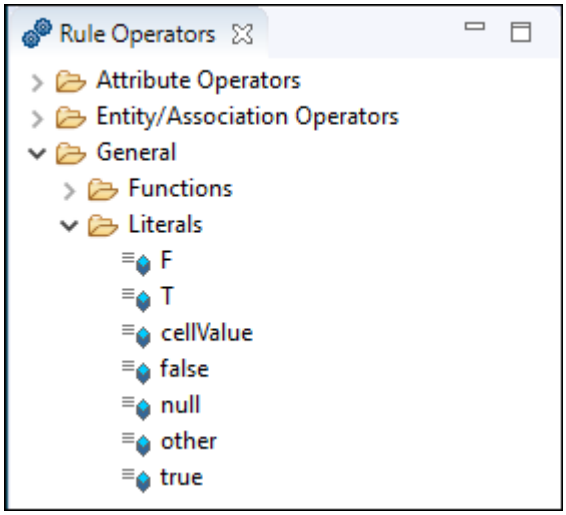
Once again, you can see that the same rule can be expressed in different ways on the Rulesheet, with identical results. The rule modeler decides which way of expressing the rule is preferable in a given situation. Progress recommends, however, avoiding double negatives. Most people find it easier to 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 48: 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 49: Rulesheet using numeric value ranges in condition values set

NumericValuesInConditions.ers

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

Rule Statements | Rule Messages

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

In this example, 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.js Studio will add the single quotation marks for you, but always check to make sure it has interpreted your entries correctly.

Figure 50: Rulesheet using String value ranges in condition values set

String Values in Conditions.ers

	Conditions	1	2	3
a	Entity1.string1	'a'..'z'	'A'..'Z'	other
b				
	Actions			
	Post Message(s)			
A	Entity1.string2	'lowercase'	'uppercase'	'other char'
B				
	Overrides			

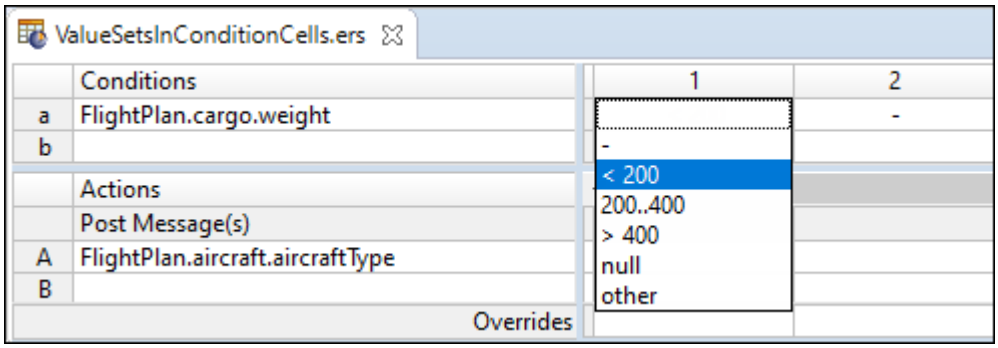
Rule Statements | Rule Messages

Ref	ID	Post	Alias	Text
1				If Entity1.string1 is lowercase, set Entity1.string2 to 'lowercase'
2				If Entity1.string1 is uppercase, set Entity1.string2 to 'uppercase'
3				If Entity1.string1 is another character, set Entity1.string2 to 'other char'

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 51: 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 52: Rulesheet with two values selected in condition cell

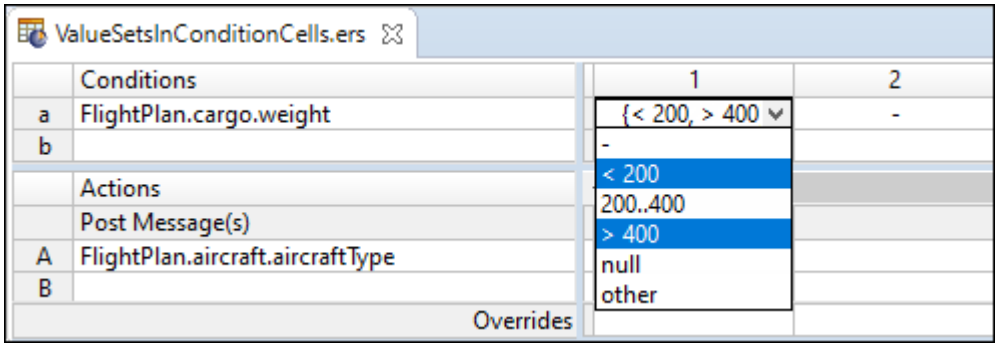
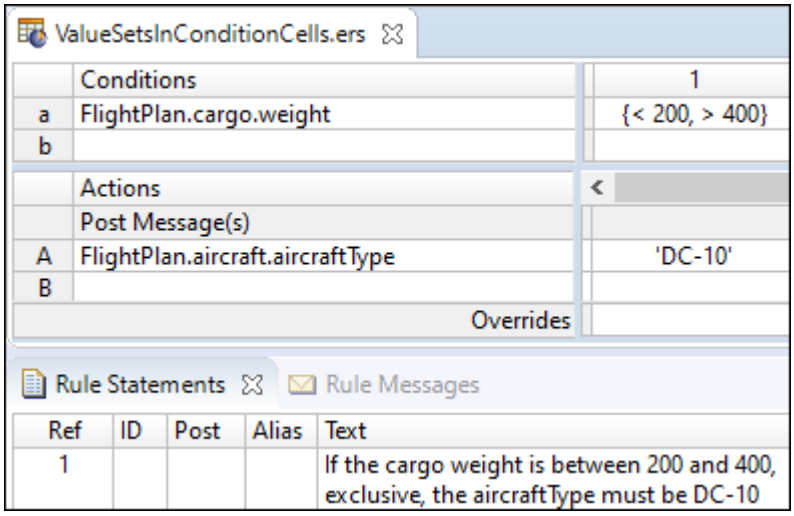


Figure 53: Rulesheet with value set in condition cell



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

Figure 54: Rulesheet with two rules instead of a value set

ValueSetsInConditionCells.ers

	Conditions	1	2
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 55: Negating a value set in a condition cell

ValueSetsInConditionCells.ers

	Conditions	1
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**

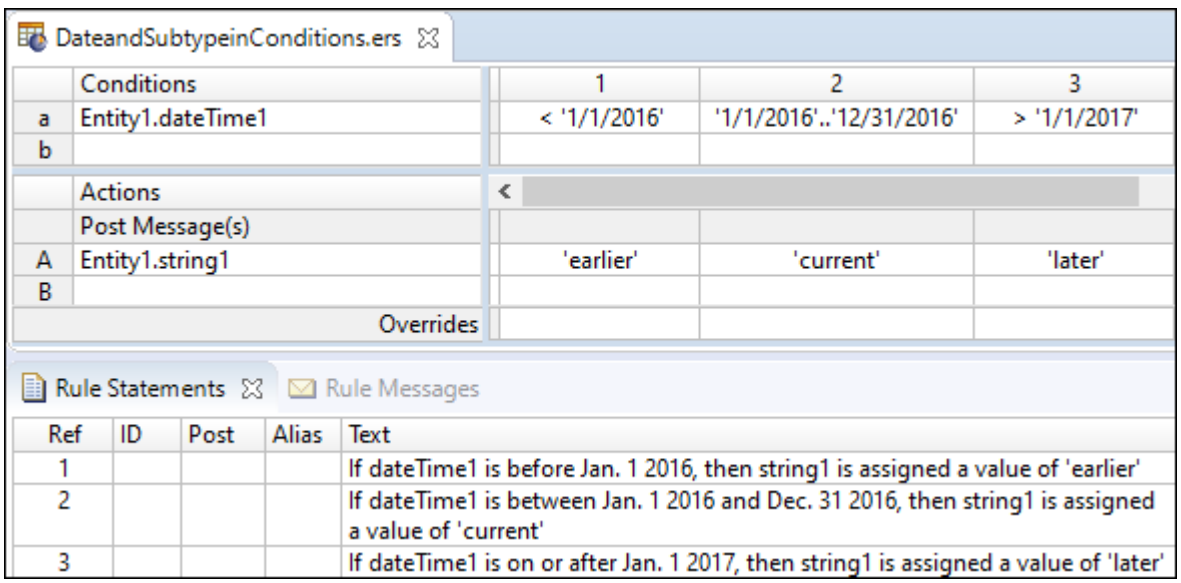
Table 5: Rulesheet condition h: Attempt to use logically overlapping expressions

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

DateTime value ranges in condition cells

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

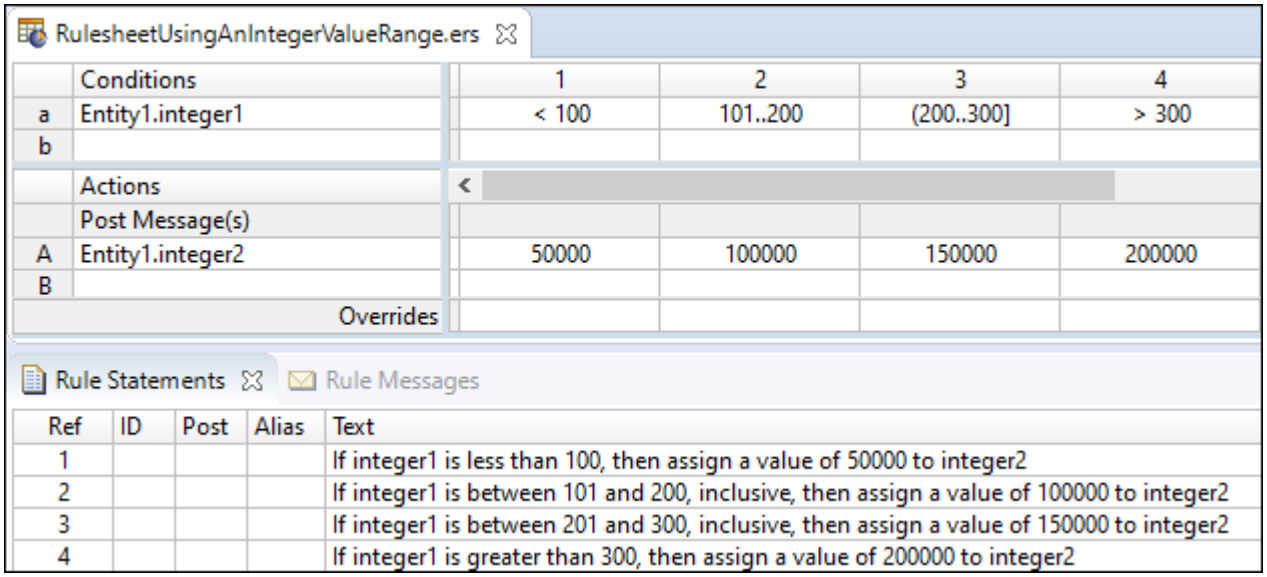
Figure 56: Rulesheet using a date value range in condition cells



Inclusive and exclusive ranges

Corticon.js 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 57: 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.js 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 58: Rulesheet with value range overlap

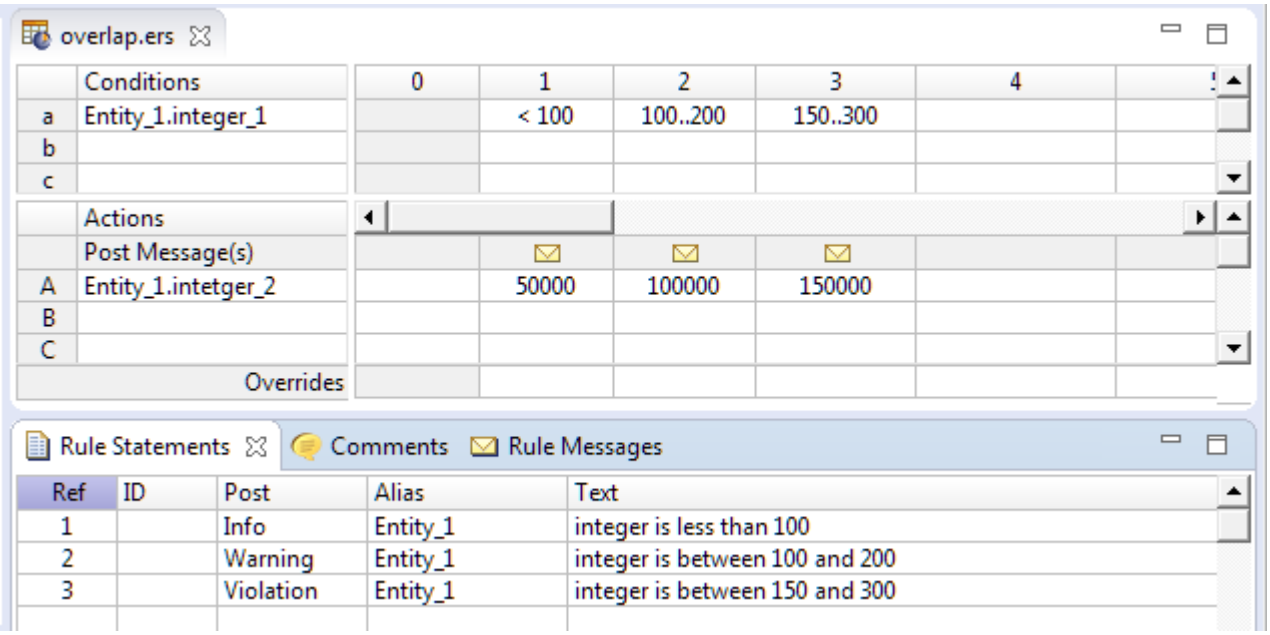


Figure 59: Rulesheet expanded with conflict check applied

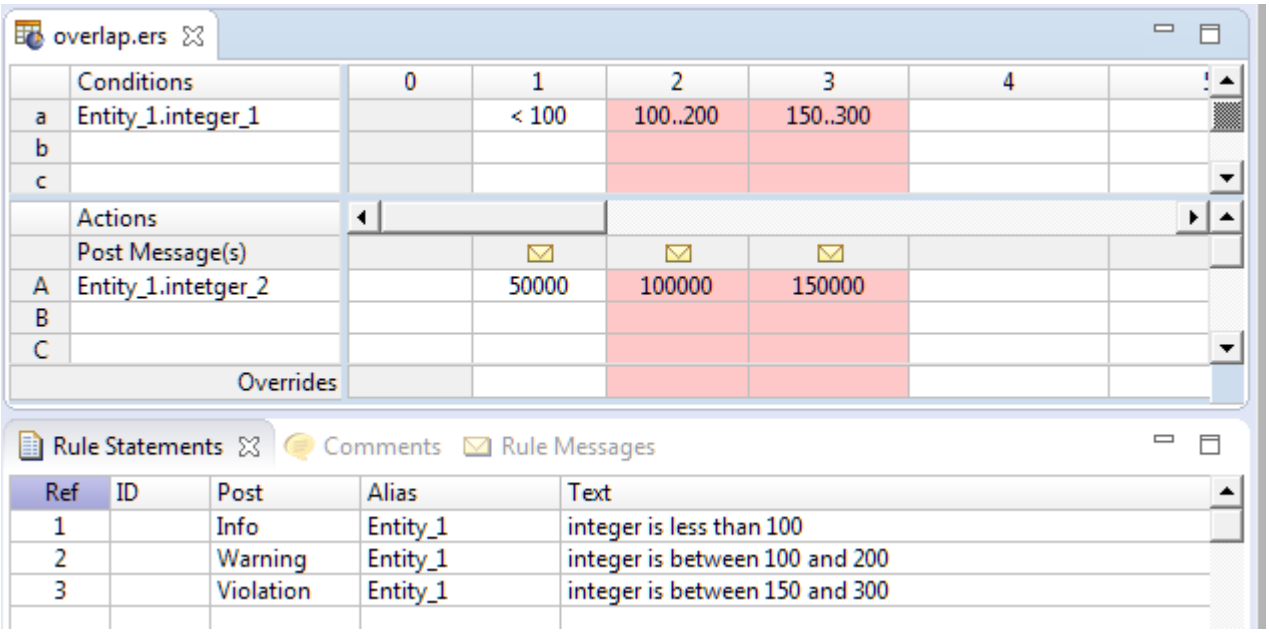
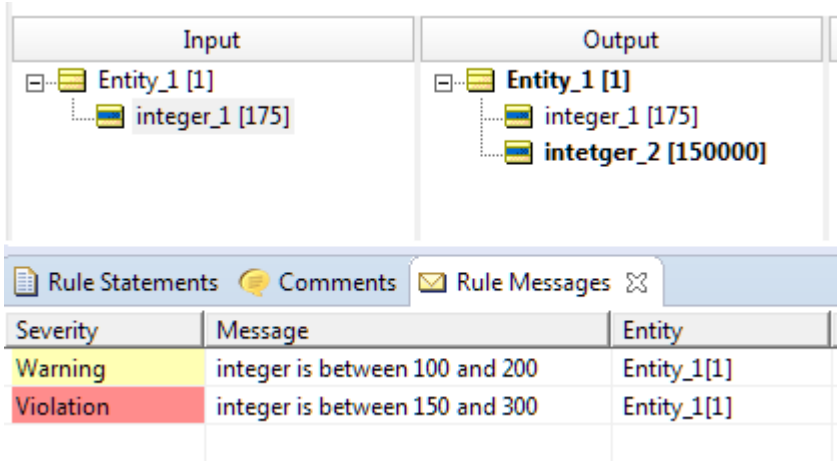


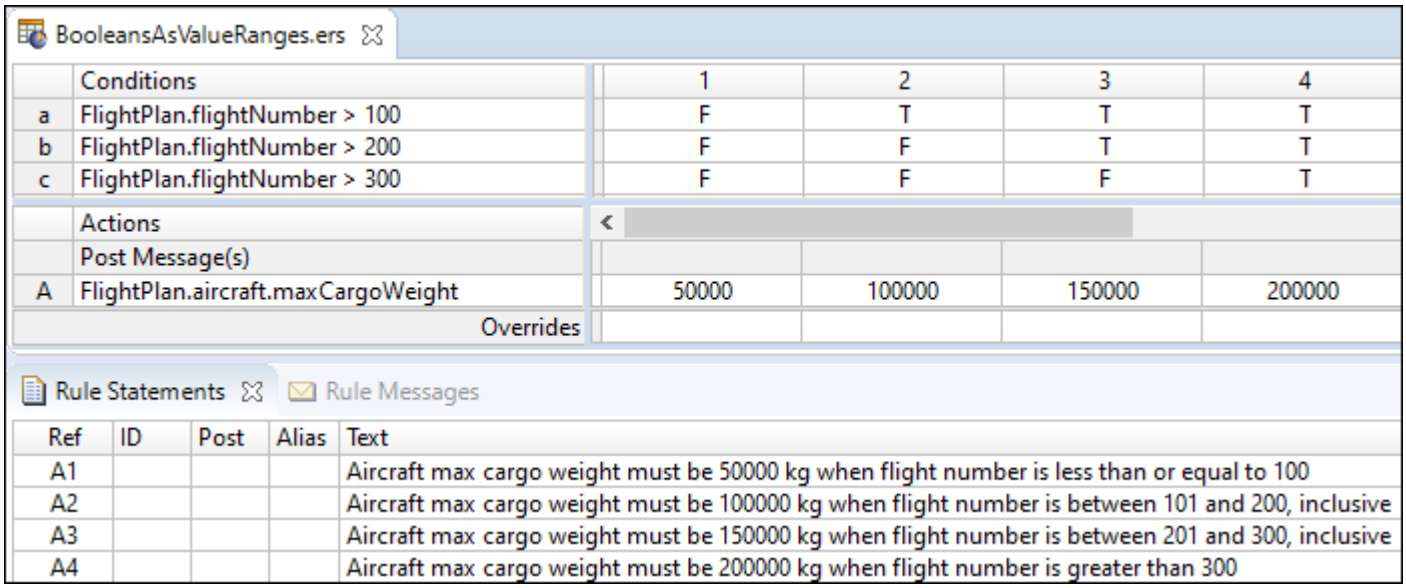
Figure 60: Ruletest showing multiple rules firing for given test data



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 61: Rulesheet using Boolean conditions to express value ranges



The rules here are identical to the rules in **Rulesheet Using Value Ranges in the Column Cells of a Condition Row** and **Rulesheet Using Open-Ended Value Ranges in Condition Cells**, but are expressed using a series of three Boolean conditions. Recall that in a decision table, values aligned vertically in the same column represent 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.js 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 62: Sample Rulesheet with rule statements containing embedded attributes

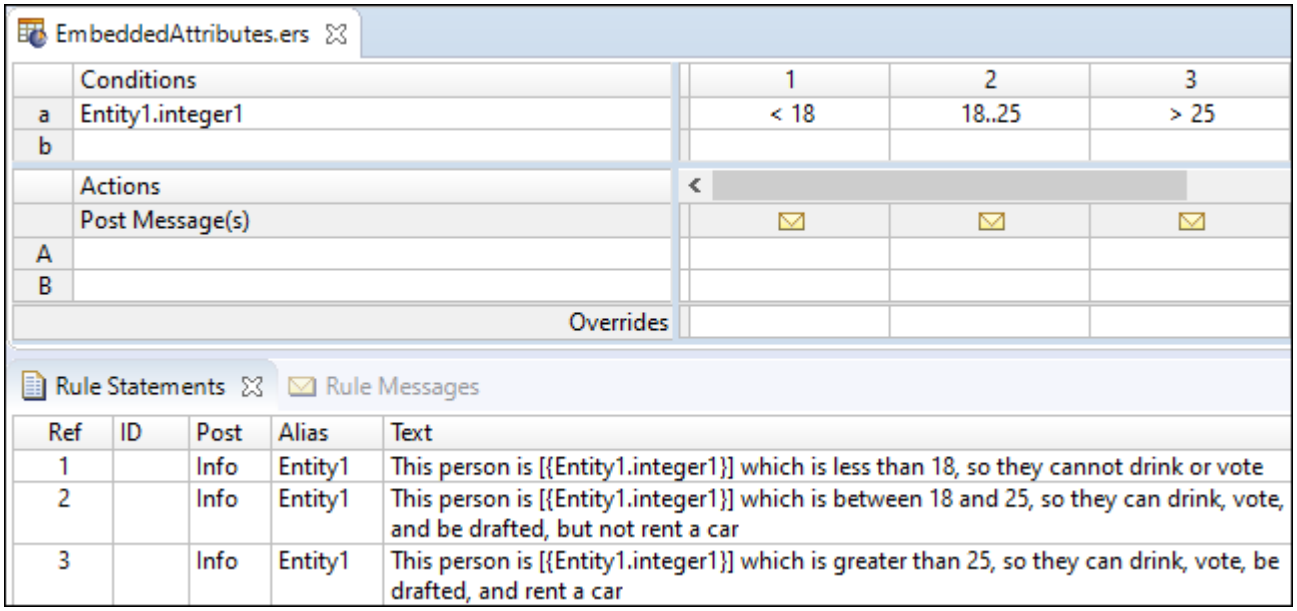
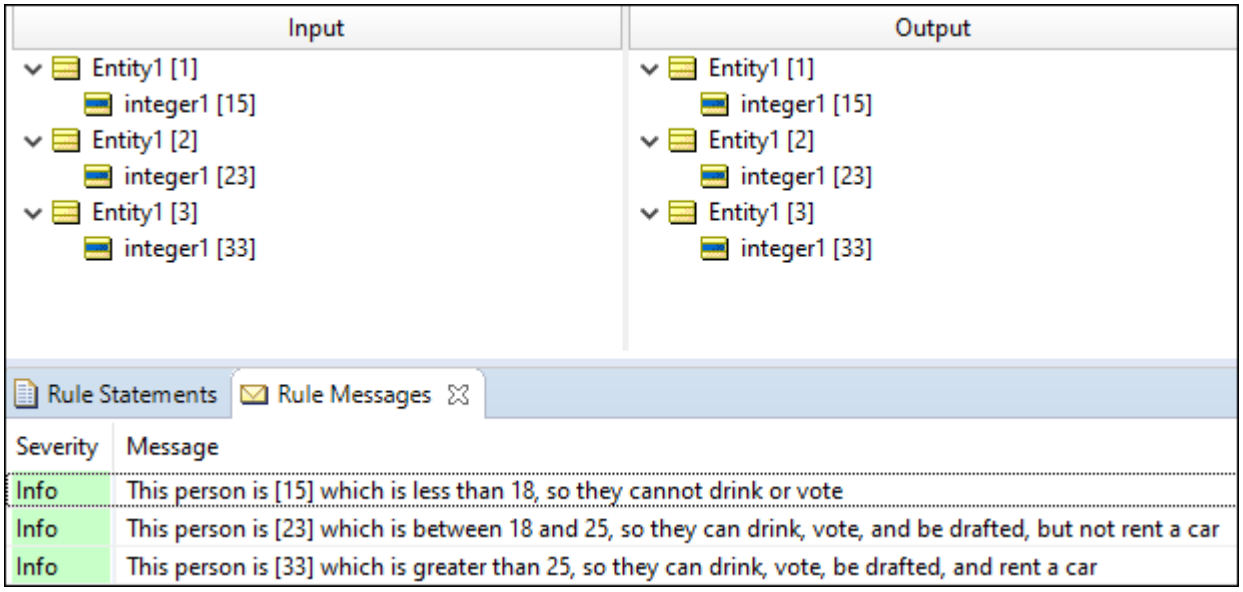


Figure 63: Rule Messages window showing bracketed embedded attributes



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.js 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.js Studio:

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

The text turns red, because Corticon.js 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.js 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.

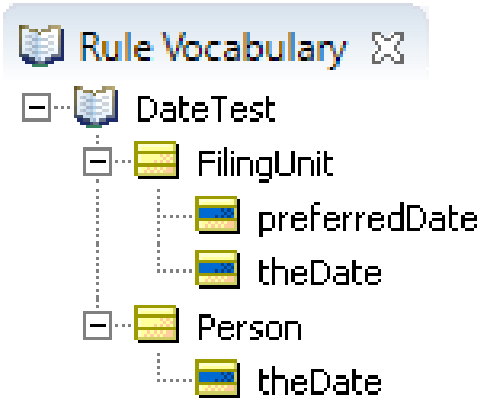
How to initialize null attributes

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 – it is assigned the result of the calculation. For example, when you are calculating Force=Mass*Acceleration, you must provide values for Mass and Acceleration. Force is the result of a valid calculation.

Initialization of attributes is often performed in Nonconditional rules, or in rules expressed in Rulesheets that execute beforehand.

How to handle nulls in compare operations

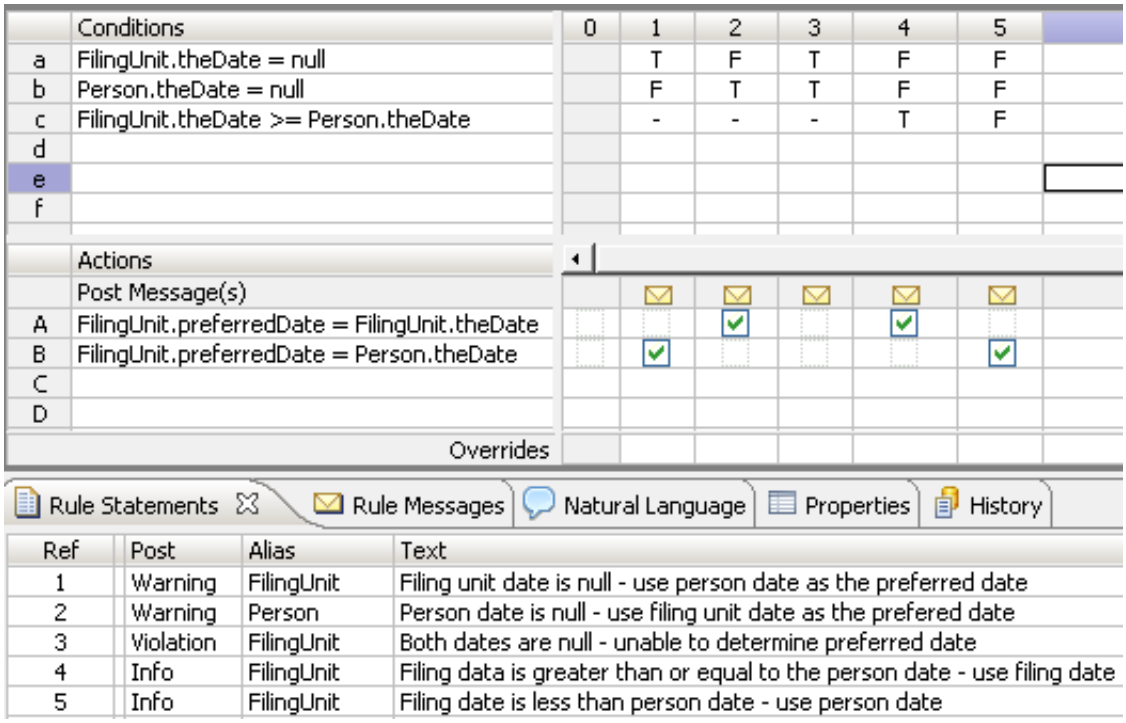
Unless the application that formed the request ensured that a value was provided before submission, one (or both) of the attributes used in a comparison test might have a null value. You might need to define rules to handle such cases. An example that describes the workaround for these cases uses the 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.js 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.

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
- Special collection operators

How Corticon Studio handles collections

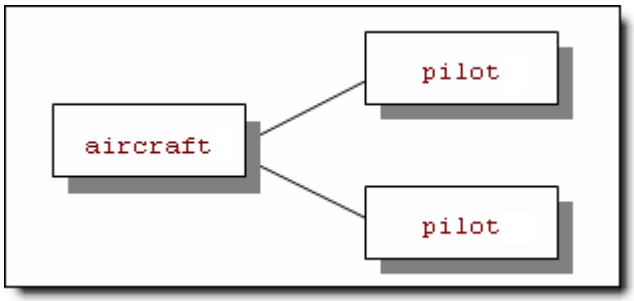
Support for using collections is extensive in Corticon.js Studio. The integration of collection support in the Rule Language is so seamless and complete that the rule modeler often discovers that rules are performing multiple evaluations on collections of data beyond what they anticipated! This is partly 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.js 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:

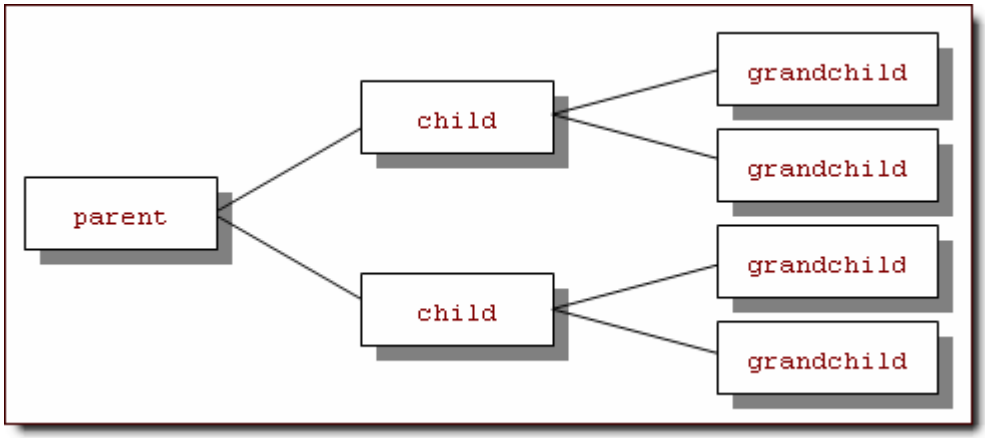
Figure 64: Visualization of a collection of pilots



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.js 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 65: Three-level collection



This collection is expressed as `parent.child.grandchild` in the Corticon.js 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.js 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.js Studio supports filtering by a special use of Filter expressions. See the Filters on page 125 topic for more details.

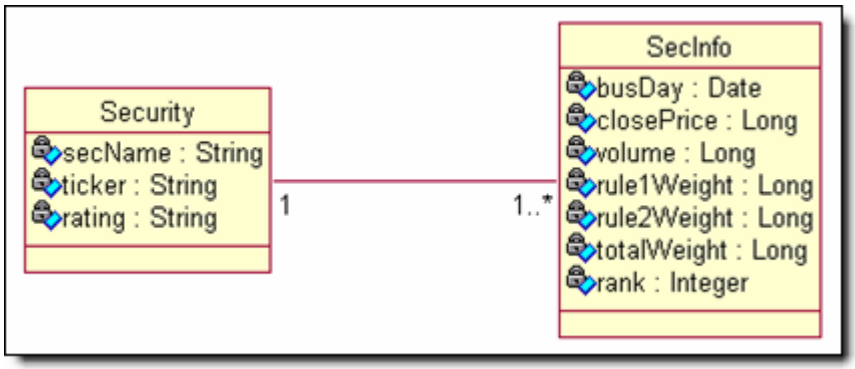
How to use aliases to represent collections

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 66: 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.

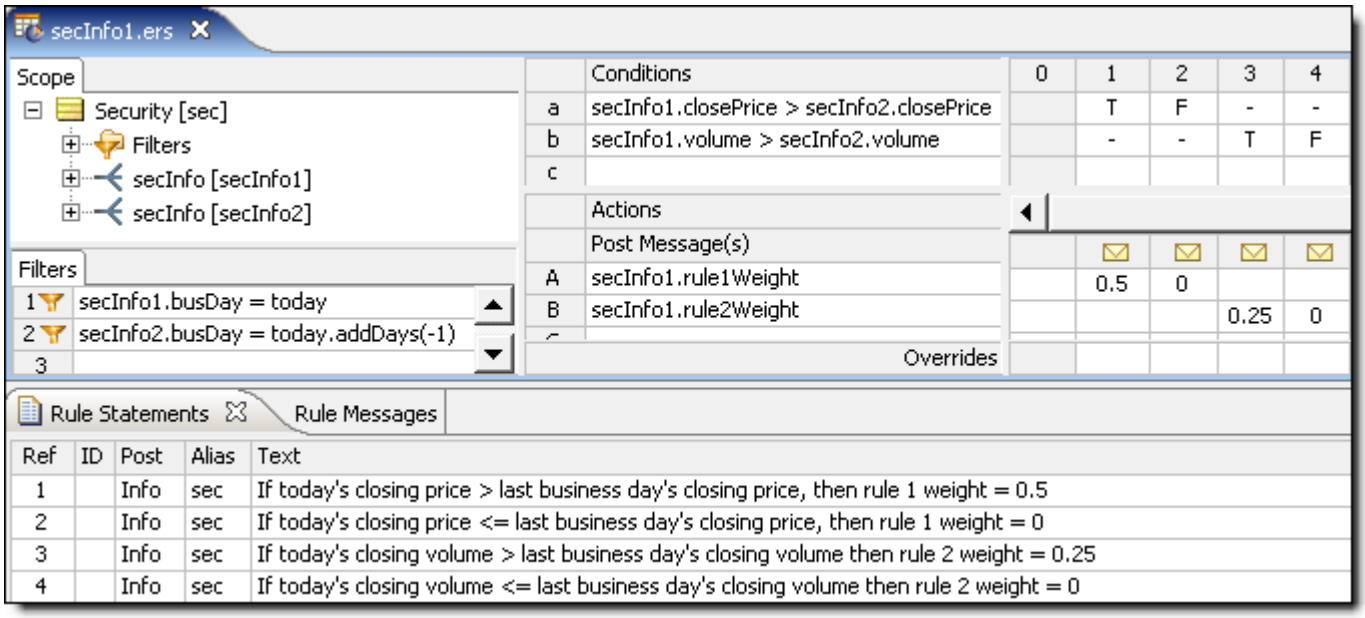
The association between `Security` and `SecInfo` is `1..*` (one-to-many) because there are multiple instances of `SecInfo` data (multiple days of historical data) for each `Security`.

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 67: Rulesheet with ranking model rules 1 and 2



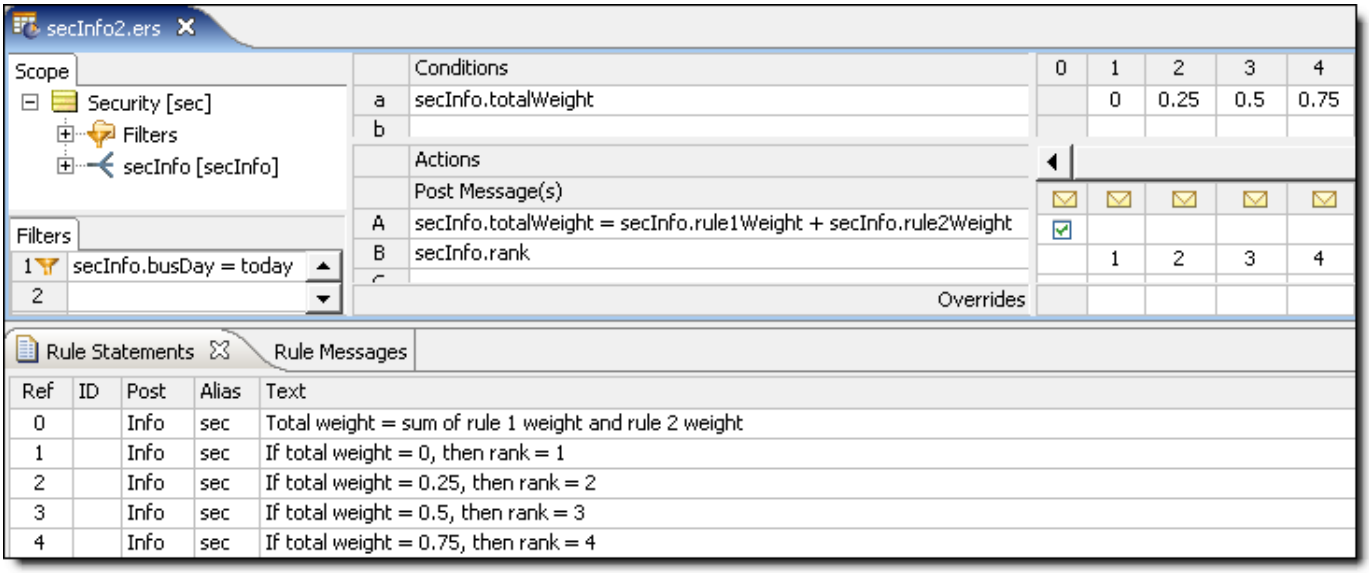
In the preceding figure, two business rules are expressed in a total of four rule models (one for each possible outcome of the two business rules). The rules are straightforward, but the shortcuts (alias values) used in these rules are different than other rules you have seen. In the Scope section, you 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 separate but identical 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 68: Rulesheet with total weight calculation and rank determination

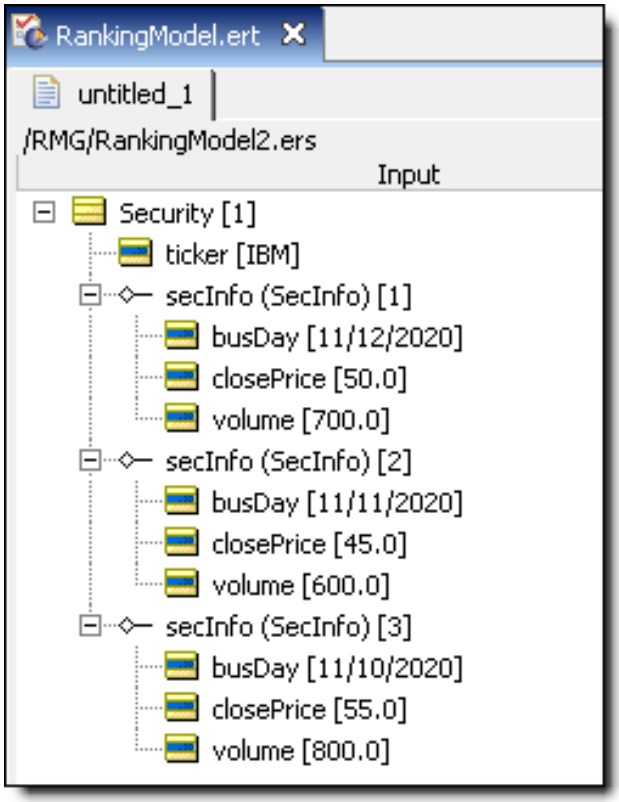


You can test your new rules using a Ruleflow to combine the two Rulesheets. In a Ruletest that executes the Ruleflow, you expect to see the following results:

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.

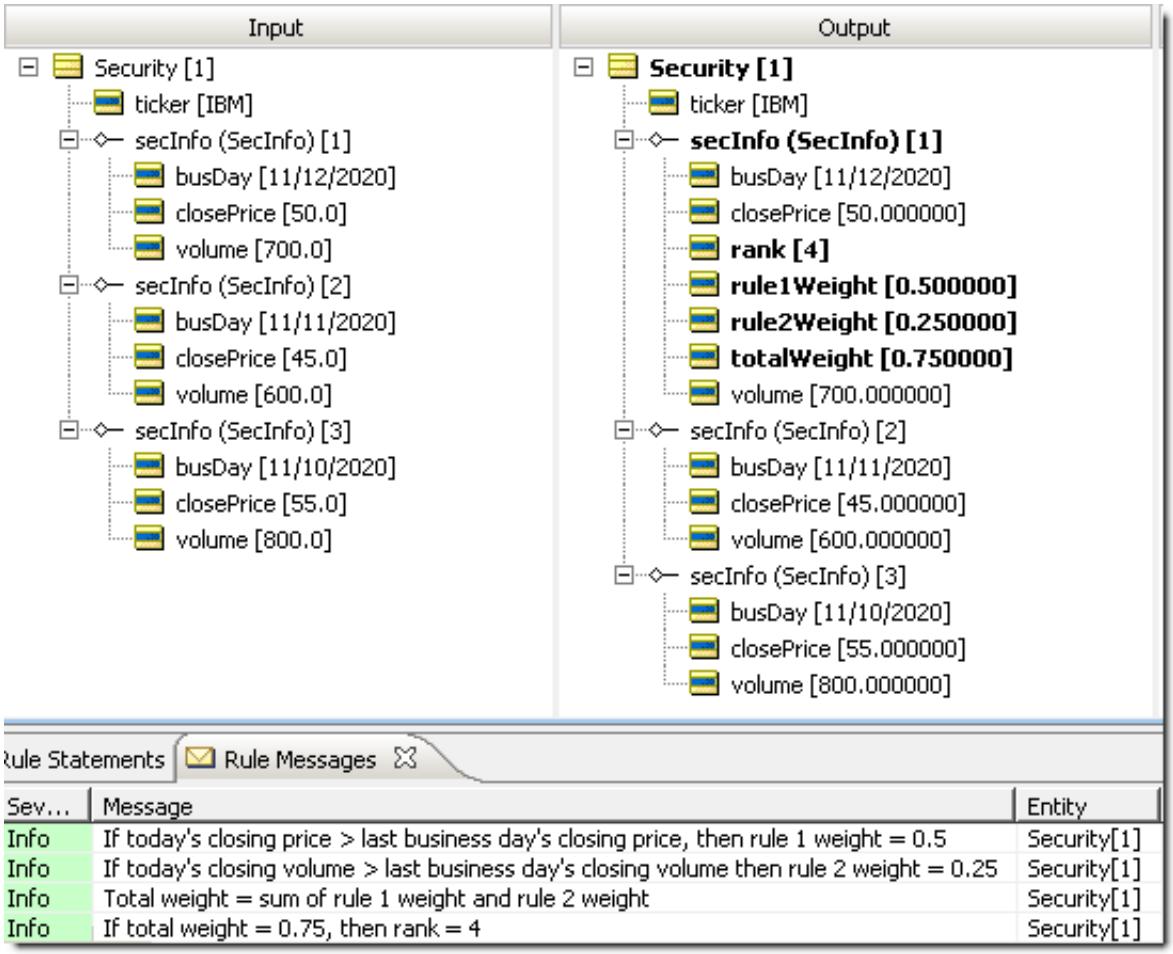
A Ruleflow constructed for testing the ranking model rules is as shown:

Figure 69: 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 70: Ruletest for security ranking model rules

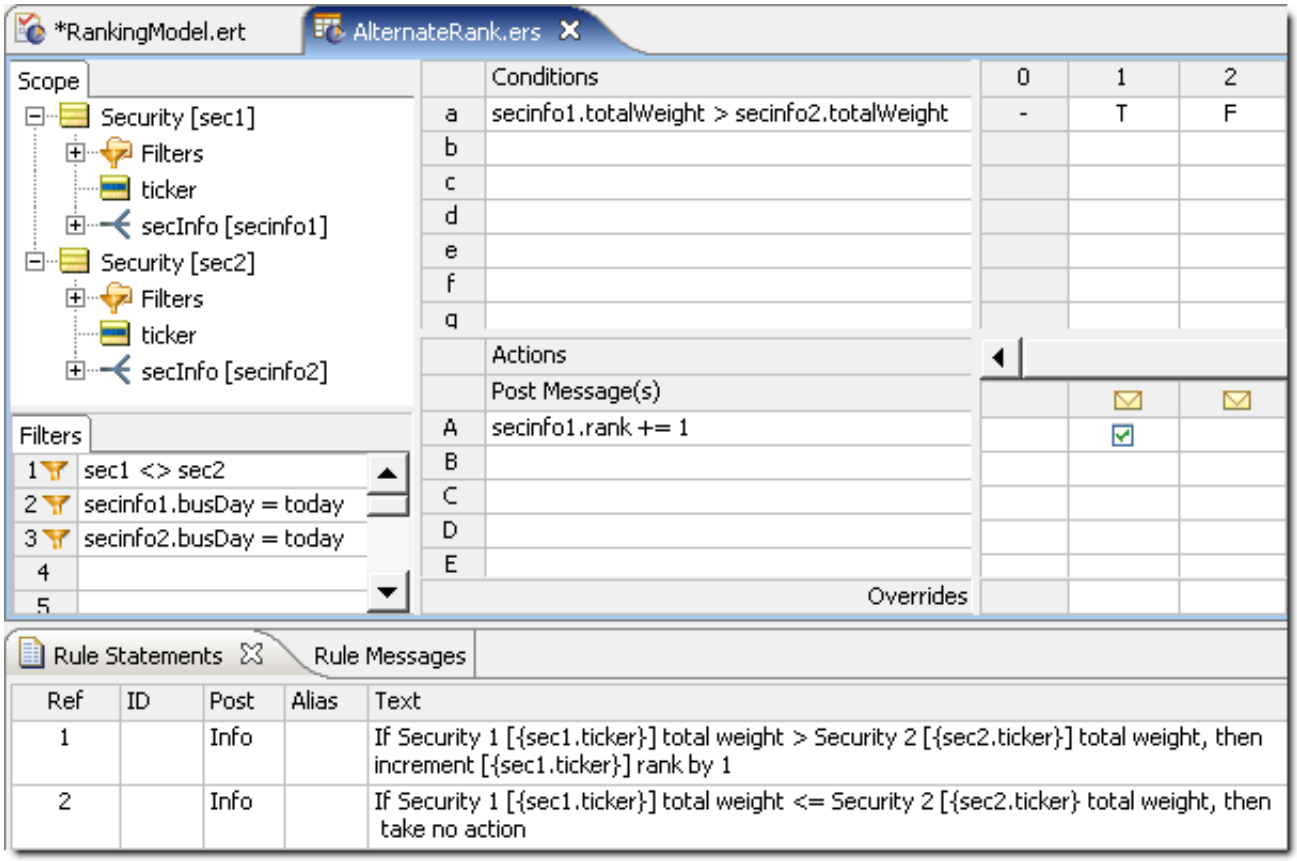


Both `closePrice` and `volume` for 11/12/2020 were higher than the values for those same attributes on 11/11/2020; therefore, both `rule1Weight` and `rule2Weight` attributes were assigned their high values by the rules. Accordingly, the `totalWeight` value calculated from the sum of the partial weights 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 67: Rulesheet with ranking model rules 1 and 2 on page 85. 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.

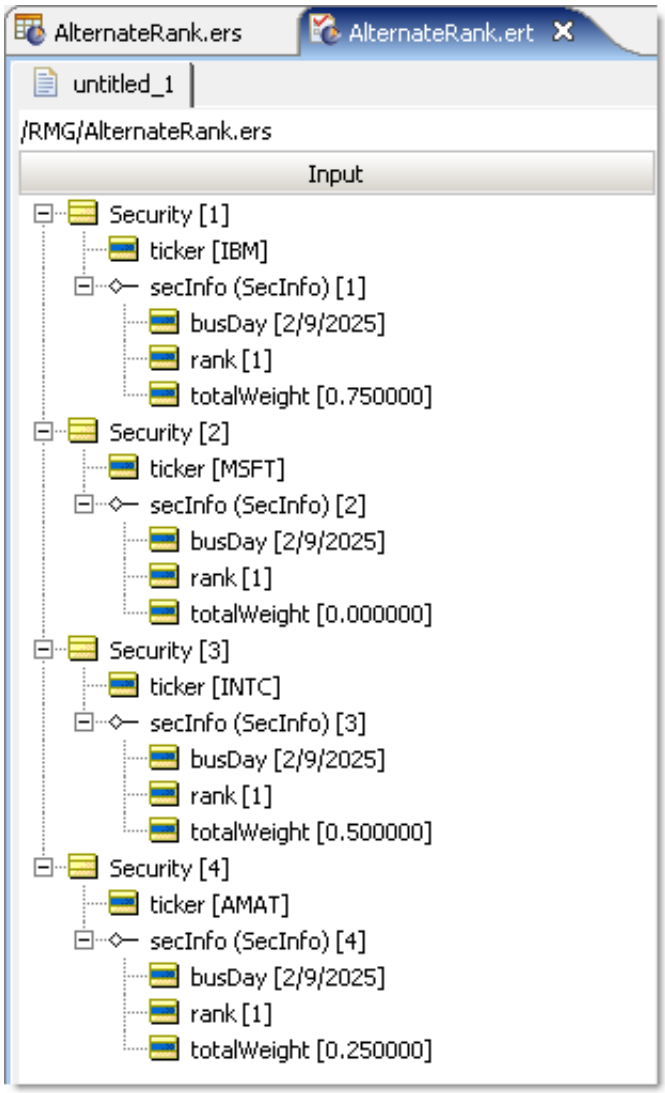
Figure 71: Rulesheet with alternate rank determination rules



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.

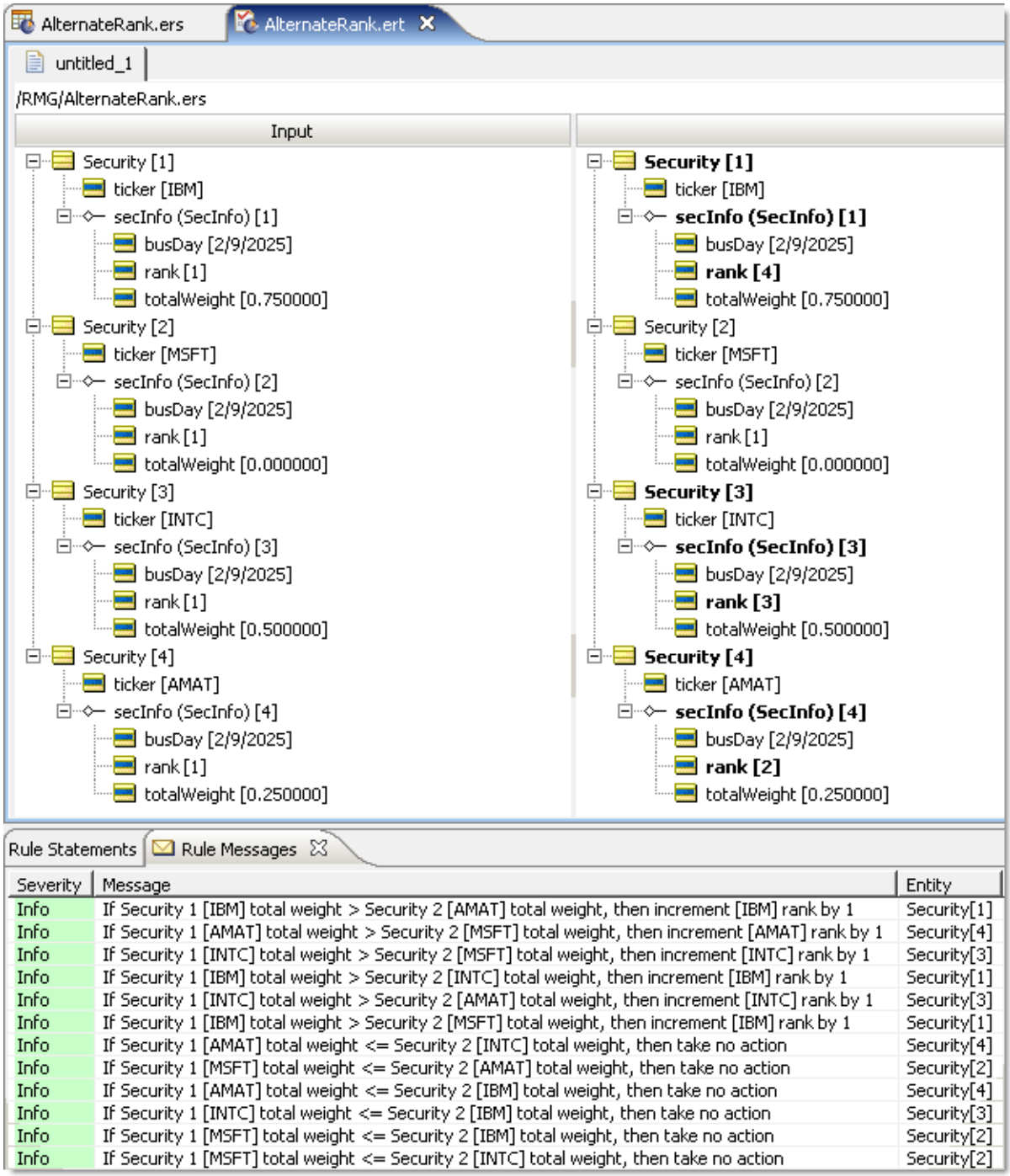
Figure 72: Input Testsheet for testing alternate security ranking model rules



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.

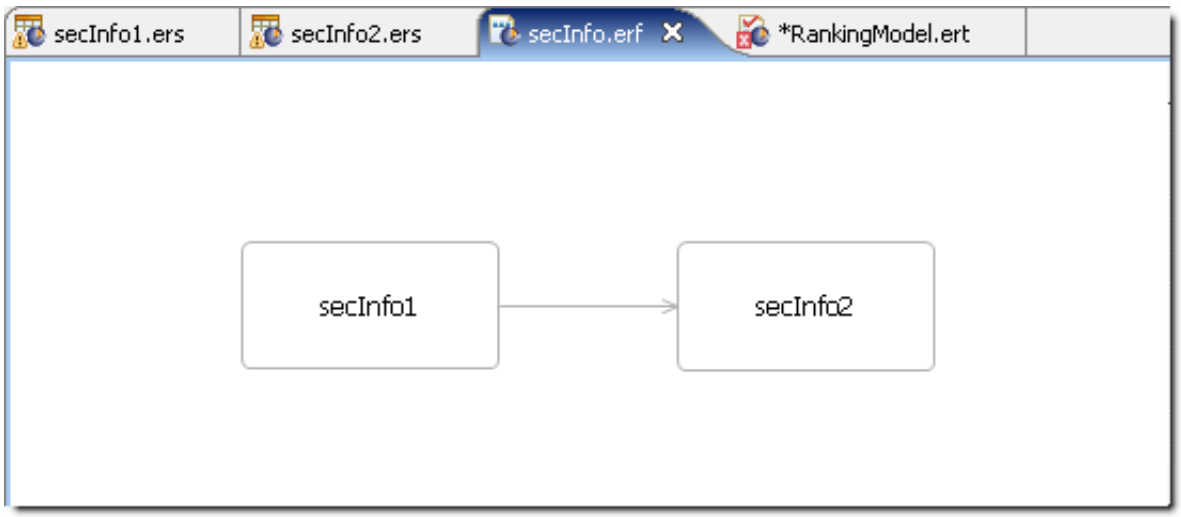
Figure 73: Results Testsheet for alternate security ranking model rules



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 74: Ruleflow to test two Rulesheets in succession



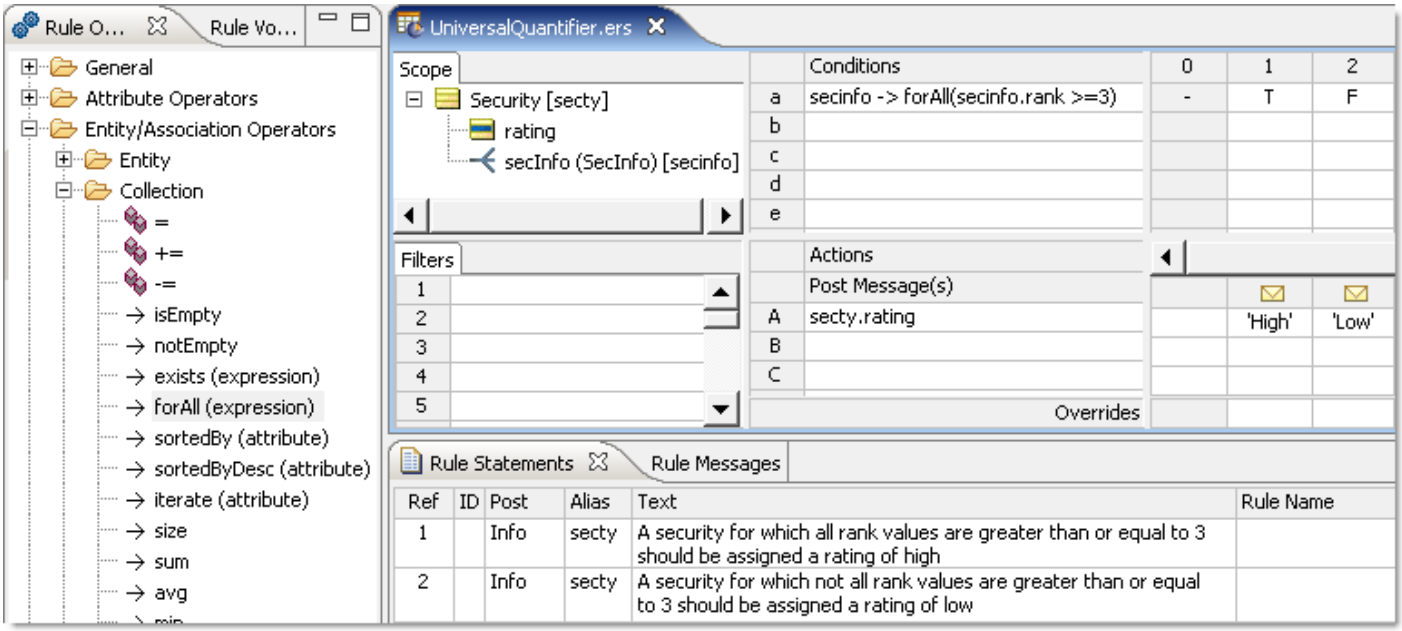
Special collection operators

There are two special collection operators available in Corticon.js 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 75: Rulesheet with universal quantifier ("for all") condition



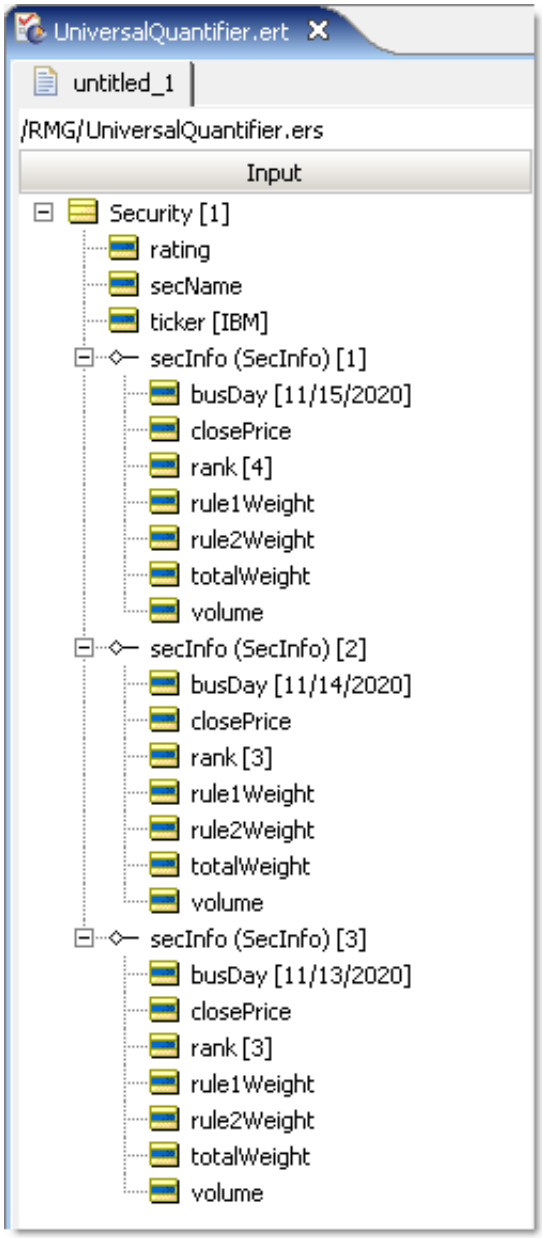
In this figure, you see the following condition:

```
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.

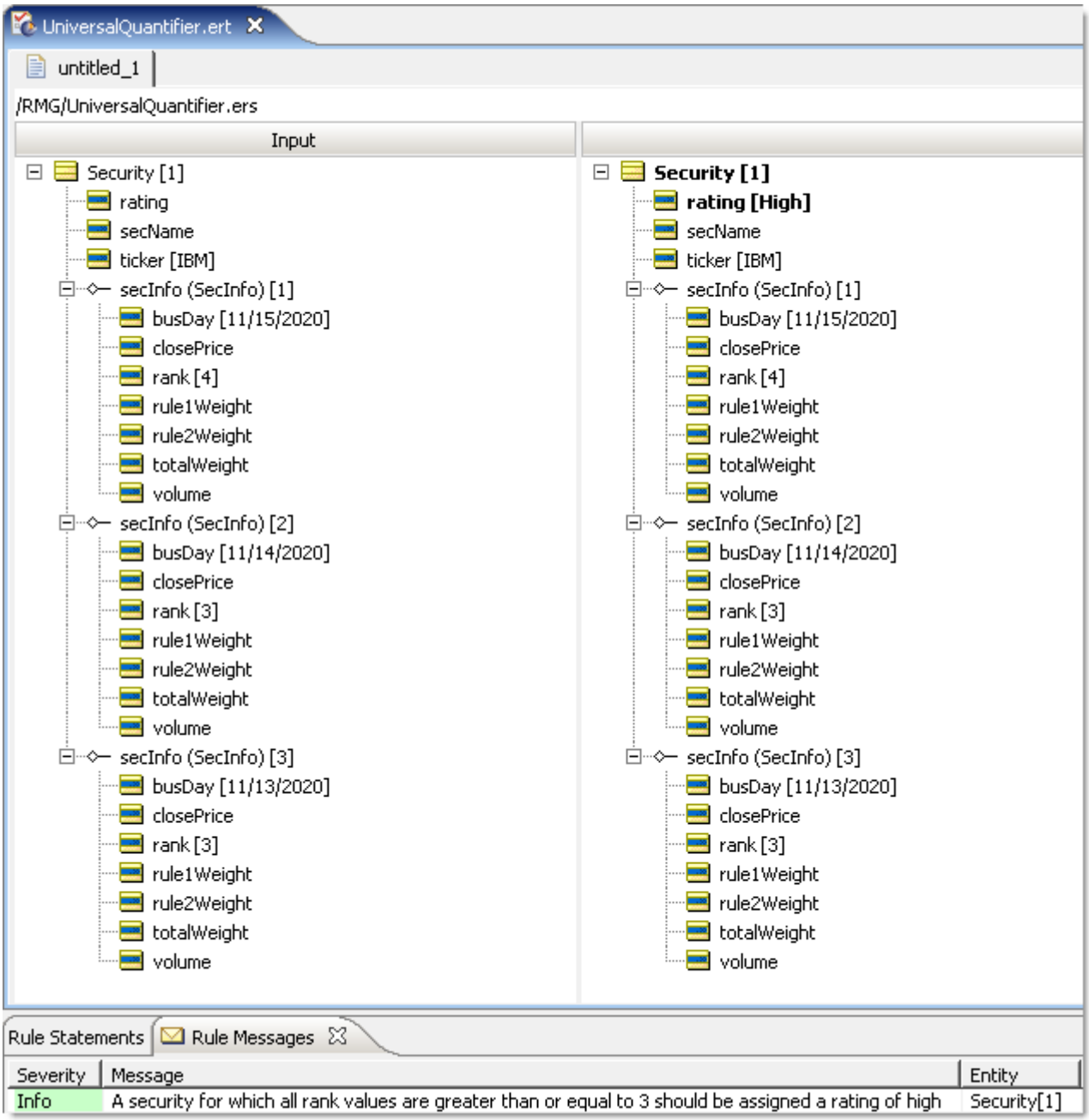
The following figure shows a Ruletest constructed to test the “for all” condition rules.

Figure 76: 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 77: Ruletest for “for all” condition rules

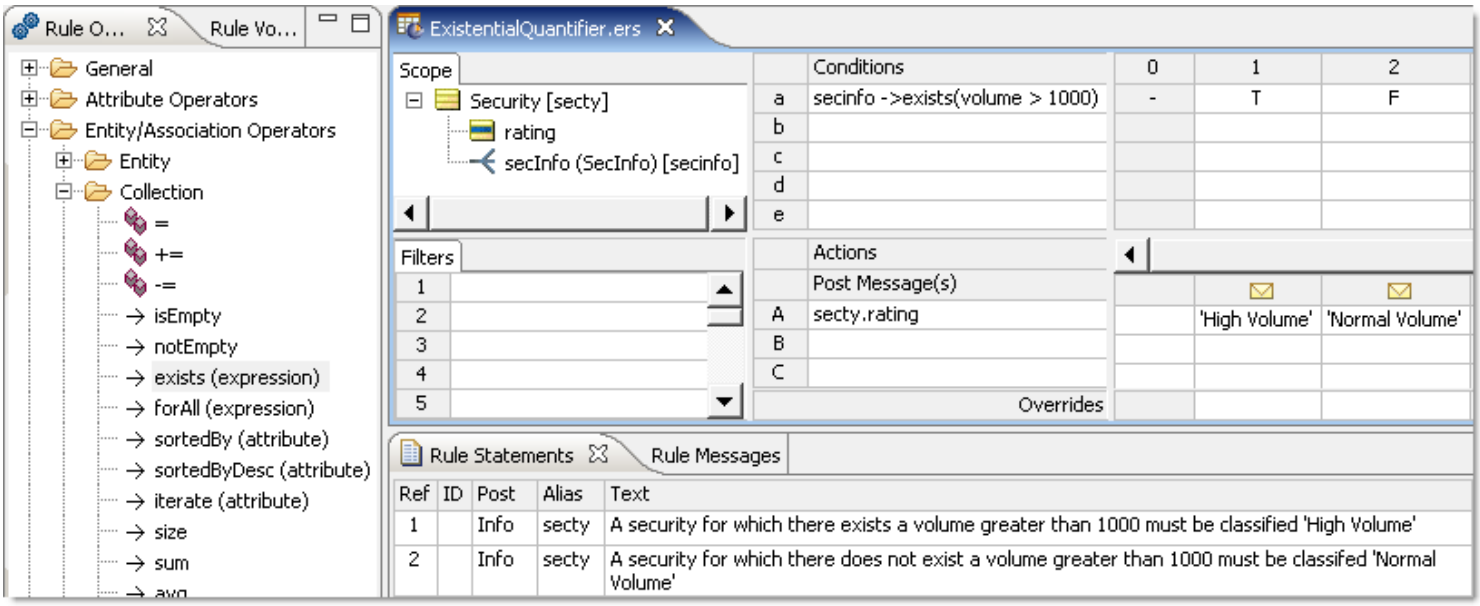


Existential quantifier

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 78: Rulesheet with existential quantifier ("exists") condition



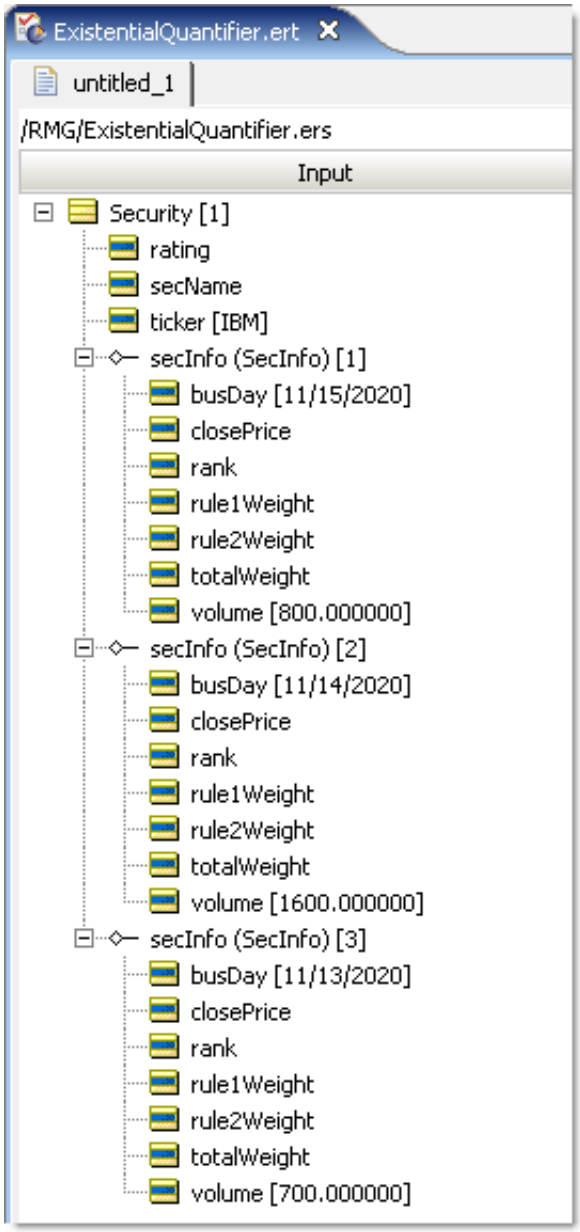
In this Rulesheet, you see the following condition

```
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.

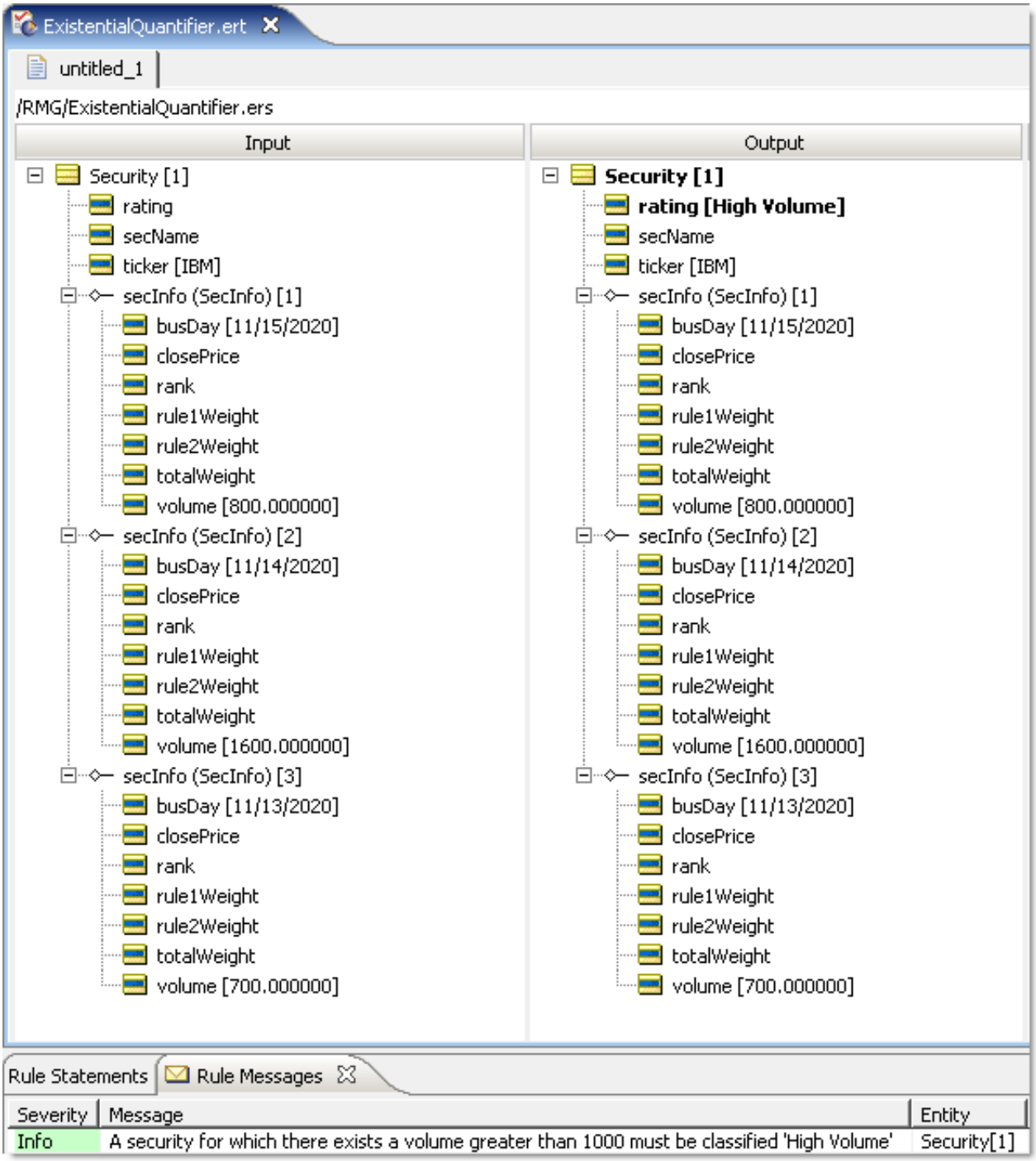
The following figure shows a Ruletest constructed to test the `exists` condition rules.

Figure 79: 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 80: Ruletest output for ("exists") condition rules



Another example using the existential quantifier

Collection operators are powerful parts of the Corticon.js 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:

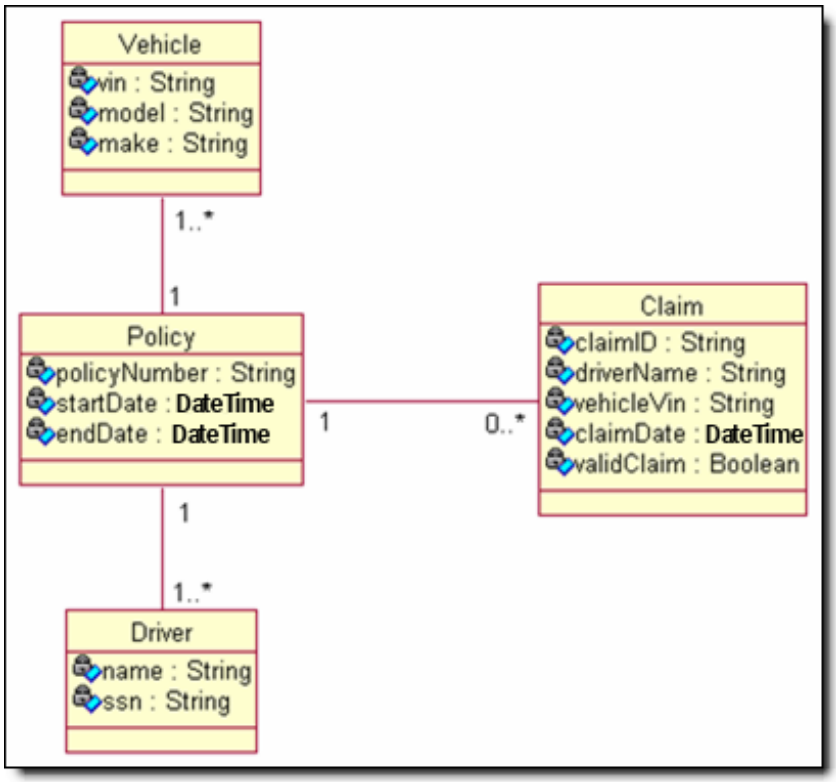
1. A claim is valid if the driver and vehicle involved in a claim are both listed on the policy against which the claim is submitted.

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` .

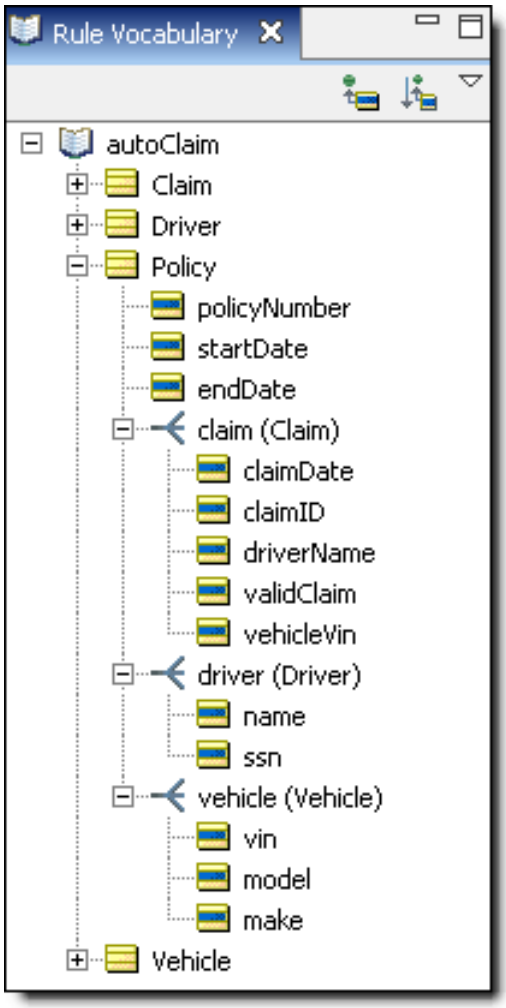
Note: Alternatively, the Claim entity could have referenced `Driver.name` and `Vehicle.vin` (by adding associations between Claim and both Driver and Vehicle), respectively, but the denormalized structure is probably more representative of a real-world scenario.

Figure 81: UML Class Diagram



This model is realized in Corticon.js Studio as:

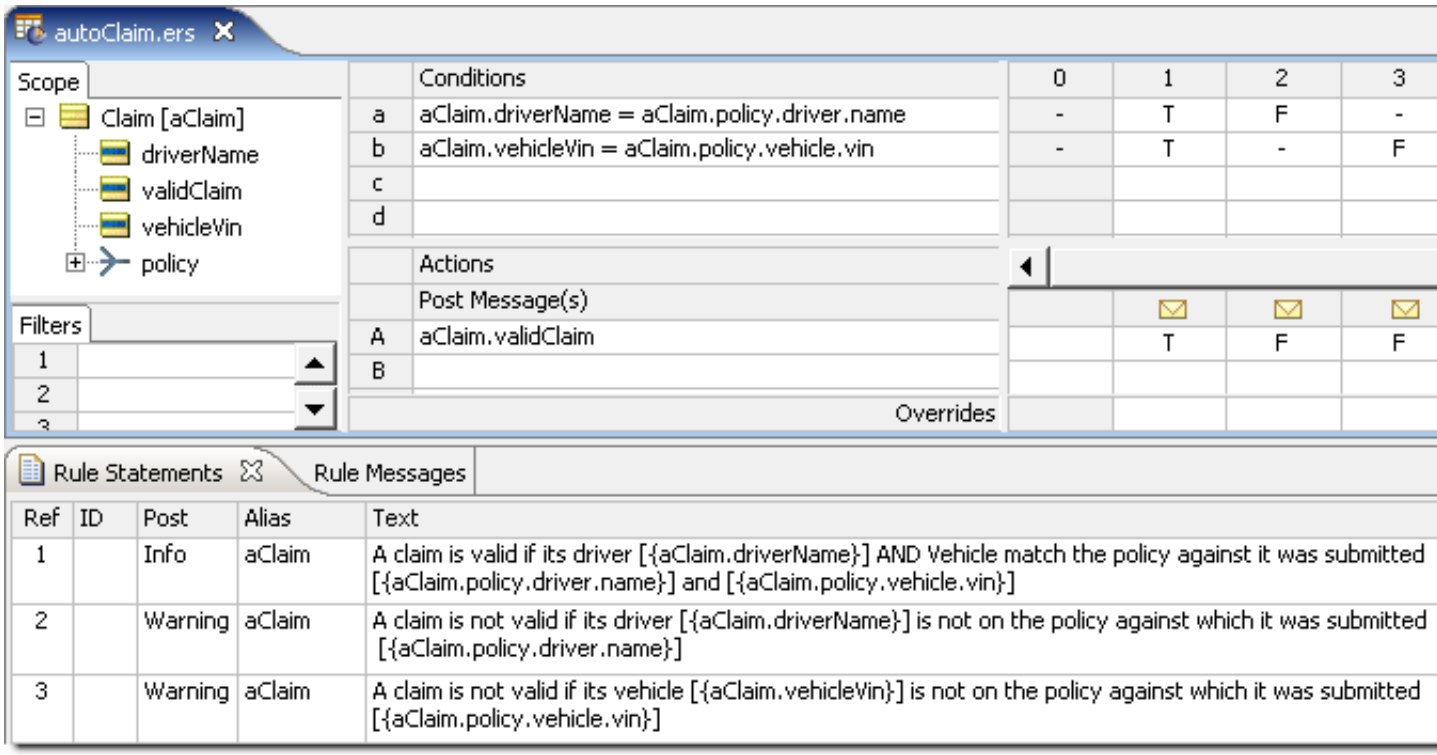
Figure 82: Vocabulary for insurance claims



Model the following rules in Corticon.js Studio, as shown:

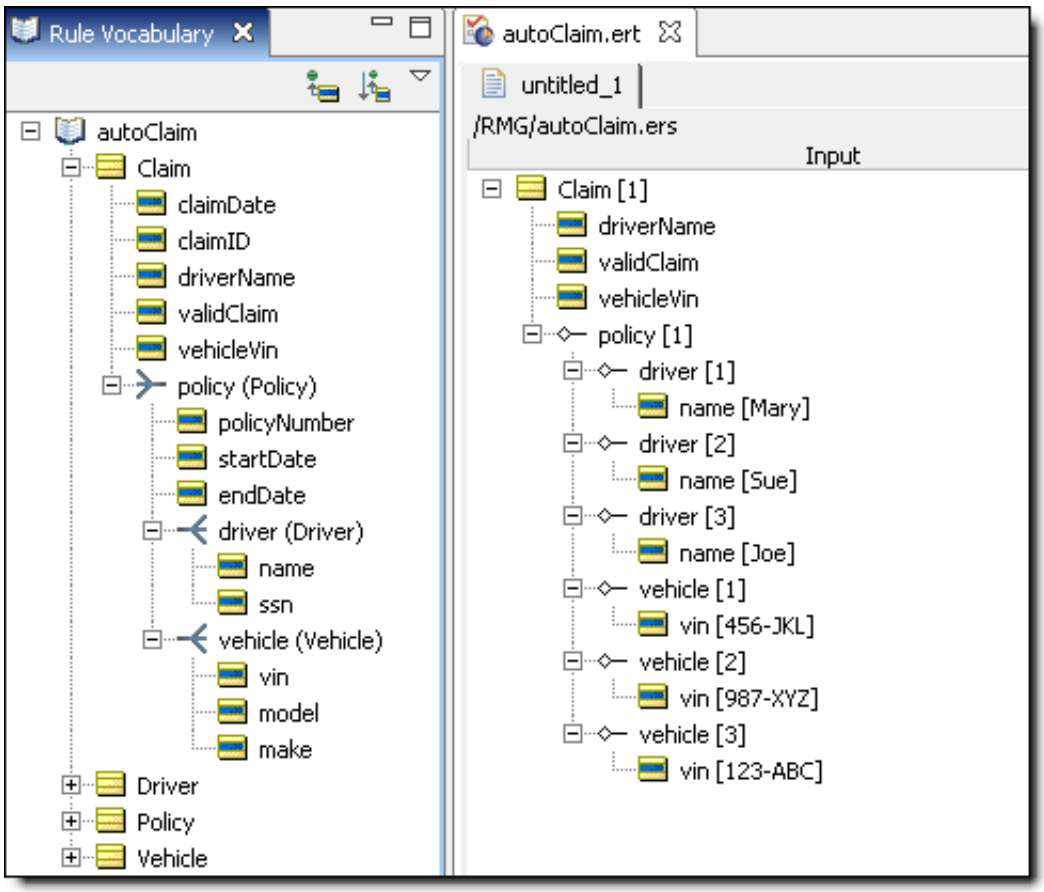
1. For a claim to be valid, the driver's name and vehicle ID listed on the claim must also be listed on the claim's policy.
2. If either the driver's name or vehicle ID on the claim is not listed on the policy, then the claim is not valid.

Figure 83: Rulesheet for insurance claims



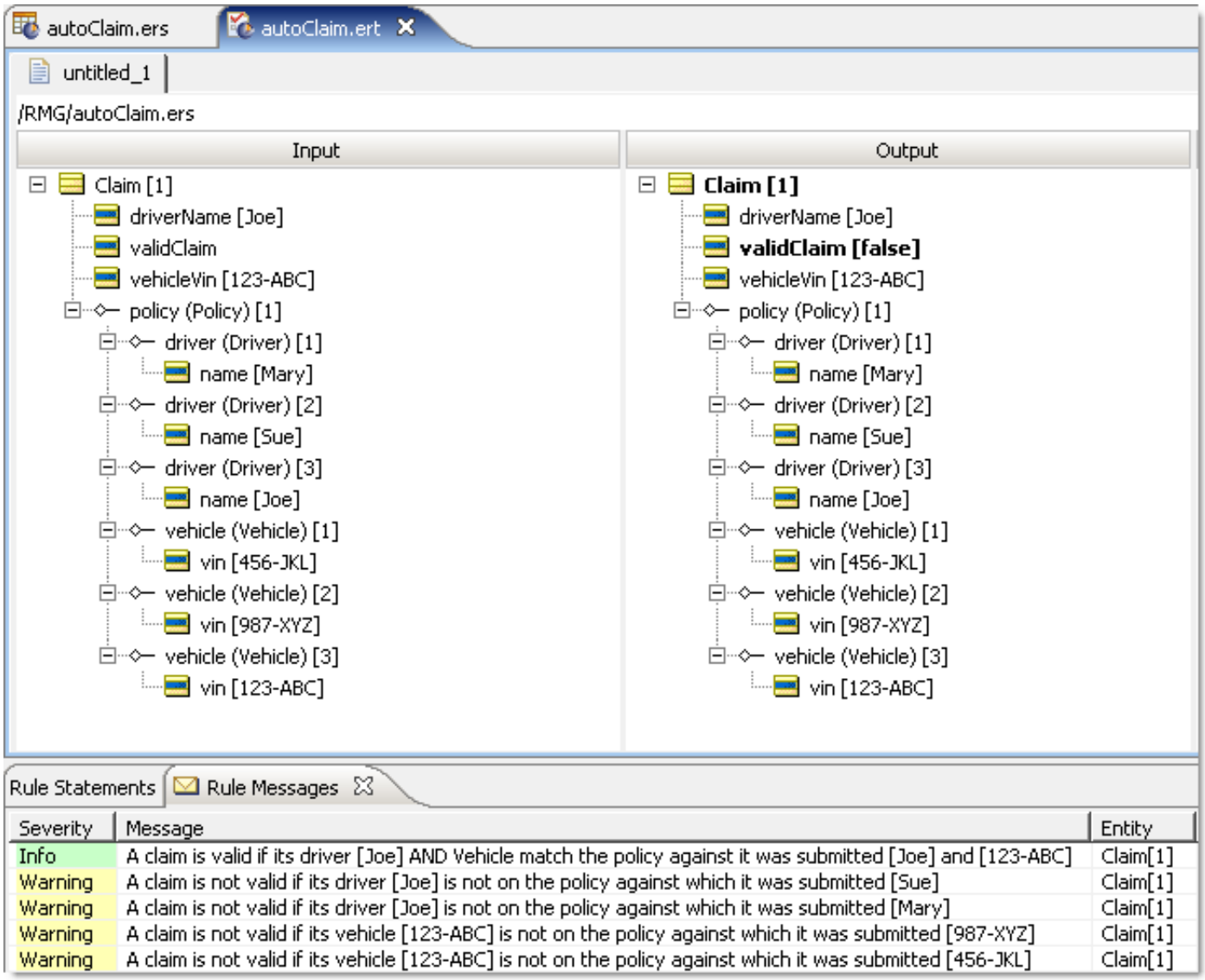
This appears very straightforward. But a problem arises when there are multiple drivers or vehicles listed on the policy. In other words, when the policy contains a collection of drivers or vehicles. The Vocabulary permits this scenario because of the cardinalities that were assigned to the various associations.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

Figure 84: 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 85: Ruletest output for insurance claims



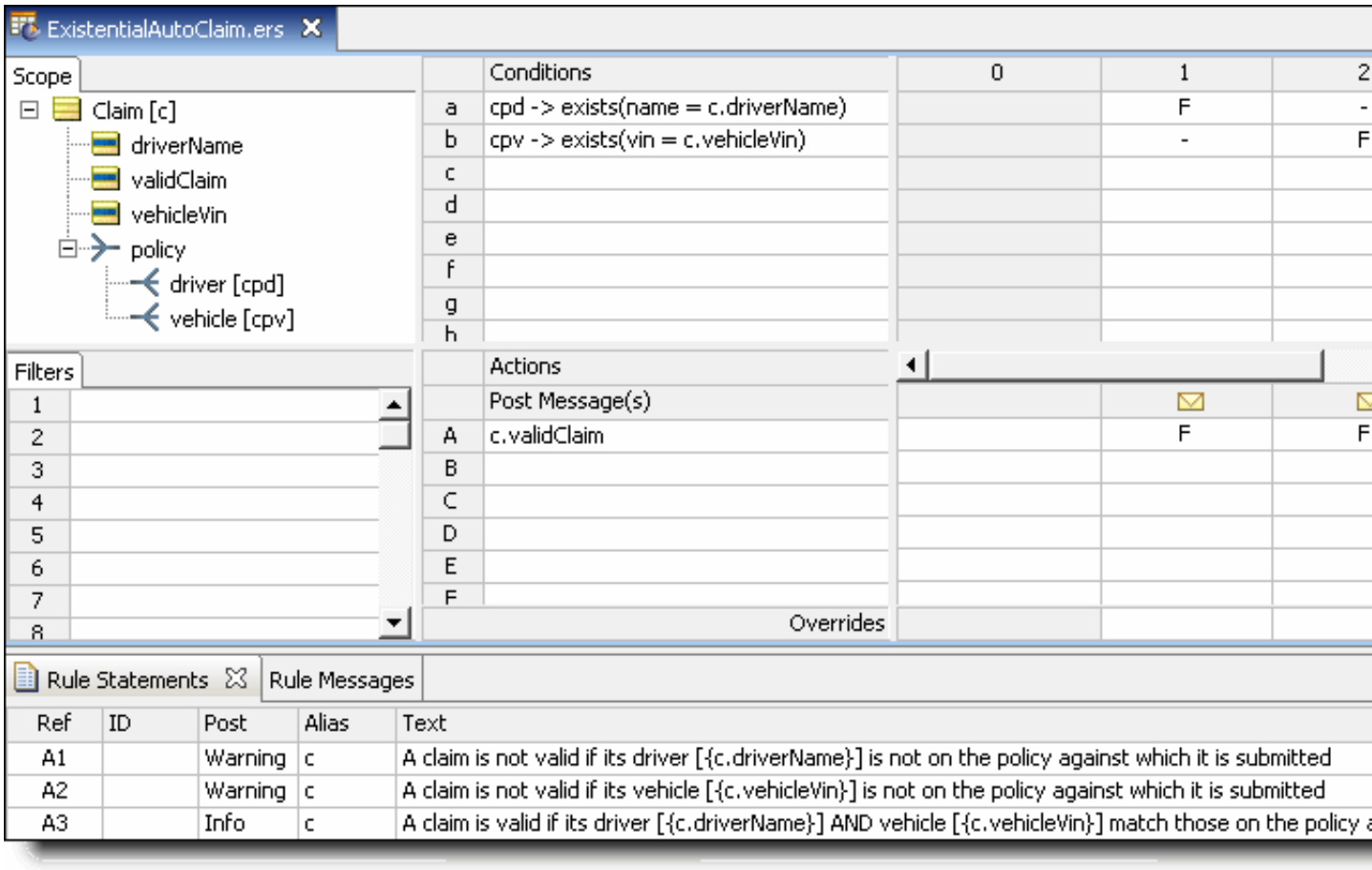
As you can see from the Ruletest results, the way Corticon.js Studio evaluates rules involving comparisons of multiple collections means that the `validClaim` attribute may have inconsistent assignments – sometimes `true`, sometimes `false` (as in this Ruletest). It can be seen from the following table 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

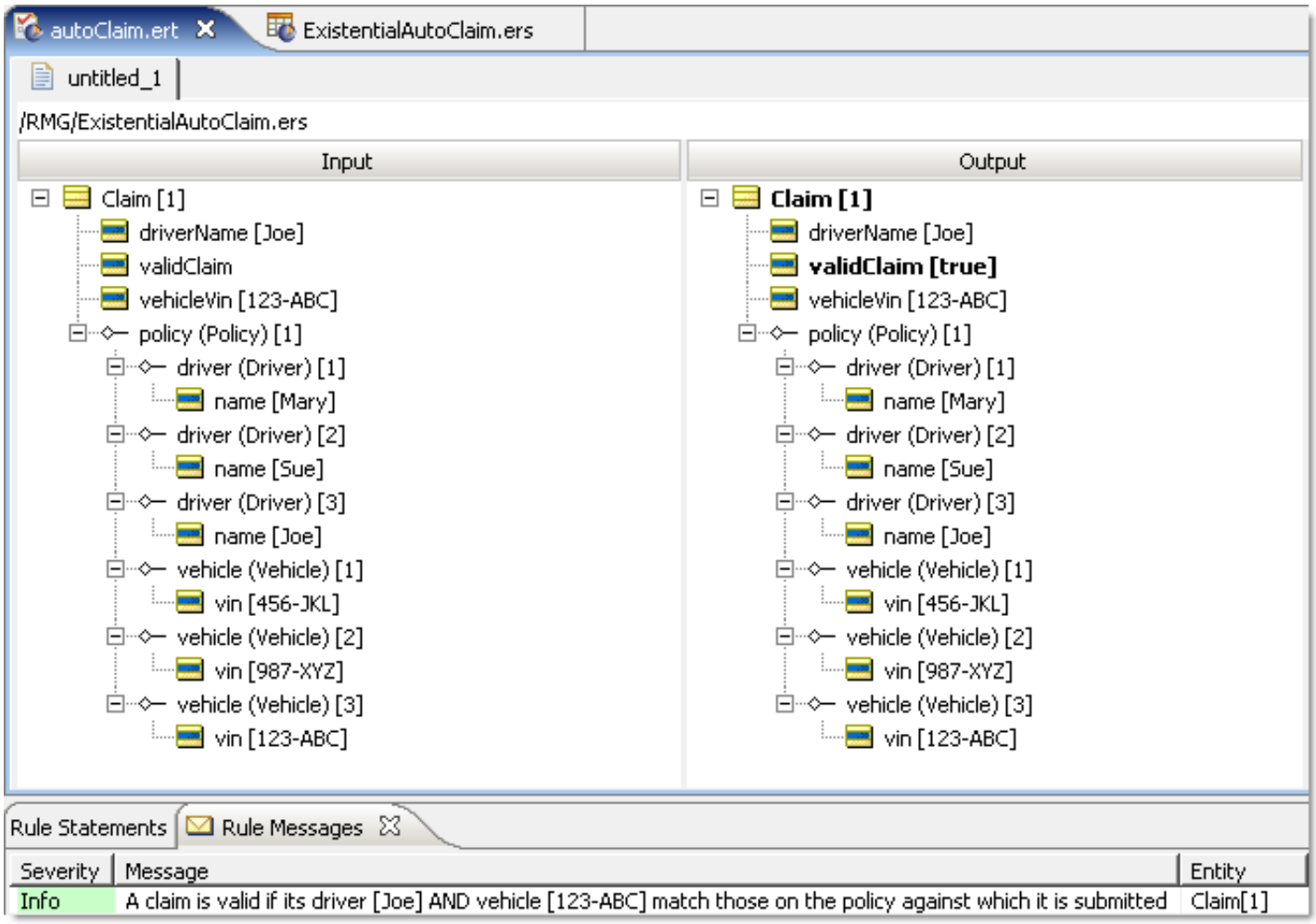
The existential quantifier will be used to rewrite these rules:

Figure 86: 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.

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.js 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.js 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.js 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

Operator precedence and order of evaluation

Operator precedence is the order in which Corticon.js 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.js 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.js'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.js 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.js's logical analysis.

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

Data type compatibility and casting

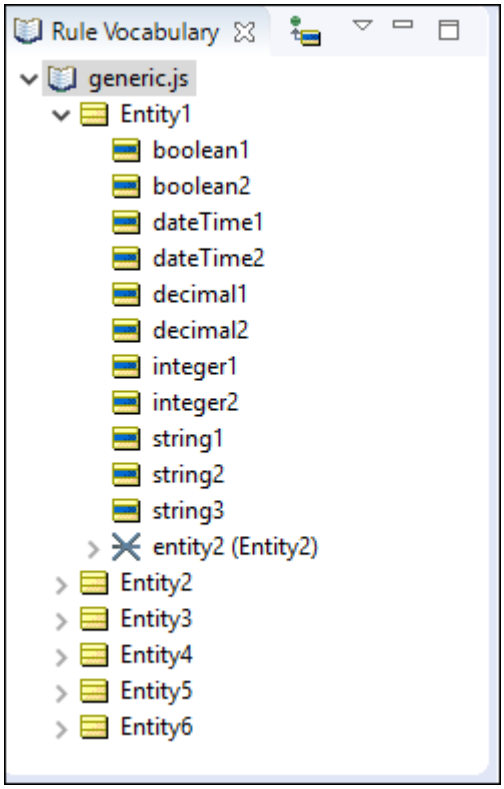
An important prerequisite of any comparison or assignment operation is data type compatibility. In other words, the data type of the equation's LHS (the data type of `A`) must be compatible with whatever data type results from the evaluation of the equation's RHS (the data type of `B`). For example,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

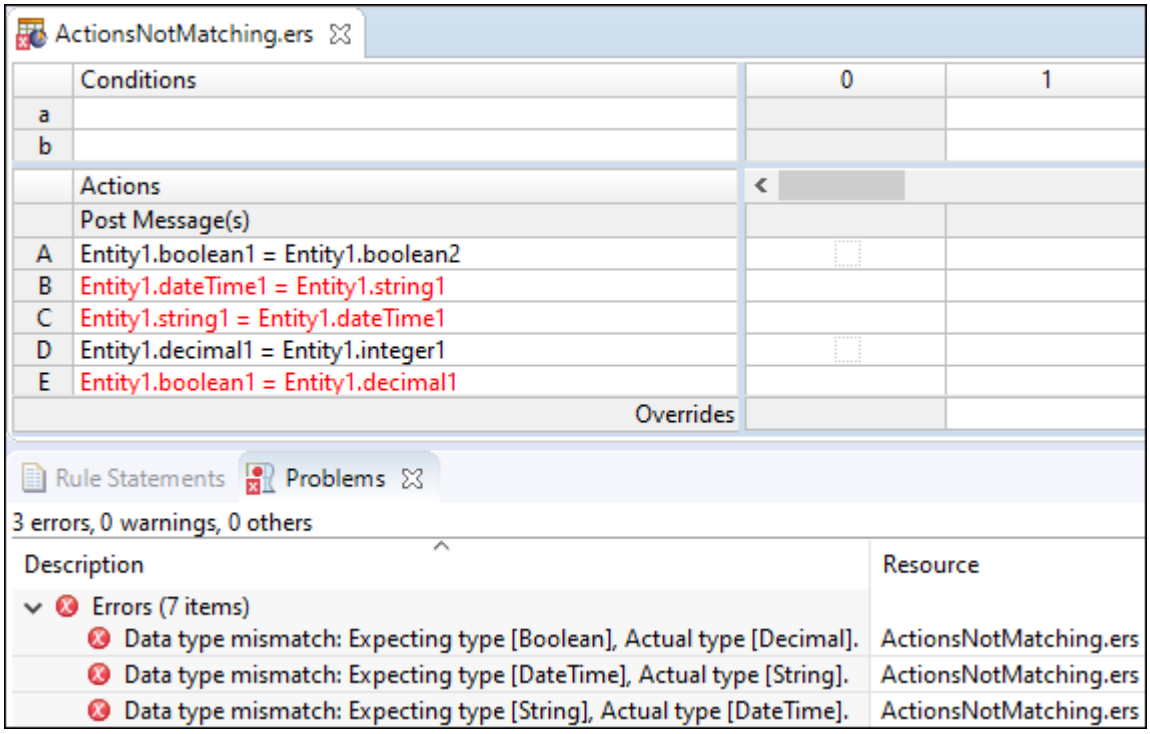
In the examples that follow, the generic Vocabulary from the *Rule Language Guide* will be used because the generic attribute names indicate their data types:

Figure 87: 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 88: Data type mismatches in assignment expressions



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

A—This expression is valid because the data types of the LHS and RHS sides of the equation are compatible. They 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.

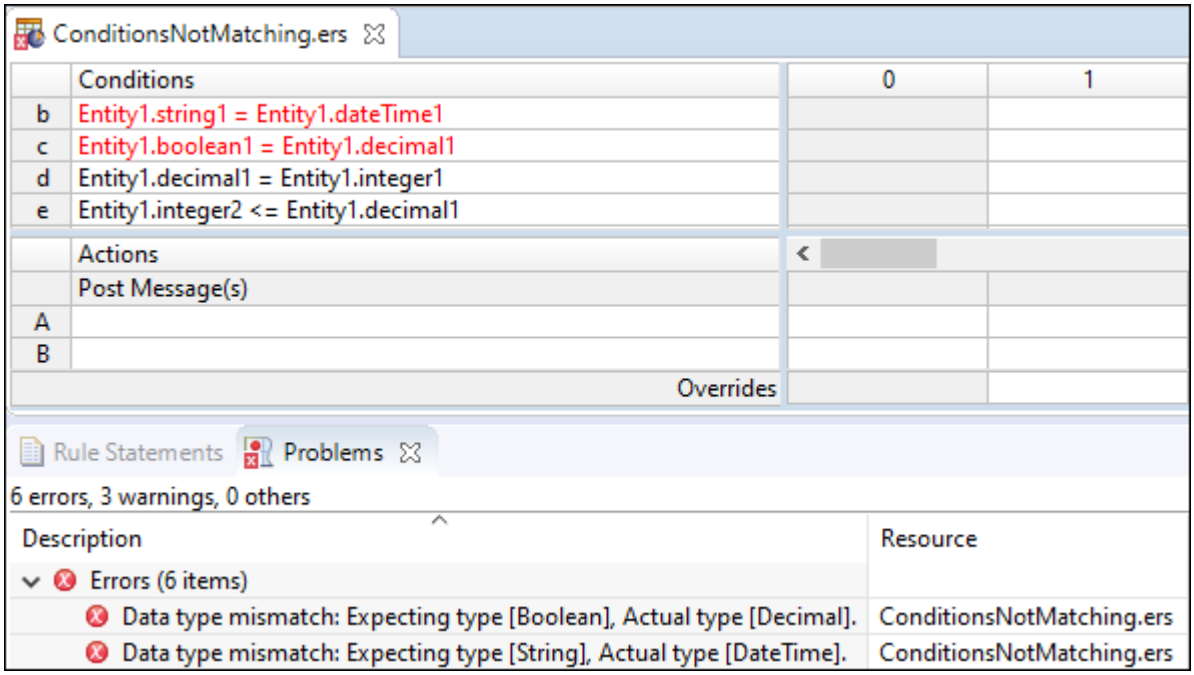
D—This expression is valid because the data types of the LHS and RHS sides of the equation are compatible *even though they are different!* This is an example of the one exception to Corticon's general rule regarding data type compatibility: Decimals can hold Integer values.

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 89: 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.js 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.js 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.

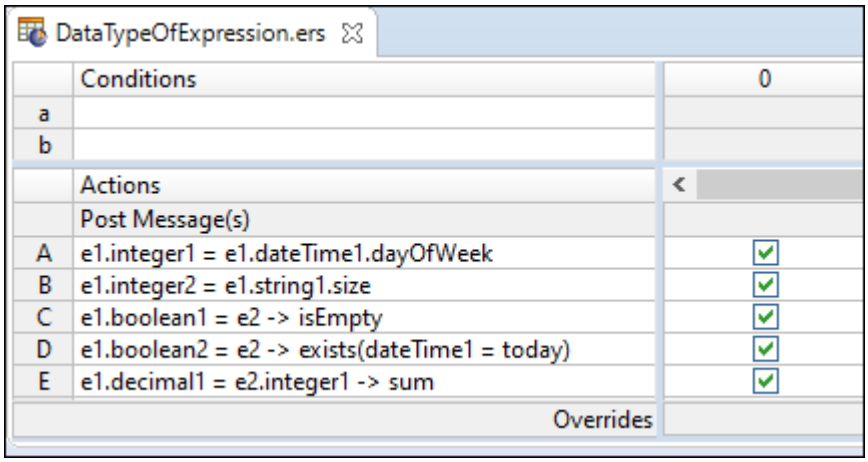
d—This comparison expression is valid because the data types of the LHS and RHS sides of the equation are compatible. This is another example of the one exception to Corticon's general rule regarding data type compatibility: Decimals can be compared to Integer values.

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 90: 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.

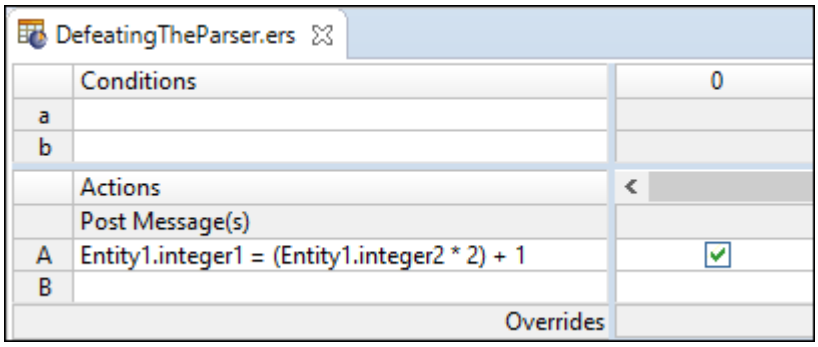
E—the RHS of this equation resolves to an Integer because the `->sum` collection operator adds up the values of all occurrences of an attribute (in this case, `integer2`) in a collection (in this case, `e2` again). Since the LHS has a Decimal data type, the assignment operation is valid. This is the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

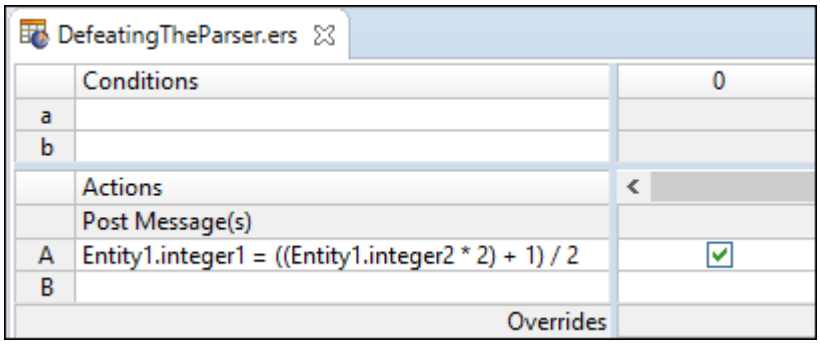
The part of Corticon.js Studio that checks for data type mismatches (along with all other syntactical problems) is the Parser. The Parser ensures that whatever is expressed in a Rulesheet can be correctly translated and compiled into code executable by Corticon.js Studio's Ruletest as well as by the 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 91: LHS and RHS resolve to integers



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 92: Will the RHS still resolve to an integer?



The minor change of adding a division step to the RHS expression has a major effect on the data type of the RHS. Prior to modification, the RHS 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 JS datatypes with casting operators

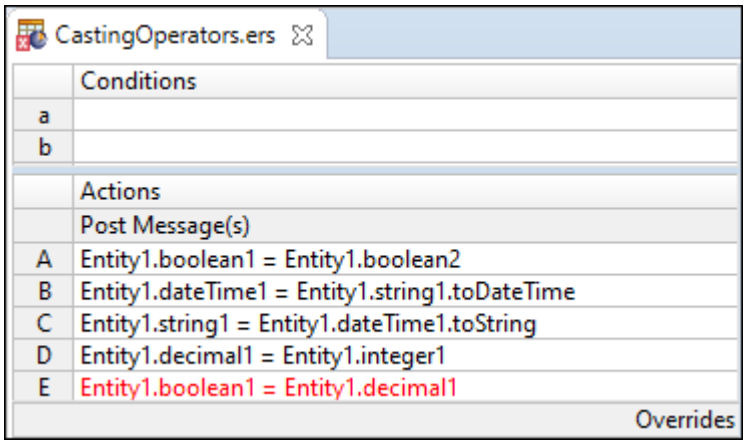
A special set of operators is provided in the Corticon.js Studio's Operator Vocabulary that allows the rule modeler to control the data types of attributes and expressions. These casting operators are described below:

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	String
`.toDateTime`	String	DateTime

The problems shown in Datatype mismatches in comparision expressions can use these casting operators to make corrections:

Figure 93: 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.

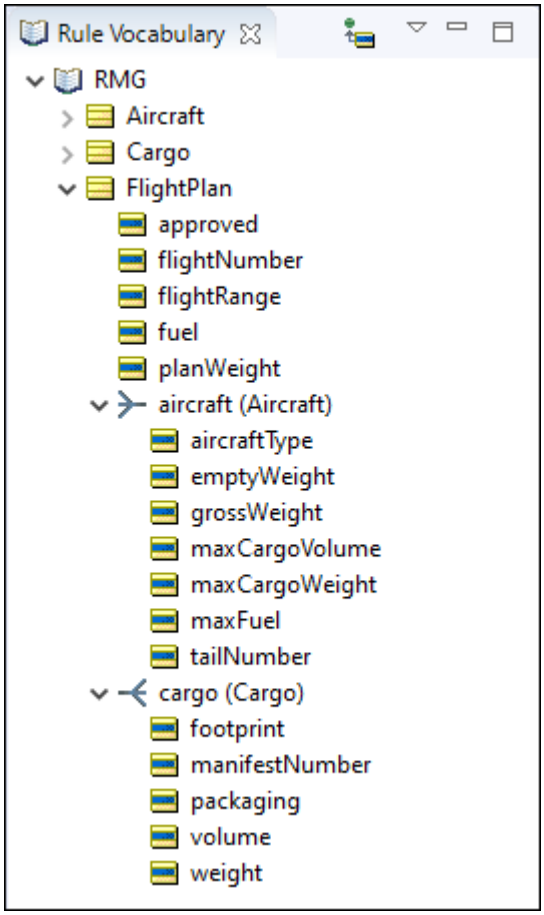
Supported uses of calculation expressions

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

- Calculation as an assignment in a noncondition on page 118
- Calculation as a comparison in a condition on page 118
- Calculation as an assignment in an action on page 120

To make the examples more interesting and allow for a bit more complexity in our 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 94: Basic Tutorial Vocabulary extended



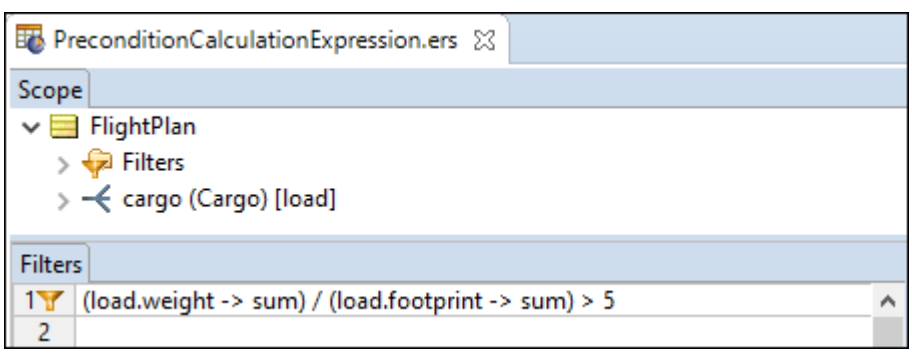
The new attributes are described in the following table:

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:

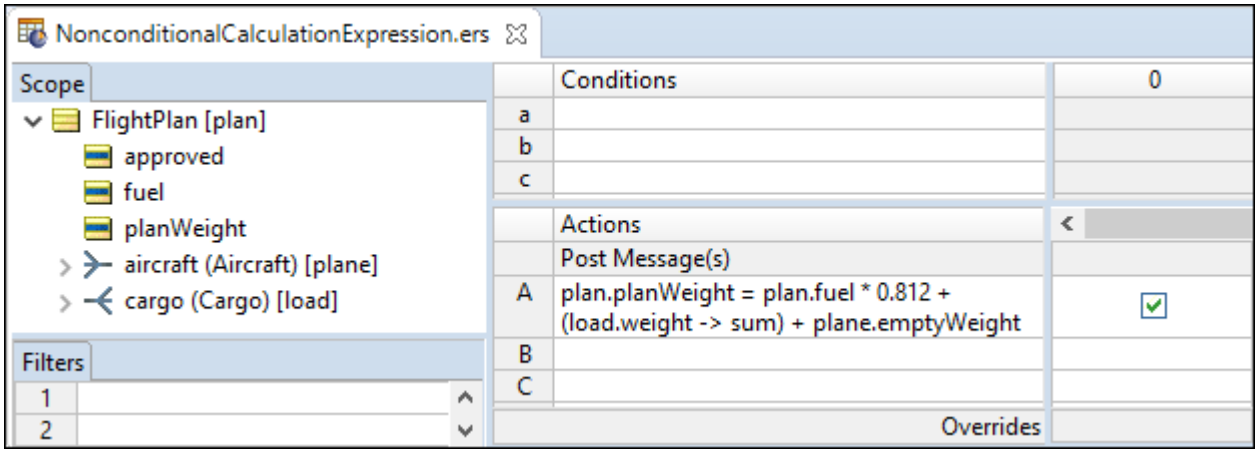


The LHS of the expression calculates the average pressure exerted by the total cargo load on the floor of the aircraft (sum of the cargo weights divided by the sum of the cargo containers' footprints). This result is compared to the RHS, which is the literal value 5. You might expect to see this type 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 95: 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.

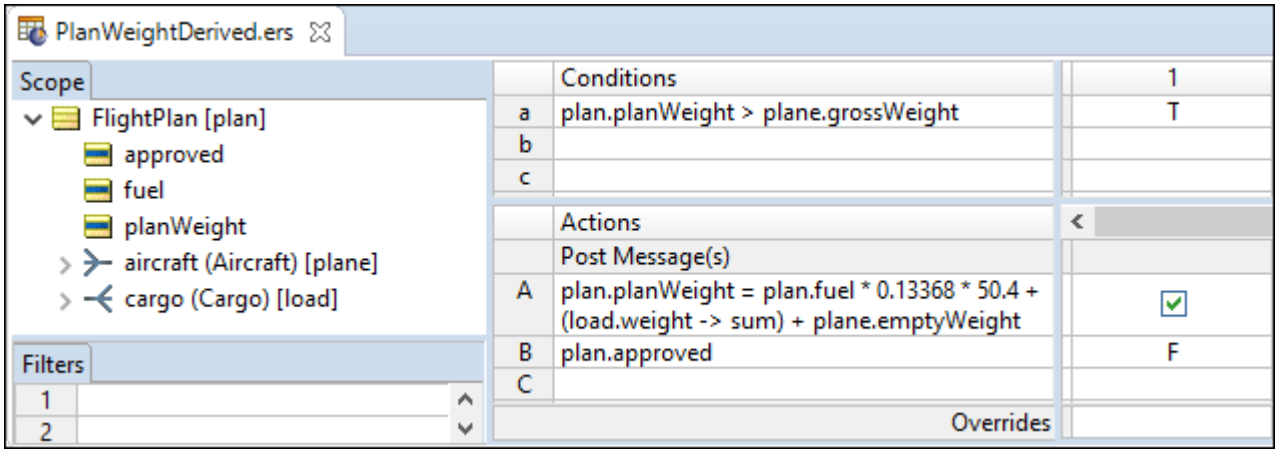
Calculation as a comparison in a condition

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 96: `planWeight` Derived and used in same Rulesheet

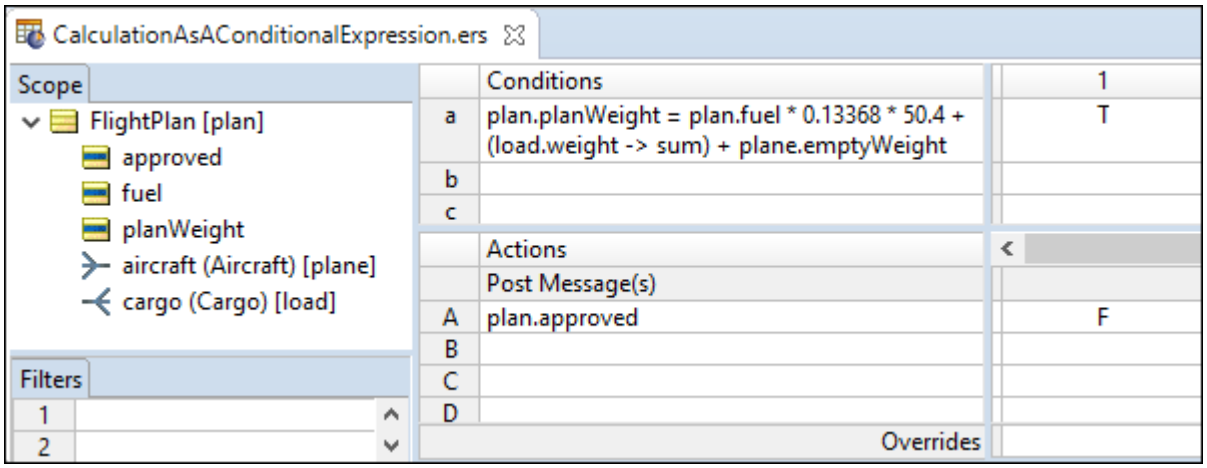


In Condition row a, `planWeight` is compared to the aircraft's `grossWeight` to make sure that the aircraft is not overloaded. An overloaded aircraft must not be allowed to fly, so the `approved` attribute is assigned a value of `false`.

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 97: Calculation in a conditional expression

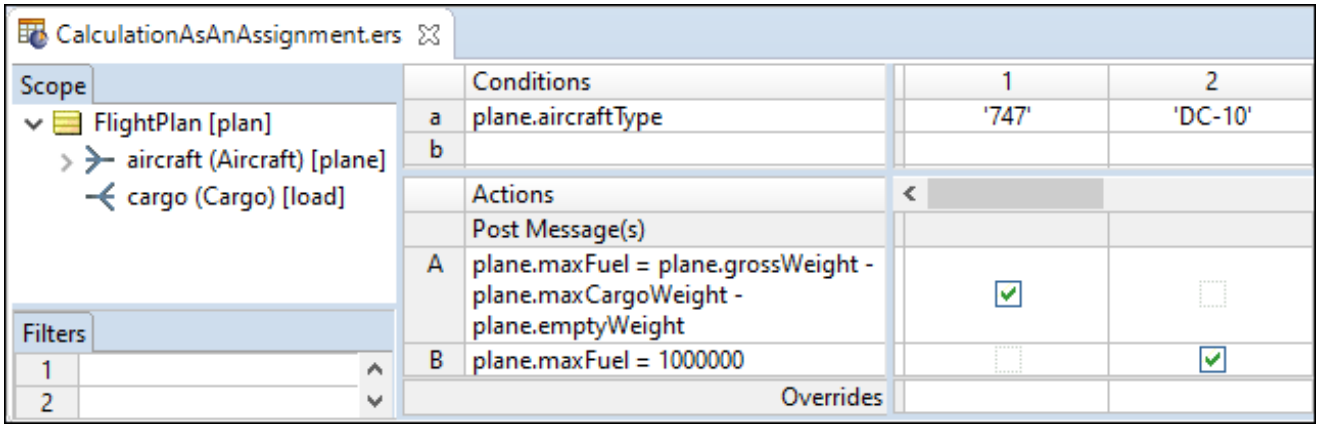


This approach might be preferable if the results of the calculation were not expected to be reused, or if adding an attribute like `planWeight` to the Vocabulary were not possible.

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 98: A calculation in an action expression



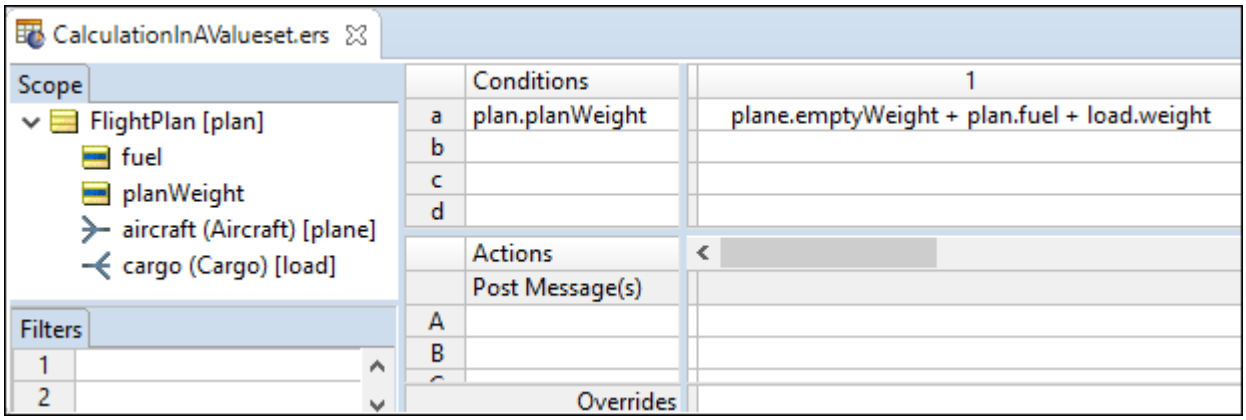
In rule 1, the `maxFuel` load for 747s is derived by subtracting `maxCargoWeight` and `emptyWeight` from `grossWeight`. In rule 2, `maxFuel` for DC-10s is assigned the literal value `100000`.

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 99: Calculation in a Values Cell and Column



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 100: Calculation in a Rule Statement

Rule Statements

Ref	ID	Post	Alias	Text
1				2 * 3 + 4

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

Figure 101: Embedding a calculation in a rule statement

Rule Statements

Ref	ID	Post	Alias	Text
1				The value of maxFuel squared is {plane.maxFuel ** 2}
				Invalid token: [**].

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.

7

Rule dependency in chaining

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.

Forward chaining

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 only mode of rule processing for all Corticon.js 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).

8

Filters

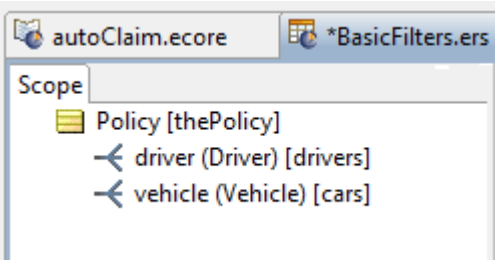
A Filter expression acts to limit or reduce the data in working memory to only that subset whose members satisfy the expression. A Filter *does not* permanently remove or delete any data; it simply *excludes* data from evaluation by other rules in the same Rulesheet.

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

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

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

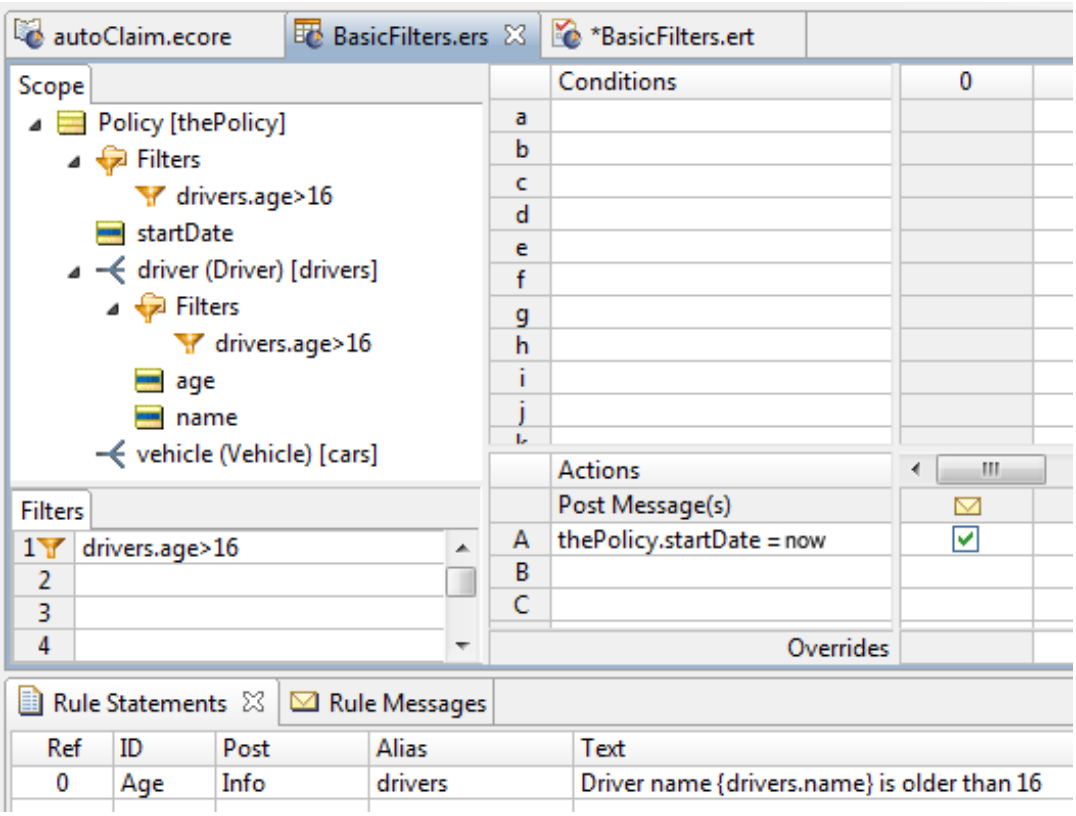
Figure 102: Aliases Declared



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

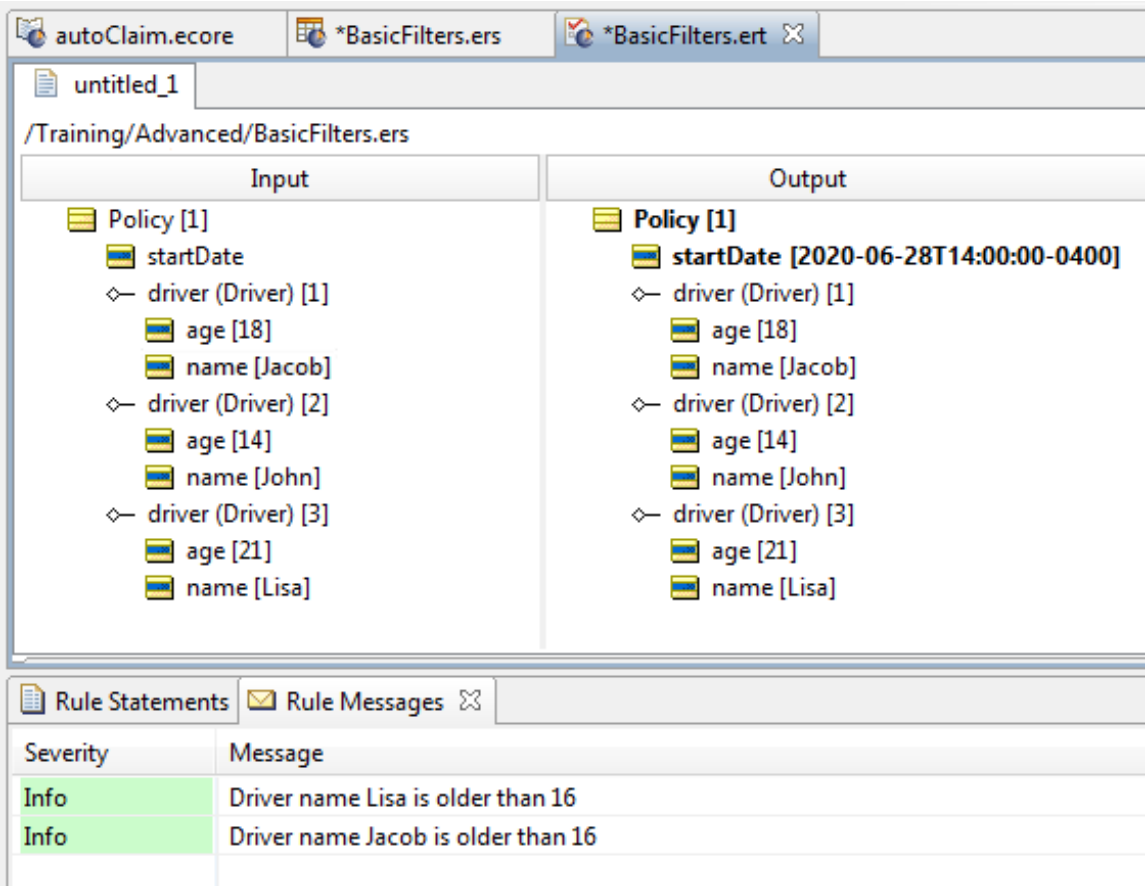
To start with, we will write a simple Filter and observe its default behavior. In the simple scenario below, the Filter expression reduces the set of data acted upon by the Nonconditional rule (column 0), which in this case merely posts the Rule Statement as a message.

Figure 103: Rulesheet to Illustrate Basic Filter Behavior



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

Figure 104: 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.

For details, see the following topics:

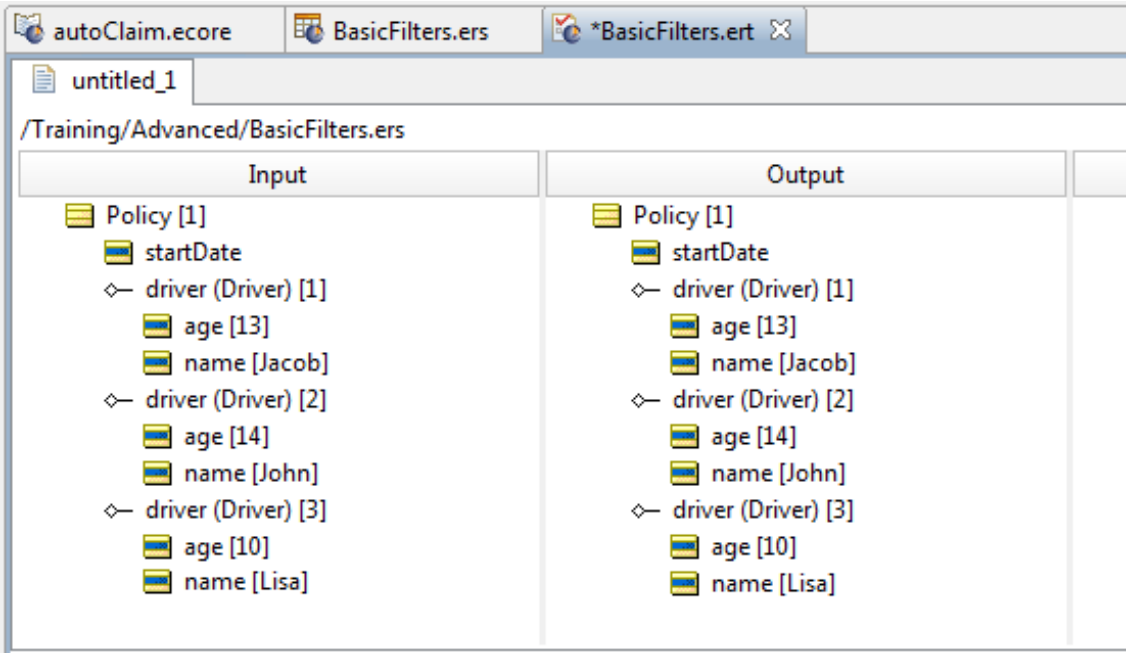
- Full filters
- Limiting filters
- Filters that use OR
- What is a precondition
- How to use collection operators in a filter

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 105: Ruletest for Full Filter

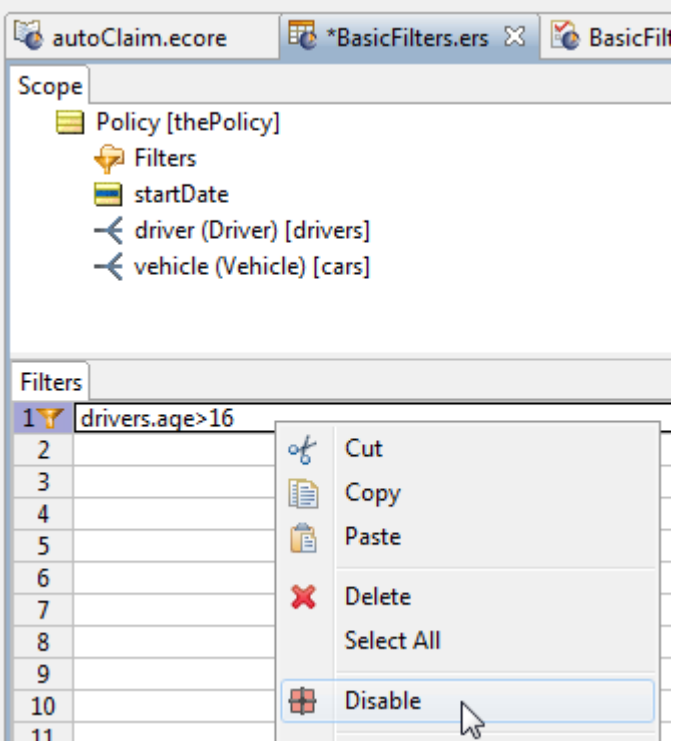


Notice two important things about this Ruletest's results: first, none of the `Driver` entities in the Input are older than 16, which means none of them survives the filter. Second, because the parent `Policy` entity does not contain at least one `Driver` that satisfies the filter, then the parent `Policy`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

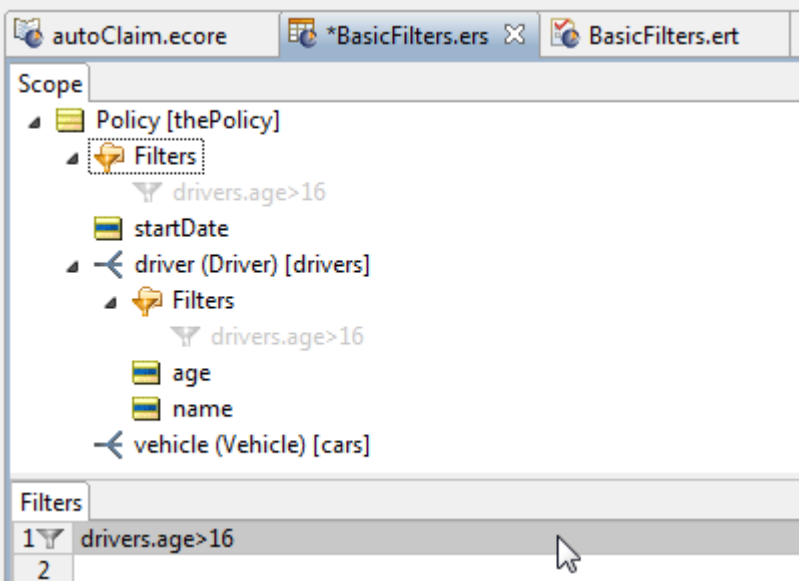
Why would you want a Filter to behave this way? Perhaps because, if these are the only drivers seeking a policy, then there must be at least one driver of legal age to warrant issuing a policy. While you will probably find that the full filter behavior is generally what you want when filtering your data, 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:



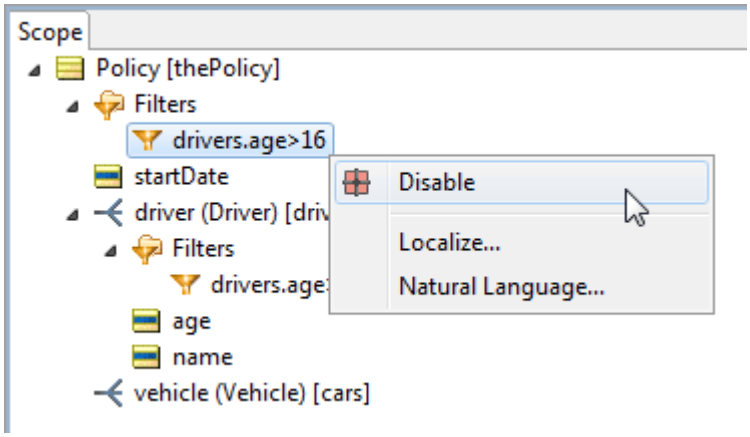
A disabled full filter is really no filter at all. You can perform the corresponding action to again **Enable** the filter.

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.

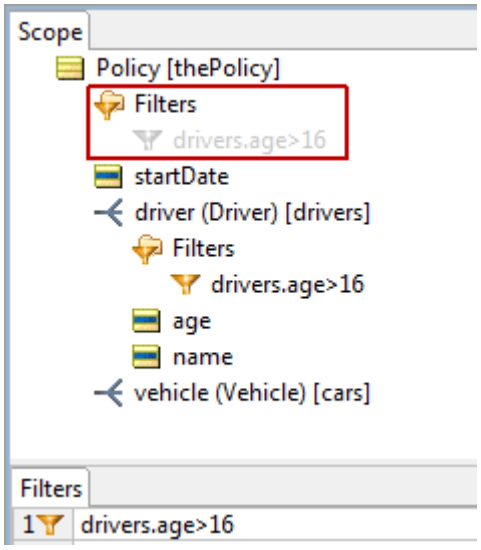
To turn a Filter expression into a limiting filter, right-click on a filter in the scope section and select **Disable** from the menu, as shown:

Figure 106: Selecting to limit a filter



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

Figure 107: 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.
```

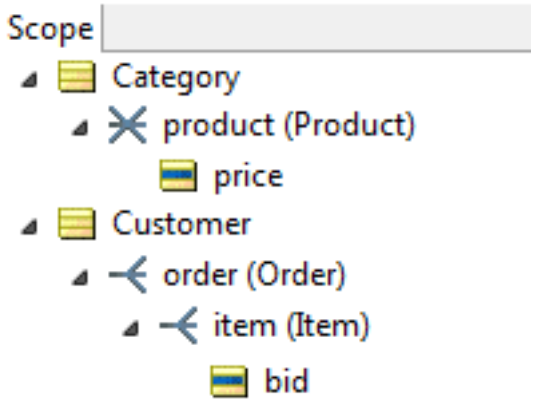
You see how one filter can apply limits to the full filter to achieve the preferred profile of what survives the filter and what gets filtered out.

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

Example of limiting filters

Consider the following Rulesheet Scope of a Vocabulary:

Figure 108: Scope in a Rulesheet that will be filtered

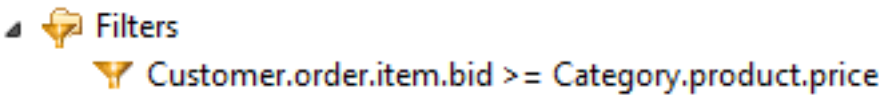


Consider the filter to be applied to data:

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

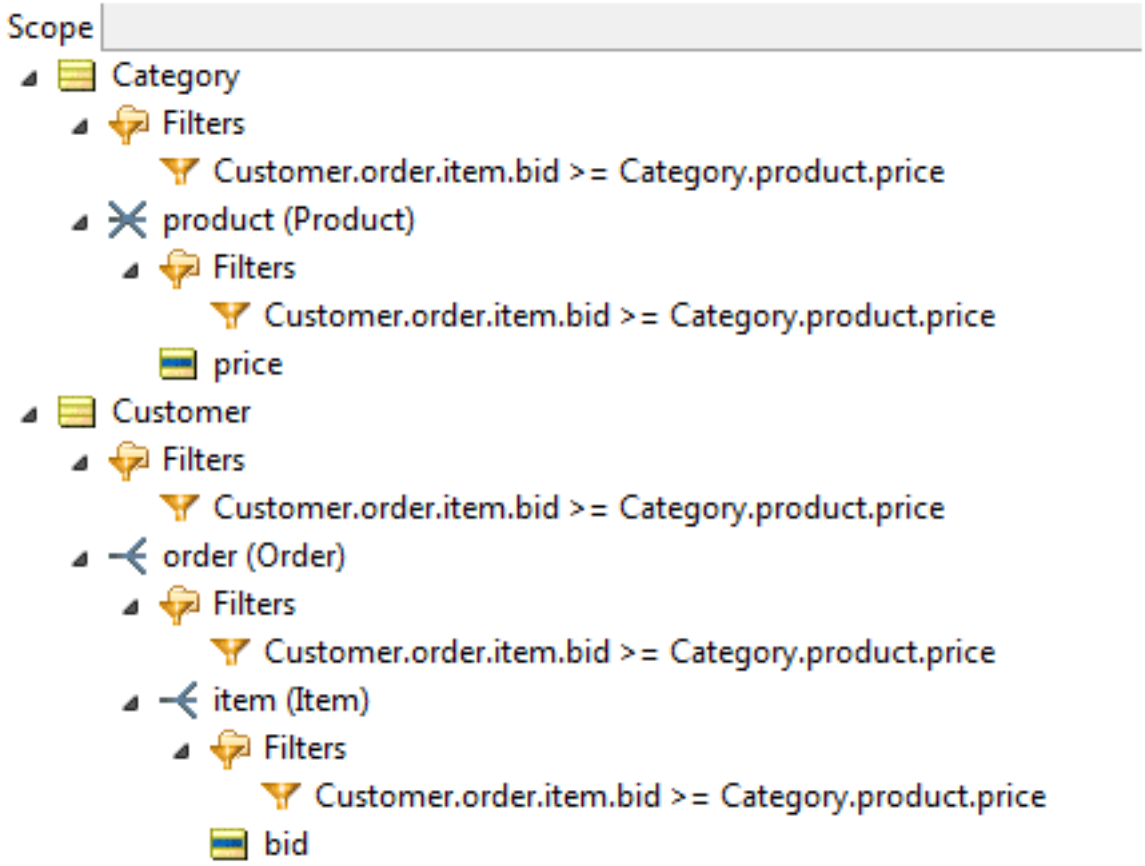
This is shown in the Rulesheet's **Filters** section as:

Figure 109: Definition of a filter



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

Figure 110: Application of the filter to the Scope's tree structure



A Ruletest Testsheet might be created as follows:

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

This filter is evaluated as follows:

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

The tuples that evaluate to `true` are:

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

The entities that survive the filter are:

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

The **Scope** section of the Rulesheet expands as follows:

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

- 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.

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.

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.

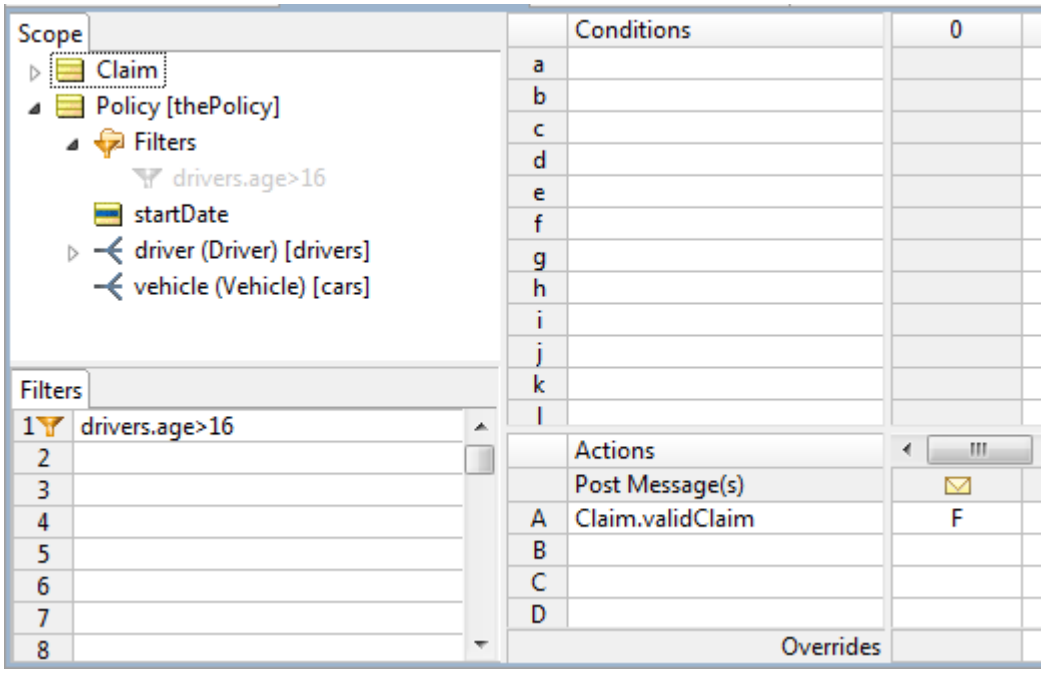
Precondition behavior of a filter ensures that execution of a Rulesheet **stops** unless *at least one* piece of data survives the filter. If execution of a Rulesheet stops because no data survived the filter, then execution moves on to the next Rulesheet (in the case where the Rulesheet is part of a 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.

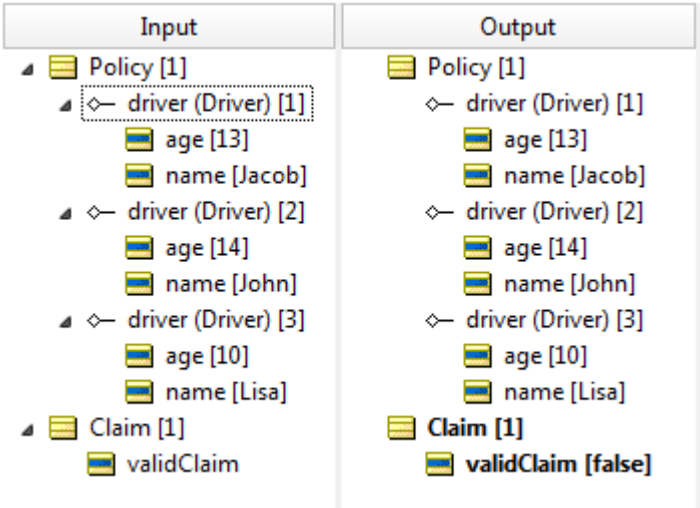
The precondition behavior of a filter is significant because it allows you to control Rulesheet execution regardless of the scope used in the rules. Take, for example, the Rulesheet shown in the following figure. The filter in row 1 is acting in its standard default mode of full filter. This means that `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 111: 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:

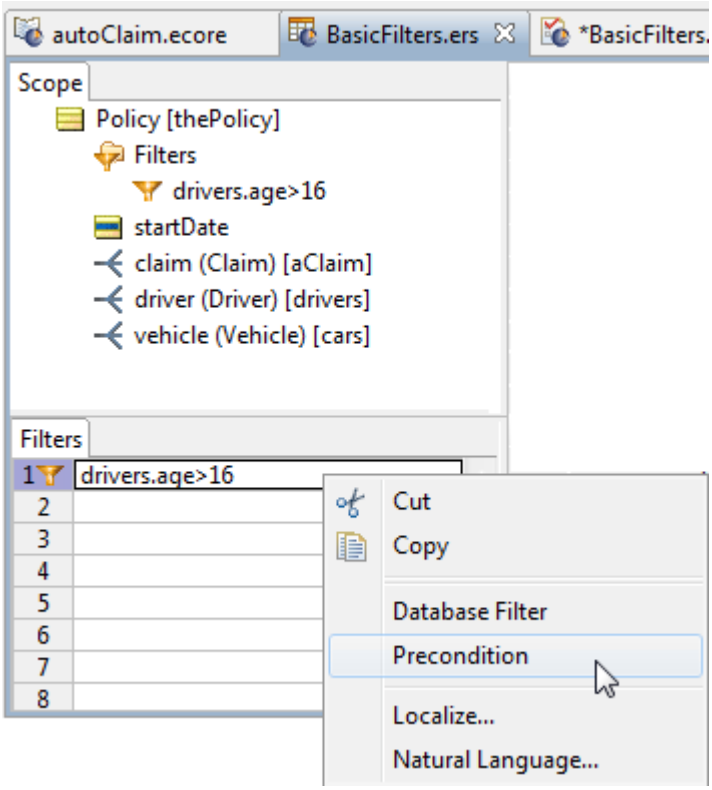
Figure 112: Rulesheet for an action unaffected by a filter



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 113: Selecting precondition behavior from the filter menu



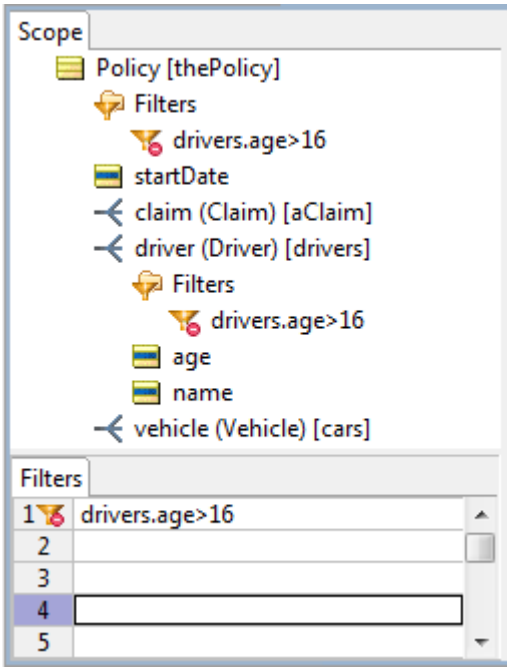
Note that the two options, **Precondition** and **Limiting Filter**, are mutually exclusive: turning one on turns the other off. A filter cannot be both a precondition **and** a limiting filter because at least one piece of data **always** survives a limiting filter, so a precondition never stops execution.

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

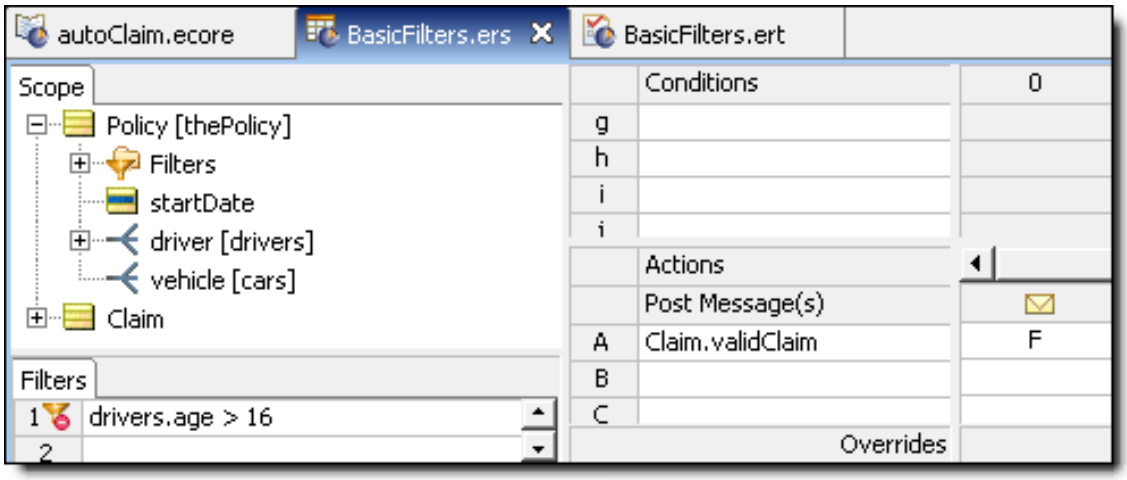
The following figure shows a filter in **Precondition** mode.

Figure 114: A Filter in Precondition Mode



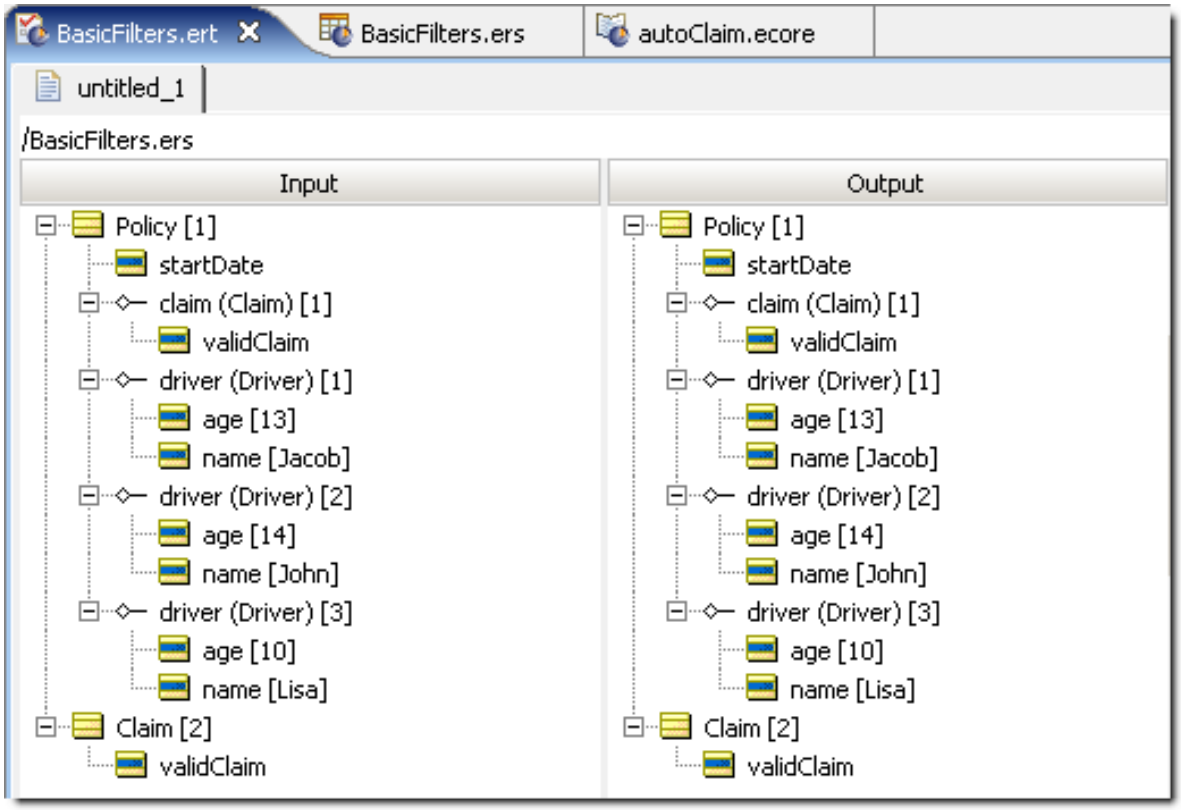
As described before, the precondition behavior of the filter causes Rulesheet execution to stop whenever no data survives the filter. So, in the original case where `Policy` and `Claim` were unassociated, a filter in Precondition mode accomplishes the business intent without artificially changing the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

Figure 115: Rulesheet with a filter in Precondition mode



A final proof is provided in the following figure:

Figure 116: 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.js.

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 117: A Condition/Action rule column with 2 Conditional rows

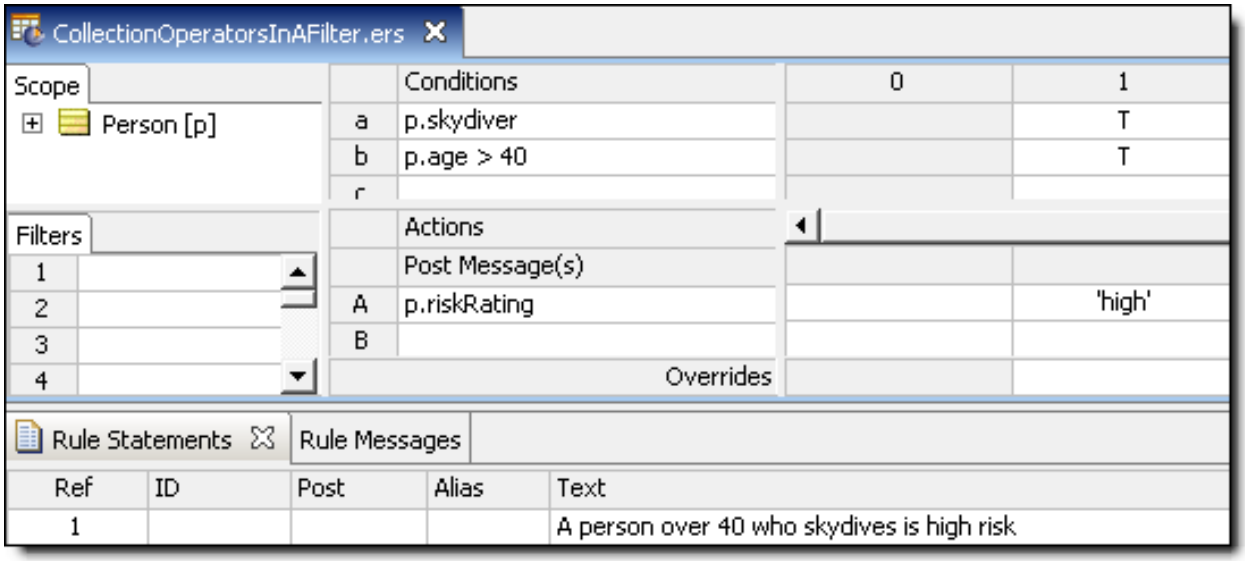


Figure 118: Rulesheet with one Condition row moved to Filters row

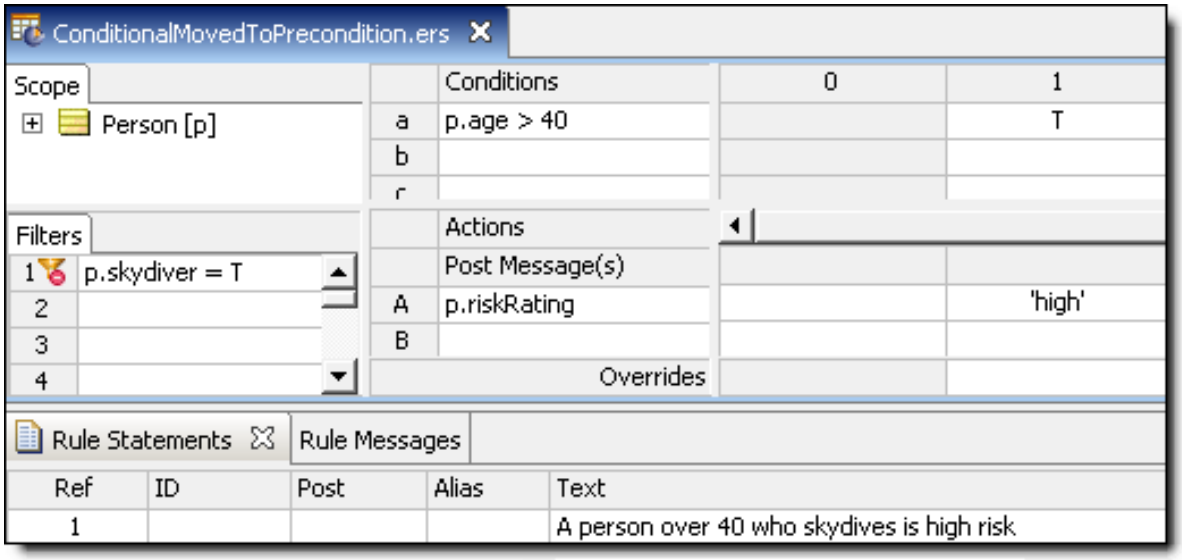
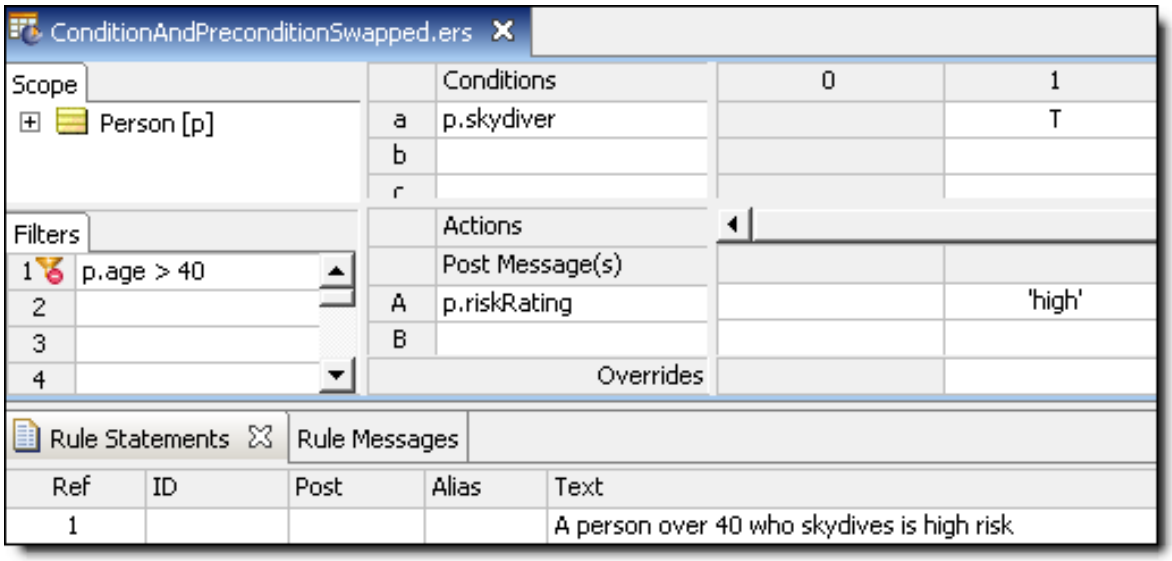


Figure 119: Rulesheet with Filter and Condition rows swapped



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

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

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

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

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

This independence of order is similar to the commutative property of multiplication: 4 x 5 = 20 and 5 x 4 = 20. Aliases work perfectly well in a declarative language (like Corticon.js's) because regardless of the order of processing, the outcome is always 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 120: Collection operator in Condition row

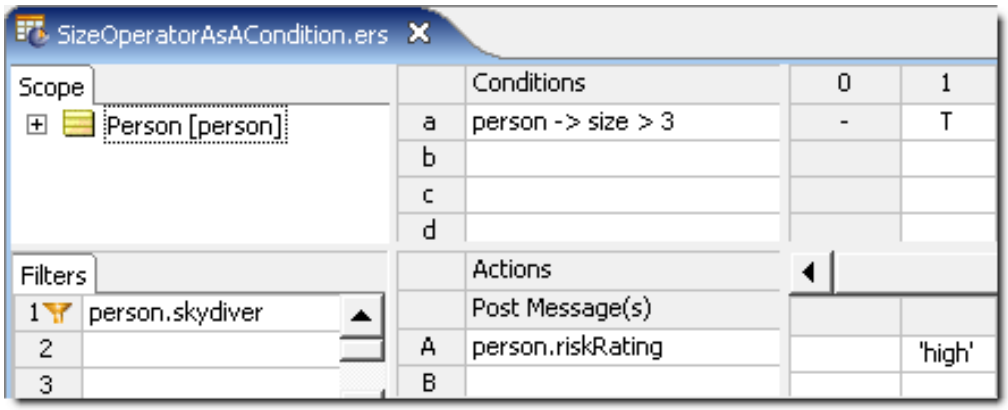
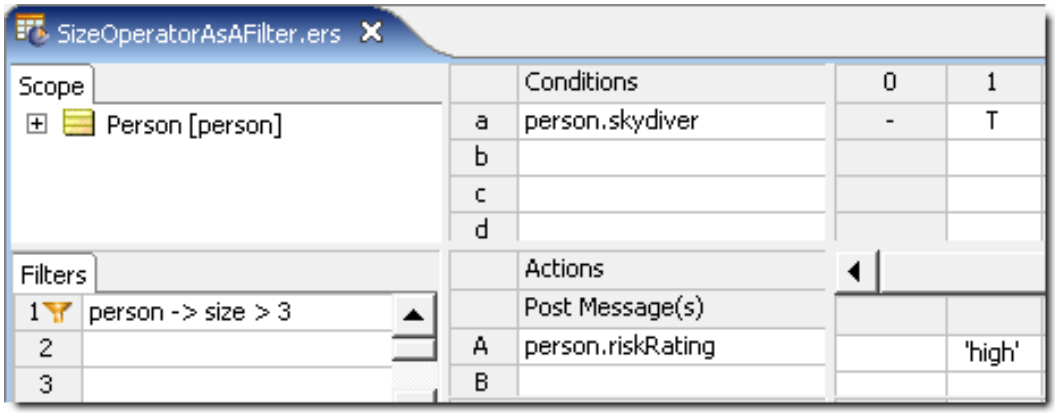
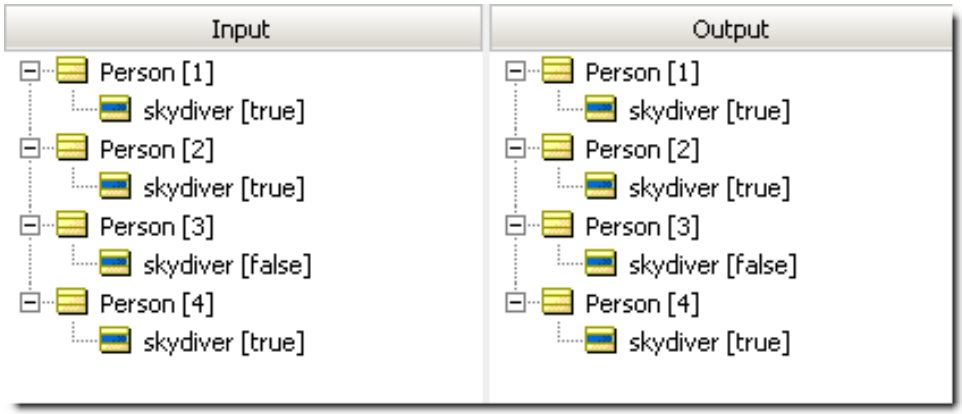


Figure 121: Collection operator in Filter row



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:

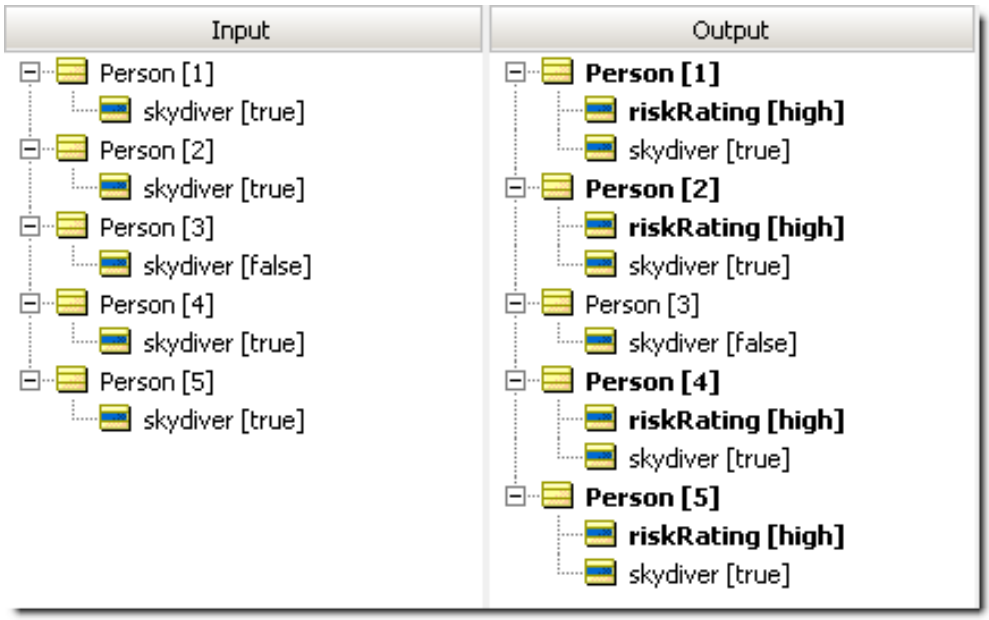
Figure 122: Ruletest with three skydivers



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:

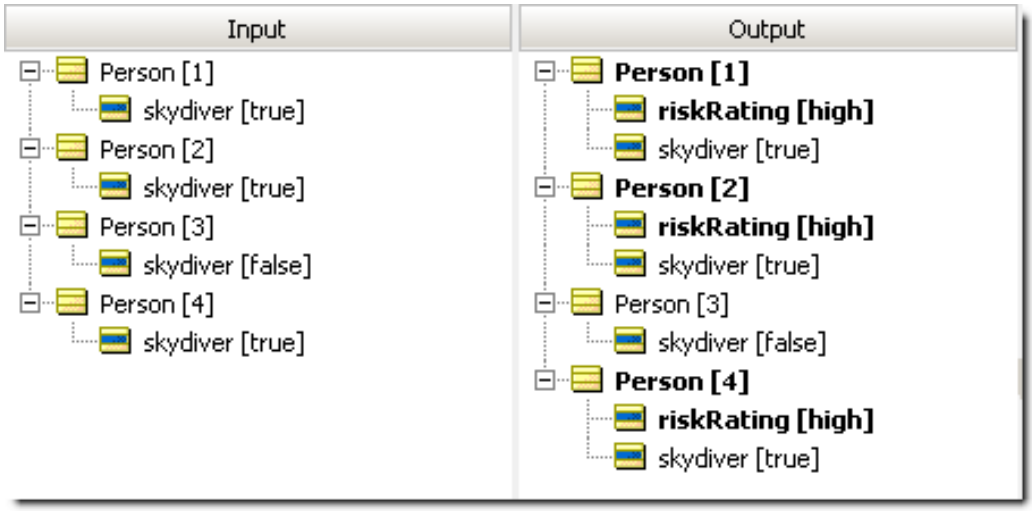
Figure 123: Ruletest with four skydivers



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 124: 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 125: Rulesheet with two conditions

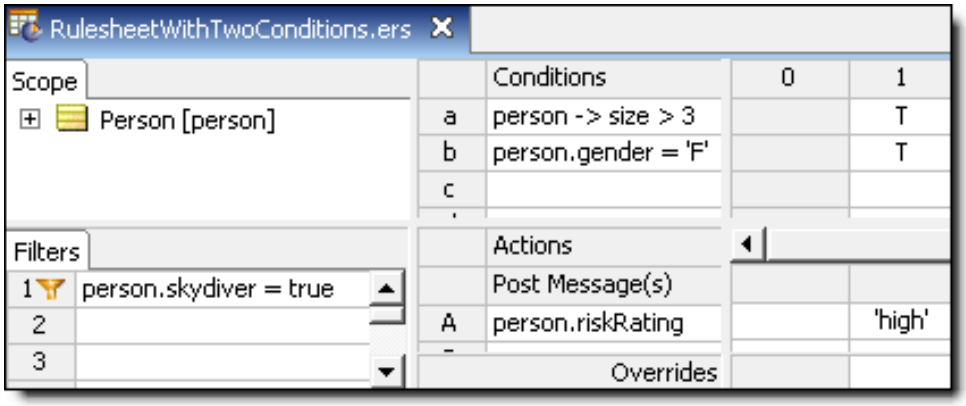
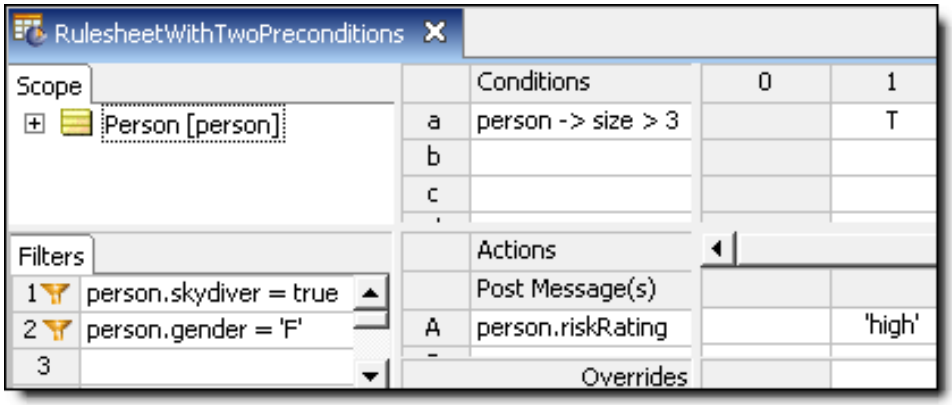
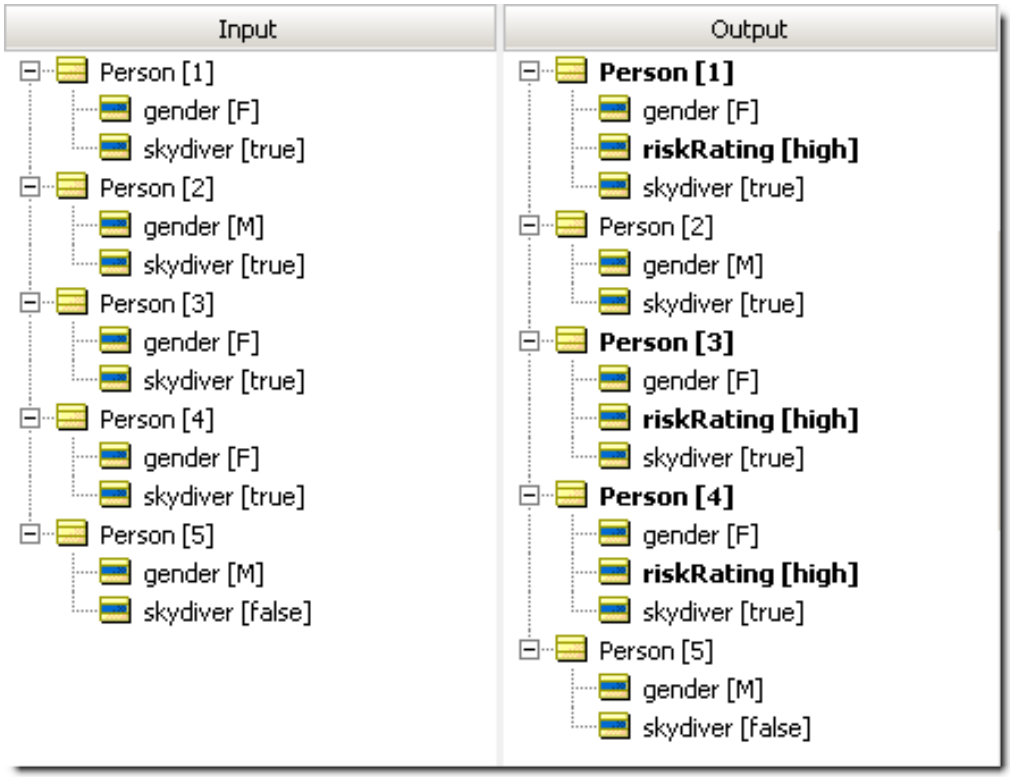


Figure 126: Rulesheet with two filters



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 127: 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.

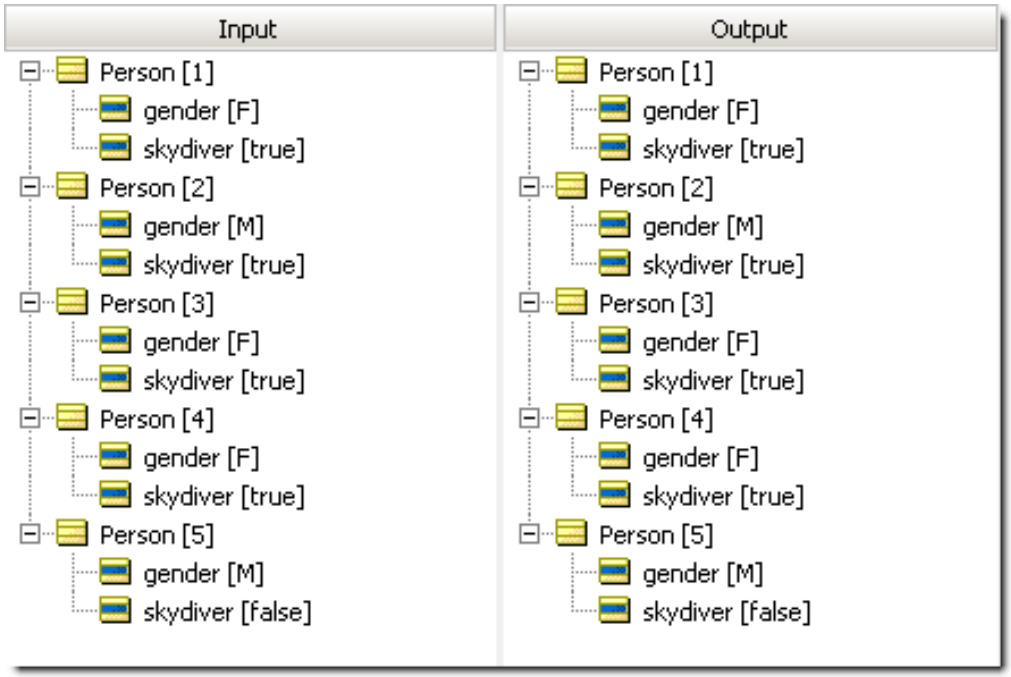
Expressed as a plain-language rule statement, the Rulesheet implements the following rule statement:

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

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 128: Ruletest for Rulesheet with two filters

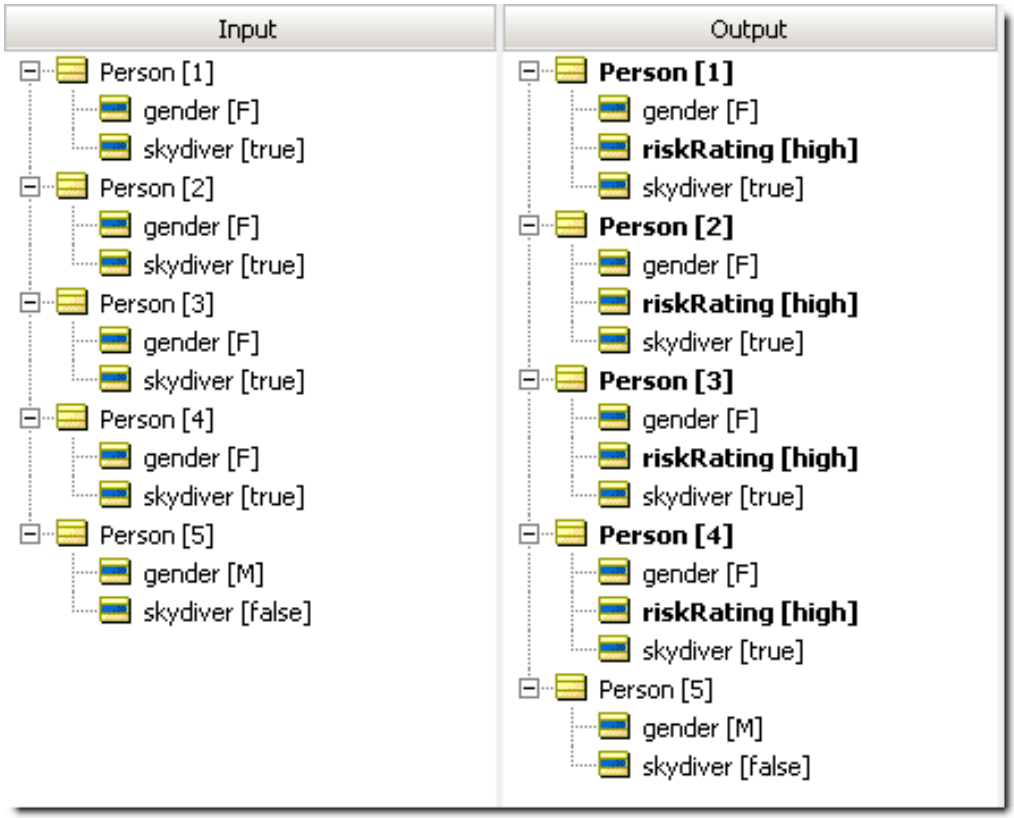


This time, no `riskRating` assignments were made to any element of collection `person`. Why? Because multiple filters are logically **AND**'ed together, forming a compound filter. In order to survive the compound filter, elements of collection `person` must be both skydivers **AND** female. Elements 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 129: 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.

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

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

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

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

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

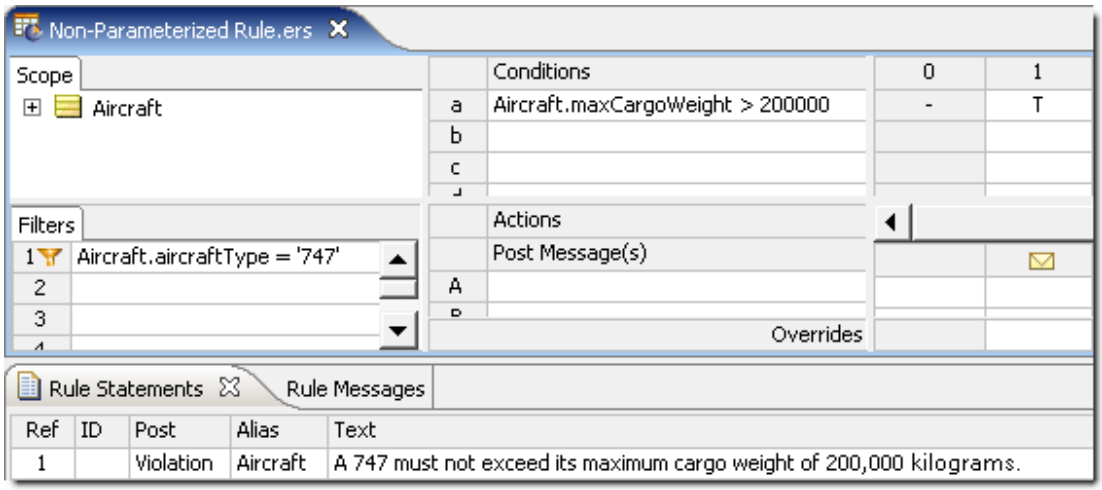
3. X aircraft must not exceed maximum cargo weight of Y kilograms.

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

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

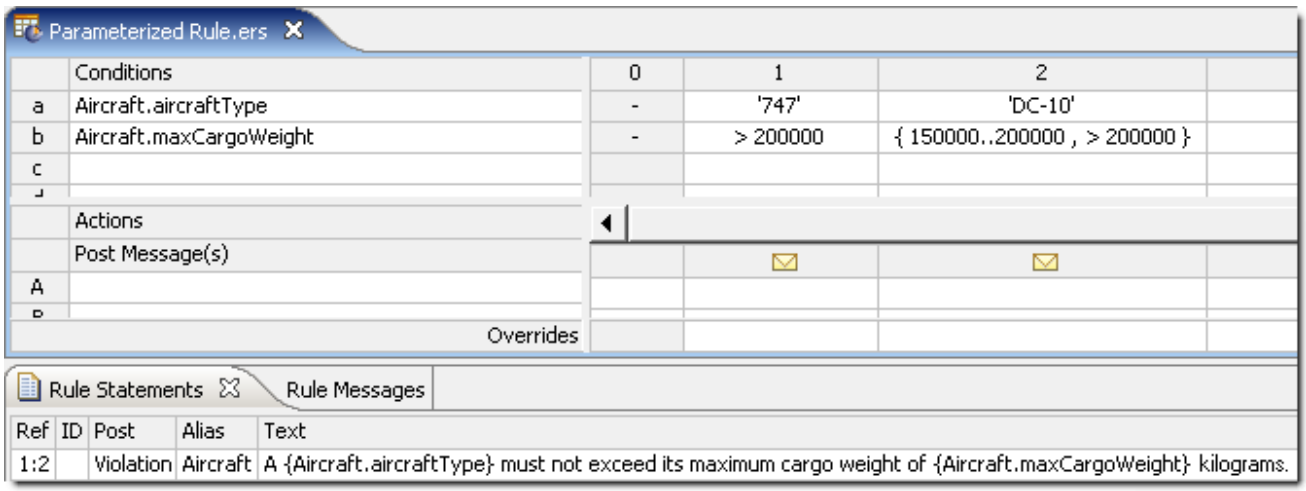
It is important to recognize these patterns because they can drastically simplify rule writing and maintenance in Corticon.js Studio. As shown in the following figure, we could build these two rules as a pair of Rulesheets, each with a Filter expression that filters data by `aircraftType`.

Figure 130: 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 131: Parameterized Rules



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

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

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.js 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.

10

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

Test, validate, and optimize your rules

Corticon.js 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 Test rule scenarios in the Ruletest Expected panel on page 174

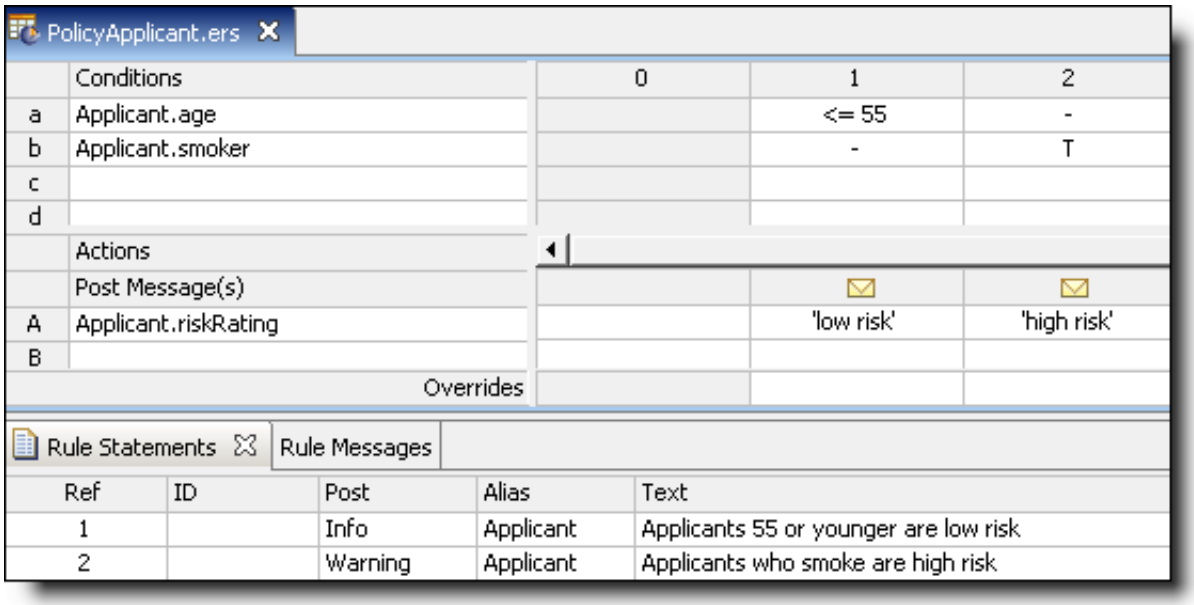
Rulesheet analysis and optimization

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.

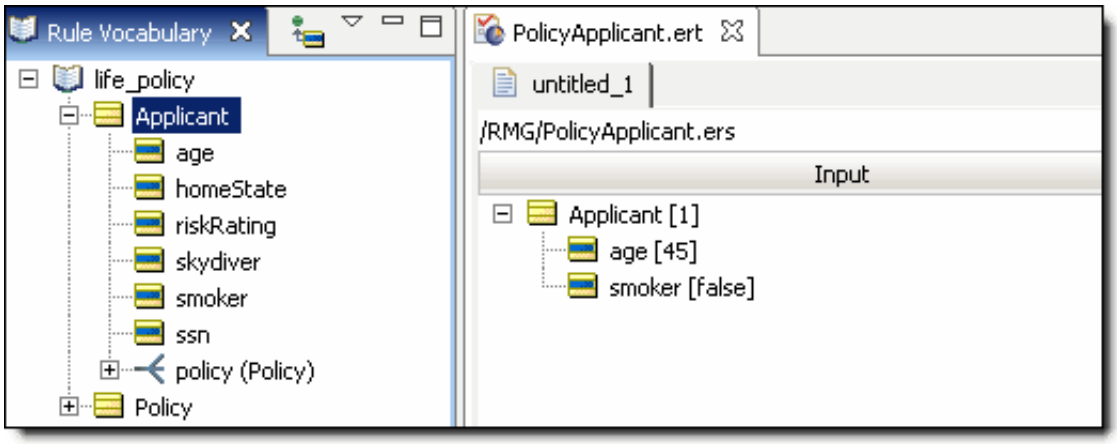
The following example illustrates the point:

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 132: Simple rules for profiling insurance policy applicants

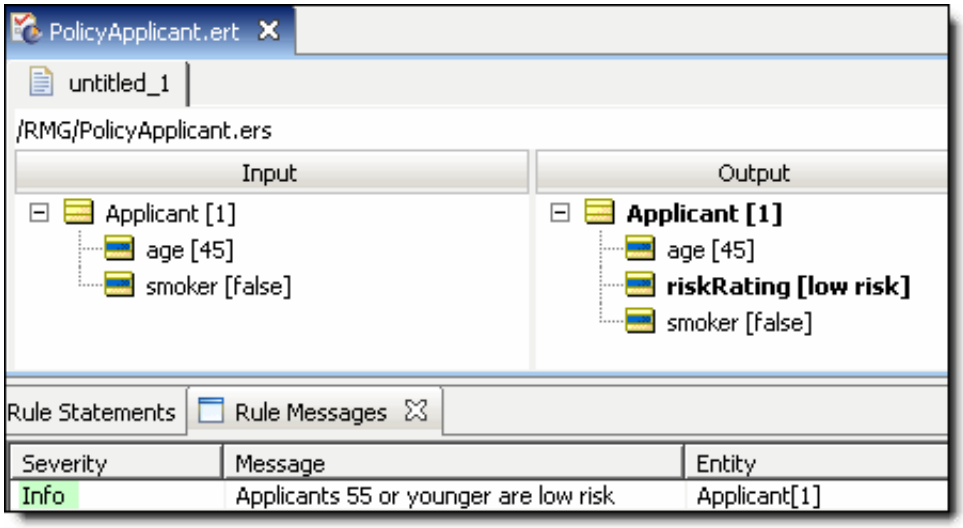


To test these rules, create a new scenario in a Ruletest, as shown:



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 133: 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?

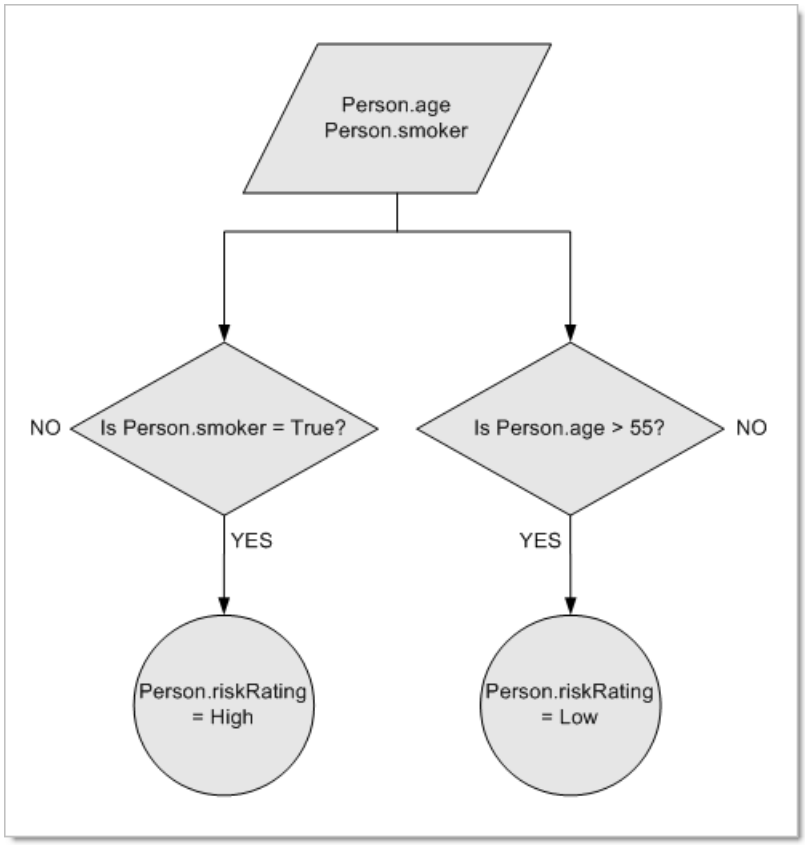
Traditional methods of analyzing logic

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:

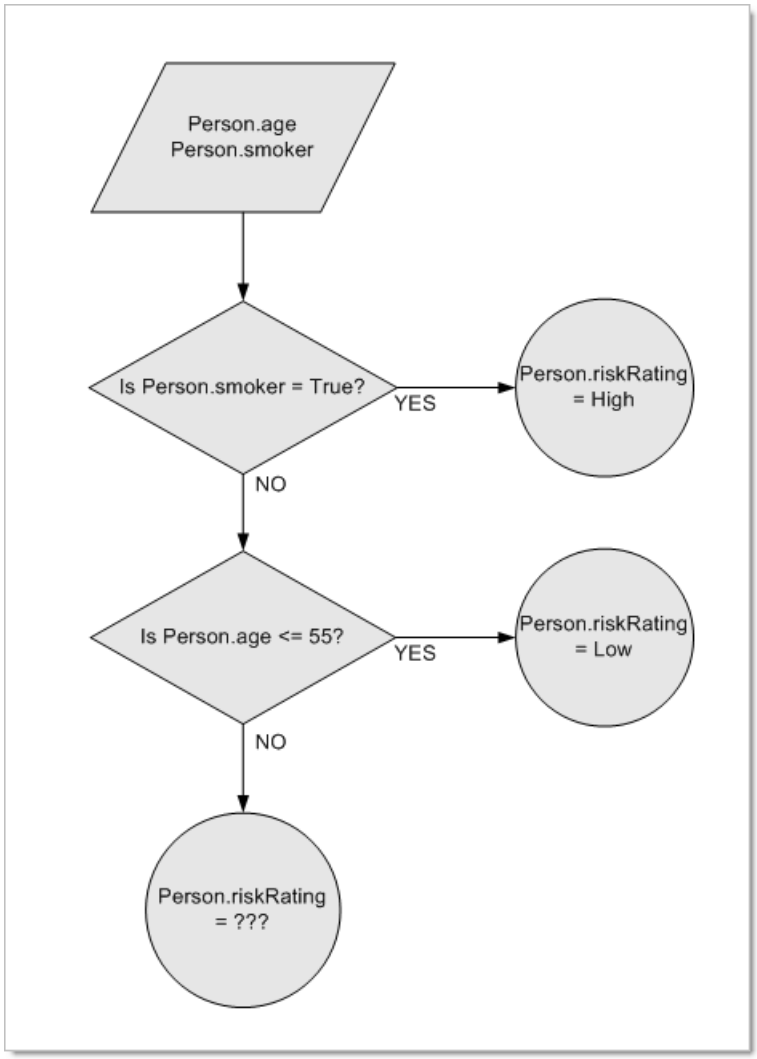
Figure 134: Flowchart with two rules



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 135: 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 8: 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 136: 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 137: 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 138: 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`	

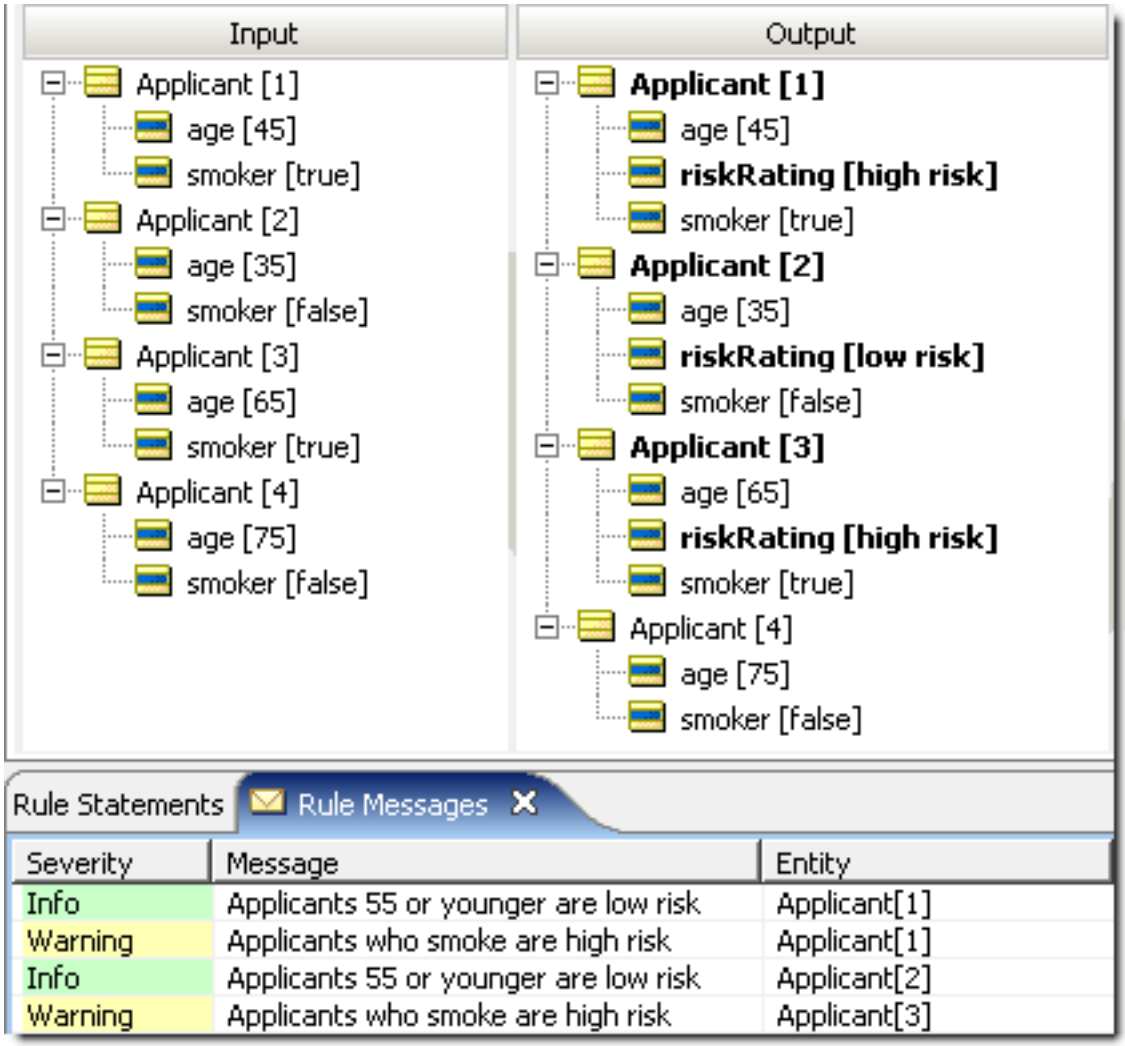
The table format also highlights two problems that were encountered earlier with flowcharts. In the figure **Rule 2 Expected Outcome**, row 1 and column 1 intersect in the upper left cell. This cell corresponds to test case #1 in the figure above. As a result, each rule tries to assert its own action – 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 139: 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.js Studio performs conflict and completeness checking automatically.

How to expand rules

Look at this table:

Figure 140: Rule 1 expected outcome

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

Then look at this Rulesheet:

Figure 141: Simple Rules for Profiling Insurance Policy Applicants

PolicyApplicant.ers

	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

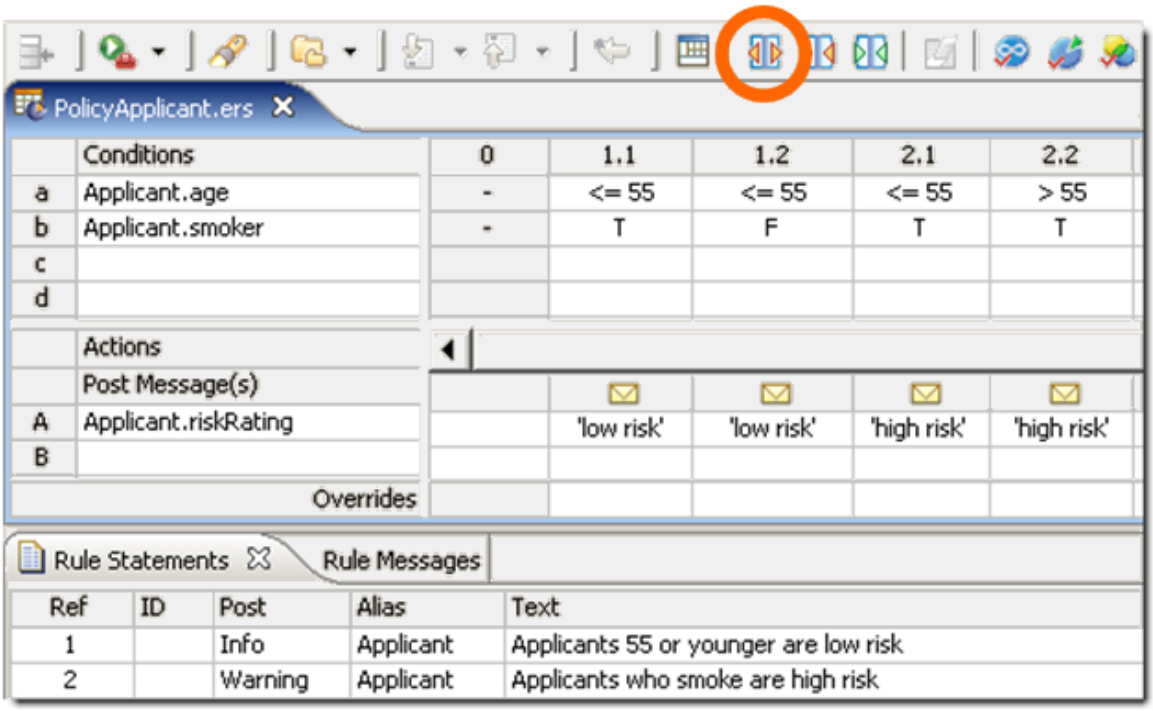
Ref	ID	Post	Alias	Text
1		Info	Applicant	Applicants 55 or younger are low risk
2		Warning	Applicant	Applicants who smoke are high risk

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.js 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.js 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 142: Expanding rules to reveal components

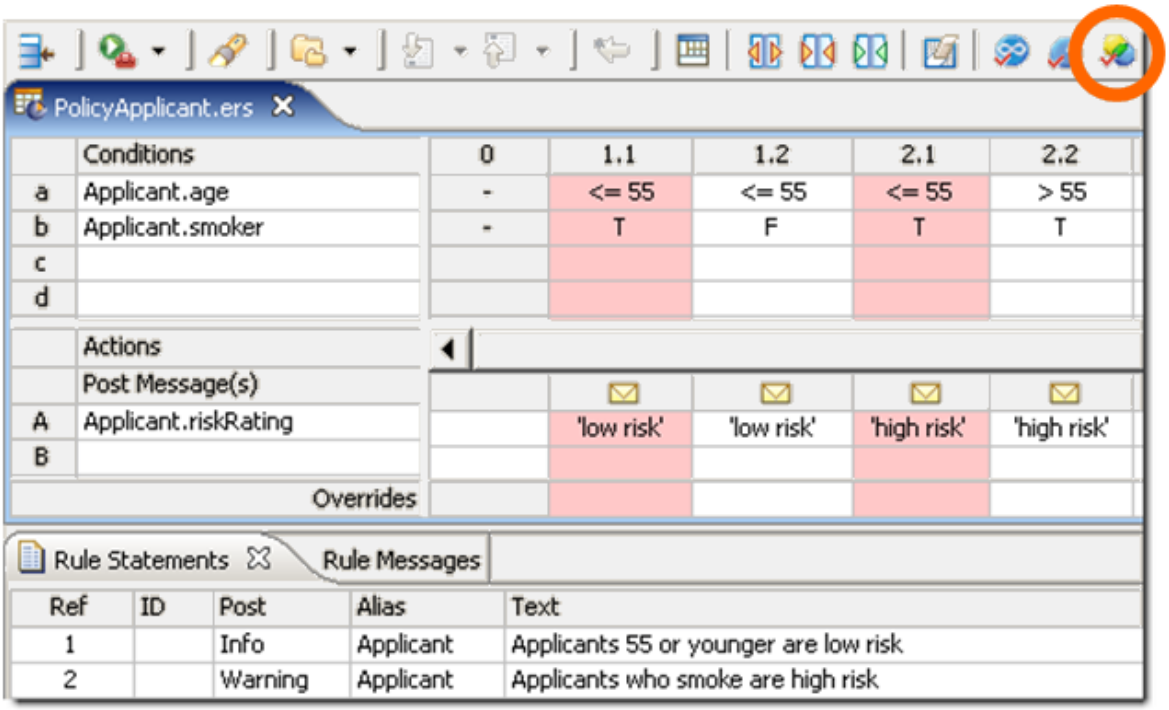


As pointed out before, the outcome is the same for each subrule. Because of this, the subrules can be summarized as the general rules shown in column 1 of Simple Rules for Profiling Insurance Policy Applicants. The two subrules collapse into the rules shown in column 1. The dash character in the smoker 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 as shown in Expanding Rules to reveal components, most of the Cross Product is displayed. Click the **Check for Conflicts** button in the toolbar.

Figure 143: Conflict revealed by the Conflict Checker



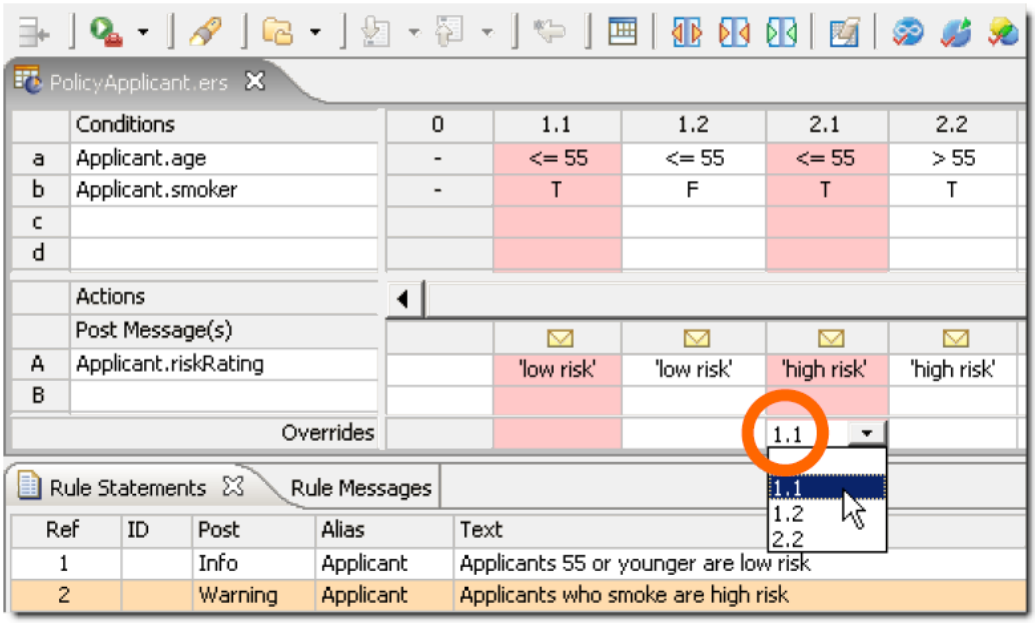
Note: The mechanics of conflict checks are described in the *Tutorial: Basic Rule Modeling* topic *"Analyze rules*.

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 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, then later with test cases. Subrules 1.1 and 2.1, the subrules highlighted in pink and yellow in Figure 143: Conflict revealed by the Conflict Checker on page 163, 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.js Studio does not instruct the rule writer how to resolve the conflict. It simply alerts the rule writer to its presence. The rule writer, ideally someone who knows the business, must decide how to resolve the problem. The rule writer has two basic choices:

1. Change the 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 183 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 144: Override entered to resolve conflict



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

> 1. Applicants aged 55 or younger are assigned a low risk rating *unless* they smoke, in which case they are assigned a high risk rating.

This modification successfully expresses the effect of the override.

If 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 have been resolved, you will see the following window:

Figure 145: 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 were recorded; if so, then the rule is overridden and processing of the rule with those entities is 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, then 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.

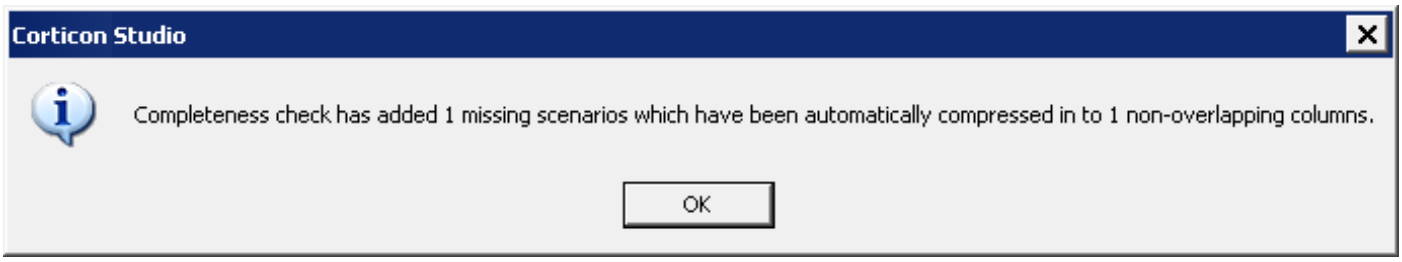
The completeness checker

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*.

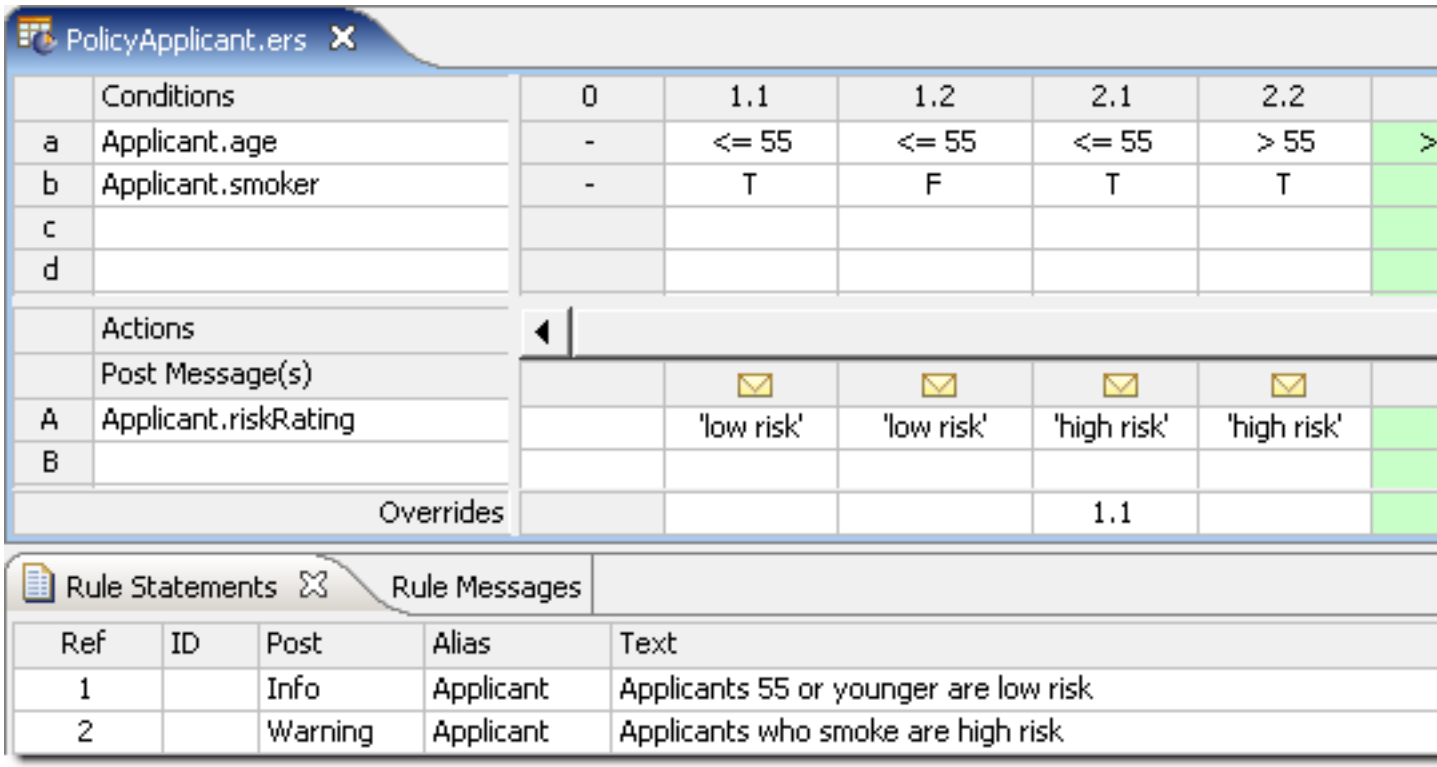
Clicking the **Check for Completeness** button, the message window is displayed:

Figure 146: Completeness Check message window



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

Figure 147: New rule added by completeness check



This new rule, the combination of `age>55` **AND** `smoker=false` corresponds to the intersection of column 2 and row 2 in Rule 2 expected outcome and test case #4 in Test cases extracted from Cross Product. The Completeness Checker has discovered 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.js 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.js 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

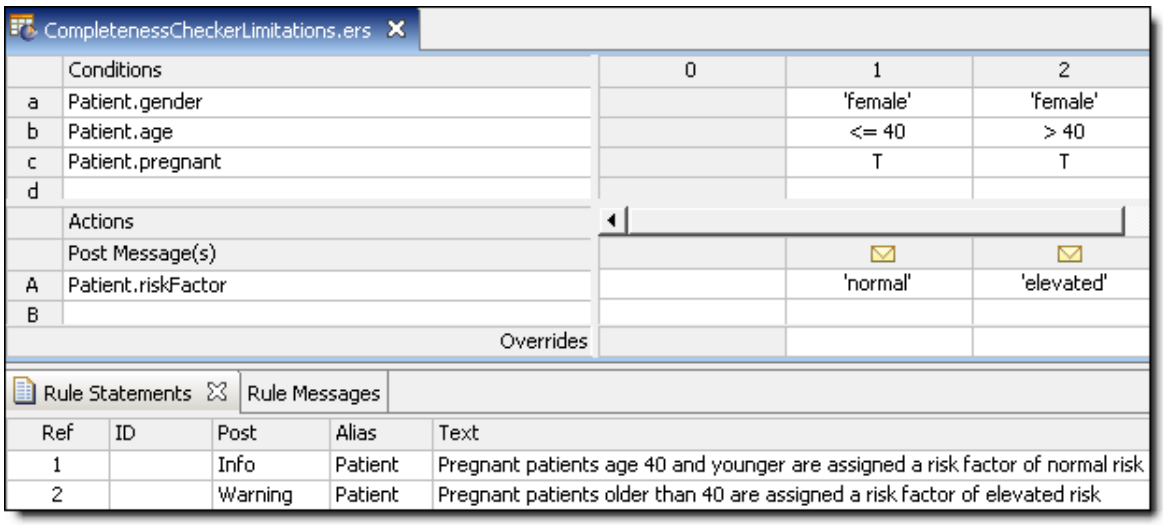
An important aspect of the Completeness Checker's operation is the automatic compression it performs on the resulting set of missing Conditions. As you can see from the message displayed in Completeness Check Message Window, the algorithm not only identifies the missing rules, but it also compresses 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

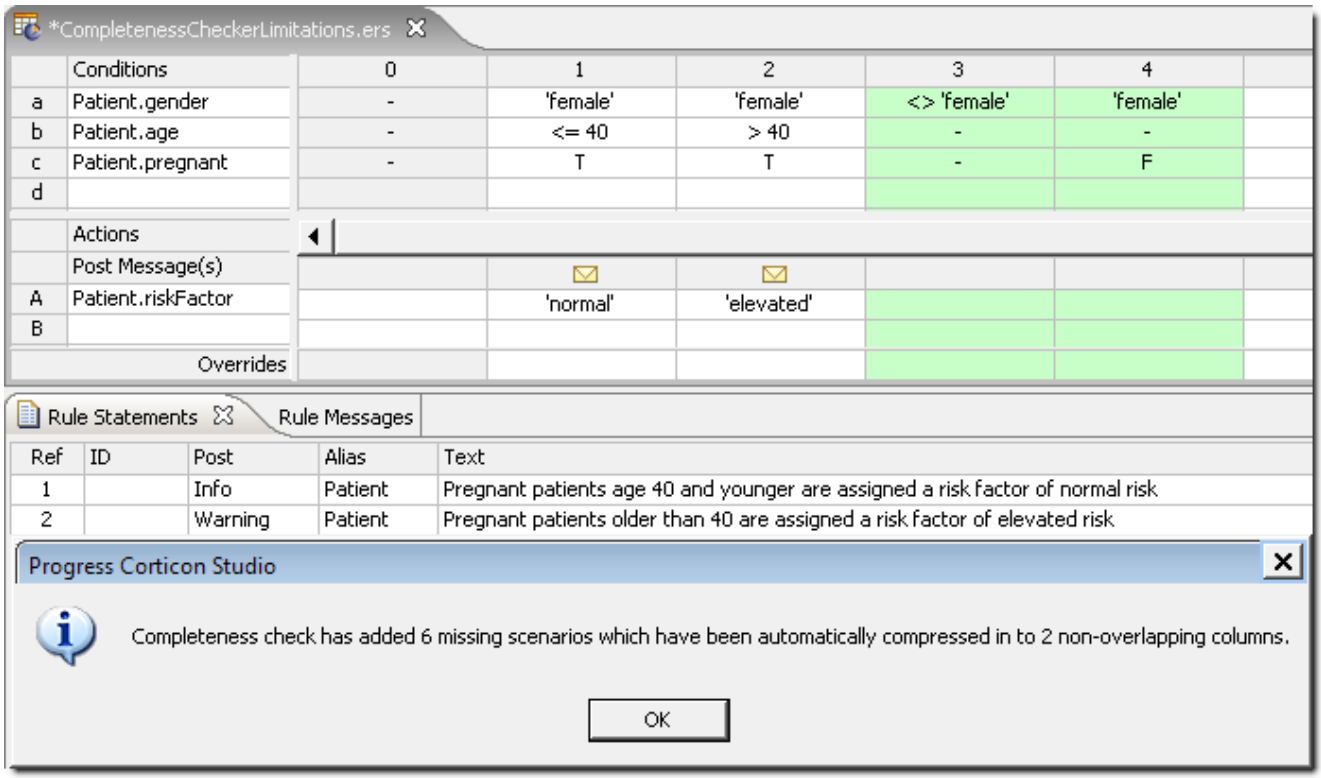
The Completeness Checker is powerful in its ability to discover missing combinations of Conditions from your Rulesheet. However, it is not smart enough to determine if these combinations make *business sense* or not. The example in the following figure shows two rules used in a health care scenario to screen for high-risk pregnancies:

Figure 148: Example prior to Completeness Check



Now, we will click on the Completeness Checker:

Figure 149: Example after Completeness Check



Notice that columns 3-4 have been automatically added to the Rulesheet. But also notice that column 3 contains an unusual Condition: `gender <> female`. Because the other two Conditions in column 3 have dash values, we know it contains component or subrules. By double-clicking on column 3's header, its subrules are revealed:

Figure 150: 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 you can say that 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. If other rules in an application prevent combinations like this from occurring, then subrules 3.1 and 3.3 can also be unnecessary. 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 can be ignored. However, say you decide to keep subrules 3.2 and 3.4 and assign Actions to them. For this example, violation messages will be posted.

However, if 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 sub-rules, into four whole-numbered general rules, select **Rulesheet >Rule Column(s)>Renumber Rules** from the **Studio** menubar.

Figure 151: Sub-rules renumbered and converted to general rules

CompletenessCheckerLimitations.ers

	Conditions	0	1	2	3	4	5	6	7
a	Patient.gender	-	'female'	'female'	<> 'female'	<> 'female'	<> 'female'	<> 'female'	'female'
b	Patient.age	-	<= 40	> 40	<= 40	<= 40	> 40	> 40	-
c	Patient.pregnant	-	T	T	T	F	T	F	F
d									
	Actions								
	Post Message(s)								
A	Patient.riskFactor		'normal'	'elevated'					
B									
	Overrides								

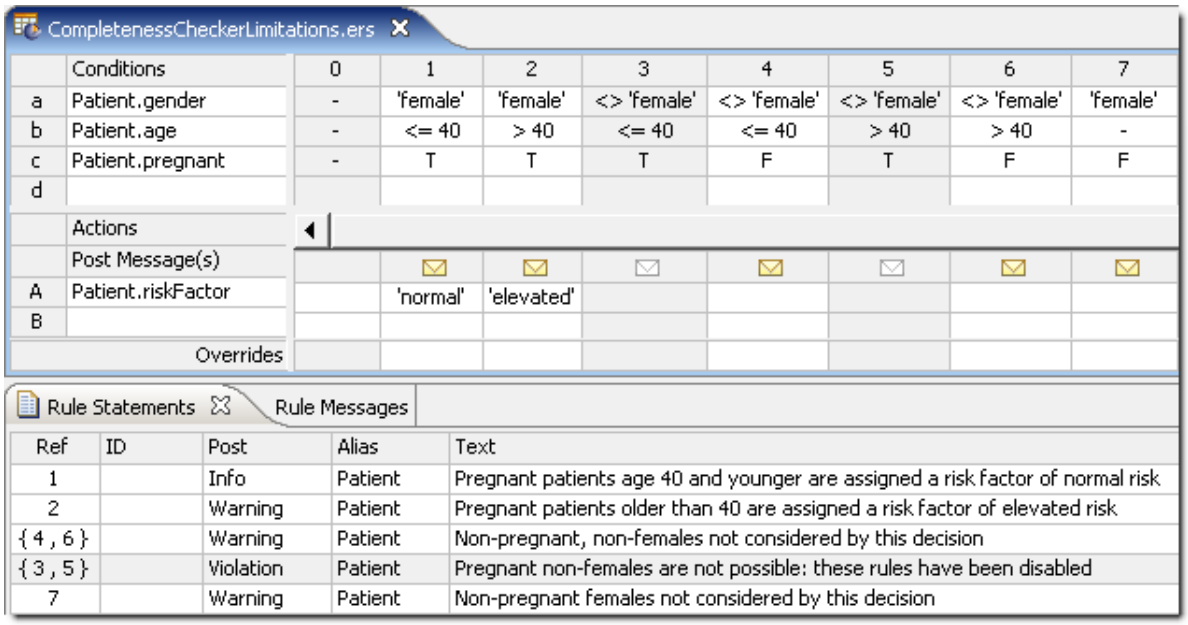
Rule Statements | Rule Messages

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

Now that the columns were 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, it is a good idea to disable 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. If you disable any columns, it is a good idea to include a Rule Statement that explains why. See the following figure for an example of a fully complete and well-documented Rulesheet.

Figure 152: Final Rulesheet with impossible rules disabled



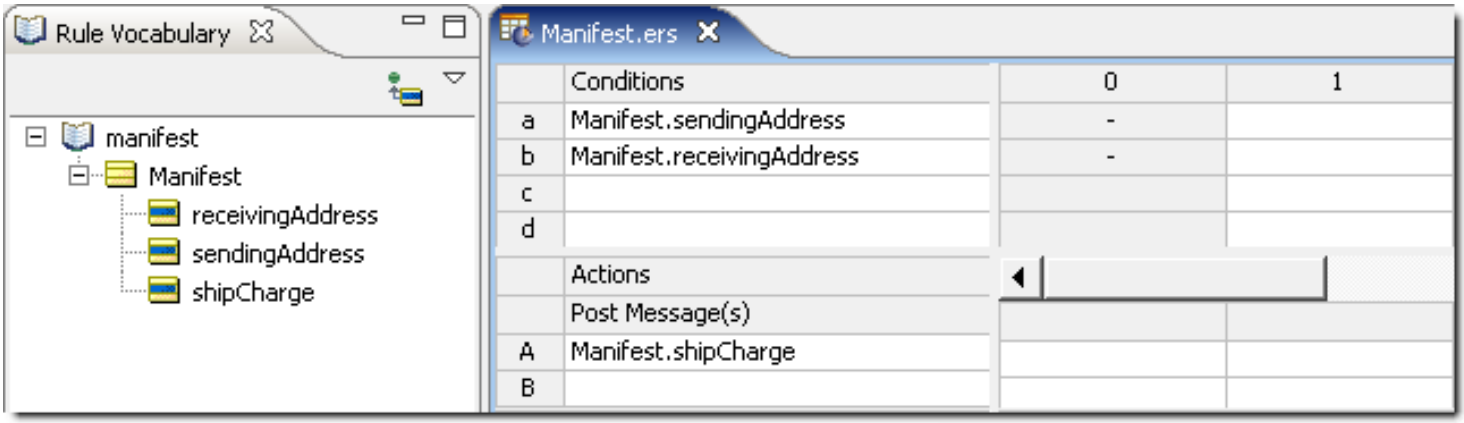
Let the expansion tool work for you with tabular rules

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

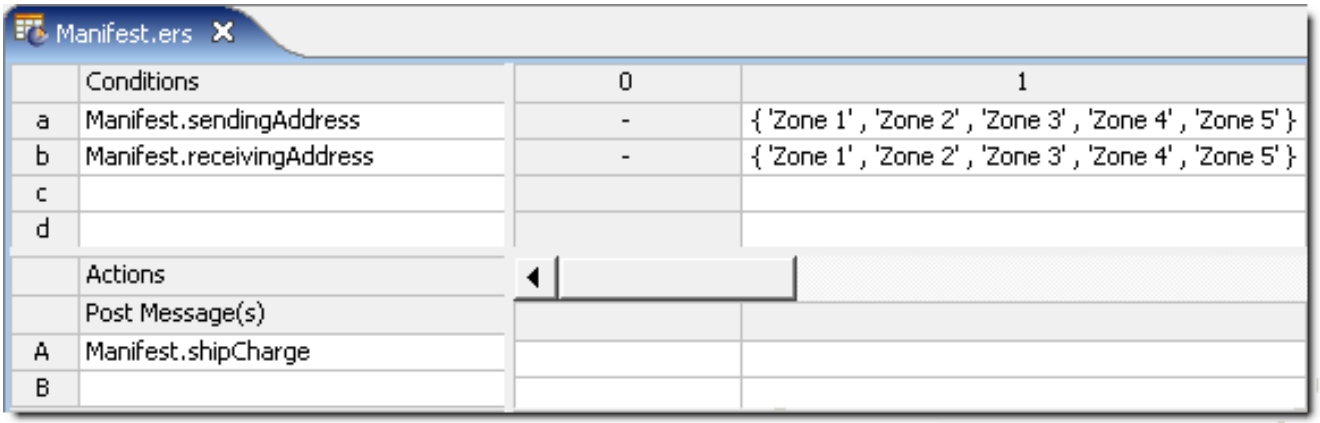
In the following figure, a simple Vocabulary with which to implement these rules was built. Because each cell in the table represents a single rule, the Rulesheet contains 25 columns (the Cross Product equals 5x5 or 25).

Figure 153: Vocabulary and Rulesheet to implement matrix



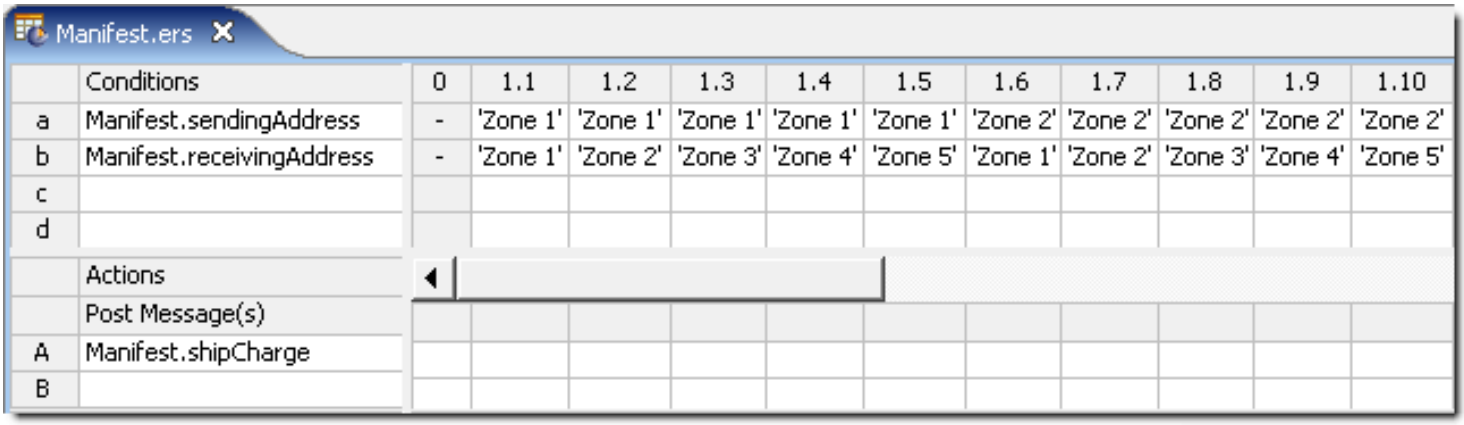
Rather than manually create all 25 combinations (and risk making a mistake), you can use the Expansion Tool to help you do it. This is a three-step process. Step 1 consists of entering the full range of values found in the table in the Conditions cells, as shown:

Figure 154: 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 155: 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 156: Rulesheet with Actions populated

Manifest.ers

	Conditions	0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10
a	Manifest.sendingAddress	-	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 2'	'Zone 2'	'Zone 2'	'Zone 2'	'Zone 2'
b	Manifest.receivingAddress	-	'Zone 1'	'Zone 2'	'Zone 3'	'Zone 4'	'Zone 5'	'Zone 1'	'Zone 2'	'Zone 3'	'Zone 4'	'Zone 5'
c												
d												
	Actions											
	Post Message(s)											
A	Manifest.shipCharge		0.25	0.35	0.45	0.55	0.65	0.35	0.25	0.35	0.45	0.55
B												

In step 3, shown in the following figure, 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 157: Rulesheet with renumbered rules

Manifest.ers

	Conditions	0	1	2	3	4	5	6	7	8	9	10
a	Manifest.sendingAddress	-	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 2'	'Zone 2'	'Zone 2'	'Zone 2'	'Zone 2'
b	Manifest.receivingAddress	-	'Zone 1'	'Zone 2'	'Zone 3'	'Zone 4'	'Zone 5'	'Zone 1'	'Zone 2'	'Zone 3'	'Zone 4'	'Zone 5'
c												
d												
	Actions											
	Post Message(s)											
A	Manifest.shipCharge		0.25	0.35	0.45	0.55	0.65	0.35	0.25	0.35	0.45	0.55
B												

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.js Studio runs low on memory, get details on increasing Corticon.js Studio's memory allotment in *"Increase Corticon.js Studio memory allocation" in the Corticon.js Installation Guide.*

Logical loop detection

Corticon.js 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.js 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 Techniques that refine rule testing on page 178.

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.js 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:

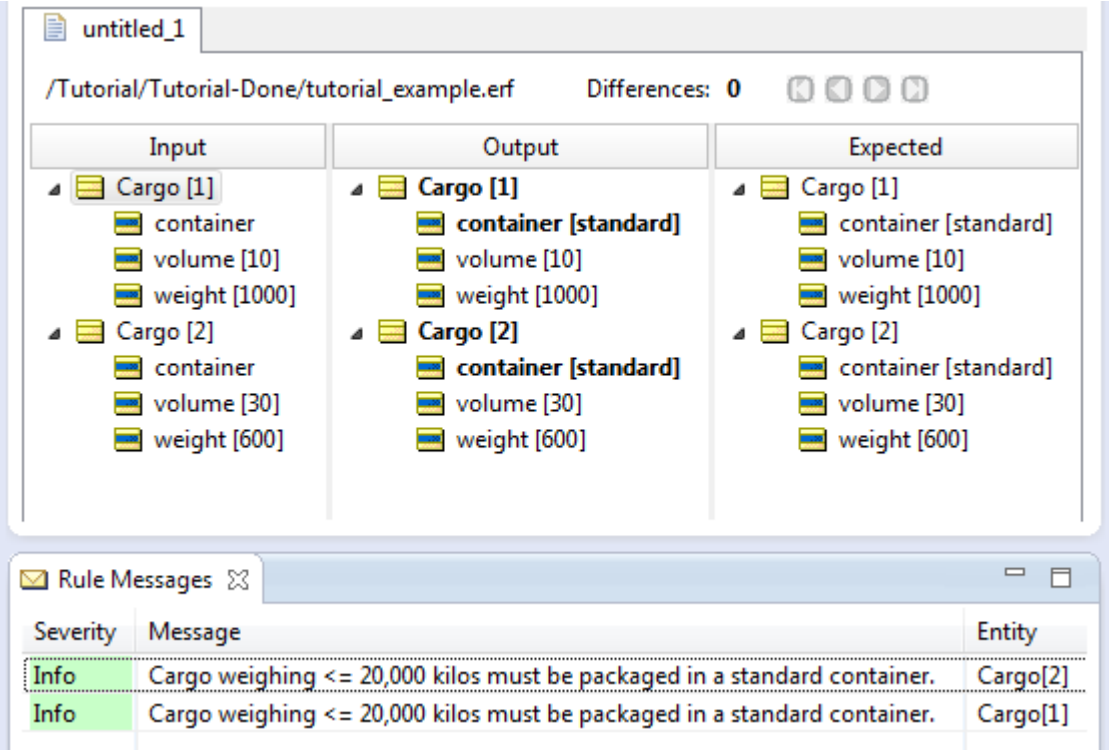
  The four buttons take you to the first, previous, next, and last discovered difference.

Review test results when using the Expected panel

The following topics present a variety of test results.

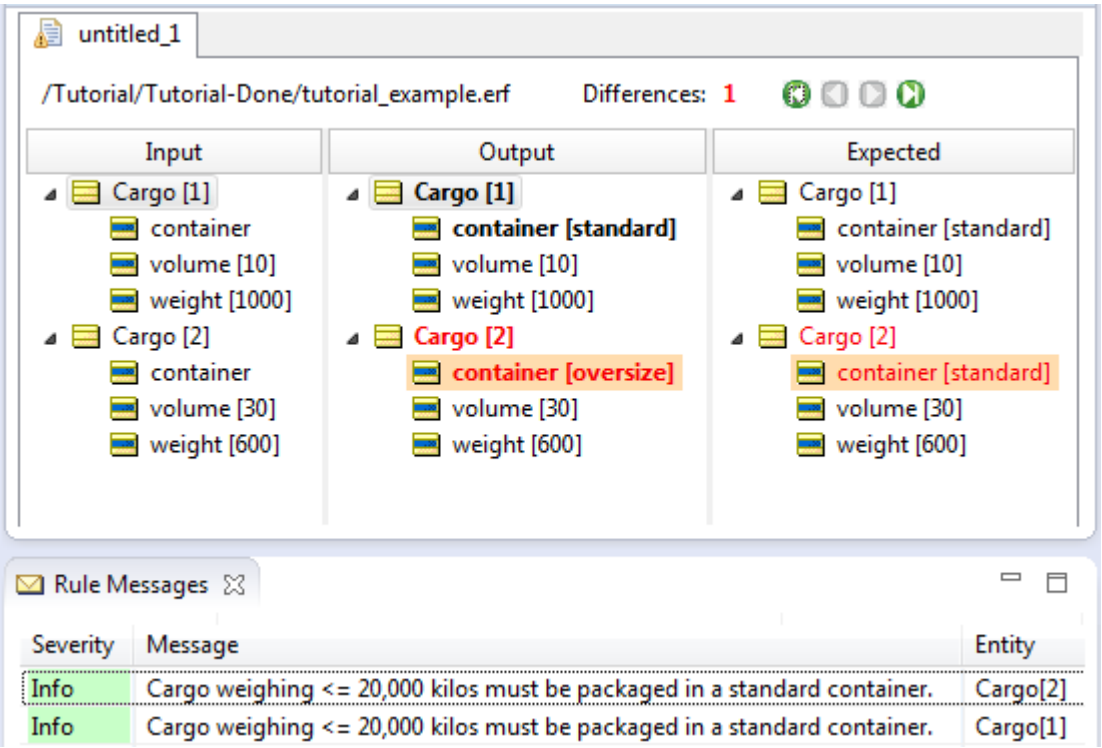
Output results match expected exactly

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

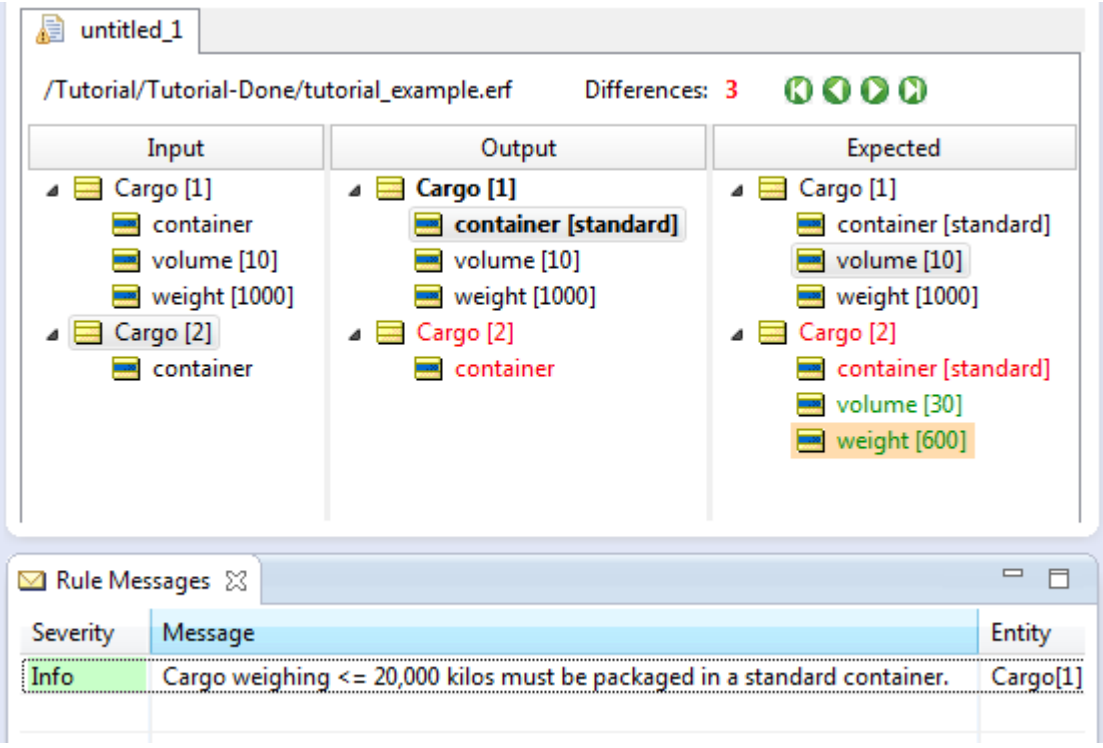
In the following example, one difference was identified. The expected value of `Cargo[2]` packaging value is `standard`, but the Ruletest produced an actual value of `oversize`. Because the **Output** does not match the **Expected** data, the text is colored red.



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.

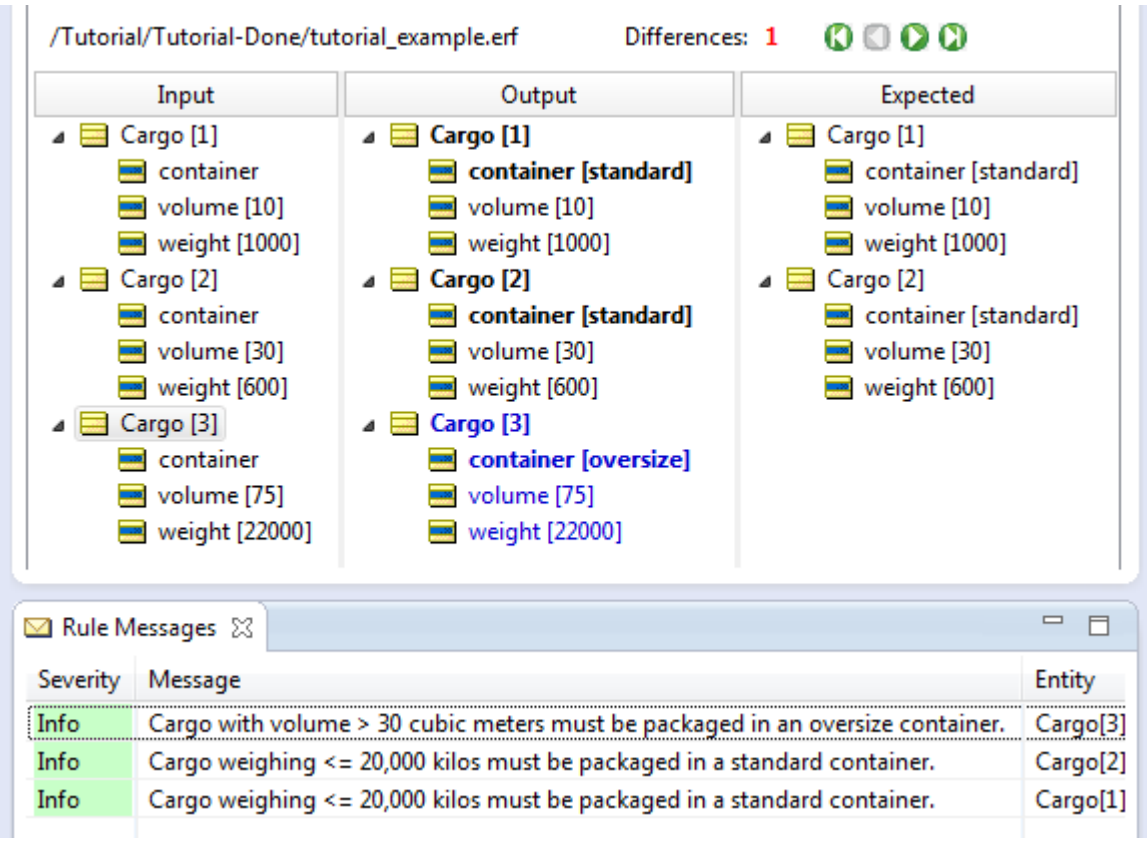
Fewer values output than expected

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.



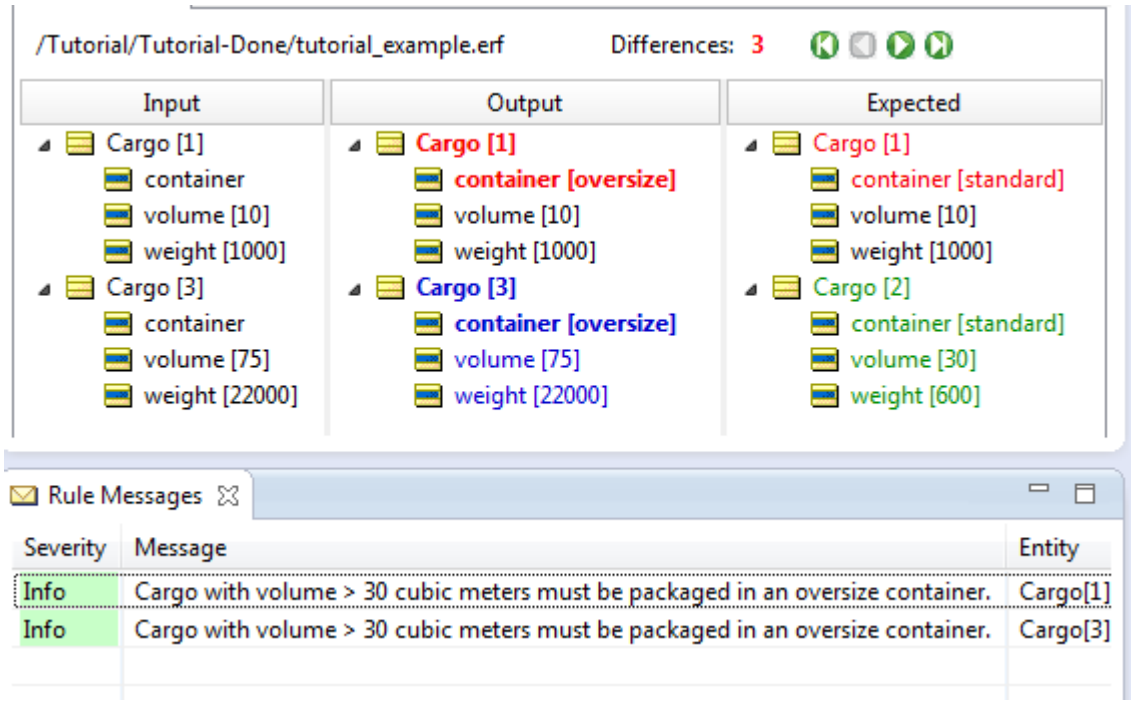
More values output than expected

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

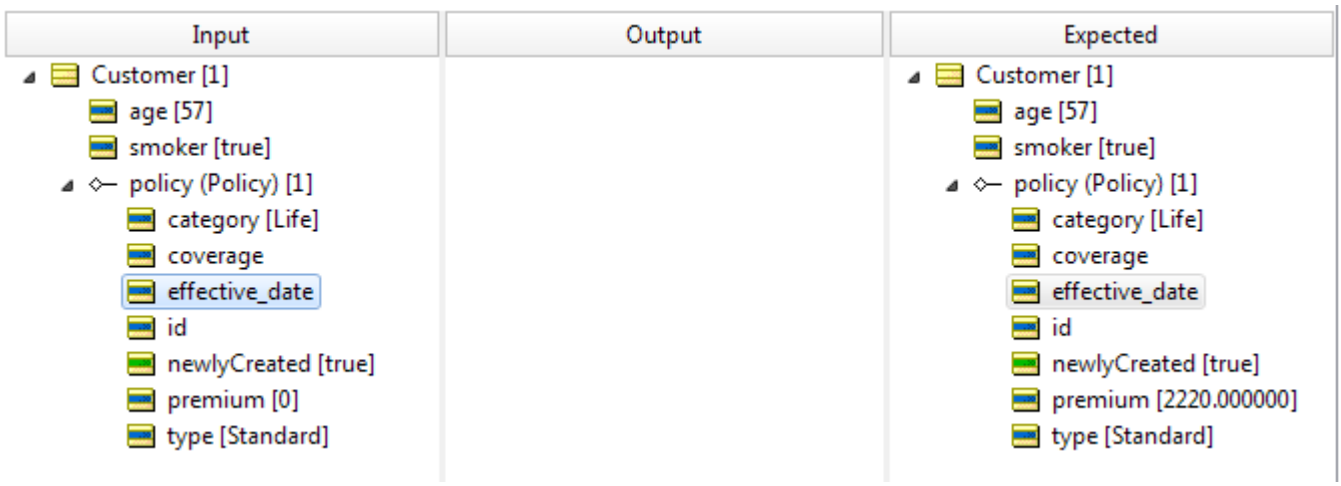
The following settings help you tune the results of comparing the output data and expected data so that irrelevant errors are minimized:

Set selected attributes to ignore validation

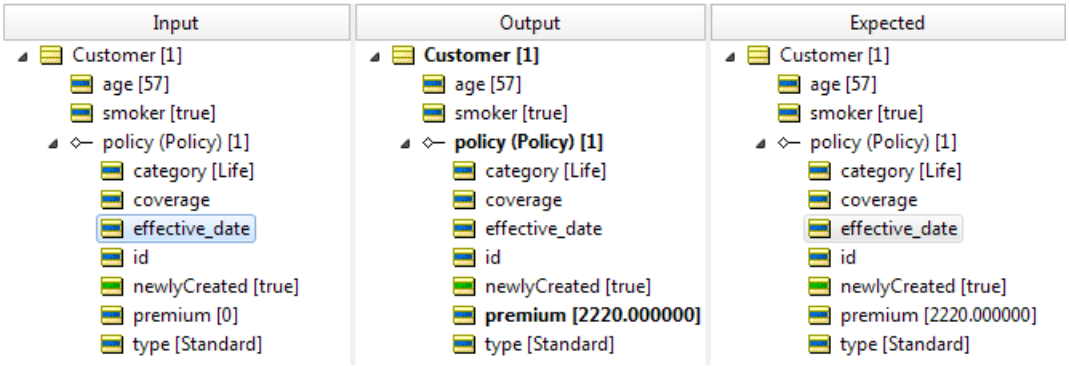
When different values are output than what was expected, it could mean that the **Expected** panel data created from **Output** data were reflecting dynamic values such as dates and time. If your Rulesheets use `now` or `today`, then the **Expected** values will evaluate as errors very soon. To handle that situation, you can choose to ignore validation for selected values in the **Expected** panel.

Consider the following example:

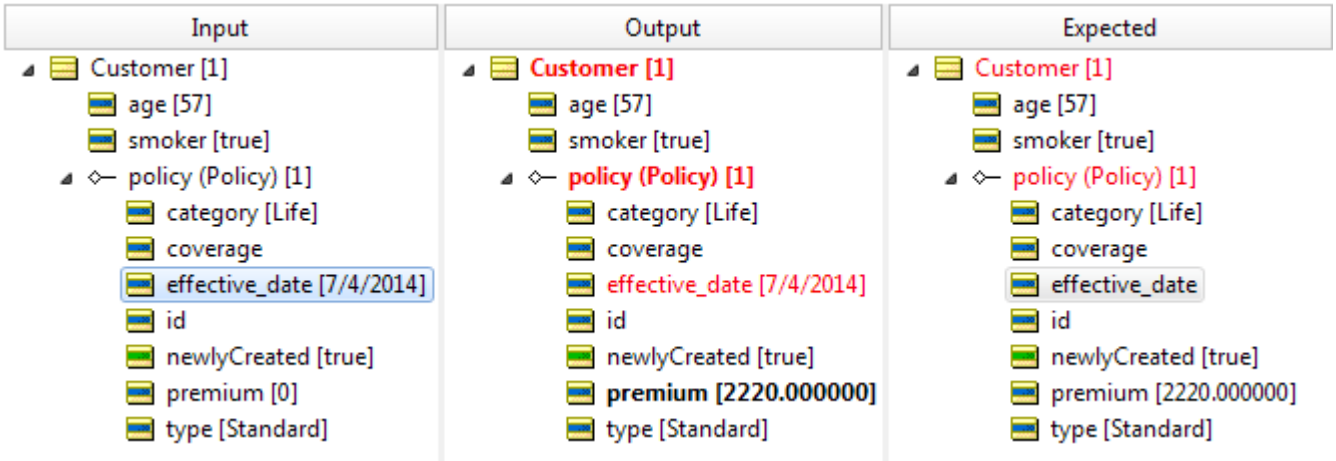
The selected attribute in this test has no input value and no expected value:



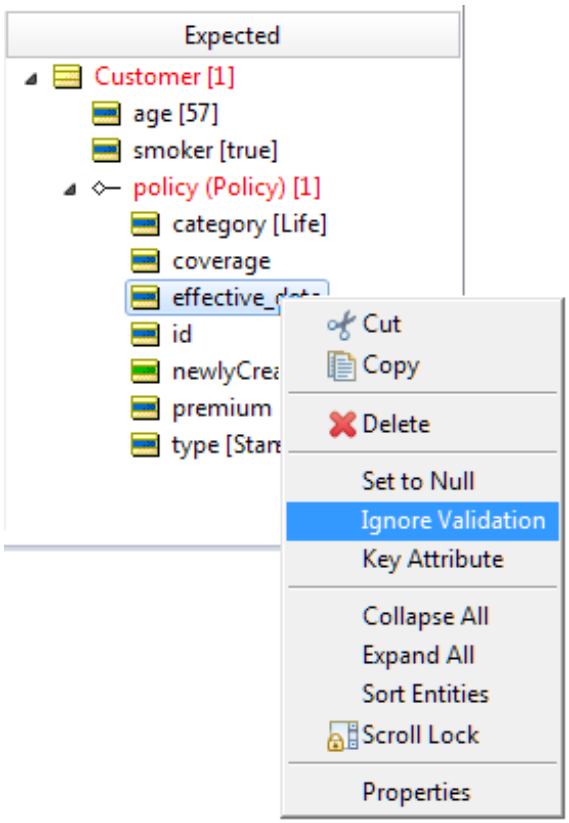
When the test runs, it is valid.



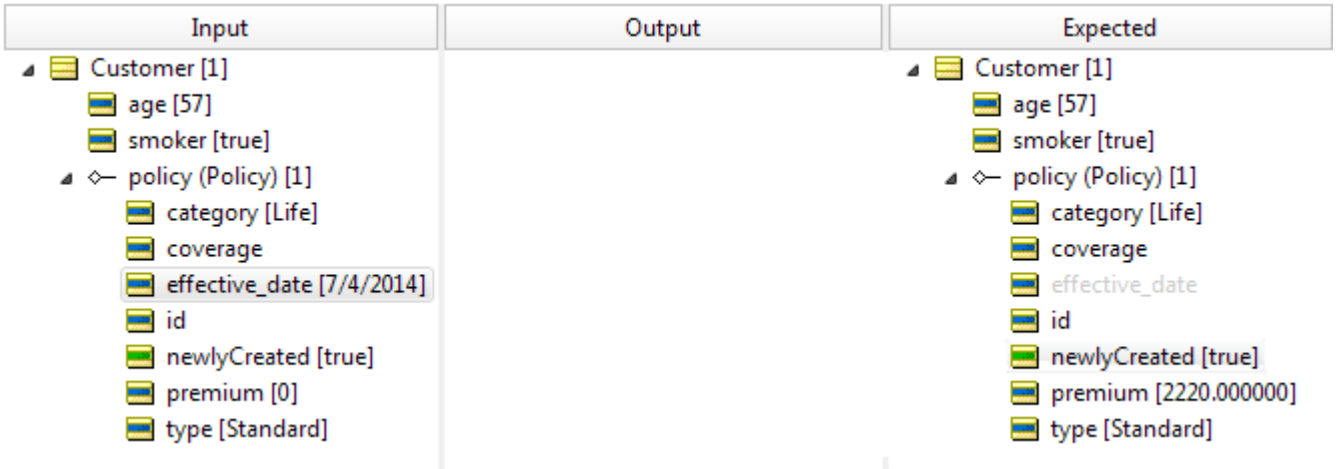
But, when the input gets a value and the output still has no value (or a different value), the test fails.



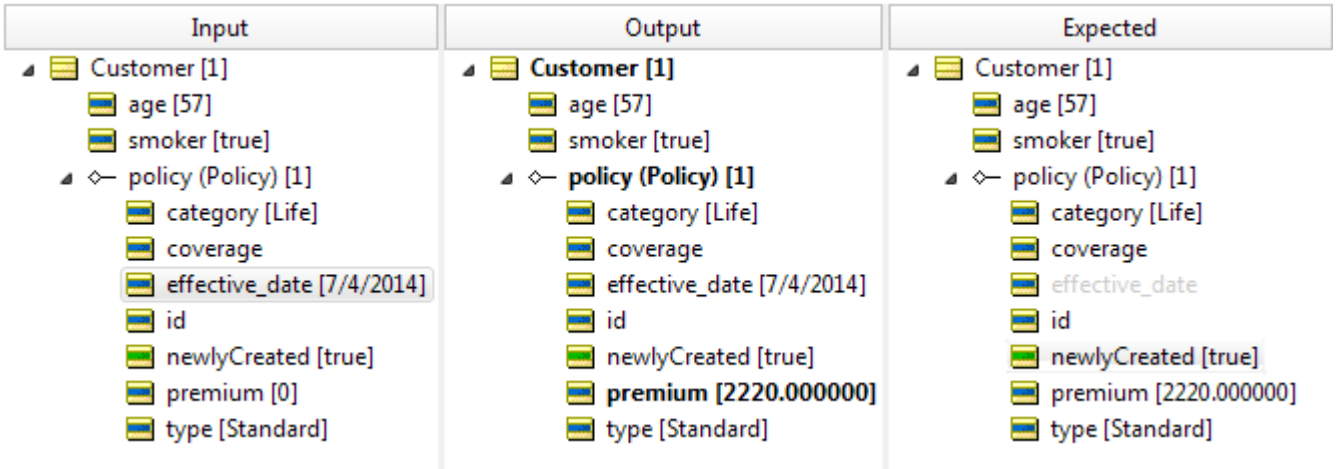
Clicking the expected attribute, you can choose **Ignore Validation**.



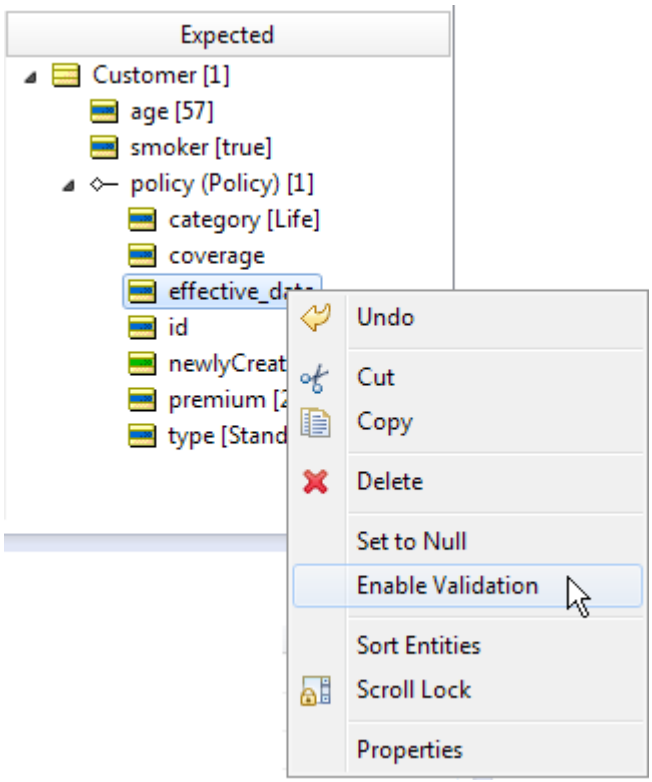
An attribute that will be ignored is greyed out.



Running the same test, the test passes.



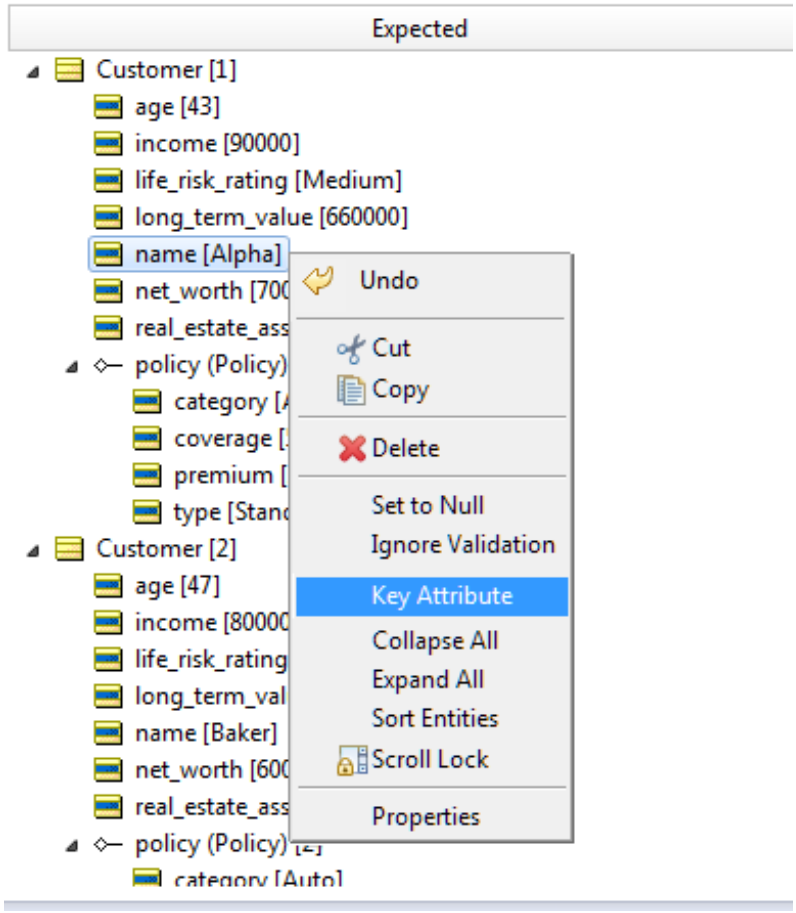
The setting can revert by selecting the attribute and then choosing **Enable Validation**.



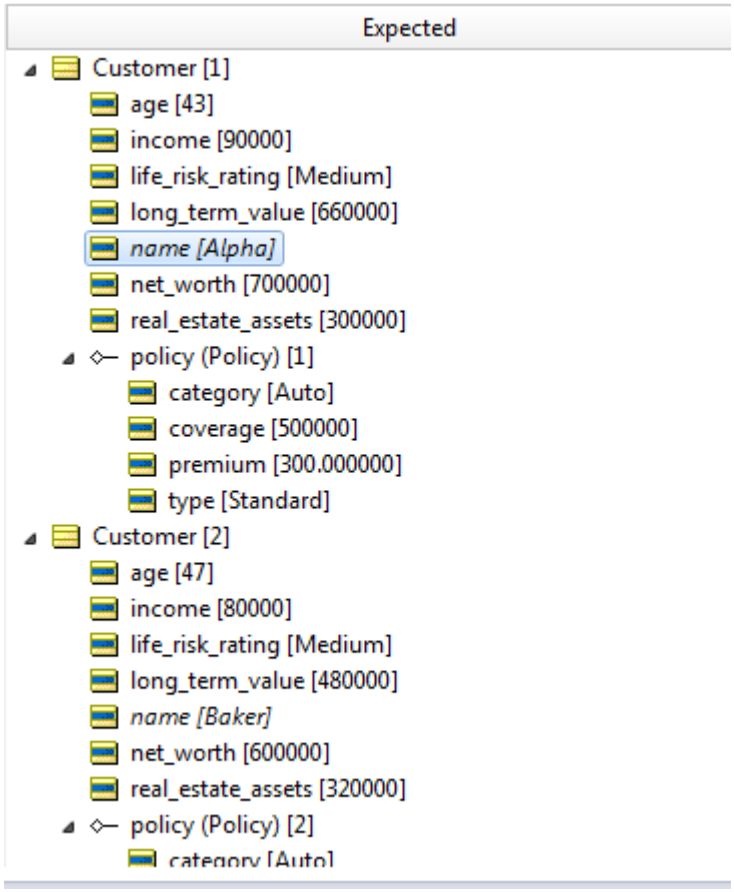
Use key attributes to improve difference detection in Ruletests

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.

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 will execute them efficiently .

The compress tool

Corticon.js 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 158: New Rule (#4) added

PolicyApplicant.ers

	Conditions	0	1	2	3	4
a	Applicant.age	-	<= 55	-	> 55	<= 55
b	Applicant.smoker	-	-	T	F	F
c						
d						
	Actions					
	Post Message(s)					
A	Applicant.riskRating		'low risk'	'high risk'	'medium risk'	'low risk'
B						
	Overrides					

Rule Statements | Rule Messages

Ref	ID	Post	Alias	Text
1		Info	Applicant	Applicants 55 or younger are low risk
2		Warning	Applicant	Applicants who smoke are high risk
3		Info	Applicant	Applicants 55 or older who do not smoke are medium risk
4		Info	Applicant	Applicants 55 or younger who do not smoke are low risk

While these 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 159: Redundancy problem exposed

PolicyApplicant.ers

	Conditions	0	1.1	1.2	2.1	2.2	3	4
a	Applicant.age	-	<= 55	<= 55	<= 55	> 55	> 55	<= 55
b	Applicant.smoker	-	T	F	T	T	F	F
c								
d								
	Actions							
	Post Message(s)							
A	Applicant.riskRating		'low risk'	'low risk'	'high risk'	'high risk'	'medium risk'	'low risk'
B								
	Overrides				1.1			

Rule Statements | Rule Messages

Ref	ID	Post	Alias	Text
1		Info	Applicant	Applicants 55 or younger are low risk
2		Warning	Applicant	Applicants who smoke are high risk
3		Info	Applicant	Applicants 55 or older who do not smoke are medium risk
4		Info	Applicant	Applicants 55 or younger who do not smoke are low risk

Clicking the **Compress Tool** has the effect shown in the following figure:

Figure 160: Rulesheet after compression

PolicyApplicant.ers

	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 161: Compressed shipping charge Rulesheet

Manifest.ers

	Conditions	0	1	2	3	4	5	6
a	Manifest.sendingAddress	-	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 1'	'Zone 2'
b	Manifest.receivingAddress	-	'Zone 1'	'Zone 2'	'Zone 3'	'Zone 4'	'Zone 5'	{ 'Zone 1' , 'Zone 3' }
c								
d								
	Actions							
	Post Message(s)							
A	Manifest.shipCharge		0.25	0.35	0.45	0.55	0.65	0.35

Value sets in Condition cells are equivalent to the logical operator **OR**. Rule 6 therefore reads:

6. A manifest with a Zone 2 sending address **AND** a Zone 1 **OR** Zone 3 receiving address costs $0.35 per pound to ship.

In deployment, the 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.

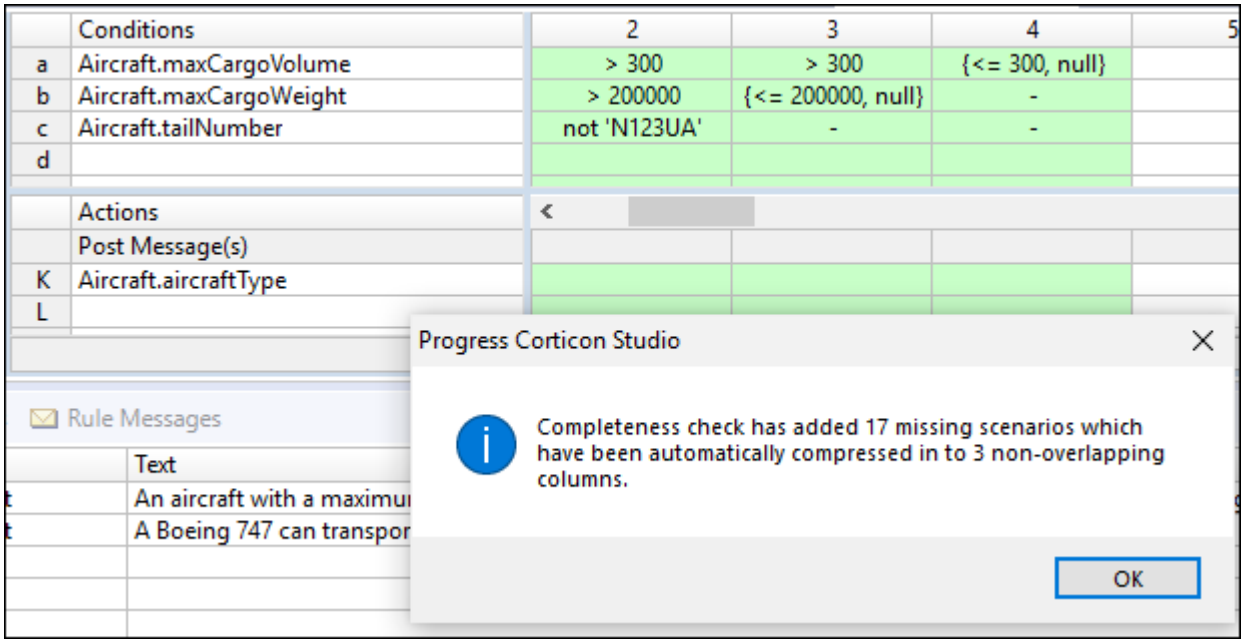
Applying modeling techniques, you might implement rule 1 as:

Figure 162: 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 163: Remaining columns produced by the Completeness Checker



Click **Expand** to fill out the Rulesheet so you can examine the 17 cross-product subrules:

Figure 164: 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 165: 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 166: 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						
	Actions					
	Post Message(s)					
K	Aircraft.aircraftType		'747'			
L						
M						
N						
O						
	Overrides					

Progress Corticon Studio

No conflicts detected

OK

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 167: 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 Decision Service performance

Why does Corticon.js 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.

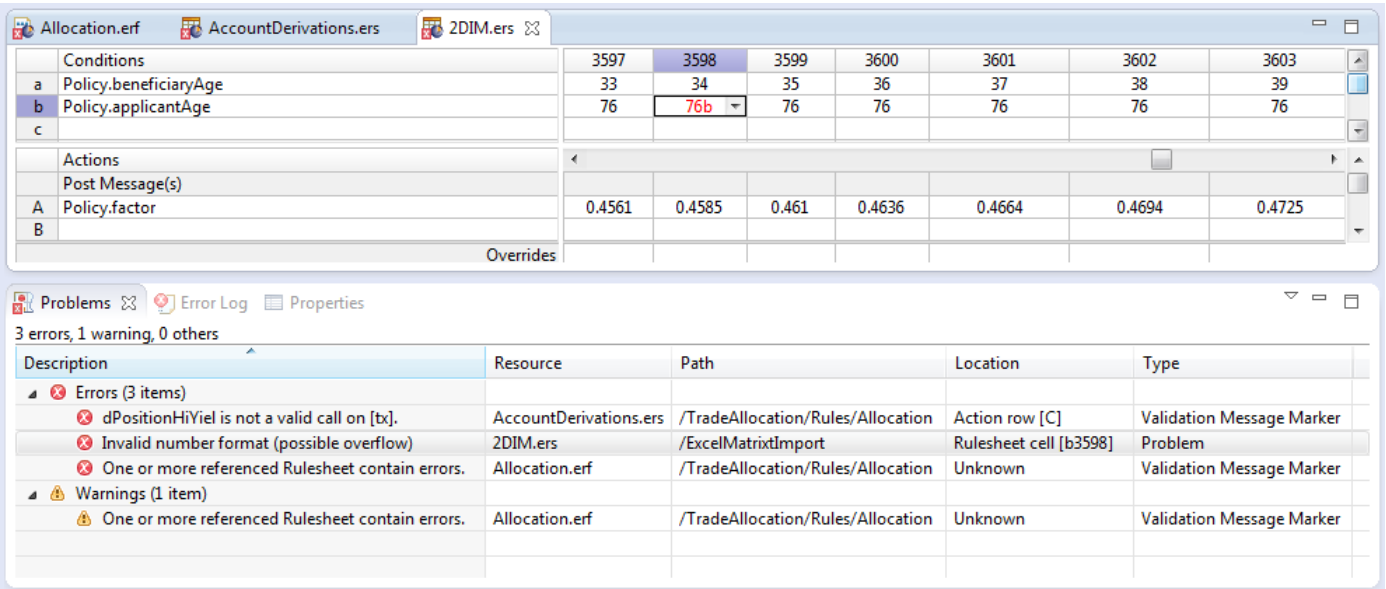
The compression performed by the Compression Tool is designed to reduce the number of rules into the smallest set of general rules (columns with dashes), even if the total number of subrules is larger than that produced by the Completeness Checker. This is important for three reasons:

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 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.js 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.

11

Advanced Ruleflow techniques and tools

Ruleflows provide techniques for combining and graphing.

For details, see the following topics:

- How to use a Ruleflow in another Ruleflow
- How to generate Ruleflow dependency graphs

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.

You can change the name of a Ruleflow on the canvas context so that it provides meaning, and you can add comments. None of these actions change the Ruleflow properties of the original Ruleflow.

With 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 their rules. That makes it easy to *reuse* either of these Ruleflows. For example, if 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.

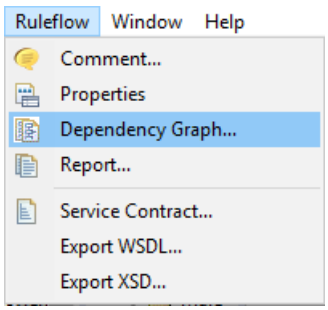
Note: Deploying Ruleflows within a Ruleflow - When this Ruleflow is deployed, the generated Decision Service will include the content of both Ruleflows. However, when either of the included Ruleflows changes, Ruleflows that include one of them are not automatically updated -- each must be redeployed to include the changes.

For more information, see *the "Ruleflows" section of the Quick Reference Guide*

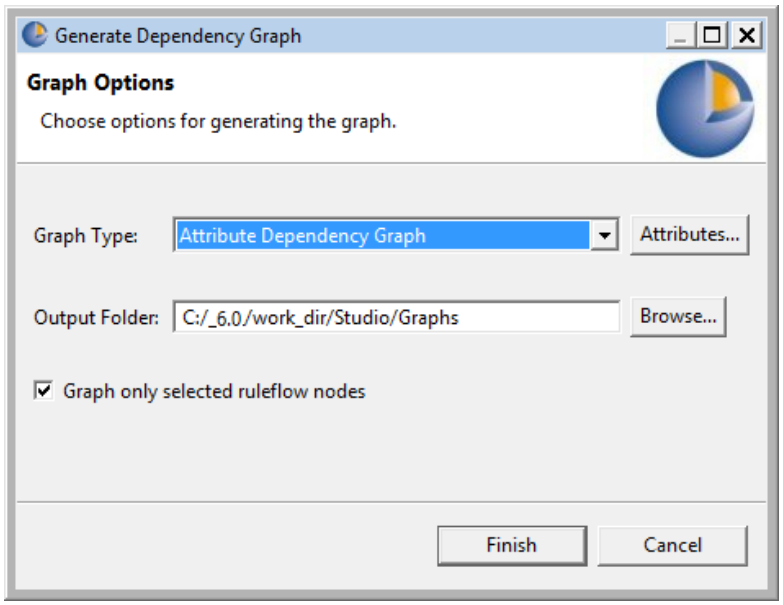
How to generate Ruleflow dependency graphs

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:



The **Generate Dependency Graph** dialog box opens:



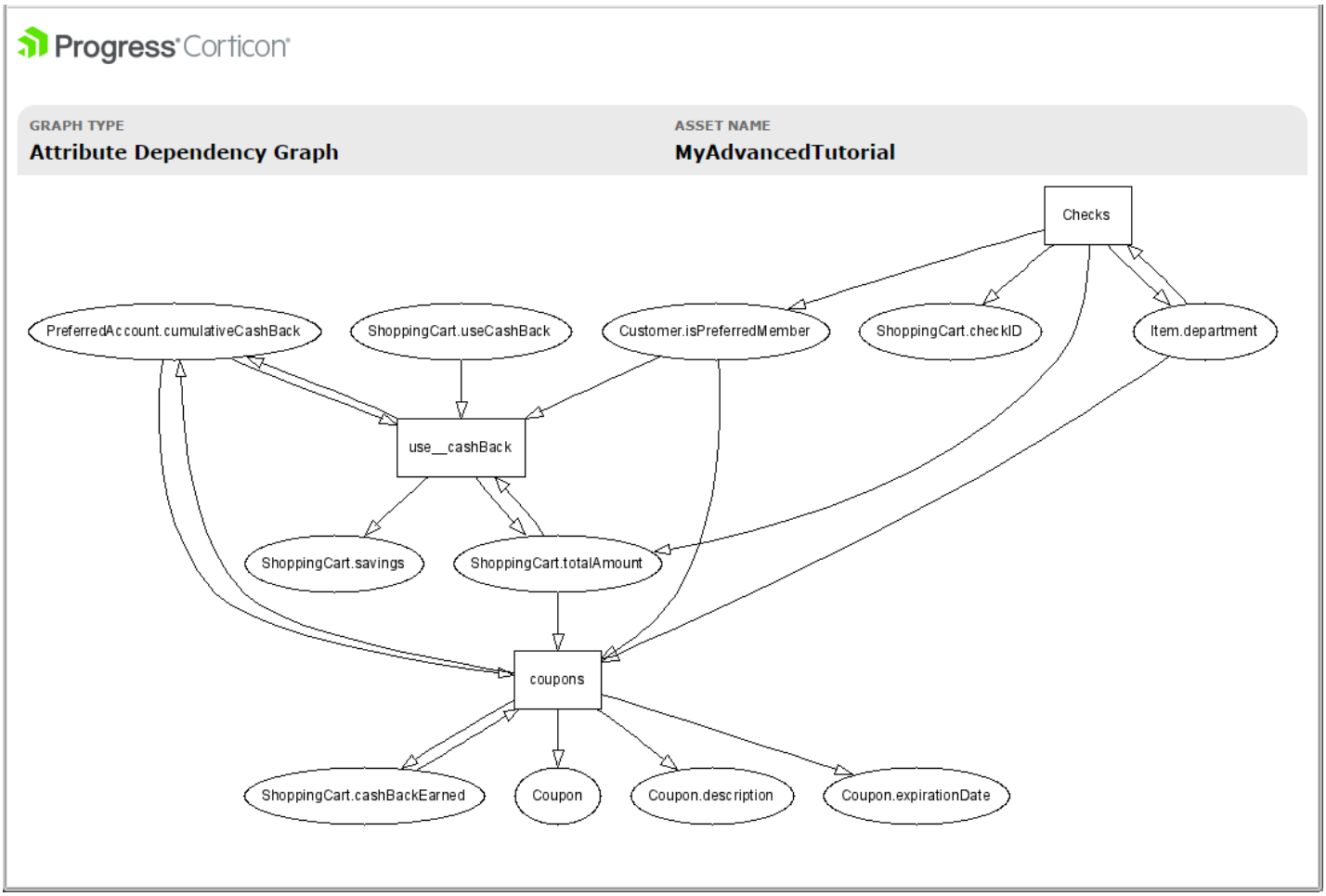
Choose the type of graph you want and the output folder. You can focus the analysis on just nodes that you selected before opening the dialog, or all nodes on the Ruleflow canvas.

Note: When no objects on the Ruleflow canvas are preselected, the option to graph only selected nodes has no effect.

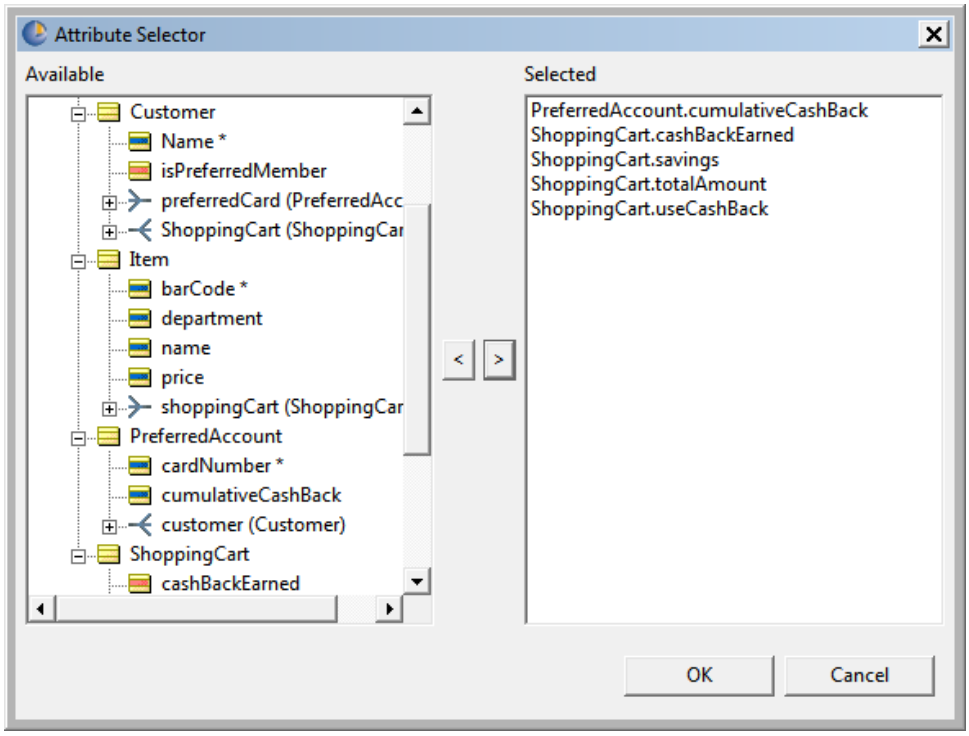
Attribute Dependency Graph

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:

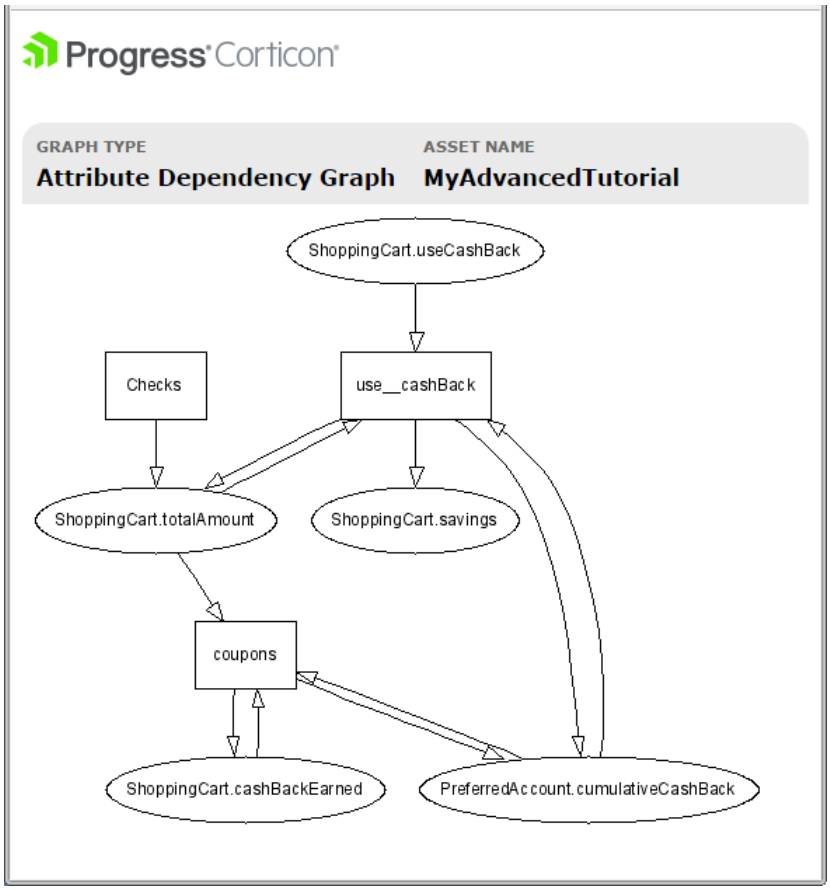


For large projects, graphs with all the attributes and dependencies can be difficult to work with. You can specify that only selected attributes are to be analyzed. Click **Attributes** to open the **Attribute Selector** dialog box, as shown:



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:

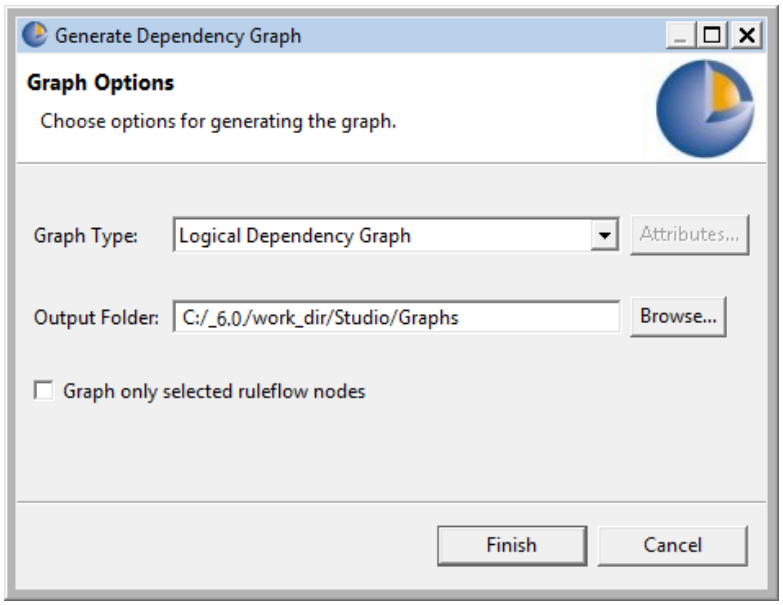


The graph image and its supporting files are saved in the output folder.

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.

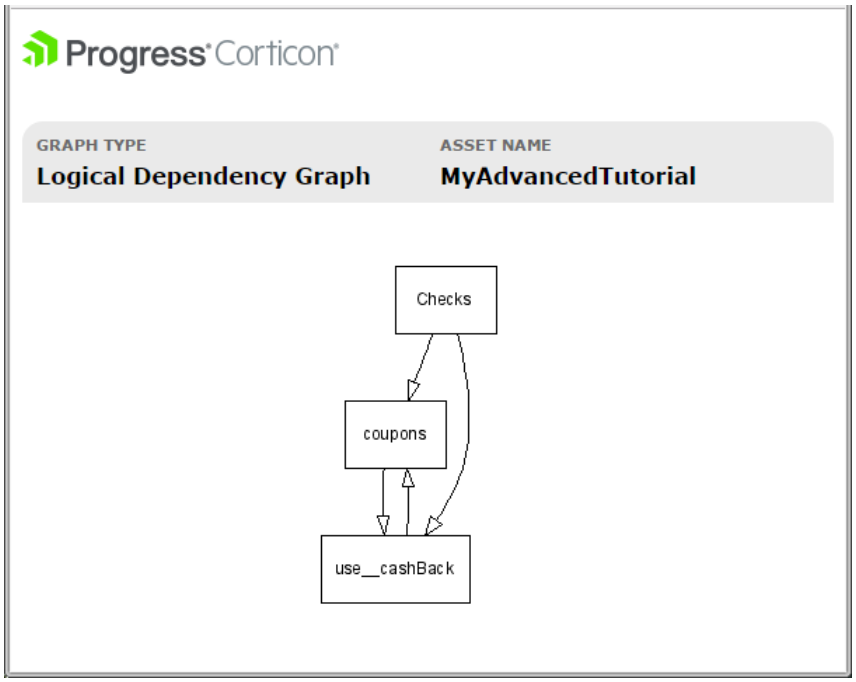
Logical Dependency Graph

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.

12

Troubleshooting Corticon.js Studio problems

When developing rules in Corticon.js Studio or deploying rules to production, you need strategies for preventing and resolving problems where rules are not producing the expected results. Considering these early in your project will help you establish best practices and prepare you for when a problem occurs. Corticon provides capabilities to help.

This section provides both best practice guidance and specific techniques to help you avoid and troubleshoot problems.

Break the project down into discrete, modular components

Corticon lets you create complex Ruleflows from smaller building blocks. When starting the development of a new rule project, create multiple simpler Rulesheets to satisfy a rules requirement rather than a single complex Rulesheet. Simple Rulesheets will be easier to validate. Create multiple small Ruleflows and assemble them into a larger Ruleflow using nested Ruleflows. Small Ruleflows are similarly easier to validate. Creating a complex Ruleflow from already validated small building blocks will minimize the chance of problems.

Use the Rulesheet logical analysis functions

Corticon provides logical analysis functions to identify gaps and conflicts in your Rulesheets. Utilizing these during rule development will minimize the chance of run time anomalies.

Create Ruletests for individual Rulesheets and Ruleflows

Creating Ruletests for each Rulesheet and Ruleflow will provide a means to unit test, or validate, each. Knowing that each of these building blocks has been well tested will give you greater confidence you will not see problems when assembling them into Ruleflows.

When modifying a rule project, run existing Ruletests

Running Ruletests will help ensure you do not introduce unintended changes in the behavior of Rulesheets or Ruleflows.

Troubleshoot rules in Ruletests

Utilize debug logging if your rules are not working as expected.

To enable debug logging in Corticon.js Studio, edit your brms.properties file and add the line:

```
loglevel=DEBUG
```

The log will detail the execution of your rules when you run Ruletests so that you can better understand the execution of your rules and identify anomalies causing incorrect results.

Restart Studio and the new log level will be used when you run Ruletests. The log information will be written to the file `CcStudio.log` in your log folder of the `[CORTICON_WORK]` directory specified during install.

Debug Logging in Deployed Rules

To enable debug logging when deploying rules, you need to set the `logLevel` to 1 in the JSON configuration object passed to your rules when executed from your JavaScript application.

```
const configuration = { logLevel: 1};
  result = decisionService.execute(payload, configuration)
```

See the generated wrapper code produced by Corticon.js Studio when packaging rules for deployment to various supported platforms.

Debugging JavaScript Platforms Differences

Corticon.js supports rule deployment to any platform with a compatible version of JavaScript. In principal, Corticon rules should execute identically. When you encounter a problem where rules are executing differently, you should perform the following tests:

1. **Execute rules in the Corticon.js Studio tester**: Corticon.js Studio runs rules for the tester by deploying them behind the scenes to an instance of Node.js and executing them with the input payload specified in the Testsheet. If your rules don't execute correctly, there is either a problem in your rules or a bug in Corticon.js.
2. **Execute rules in a browser**: Corticonj.js Studio provides the option to package rules for browser deployment. As part of this Corticon.js produces a simple HTML file, `browser.sample.html`, for testing rules in a browser that presents a simple form where you can provide an input JSON payload, pass it to your rules, and then see the results.

   If your rules execute correctly in the Corticon.js Studio tester but not correctly in the browser, there is a difference in behavior of the underlying JavaScript engine and you should contact Corticon support.
3. **Execute rules on your target platform**: Your rules might execute correctly in both in the Corticon.js Studio tester and in a browser, but behave differently on your target platform. There might be an incompatibility in the JavaScript engine on your platform or some platform specific environment issues are interfering with rule execution; for example, insufficient memory. You could contact Corticon support but if it is a problem unique to your platform, their options for assistance will be limited.

Preparing to Call Support

When you need to contact Progress support for assistance with Corticon.js rule execution errors, be prepared to provide any related debug logs, as well as your rule assets and a sample JSON payload demonstrating the incorrect behavior.

For details, see the following topics:

- Where did the problem occur
- Use Corticon Studio to reproduce the behavior

- How to compare and report on Rulesheet differences

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.js 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.js 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.

Analyze Ruletest results

This section assumes:

- Your Ruletest produced none of the errors listed above, or
- You or Corticon.js 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 deployment.

  The log captures errors and exceptions caused by certain rule and request errors.

- 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 168: Rule messages when metadata is enabled in Studio

Rule Messages

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
```

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.

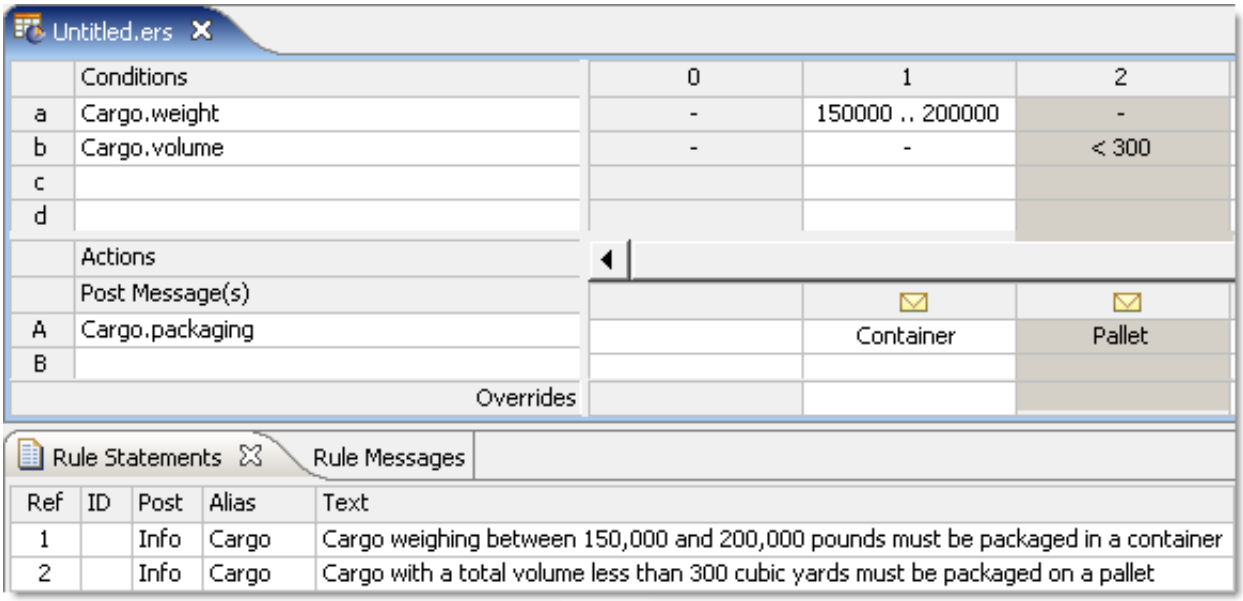
An important tool to help identify the breakpoint is the Ruletest's message box. By choosing values for `Post` and `Alias` columns in the **Rule Messages** window, you can generate a trace or log of the rules that fire during execution. The message box in a Ruletest displays those messages in the order that they were generated by . 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 . 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 169: 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 170: 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.js Technical Support. As always, be prepared to send the product version used, and the set of Corticon.js files (`.ecore`, `.ers`, `.erf`, and `.ert`) that will enable us to reproduce the problem.

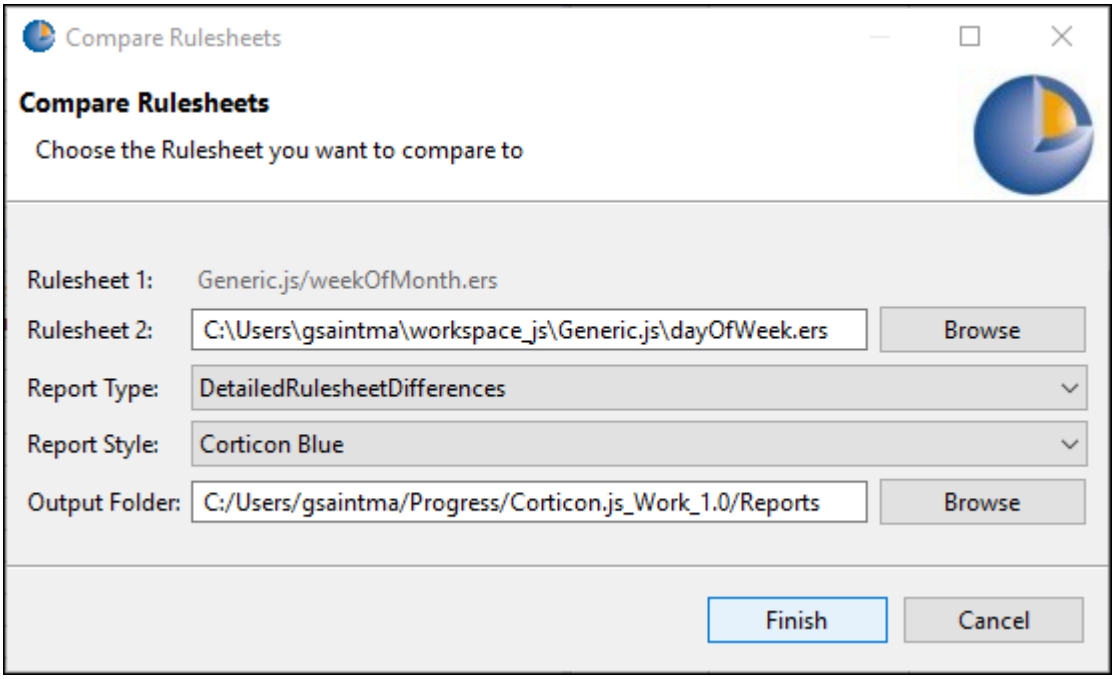
How to compare and report on Rulesheet differences

When the execution of your rules is not producing the expected results and your not sure what changed, Corticon.js 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 on 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 has 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 illustrated.



   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.js_WORK_DIR]/Reports`. You can create a root location such as

`C:\Corticon.js_Reports` 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.js__WORK_DIR]\Reports\XSLT\` in folders according to the asset types. You can copy the files to use as templates or change them to create report types that are then offered in the Report Type dropdown menu for the asset type.
- The style files are located at `[CORTICON.js_WORK_DIR]\Reports\CSS\`. You can copy a stylesheet file to use as a template to create custom report styles that are then offered in the **Report Style** dropdown menu.

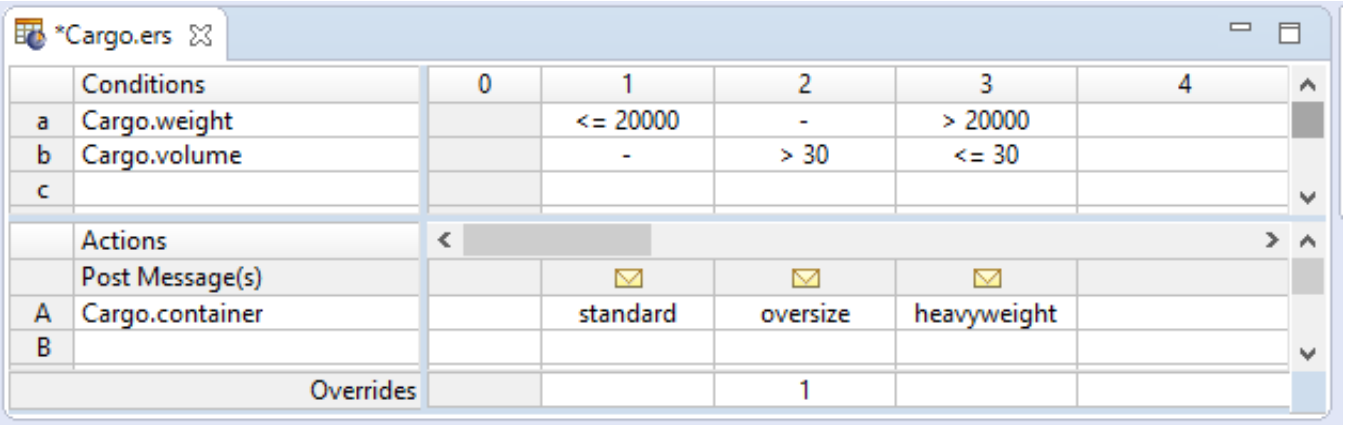
Reading a differences report

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

*Cargo.ers

	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	
B						
	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 has been disabled

*Cargo.ers

	Conditions	0	1	2	3	4
a						
b						
c	Cargo.volume		-	> 30	<= 30	
d	Cargo.weight		<= 20000	-	> 20000	
	Actions					
	Post Message(s)					
A	Cargo.container		standard	oversize	heavyweight	
B						
C						
	Overrides			1		

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.

Conditions

Rulesheet1	Rulesheet2
a.	
b.	
c. Cargo.volume Disabled	b. Cargo.volume
d. Cargo.weight	
	a. Cargo.weight
	c. Cargo.needsRefrigeration

A

Customize Corticon.js Studio

Corticon.js Studio provides a small set of properties to change its default behavior. These properties are defined in the `brms.properties` file in the `[CORTICON_WORK_DIR]` folder.

About the brms.properties file

- It is good practice to back up the file before you start to make changes.
- If you delete the file, it does not get recreated at restart. However, as 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 property settings in a text file, and then save it to its proper location as `brms.properties`.
- An update of the installation will preserve a modified `brms.properties` file, and will add the default file if none is present.

How to modify properties in the 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 # 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:

```
#com.corticon.javascript.studio.tester.decimalscale=6
```

and then change it to

```
com.corticon.javascript.studio.tester.decimalscale=2
```

Saving and applying 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` to use.

Note: The overrides and license specified are stored in the Studio Workspace. If you change the Workspace, those overrides or defaults will take effect.

For the revised settings to take effect, save the edited file, and then restart the Corticon Studio.