1Integrate

Training

Rules and Basic Concepts

Product version: v 1.6

Document version: v 1.2

Document date: 02/11/2017

1Spatial
Tennyson House
Cambridge Business Park
Cambridge
CB4 0WZ
United Kingdom

Phone: +44 (0)1223 420414

Fax: +44 (0)1223 420044

Web: www.1spatial.com

Contents

1 Introduction

The 1Integrate Rules and Basic Concepts training course covers the following topics:

Part 1: Overview:

- Key concepts and overview
- 1Integrate workflow
- 1Integrate interface
- Data import
- Rules interface
- Rule nodes

Part 2: Rule building:

- Simple attribute check
- Rule on two classes
- Rule refinement
- IF...THEN...ELSE structure
- Multiple existence conditions
- For All structure
- Built-in functions
- Rule templates

Part 3: Exercises:

- Transport network data
- Utilities data
- MidCoast water case study

Course Objectives

At the end of this course you will be familiar with the 1Integrate interface for connecting to an external spatial data source and assessing the quality of the data against a set of defined business rules.

You will also have a thorough understanding of the 1Integrate rule builder for creating and refining rules.

Prerequisites

The course assumes you have the skill set and software tools described below.

Skills:

- Experience of GIS or a map production environment
- Experience of working in Oracle or another database environment
- Basic knowledge of object-oriented design concepts

1Spatial tools:

- 1Integrate

2 Key Concepts

1Integrate is a spatial processing, analysis and compliance engine that ensures the quality and consistency of spatial data. This solution is delivered in a J2EE architecture that offers web access through an Application Server.

1Integrate enables users to rapidly analyse scattered spatial data to establish its operational purpose and facilitate its reuse by providing rules-based conformance checking.

Spatial data can be certified to enable critical business decisions to be made, safe in the knowledge that it is fit for a particular purpose because it is tested and documented to be so.

1Integrate consists of:

- A knowledge management tool (collaboration)
- A spatial rules engine (rules-based paradigm)

1Integrate's component-based Web architecture provides:

- J2EE application, container managed
- Full function SOAP interfaces
- BPEL compatibility
- AJAX Web interface
- XML rules language
- Structured knowledge base

Objectives

By the end of this chapter you will be able to:

- Understand the 1Integrate concepts and architecture
- Understand the types of conformance checking 1Integrate can perform

Errors in Spatial Data

1Integrate greatly reduces the time taken to check for errors and provides a rigorous, efficient, and cost-effective mechanism to measure and maintain spatial data quality.

When an operator is viewing the data, errors in the graphical representation can easily be missed.

In the following example, the data looks connected but when zoomed in, the error is more apparent.

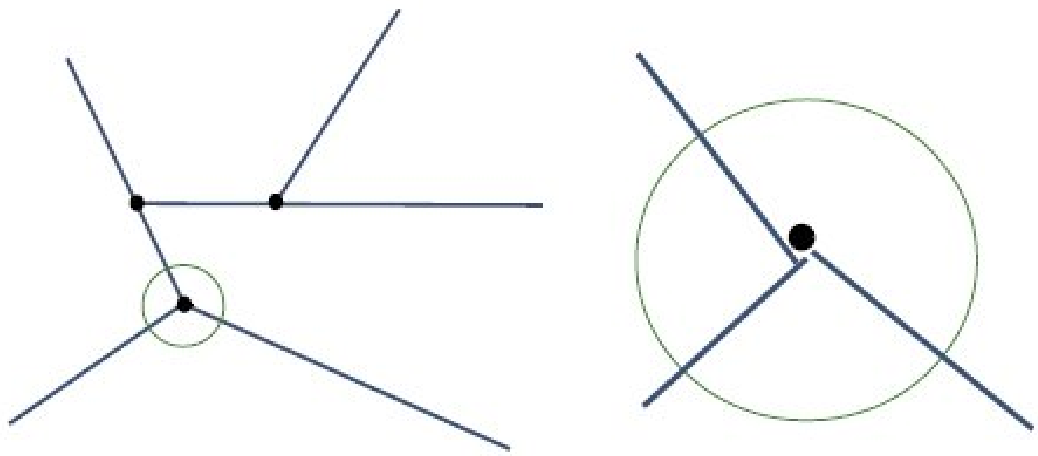


Figure 2-1: *Data is seen to not be connected when zoomed in*

Similarly, errors in logical representation can be easily missed.

In the following example, the subtype attribute for these two objects should be the same. Again, this type of error could easily be missed and could have costly consequences.

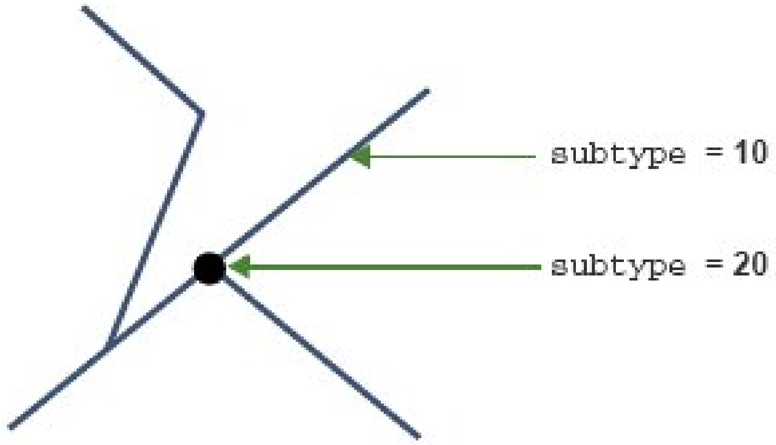


Figure 2-2: *Data has different subtype attributes*

Philadelphia data

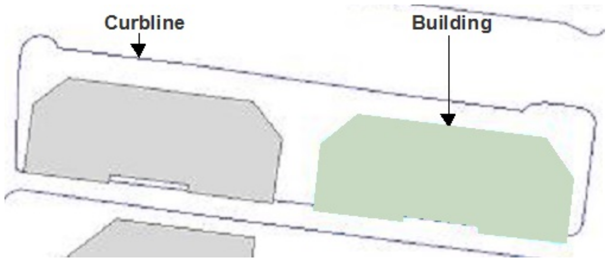
The data used in the first part of this training course consists of various layers that cover downtown Philadelphia.

The layers, owned by several departments and used in isolation, have been brought together to see how well they adhere to business rules. The layers and original owners are:

- **Buildings** (Water Department) - building outlines captured as polygons using photogrammetry
- **Curblines** (Streets Department) - curbline outlines captured as polygons using photogrammetry
- **Street centre lines** (Streets Department) - derived from TIGER data
- **Zoning** (Planning Department) - polygons that describe building usage (for example residential, industrial) captured from small-scale, hard copy maps at a time when the accurate building information was not available
- **Fire stations** (Planning Department) - fire station points placed within a one block proximity of their actual locations within the city block (+/- 200 ft accuracy)

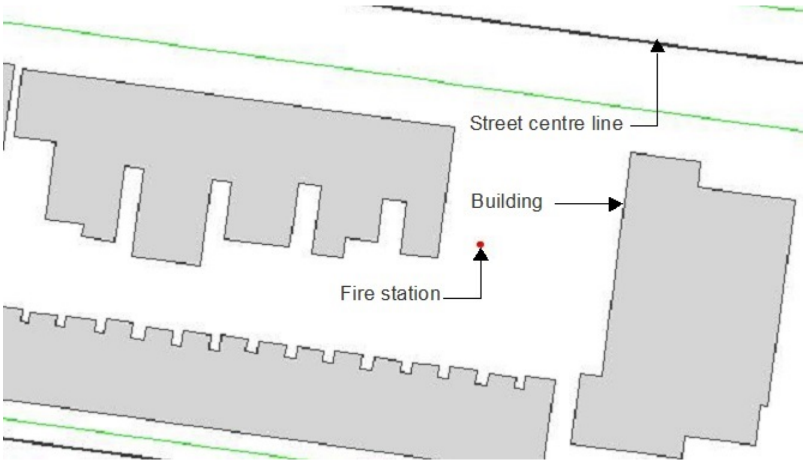
During the course, various rules will be created to assess the quality of the data.

For example, if the building and curbline layers were to be used for planning, it could be a requirement that the building polygons should not overlap the curbline polygons.



Also, the fire stations points were placed within a one-block proximity of their actual locations within the city block (+/- 200 ft accuracy); therefore, it is possible that when the fire stations and building layers are examined, some fire stations may be located in the wrong building or no building.

It would also be possible to check the fire station address attribute with the nearby street centre line name(s) to ensure that it has been located in the correct area.



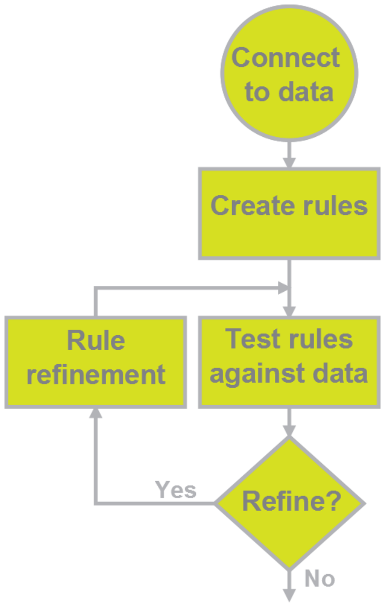
Summary

1 Integrate:

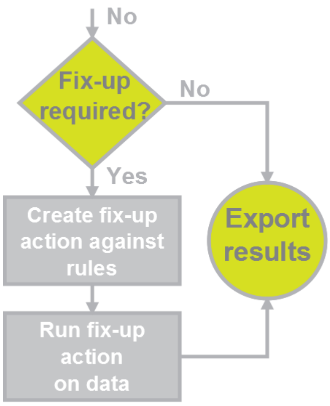
- Rapidly analyses scattered spatial data to establish its fitness for purpose
- Enables rigorous control of data quality and provides substantial return on investment by automating traditionally time-consuming, manual and expensive data quality tasks

3 1Integrate Workflow

1Integrate and 1Spatial Management Suite can be used by following the flow diagrams below:



The grey items in the following diagram are not covered in this course:



Objectives

By the end of this chapter you will be able to:

- Understand the 1Integrate workflow
- Run a session consisting of a set of Check Rules tasks and view the results

Key Concepts

Task

A task is a phase in the 1Integrate workflow.

1Integrate can run many types of tasks, but we are going to focus on the following task types for this course:

- Open Data
- Check Rules
- Pause

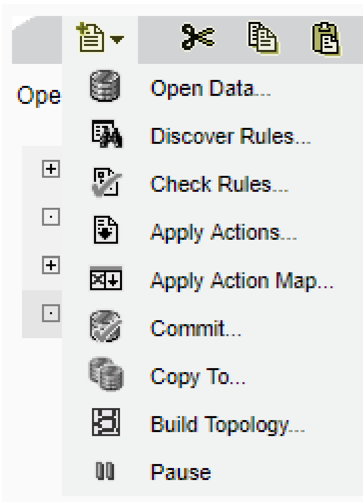


Figure 3-1: *Adding tasks to a session*

Tasks execute asynchronously so that you can monitor progress and abort sessions. They continue processing even if you log out or shut down the client computer. All status values and reports associated with the task are retained.

Session

A session is a sequence of tasks that make up a workflow in 1Integrate.

For example, starting with one or more Open Data tasks and one or more validation rules. A session is a sequence of processing tasks, starting with one or more Open Data tasks.

Sessions workspace

The Sessions workspace has two tabs: **General** for metadata and **Tasks** for adding, deleting and reordering tasks in a workflow.

Multi-level undo and redo may be used to correct mistakes.

The Session Description allows for the details and behaviour of a session to be easily read, saved and shared.

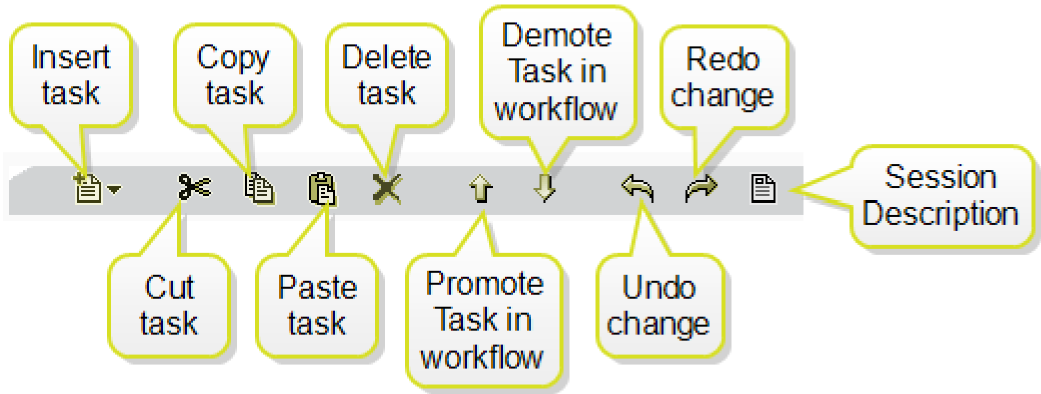


Figure 3-2: *Session workspace toolbar*

Media player controls are used to run tasks:

The play button starts a task or resumes a paused task

The pause button suspends execution of a task

The rewind button rewinds the most recent task performed

The stop button discards all data and rewinds to the start of the first task

In the list of tasks, the selected task has a grey background, completed tasks have a green background and incomplete tasks have a light grey background.

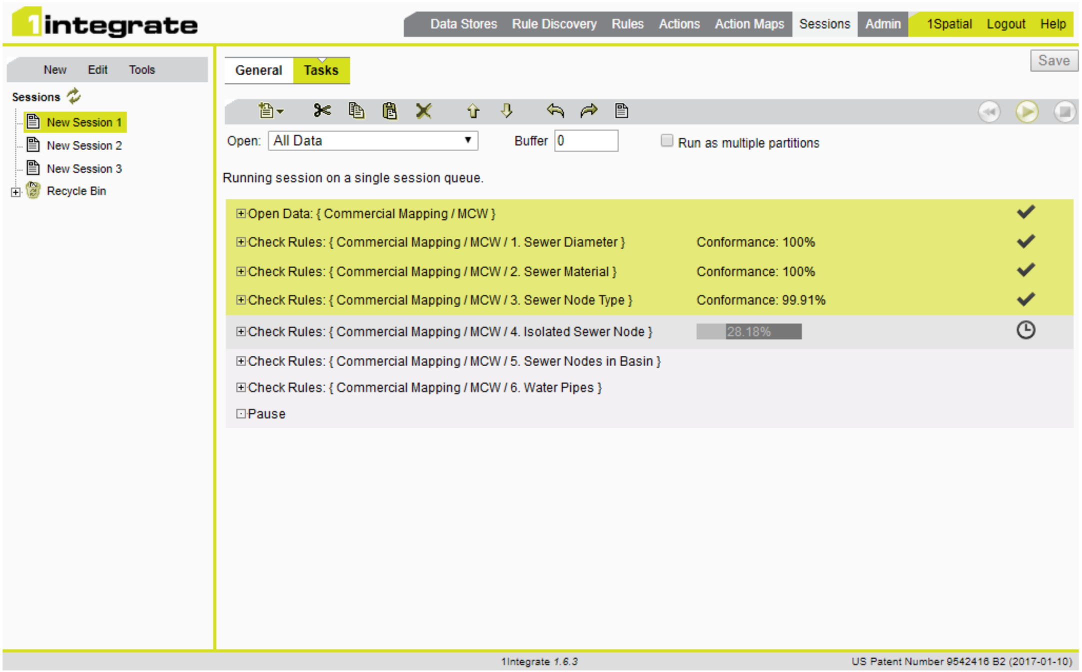


Figure 3-3: *A session in progress*

Note: By adding a **Pause** after the last task, the session stays active. If all tasks are completed without a Pause at the end, the data is removed from the 1Integrate cache and the session ends. When a session completes at a Pause task, a Rewind hyperlink appears so you can backtrack through the flow of your session.

Grid processing

Grid processing allows sessions to leverage the resources of multiple machines and multiple core machines. The hardware resource, or grid, will run a 1Integrate Session Queue, which is the software that manages 1Integrate sessions.

A single instance of the user interface and Web service interface interacts with all session queues on the grid, allowing a single session to be automatically allocated to a session queue when it is started. During the startup of multiple grid partitions the session is split and each partition is executed on separate session queues.

Multiple grids are processed in parallel allowing faster start and stop performances. Discovery of new session queues and handling of new, failed and stopped session queues is automatic, with failover to alternative session queues if required.

Exercise: Run a session

In this exercise, you will run a session that consists of an Open Data task followed by several Check Rules tasks.

1. Your instructor will allocate a session to you. Select the session and view the tasks.
2. Expand the Open Data task. Clear All classes and view the classes in the data store. Reselect All classes.
3. Click  to start the session.
4. When the Open Data task completes, expand the task and click Results: XML.

   How many objects were loaded into the cache?

   Your results

5. Click  again to start the Check Rules tasks.

   The session runs up to the final Pause task and the Rewind hyperlinks are then displayed.

Viewing results

Every **Check Results** task has an HTML report that gives an overview of the conformance results, with the IDs of the objects which fail the rule.

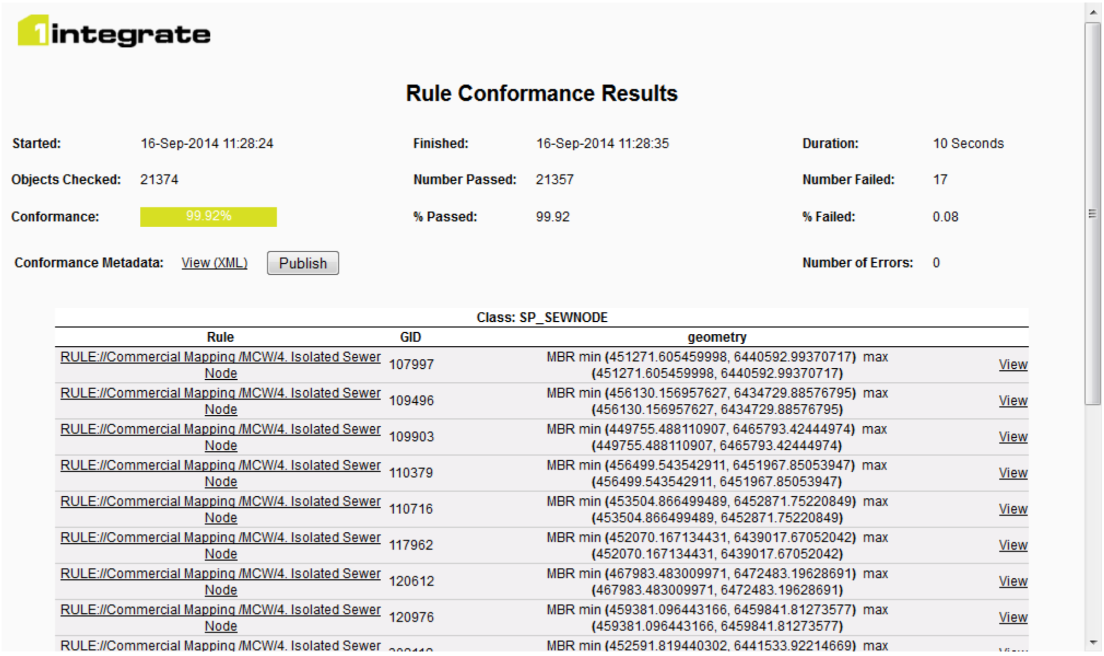


Figure 3-4: *Rule Conformance Results*

Summary

This chapter introduced the Sessions interface and shown how to order tasks to create a workflow. You have created a session and run some rules against a set of data.

4 1Integrate Interface

Objectives

By the end of this chapter you will be able to:

- Use 1Integrate to navigate between interfaces and tabs
- Use the Edit menu to manipulate new elements (e.g. folders)

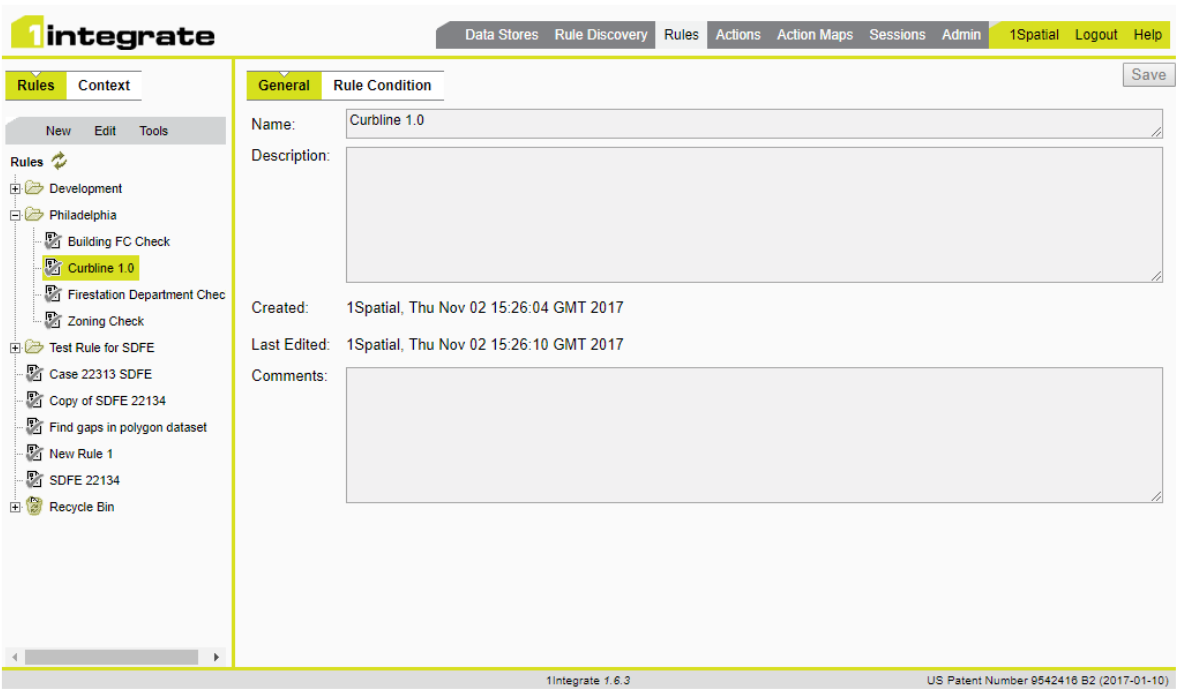
Key Concepts

1Integrate GUI

The graphical user interface (GUI) is organised into four interfaces that access the different parts of the system.

- **Data Stores** (described in "Data Import" on page 25)
- **Rules** (described in "Rule Nodes" on page 38 and "IF...THEN...ELSE" on page 68)
- **Sessions** (described in "1Integrate Workflow" on page 14)
- **Admin**

The following image is an example of the Rules interface and the **Curblines 1.0** rule in the **Philadelphia** folder.

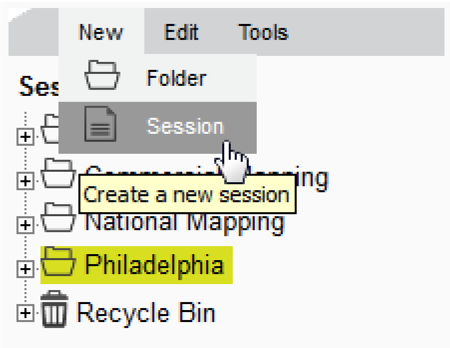


Each interface has a navigation pane on the left, and on the right a workspace holding details of a selected item. The navigation pane and workspace can be dragged and resized.

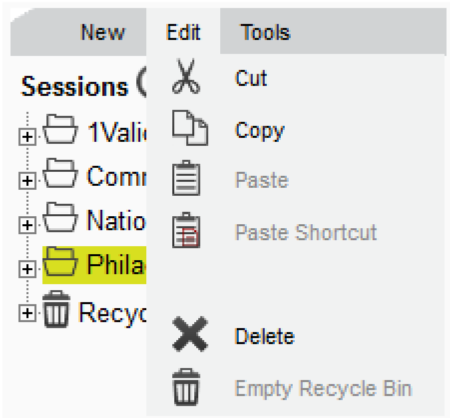
Navigation Pane

In all the interfaces the navigation pane contains three standard menus, **New**, **Edit**, and **Tools**.

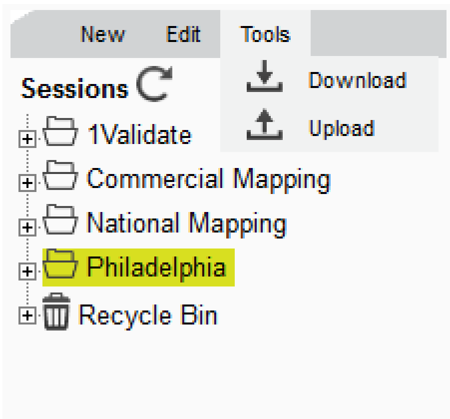
New is used to insert a folder or item:



Edit is used to organise the navigation pane:



Tools is used to provide rule backup functionality:



Folder

A folder is a logical grouping of items in the navigation pane. The 1Integrate interface implements standard AJAX functionality.

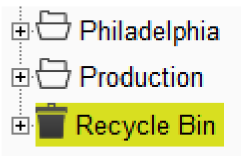
Note: 1Integrate folders do not correspond directly to file system directories.

General Tab

The workspace always has a **General** tab containing metadata:

- Name: A unique name within the current folder. There are no restrictions on the characters or length of a name
- Created: An uneditable field showing the name of the user who created the item and the date of creation
- Last edited: An uneditable field showing the name of the user who last edited the rule and the date of modification
- Description: An optional description in free format text
- Comments: Optional comments in free format text

Recycle Bin



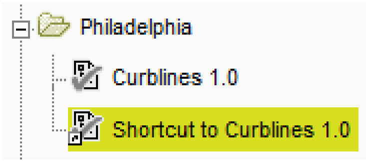
Deleted items are moved into a Recycle Bin, which is located at the bottom of the navigation tree on each top level tab. The Recycle Bin is shared by all users of the system and cannot be deleted.

An item in the Recycle Bin is removed from the navigation tree, but may still be used.

Note: If you empty an item from the Recycle Bin it is permanently deleted.

Referenced items, such as a data store or rule used in a session, cannot be emptied from the Recycle Bin until they are removed from the session or rule that references them.

Shortcut



A shortcut allows an item to be referenced from many places. Any change made through a shortcut is applied to the original item.

Shortcuts to the same item can be saved in folders for various uses. For example, a rule could be created to be used many times in different rule sets. By creating a shortcut, the rule does not need to be replicated in each folder.

> **Note:** Original items cannot be emptied from the Recycle Bin if a shortcut references it outside of the Recycle Bin.

Save

A Save button is enabled after you have made a change.

> **Note:** To finish editing a text field, select another field or click outside the field that you are currently editing.

Exercises

> **Exercise: Create a working environment**
>
> Create a folder called [your_name] in the following interfaces for use in this training course:
>
> - Data Stores
> - Rules
> - Sessions
>
> > **Note:** Text is entered into the 1Integrate pages when you tab out to another field.

> **Exercise: Edit functions**
>
> Become familiar with the edit functions used to copy, paste, and cut folders and folder elements.
>
> Delete anything you create except the original folders created above and empty the recycle bin.

Summary

1Integrate has a number of features that make the interface intuitive and easy to use.

Tools such as the Recycle Bin, shortcuts, and copy and paste should already be familiar as they replicate the features found in many applications.

The interface reflects the 1Integrate workflow with the interfaces for each phase of the process. "1Integrate Workflow" on page 14 describes the phases in more detail.

5 Data Import

Data stores and schema mappings connect 1Integrate to an external data repository.

Objectives

By the end of this chapter you will be able to:

- Set up a data store in 1Integrate
- Use the schema mapping interface
- Use the Session interface to open the created data store

Key Concepts

1Integrate cache

1Integrate's dedicated geospatial cache can hold multiple spatial datasets.

The cache is object-oriented, therefore we refer to classes and objects as opposed to feature types and features as in non object-orientated systems.

Data store

A data store is an external source of data, usually including a geographic component.

Rules are applied to data drawn from one or more data stores. These can be read into 1Integrate in isolation, or if the data is coincident, multiple data stores can be read in together.

The data is read into 1Integrate using a schema mapping definition and then processed against a set of rules that analyse relationships between data stores as well as within a data store.

Data stores workspace

The data stores interface defines connections to external data. Information is split into tabs:

- **General** - standard metadata relating to the selected item.
- **Input Details** - connection details to the database for importing data into 1Integrate.
- **Input Mapping** - mapping from the schema of the external data source to the 1Integrate session schema.
- **Output Details** - connection details to a database for exporting data from 1Integrate.
- **Output Mapping** - mapping from the 1Integrate session schema to the schema of the external data source.

> **Note:** Output mapping is only used for **Copy To** tasks in a session. If using a **Commit** task then the inverse of the input mapping is used and the output mapping is ignored.

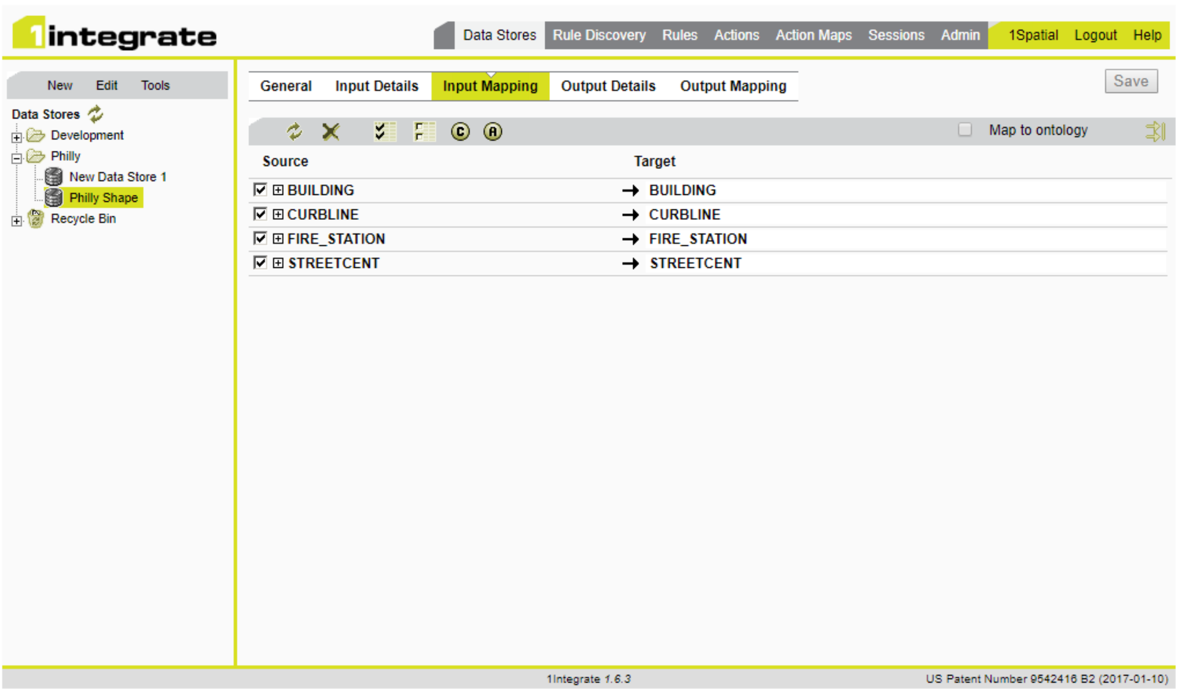


Figure 5-1: *Data stores workspace*

Database connection

For an Oracle database connection, the input details interface requires connection information that includes:

- Data Store Type
- User Name
- Password
- Net Service Name

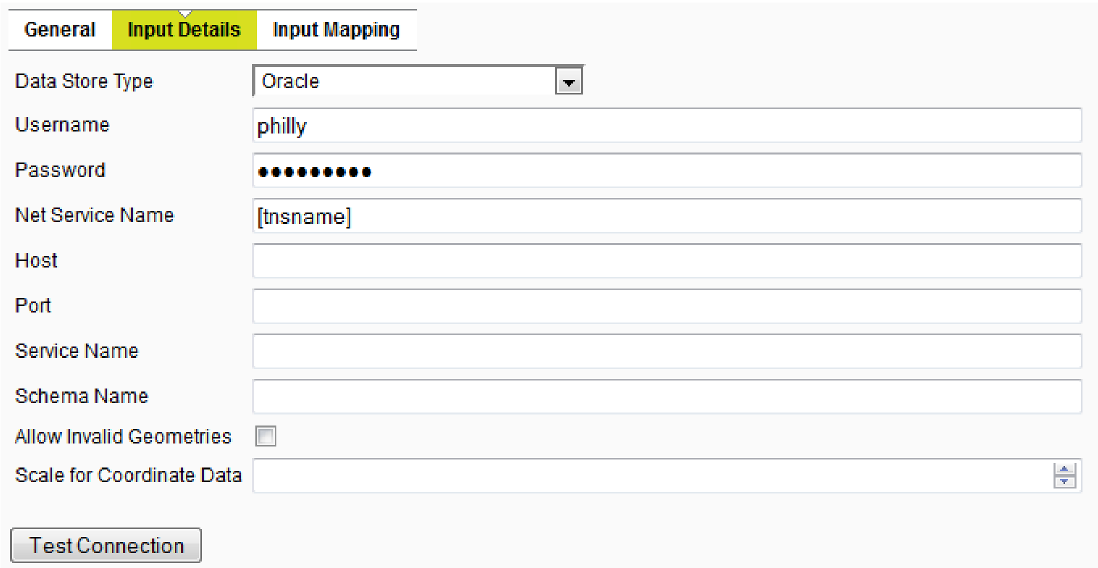


Figure 5-2: *An Oracle connection using the Net Service Name*

Alternatively, the following connection information could be provided (where Host, Port and Service Name are used instead of Net Service Name):

- Data Store Type
- Username
- Password
- Host
- Port
- Service Name

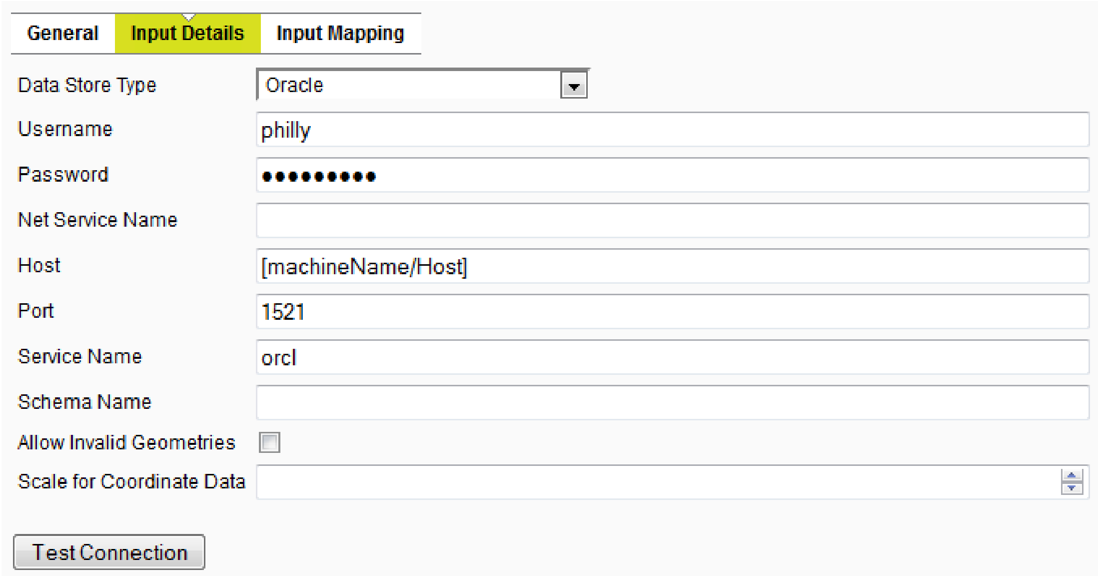


Figure 5-3: *An Oracle connection using the Host, Port and Service Name*

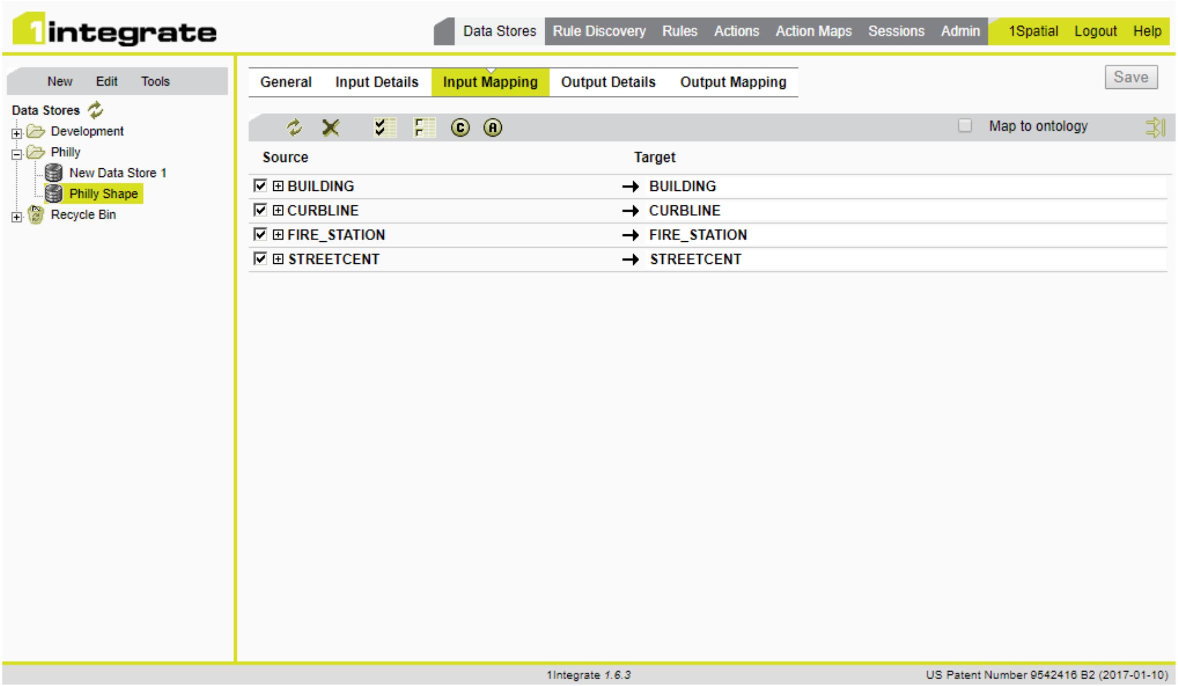
Schema mapping

The schema mapping defines how data is converted between the schema of a data store and 1Integrate's internal schema. The schema used by 1Integrate is a union of all data stores from which data has been opened.

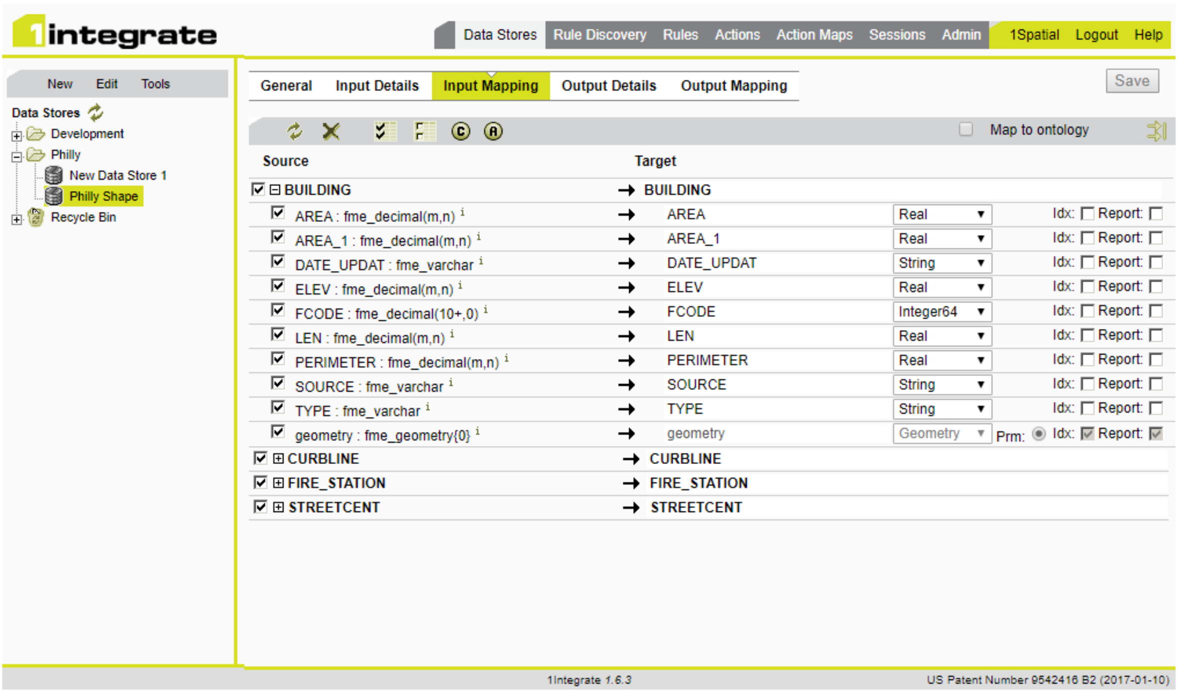
1Integrate supports multiple inheritance in external schemas. In these cases, where base classes are referenced, 1Integrate will deduce the appropriate set of classes.

Input mapping tab

Source and target maps are defined in the Input Mapping tab:



The following image shows the classes expanded to display attributes:



Source map

The source map is the schema from the external data source. The Source column in the Input Mapping tab shows the tables and columns from this source.

Only the checked items are analysed in 1Integrate.

Target map

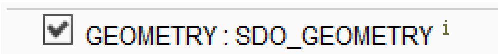
The target map is the original schema as represented in the 1Integrate cache. The Target column in the Input Mapping tab maps tables and columns to classes and attributes in 1Integrate.

You can override the name and type of each attribute. By default, 1Integrate is populated with the same names and data types as the external data source.

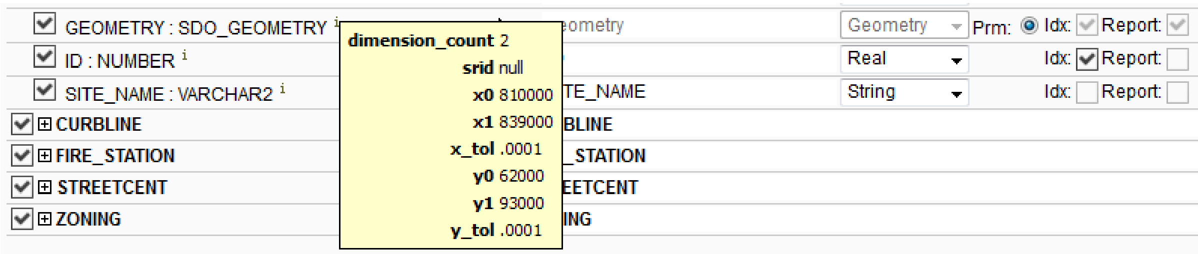
1Integrate also indexes geometry columns and the primary keys of the source data. Indexes are shown when idx is selected.

GEOMETRY attribute

You can view the metadata for each class containing a GEOMETRY attribute by clicking the superscript i (ⁱ):



For example:



The GEOMETRY attribute populates 1Integrate with the name geometry. If a table has multiple geometries, the name geometry exists for the primary geometry attribute only, while the secondary geometry attribute will have its original name. The user can change which geometry should be used as the primary geometry attribute.

Exercises

Exercise: Set up a datastore

In this exercise you will create a data store that will be used for the first part of the rule building exercises.

1. Go into the Datastores interface and open the datastore folder you created earlier.
2. Create a new data store called `[your name]_Philly`.
3. On the Input Details tab, use the connection parameters provided for the Philadelphia (philly) data.
4. Click Test Connection to ensure you are connected to the external data source.
5. On the Input Mapping tab, load the schema and select all classes and attributes.

   Expand the classes to view the attributes. Pick a class and view the metadata information on the geometry column.
6. Save the data store.

Exercise: Create a session

1. Navigate to the Sessions interface and in your folder create a session called `[your name]_Philly`.
2. Add an Open Data task and select the data store you created in the previous exercise.

   Ensure **All Classes** is selected.
3. Add a Pause task.
4. Run the session.
5. View the XML report for the Open Data task.

   How many objects were loaded?

Your results

Summary

In this chapter you have learned how to import your data into the 1Integrate cache, and how to map the data's schema to a 1Integrate target schema.

You have imported and mapped the Philadelphia data for the exercises in the following chapters.

6 Rules Interface

The rules interface provides a graphical user interface for building business rules to which your data should conform.

You may choose to define many different rules that apply to the same class of object.

Objectives

By the end of this chapter you will be able to:

- Understand the layout of the rules interface and the tools that can be used within it

Key Concepts

Rules workspace

The Rules workspace has two tabs: **General** for metadata and **Rule Condition** for defining rule clauses.

Multi-level undo and redo may be used to correct mistakes.

You can reorder nodes by dragging and dropping. You can also cut and paste all or part of a rule into another rule.

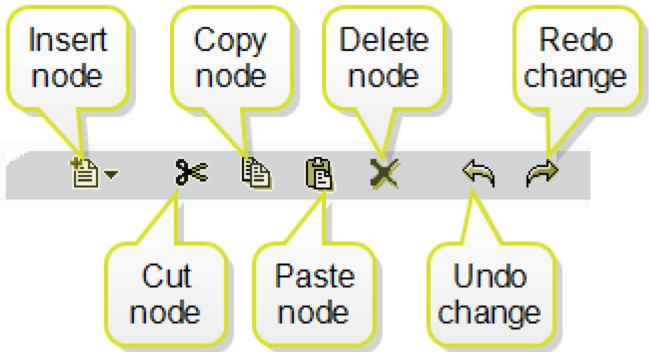


Figure 6-1: *Rules workspace toolbar*

The workspace is split into three panels:

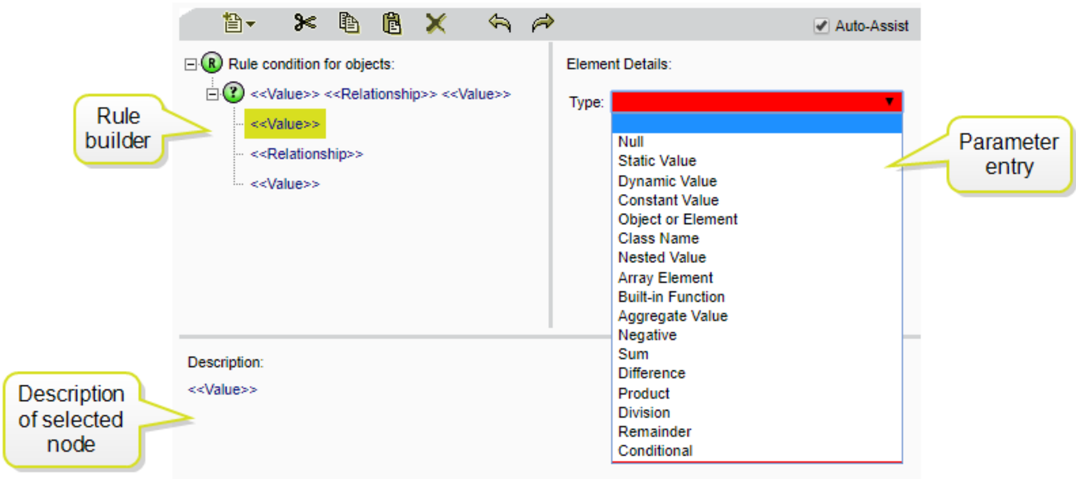


Figure 6-2: *Rules workspace panels*

Auto-assist

Auto-assist inserts placeholders for the values and relationships required by a node’s syntax.

When auto-assist is selected, placeholders show the child nodes required for a parent node.

The placeholders are:

- <<Condition>>
- <<Value>>
- <<Relationship>>

Select the placeholder to view a list of available elements.

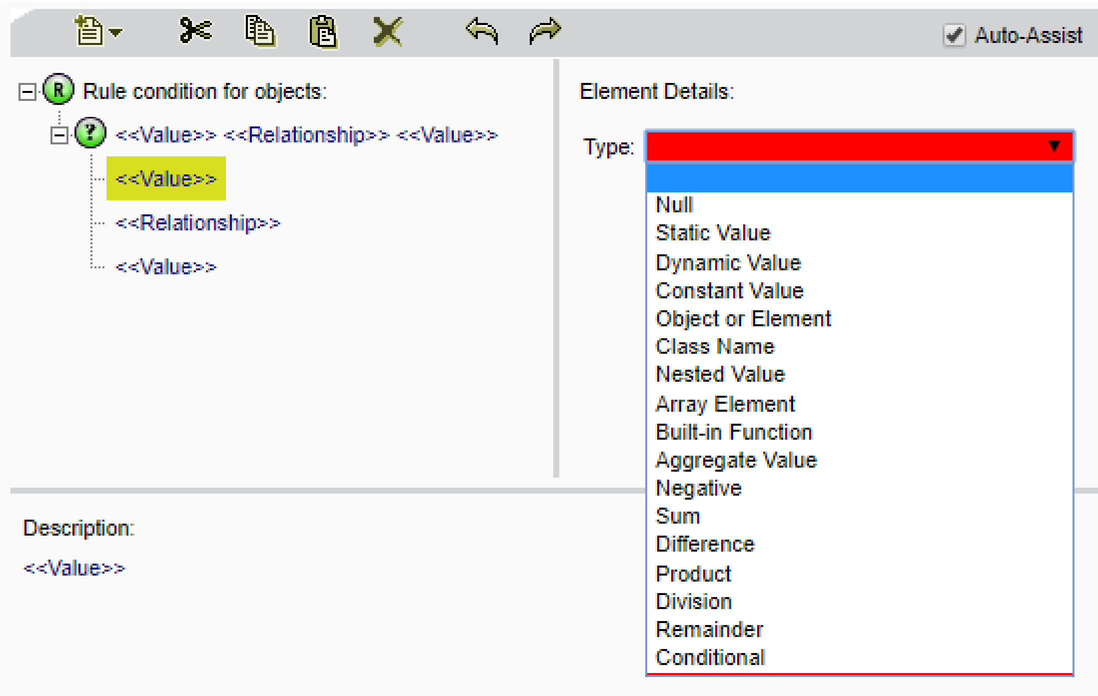


Figure 6-3: *Value and relationship placeholders created by auto-assist*

Note: Auto-assist is selected by default. When auto-assist is not selected you must add the values and relationships from the Insert menu using the correct syntax.

Context

The schema entities and relationships defined for a data store are known as the context.

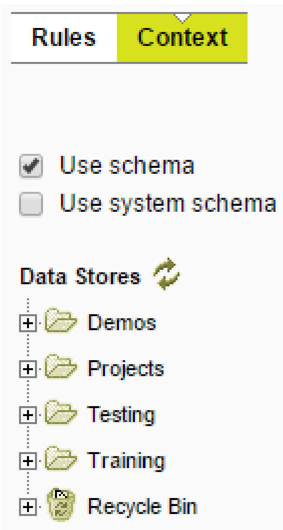


Figure 6-4: *Context for a data store*

If Use Schema and a data store are selected from the Context tab on the navigation pane, available classes and attributes are listed as parameters for any of the following types of value:

- Dynamic Value
- Object or Element
- Class Name
- Nested Value
- Array Element
- Aggregate Value

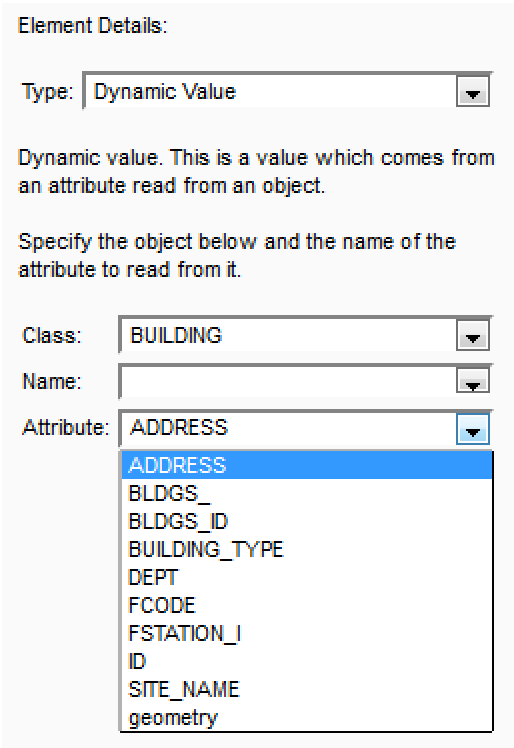


Figure 6-5: *Attributes available from the schema*

> **Note:** If the context is not defined, class and attribute names are free format text.

Summary

The Rules interface allows you to create and save rules on your data. Text and graphical representations of the rule are displayed.

The interface has an auto-assist feature which shows the structure for your rule. You fill out the rule details in the parameter entry section and if Use

Schema is selected, 1Integrate suggests schema names when setting up the rules.

"Rule Nodes" on page 38 describes the elements that make up a rule.

7 Rule Nodes

In 1Integrate rules are displayed as a tree of nodes. There are several types of nodes, which are described in this chapter.

Objectives

By the end of this chapter you will be able to:

- Understand the different types of nodes used in the 1Integrate rule builder

Key Concepts

Condition node

A condition is a high-level logical test that defines the syntax for a rule.

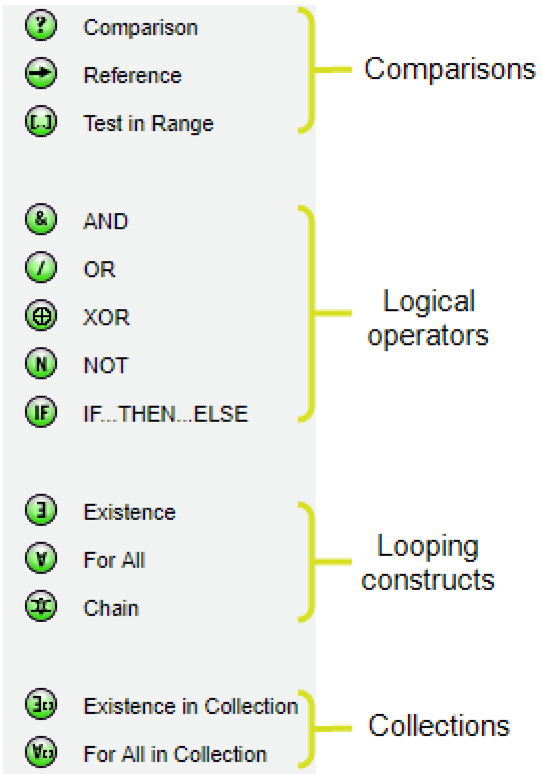


Figure 7-1: *Condition nodes*

Value node

A value is a constant, a reference to objects or a calculation, that is compared using a relationship.

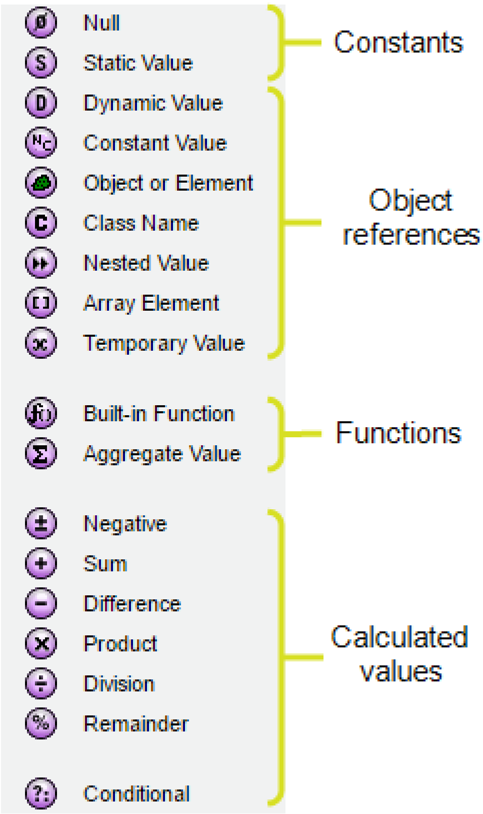


Figure 7-2: *Value nodes*

There are a number of possible types of value:

- null
- integer
- real
- boolean
- string
- geometry
- array
- complex attribute
- object (a feature from the data store)
- collection of objects

Arrays can contain integer, real, boolean, string, geometry, and complex attribute values, but not null values.

A complex attribute is composed of named parts, each of which may be a null, integer, real, boolean, string, geometry, array, or complex attribute value.

Scalar relationship node

A scalar relationship compares any two values that can be boolean, integers, real numbers, or literal strings. The result is evaluated as true or false.

For further details see "Scalar Relationships" on page 132.

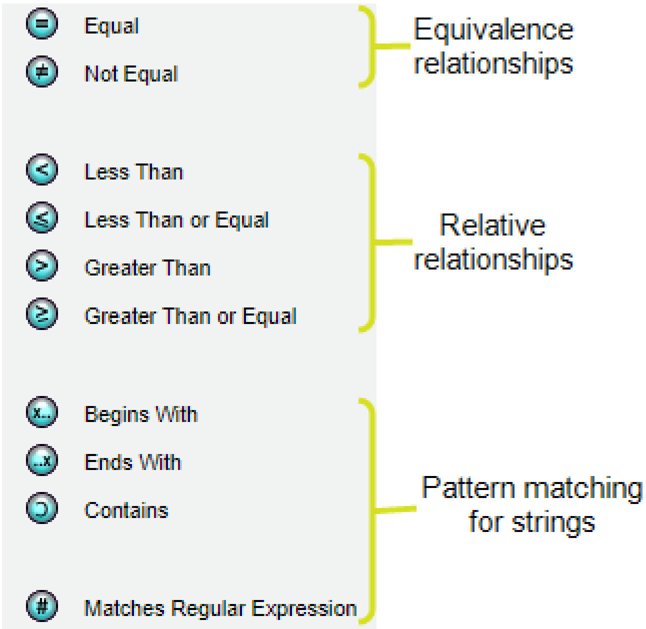


Figure 7-3: *Scalar relationship nodes*

Spatial relationship node

A spatial relationship compares two geometries.

For further details see "Spatial Relationships" on page 133.

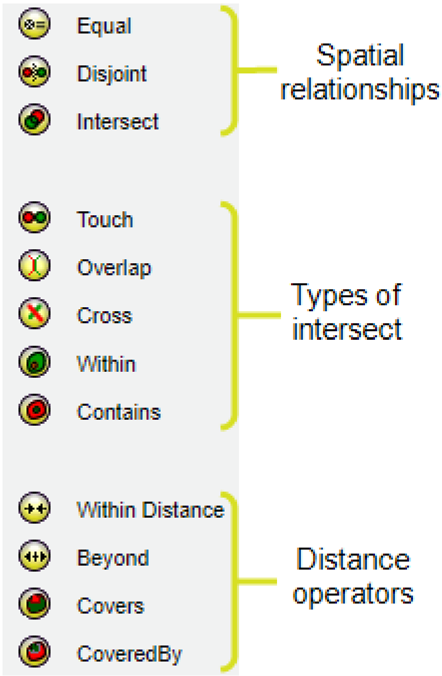


Figure 7-4: *Spatial relationship nodes*

Summary

This chapter has introduced the concept of a node in the rule builder and shown the different nodes available when building rules. Some of these nodes will be used in this course.

8 Simple Attribute Check

This exercise provides step-by-step instructions to create a rule in 1Integrate that performs a simple attribute check.

We will also introduce key concepts for rule building: root nodes, child nodes and class labels.

Objectives

By the end of this chapter you will be able to:

- Use the rule builder to create a rule using a comparison condition to perform a simple attribute check
- Run the created rule in the Session interface and view the results

Key Concepts

Root node

The root node is the parent for all nodes in a rule and appears at the top of the rule tree.

It defines top level objects and has one child node which is a condition defining the rule structure.

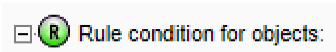


Figure 8-1: *The root node of a rule*

Class label

The class label identifies classes to be queried. If the class label is omitted, all classes are queried. A class label is used on the root node.

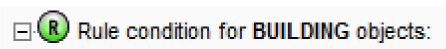


Figure 8-2: *A root node where the class label is set to BUILDING*

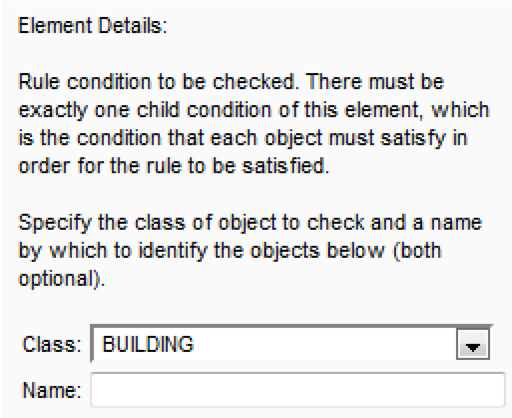


Figure 8-3: *Element details, showing the class label is set to BUILDING*

Child node

A child node calculates a value or performs a test which feeds into a rule.

For example, a Comparison has three child nodes: two values, and a relationship.

> **Note:** These child nodes are created automatically if using Auto-Assist (see "Auto-assist" on page 34).

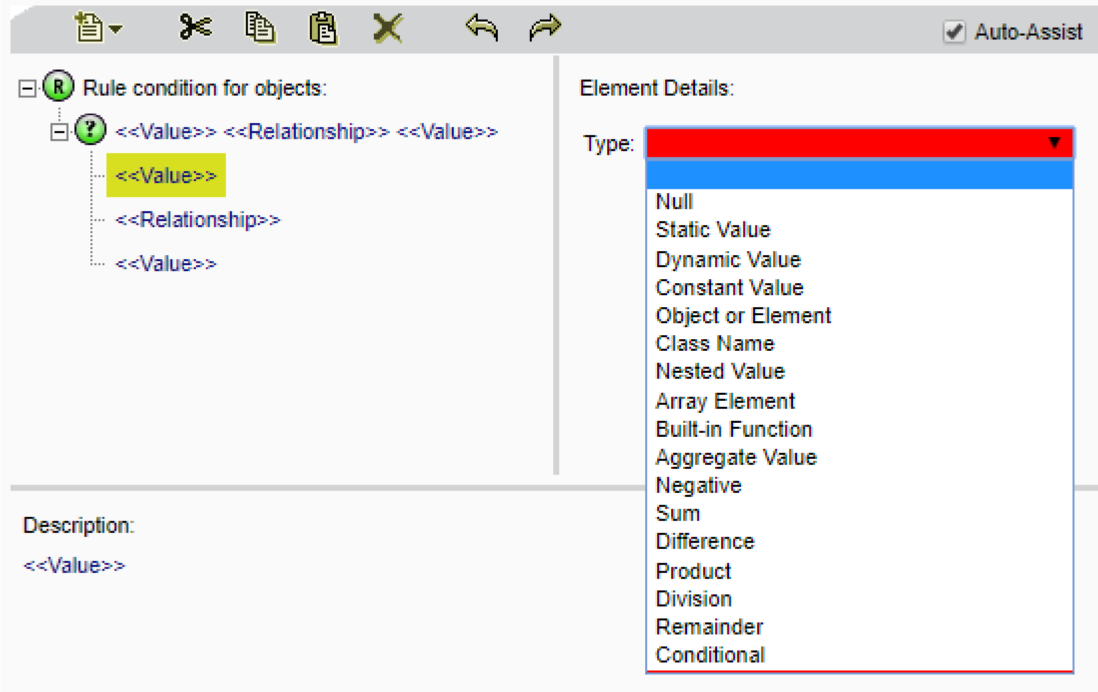


Figure 8-4: *Selecting the element type for the selected placeholder*

Select the type of value or relationship for the child node in the parameter entry pane.

Worked Example: Building FC Check

The example shows a simple rule using a scalar relationship and a static value.

The rule description will look like:

`Check for BUILDING objects that BUILDING.FCODE equals 1810`

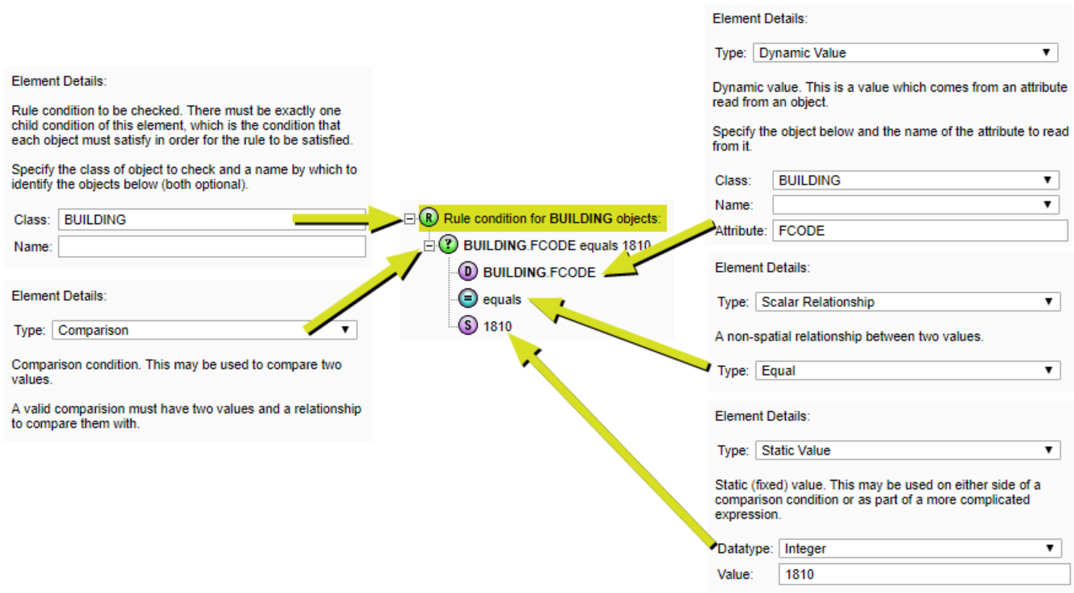


Figure 8-5: *Building FCODE Equals 1810*

Preparation

1. Select **Auto-assist** (see "Auto-assist" on page 34).
2. Select **Use schema** and the **Philly** schema (see "Context" on page 35).
3. In your Rules folder create a new folder called `Philly`.

Create the Rule

> **Note:** When you create a new rule, it is good practice to give a full description of the rule in Description on the General tab, as this will aid you when you return to the rule at a later date.

1. In the navigation pane select your **Philly** folder.
2. Select **New** > **Rule**.

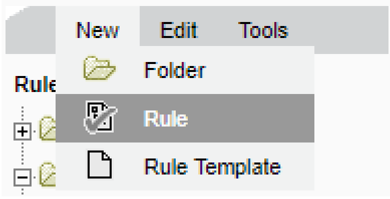


Figure 8-6: *New > Rule*

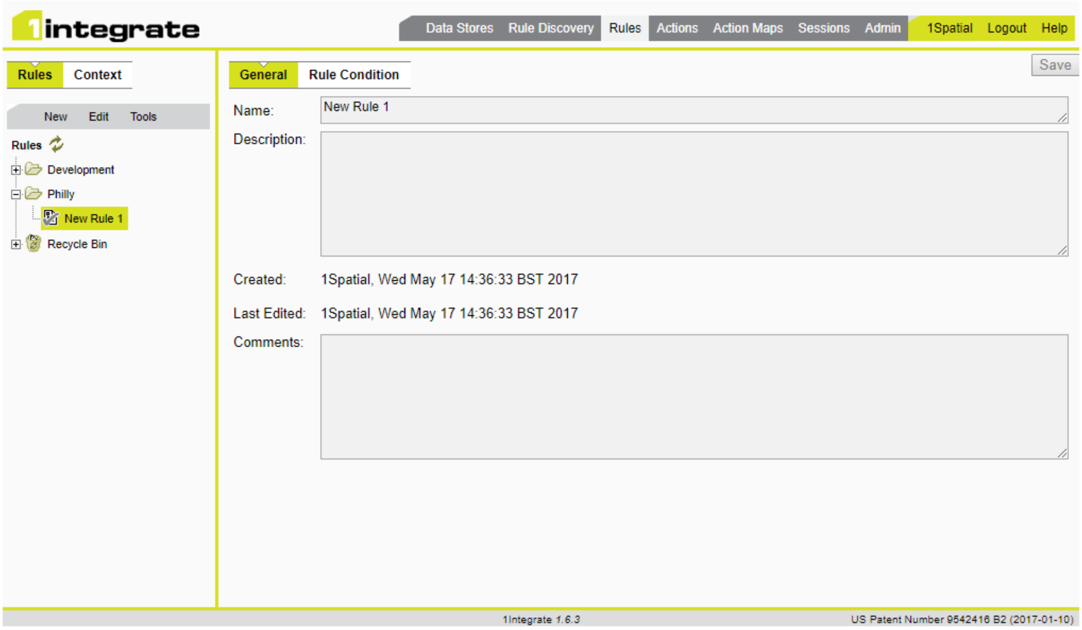


Figure 8-7: *A new rule is created*

3. In **Name** type `Building FC check`.
4. Click **Save**.

Define the Rule Structure

Set up a class on the root node

Note: If **Context** > **Use schema** is selected on the navigation pane, you can only select from classes and properties defined in the current schema. Rules are independent of an external schema and can be reused with different data sources by resetting the context.

1. Select the root node:

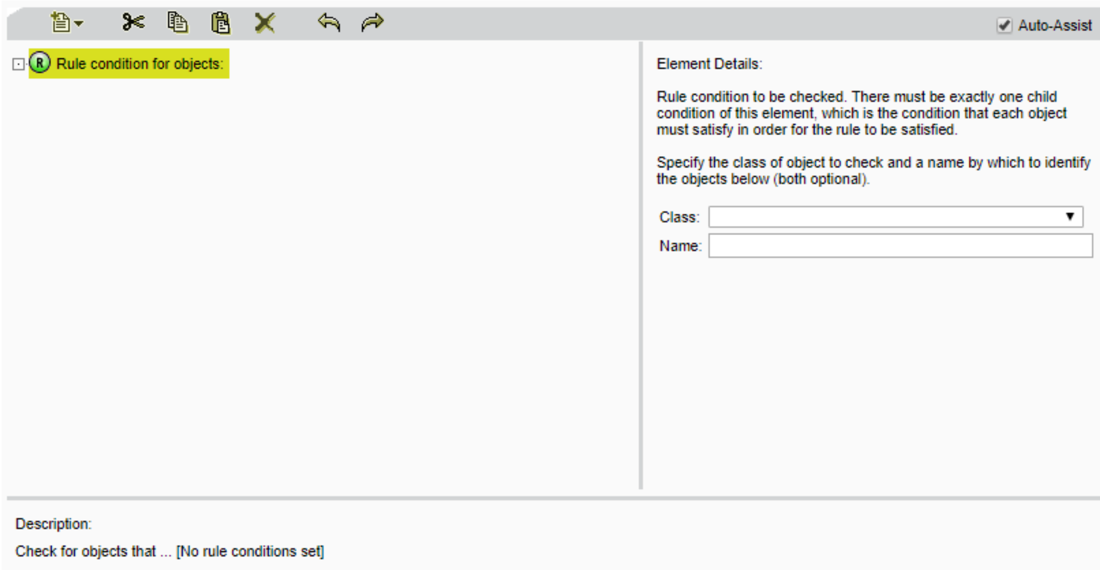


Figure 8-8: *Setting a class on the root node*

2. In **Class** select **BUILDING**.

Set up a condition

1. Click .
2. Select **Condition** > **Comparison**.

Figure 8-9: *Condition > Comparison*

The syntax for a comparison condition is displayed, with placeholder values.

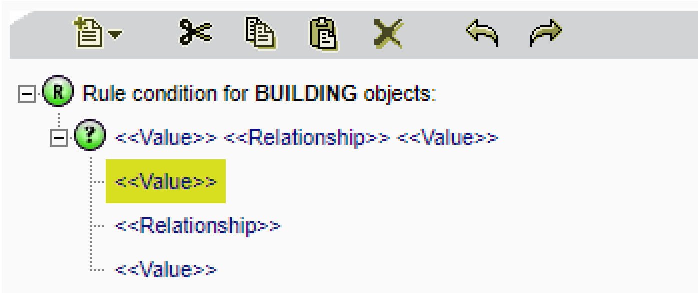


Figure 8-10: *Placeholders for a comparison condition*

Set up the comparison

Set up a dynamic value

1. In **Type** select **Dynamic Value**.

   The rule builder displays error messages for the unfilled values.

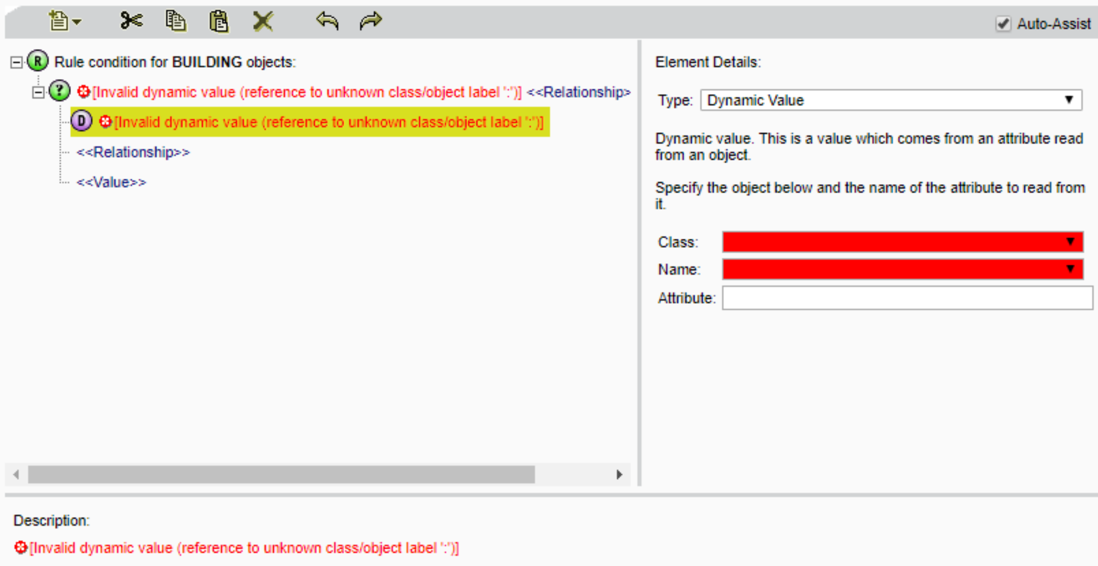


Figure 8-11: *Incomplete dynamic value*

2. In **Class** select **BUILDING**.
3. In **Attribute** select **FCODE**.

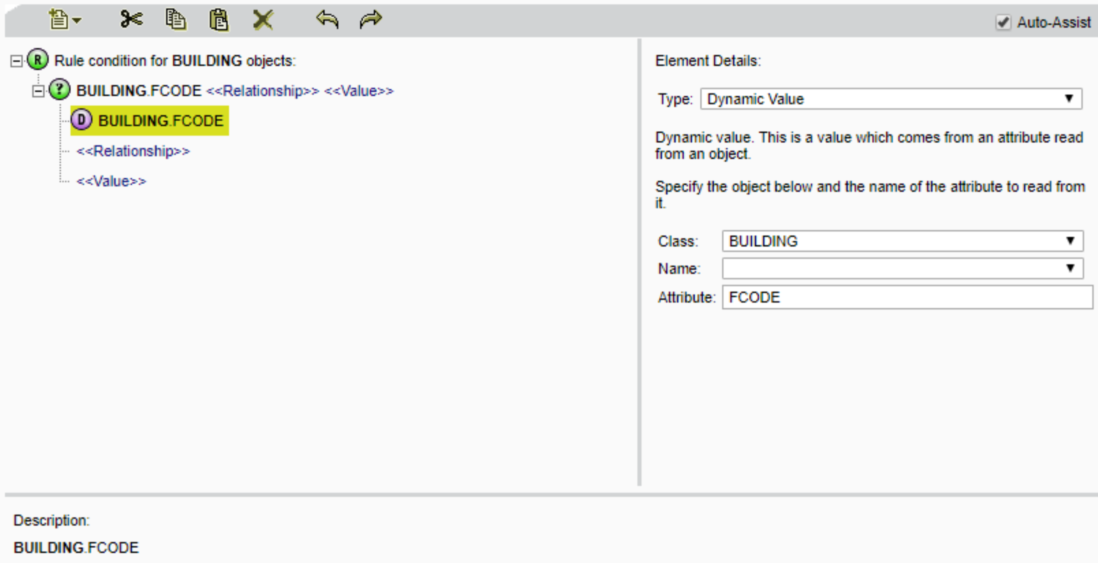


Figure 8-12: *BUILDING.FCODE is set as the first node.*

Set up a relationship

1. Select the **<<Relationship>>** placeholder.

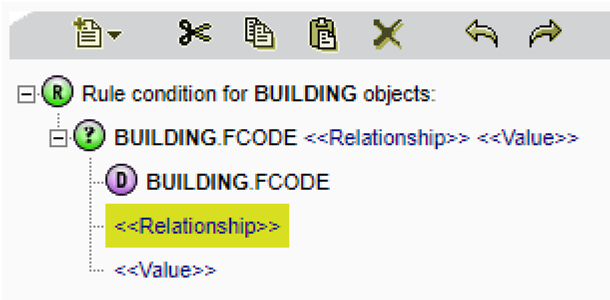


Figure 8-13: *<<Relationship>> placeholder*

2. In **Type** select **Scalar Relationship**.

   Another Type list is displayed.

3. In the lower **Type** list select **Equal**.

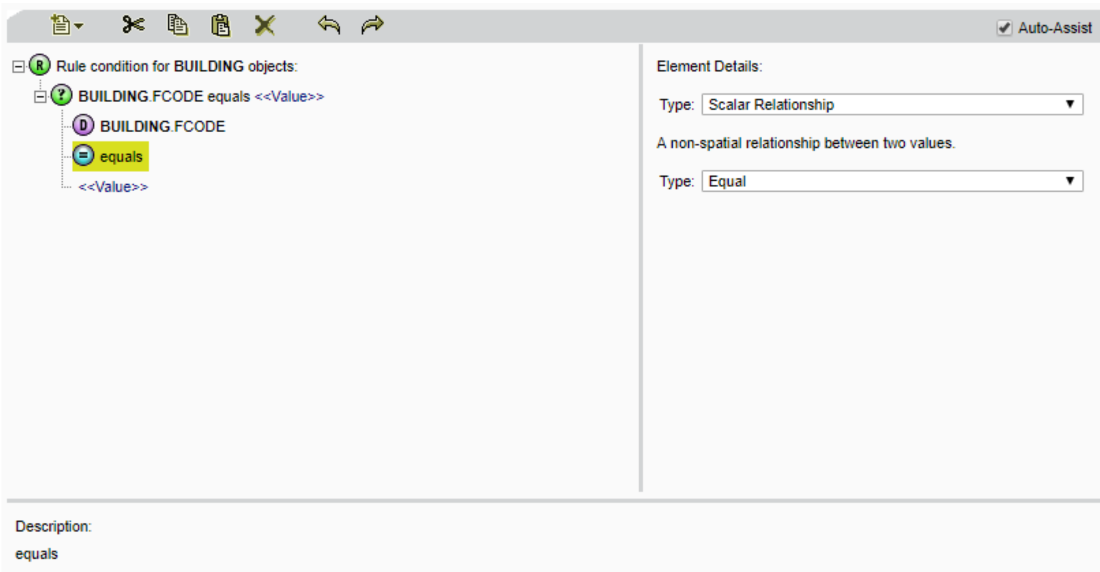
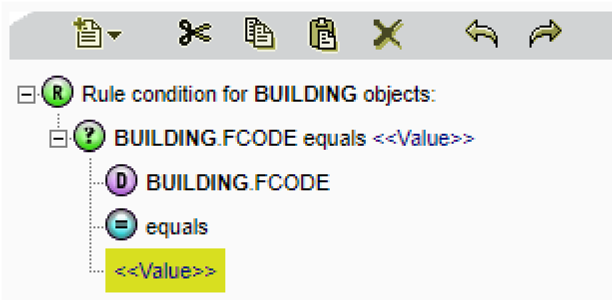


Figure 8-14: *Equal is set as the second node*

Set up a static value

1. Select the **<<Value>>** placeholder.



2. In **Type** select **Static Value**.

   Datatype and Value are displayed.

3. In **Datatype** select **Integer**.
4. In **Value** type `1810`.

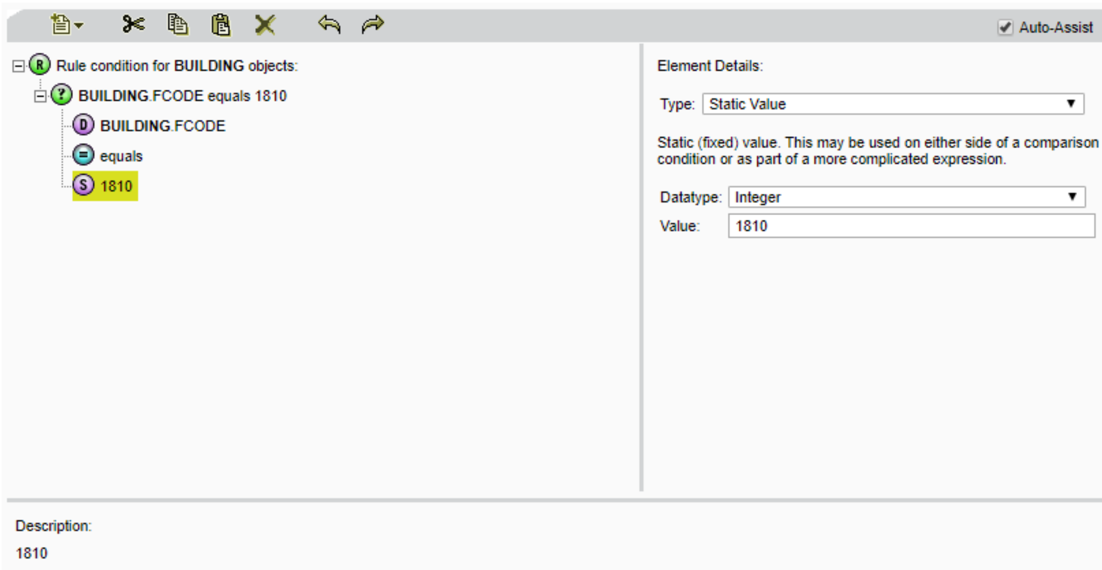


Figure 8-15: *1810 is set as the third node and the rule is now completed*

The rule now matches the description we were expecting in "Worked Example: Building FC Check" on page 44.

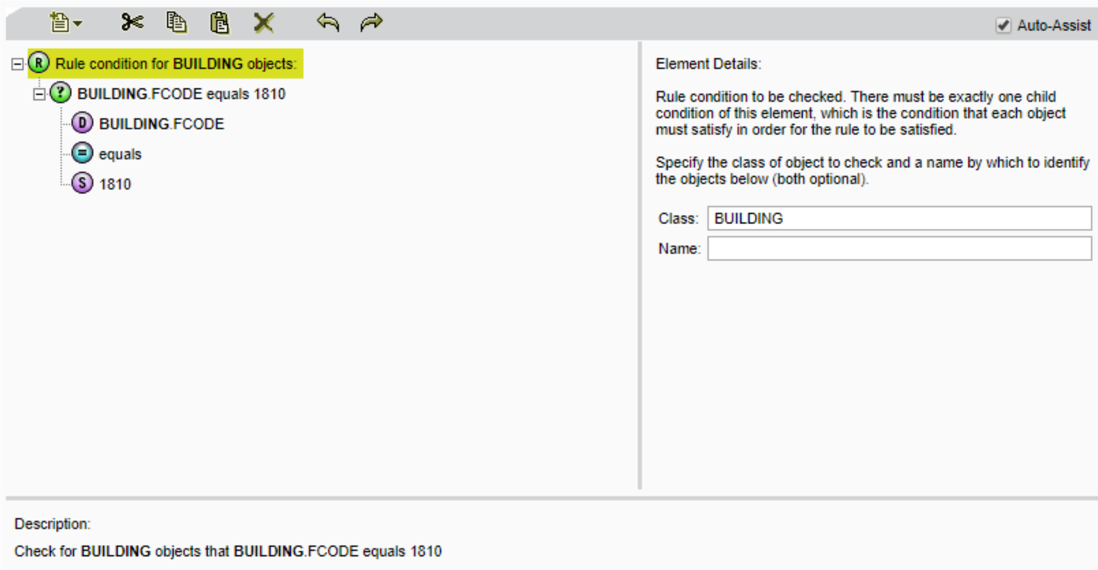


Figure 8-16: *Completed rule*

Run the rule check

Save the Rule

1. Click **Save**.

   The rule is saved.

Add a Check Rules task

1. In the **Sessions** interface select the **[your name]_PHILLY** session.
2. Click  and add a **Check Rules** task.
3. Navigate to your **Philly** rule folder
4. Select the **Building FC check** rule.
5. Click **OK**.
6. **Building FC check** is added to your session in a Check Rules task.
7. Add a **Pause** task to your session then run the session (see "Run a session" on page 18 for a reminder of how to do this).

Results

How many non-conforming objects are there?

Your results

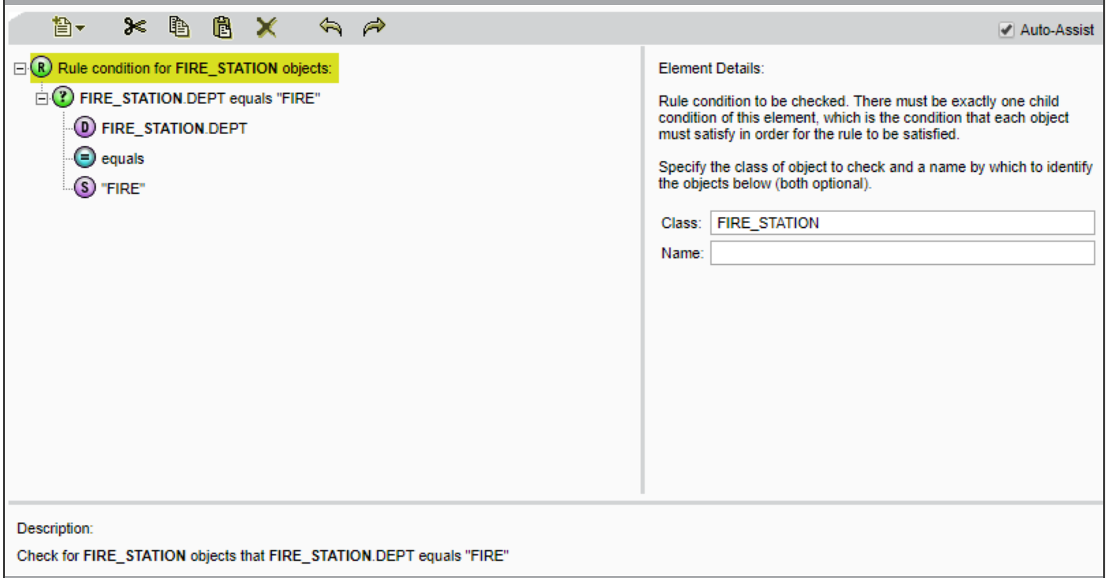
View the results of the Check Rules task in the HTML report.

Exercises

Exercise: Create a rule using a string attribute

In this exercise you create a rule in 1Integrate that performs a simple attribute check.

Create a new rule called `Firestation Department Check`. This rule will check for **DEPT** attributes on the **FIRE_STATION** class which are equal to **"FIRE"** (a string).



After you create the rule, create a session and run a Check Rules task.

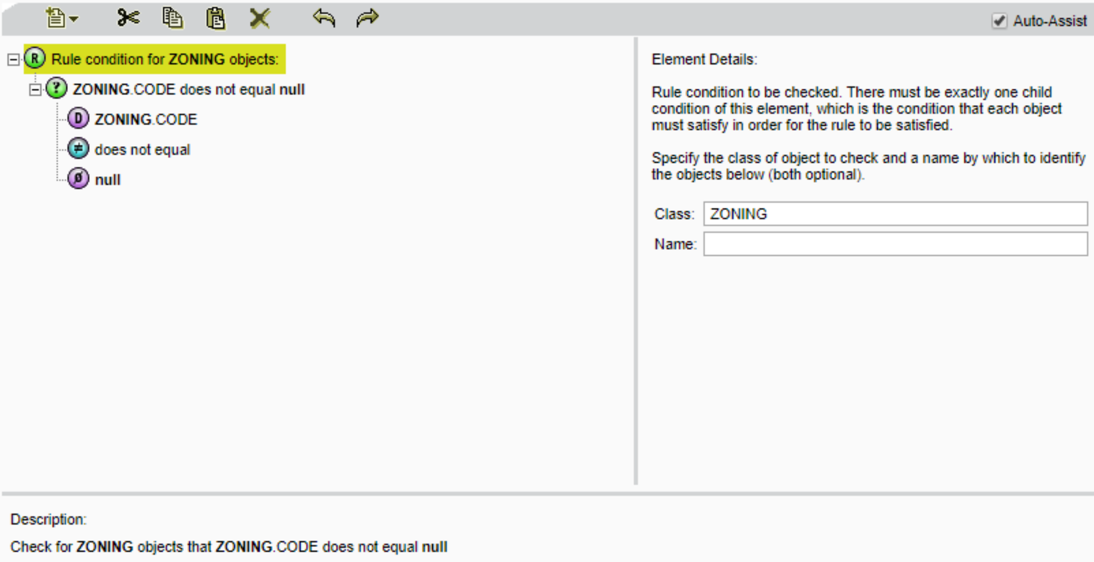
How many nonconforming objects are reported?

Your results

Exercise: Create a rule using a check for a null value

In this exercise you create a rule in 1Integrate that performs a simple check to ensure an attribute contains a value.

Create a new rule called `Zoning Check`. This rule will check for **CODE** attributes on the **ZONING** class that are **not null**.



After you create the rule, create a session and run a check rules task.

How many nonconforming objects are reported?

Your results

Summary

In this chapter we have built and run the first rules using a simple attribute check. We have used a comparison to check a dynamic value against a static value and then run the rule against the Philadelphia data.

Subsequent chapters will introduce you to more complex rule structures.

9 Rule on two Classes

This chapter provides instructions to create a rule in 1Integrate that checks the spatial relationship between objects. **BUILDING** objects should not overlap **CURBLINE** objects. The buildings should be inside the curblines.

In order to understand this rule we introduce the geometry attribute, and explain the concept of point, line and area geometries.

Objectives

By the end of this chapter you will be able to:

- Use the rule builder to create a rule on classes with the use of an existence clause
- Begin to understand and use the spatial operators within a rule
- Understand the difference between point, line and area geometries

Key Concepts

Point geometry

A point is a 0-dimensional geometry that has an interior and an exterior, but no boundary.

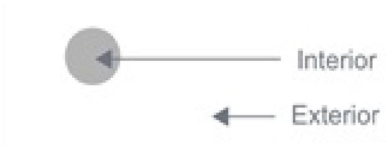


Figure 9-1: *A point geometry*

Line geometry

A line is a one dimensional geometry that has an exterior, an interior, and a boundary.

As the boundary is at the end points of a line, if the line is a closed ring there is no boundary.

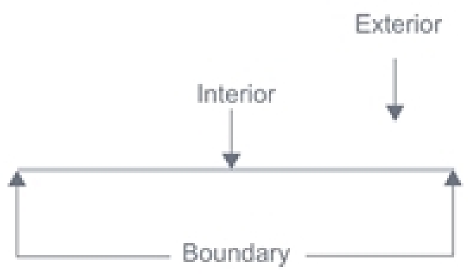


Figure 9-2: *A line geometry*

Area geometry

An area is a two dimensional geometry that has an exterior, an interior, and a boundary.

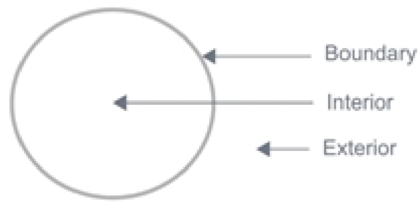


Figure 9-3: *An area geometry*

Worked Example: CURBLINE 1.0

The rule that will be created in this exercise will be:

`Check for CURBLINE objects that there are no BUILDING objects for which BUILDING.geometry overlaps CURBLINE.geometry`

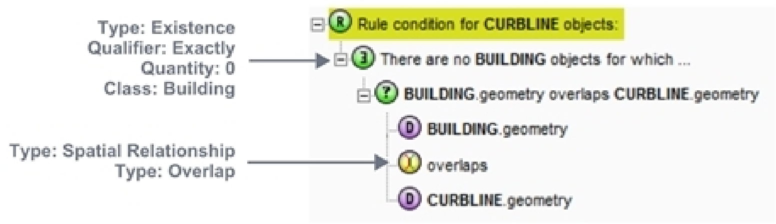


Figure 9-4: *A rule checking for CURBLINE objects overlapping BUILDING objects*

Preparation

1. Select **Auto-assist** (see "Auto-assist" on page 34).
2. Select **Use schema** and the **Philly** schema (see "Context" on page 35).
3. Create a new rule with the name `Curbline 1.0`.

Define the Rule Structure

The Existence condition checks how many objects satisfy a set of conditions. The conditions are set up on two classes, a parent node and the node of the Existence condition and you can specify *At least*, *At most* or *Exactly* a number of objects.

To check for the absence of objects, we need to look for exactly 0 objects.

1. Set up the CURBLINE class on the root node (see "Set up a class on the root node" on page 45).
2. Insert an existence condition using the following values:

Element details	Description
Type	Existence
Qualifier	Exactly
Quantity	0
Class	`Building`

3. Select the **<<Condition>>** placeholder and create a **Comparison** condition for this node (see "Set up a condition" on page 47).

Set up the comparison

Spatial relationship

A spatial relationship operator is applied to the boundaries and interiors of two geometries.

The two spatial relationships are:

- Disjoint: check that neither the boundaries nor the interiors of two geometries intersect
- Intersect: check that either the boundaries or interiors of two geometries intersect.

  There are eight types of intersect (detailed in "Relationships" on page 132):

  - Equal
  - Touch
  - Overlap
  - Cross
  - Within
  - Contains
  - Covers
  - Covered by

Select the upper **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`BUILDING`
Attribute	`geometry`

Select the **<<Relationship>>** placeholder and create a **Spatial Relationship** using the following values:

Element details	Description
Type	Spatial Relationship
Type	Overlap

Select the lower **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic value
Class	`CURBLINE`
Attribute	`geometry`

The rule now matches the text in "Worked Example: CURBLINE 1.0" on page 57:

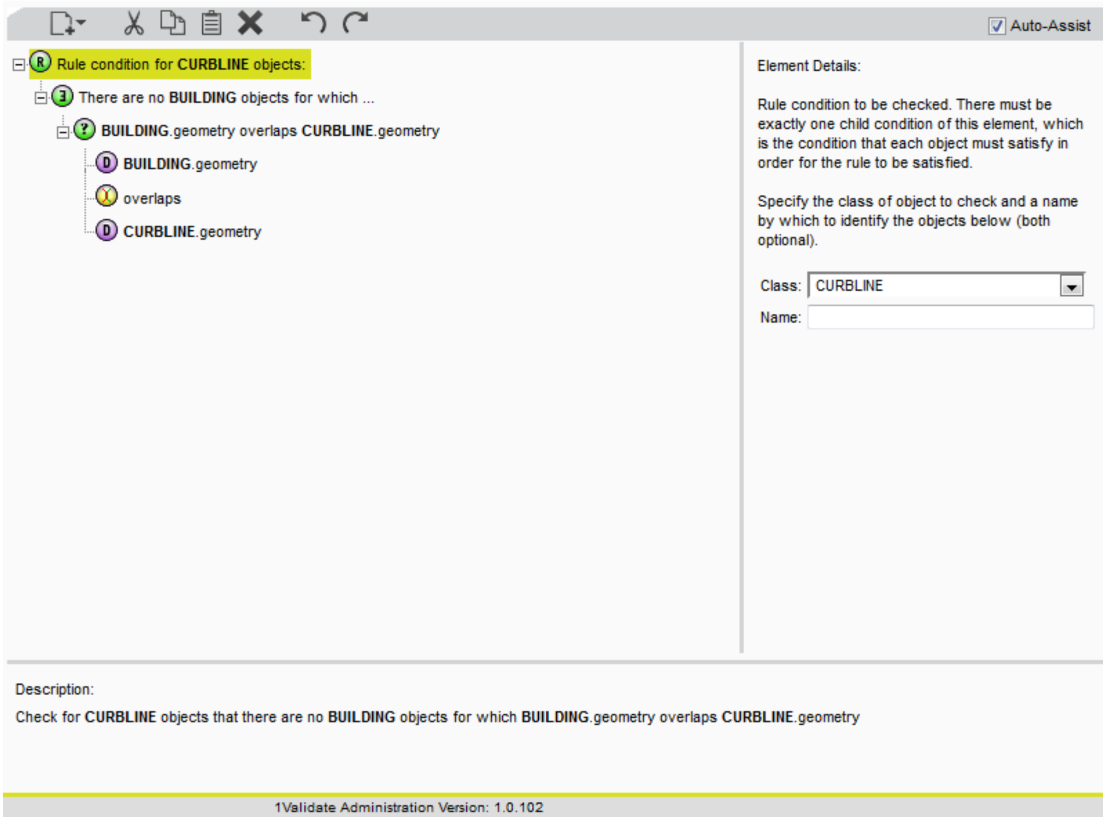


Figure 9-5: *Completed rule*

Run the rule check

1. Save the rule.
2. Add a **Check Rules** and a **Pause** task to your session.
3. Run the session.

Results

How many nonconforming objects are there?

Your results

The CURBLINE 1.0 rule checks that curblines do not overlap buildings and have a number of nonconforming objects.

To observe how the rule appears on a map, click View in the row with ID 19850 from the Rule Conformance HTML Results page. The cache viewer shows all data read via the open data tasks into a session with a single partition where the interface and session queues are pointing at the same cache dataroot. In addition to the on-screen tools you can pan (left-click and drag) and zoom to area (hold Shift and select an area) on the map.

This failure is located near the address 3300 Grays Ferry Ave, Philadelphia. The curbline object with the ID 19850 is along Schuylkill Expressway.

Figure 9-6: *Nonconforming object 19850*

Note: You should check the `ELEV_1` code when checking the data.

Exercises

Exercise: STREETCENT objects

`STREETCENT` objects should not cross `BUILDING` objects.

How many do?

Your results

Summary

In this chapter we saw how you can run a check against objects in two classes. You also set up a spatial relationship between two object geometries.

The rules in this chapter and the previous chapter are the basic structures from which you will now learn to create more complex rules.

10 Rule Refinement

The exercise in this chapter builds on the previous chapter by adding an attribute check to the spatial relationship.

In the previous exercise, it was found that valid overlaps exist where the curbline is part of the freeway, which is elevated. This allows buildings and curblines to overlap.

In this exercise we will add a comparison clause to only check curbline objects where the `ELEV_1` code is set to 0 (no elevation).

Objectives

By the end of this chapter you will be able to:

- Refine rules by adding additional clauses to an existing rule
- Understand how multiple clauses can be used to create more complex and focused rules

Key Concepts

Clause

A rule can contain a number of clauses beneath the root node.

A clause uses a *comparison*, the results of which feed back into the main rule.

The following logical operators create clauses:

- AND
- OR
- IF...THEN...ELSE
- Existence
- For All
- Chain
- XOR
- Existence in collection
- For All in collection

Worked Example: CURBLINE 2.0

The rule created in this exercise is:

```
Check for CURBLINE objects that there are no BUILDING
objects for which BUILDING.geometry overlaps
CURBLINE.geometry and CURBLINE.ELEV_1 equals 0
```

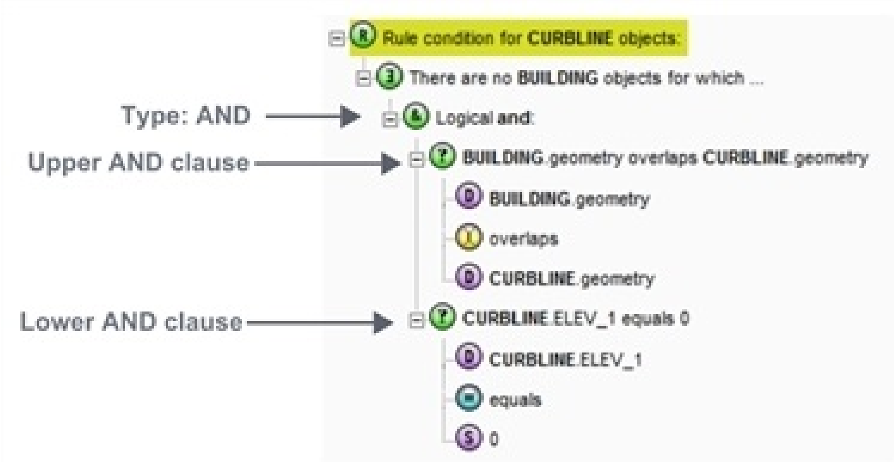


Figure 10-1: *A rule checking for CURBLINE objects overlapping BUILDING objects and the CURBLINE has no elevation*

Preparation

1. Select **Auto-assist** (see "Auto-assist" on page 34).
2. Select **Use schema** and the **Philly** schema (see "Context" on page 35).

Refining an existing rule

Copy the `Curbline 1.0` rule and rename the copy to `Curbline 2.0`.

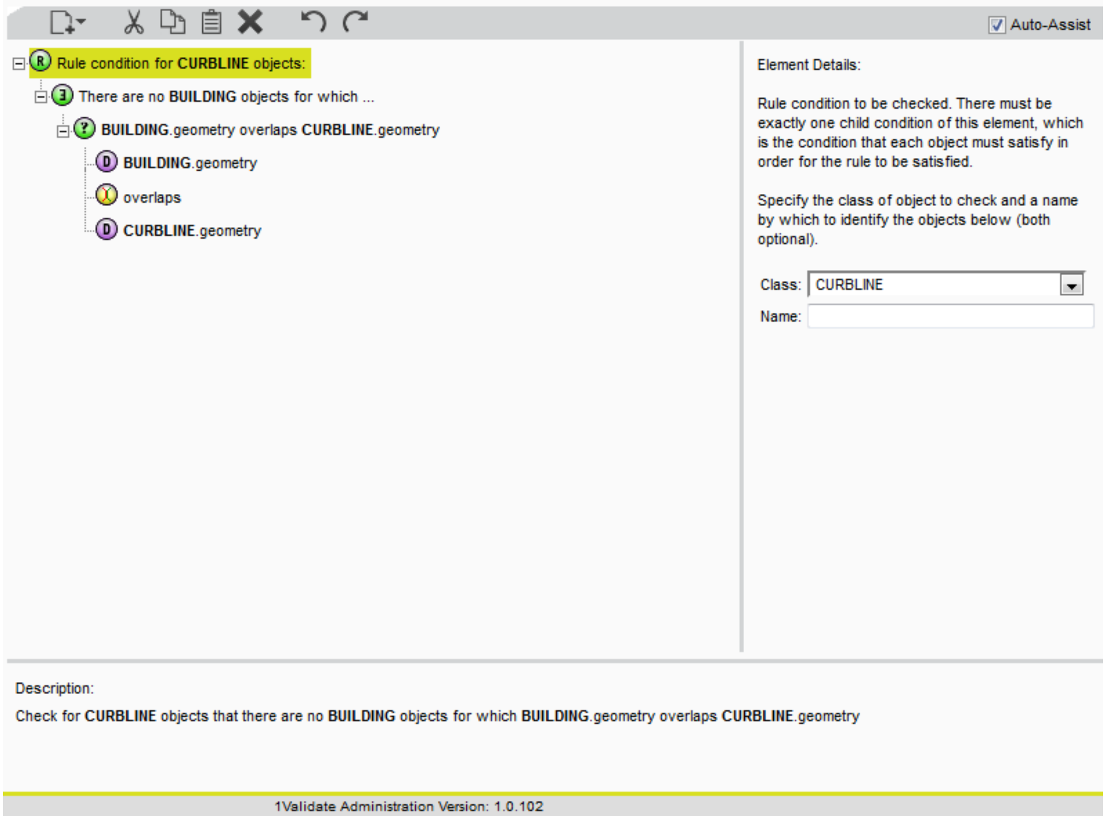


Figure 10-2: *Copied rule*

Create an AND condition

1. Select the Comparison node in the rule (BUILDING.geometry overlaps CURBLINE.geometry).
2. Click  then select **Condition** > **Comparison**.

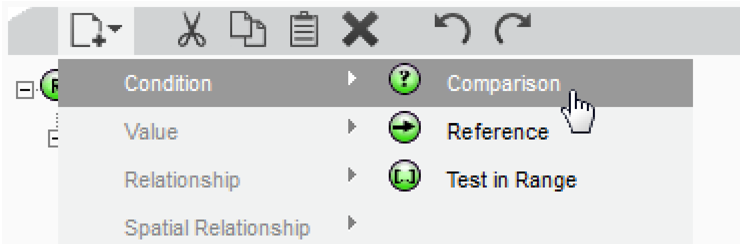


Figure 10-3: *Add a comparison*

A **Logical AND** is inserted above the existing Comparison node and a *second* Comparison is inserted below it (but at the same level).

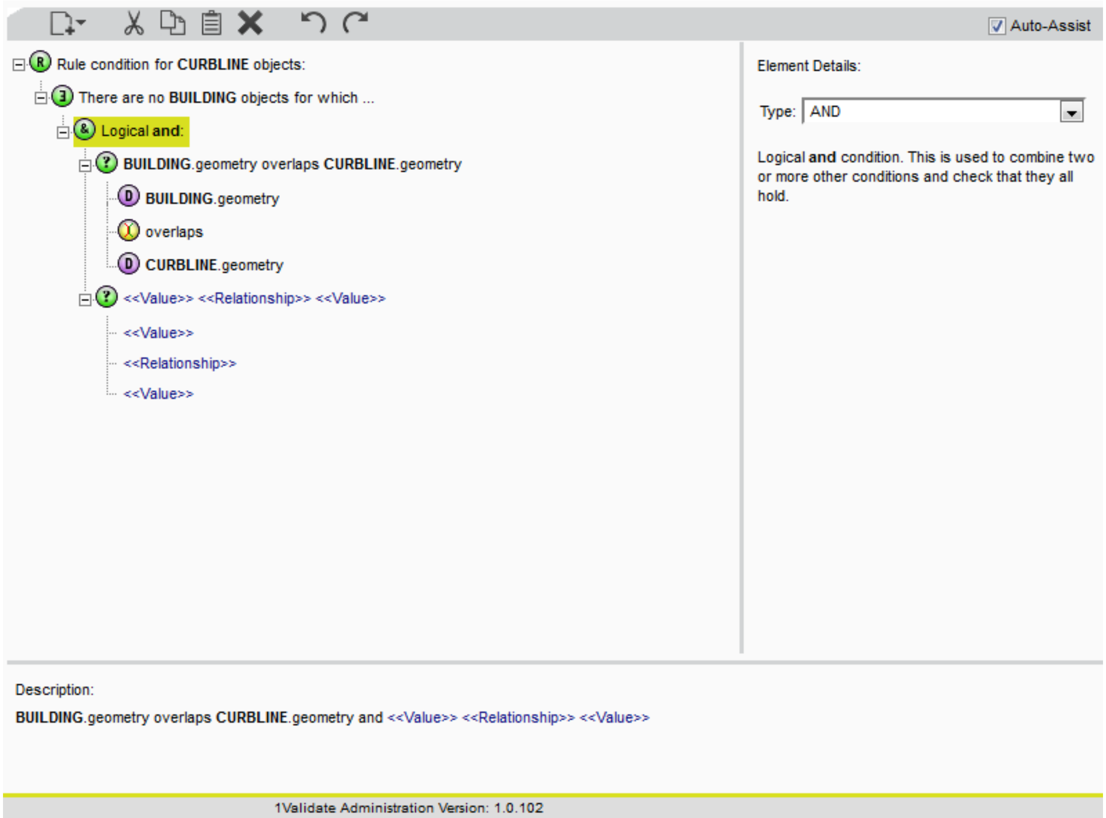


Figure 10-4: *Logical AND added to the rule*

3. Select the upper **<<Value>>** and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`CURBLINE`
Attribute	`ELEV_1`

4. Select **<<Relationship>>** and create a **scalar relationship** using the following values:

Element details	Description
Type	Scalar Relationship
Type	Equal

5. Select the lower **<<Value>>** and create a **Static Value** using the following values:

Element details	Description
Type	Static Value
Datatype	Integer
Value	0

The rule now matches the text in "Worked Example: CURBLINE 2.0" on page 62.

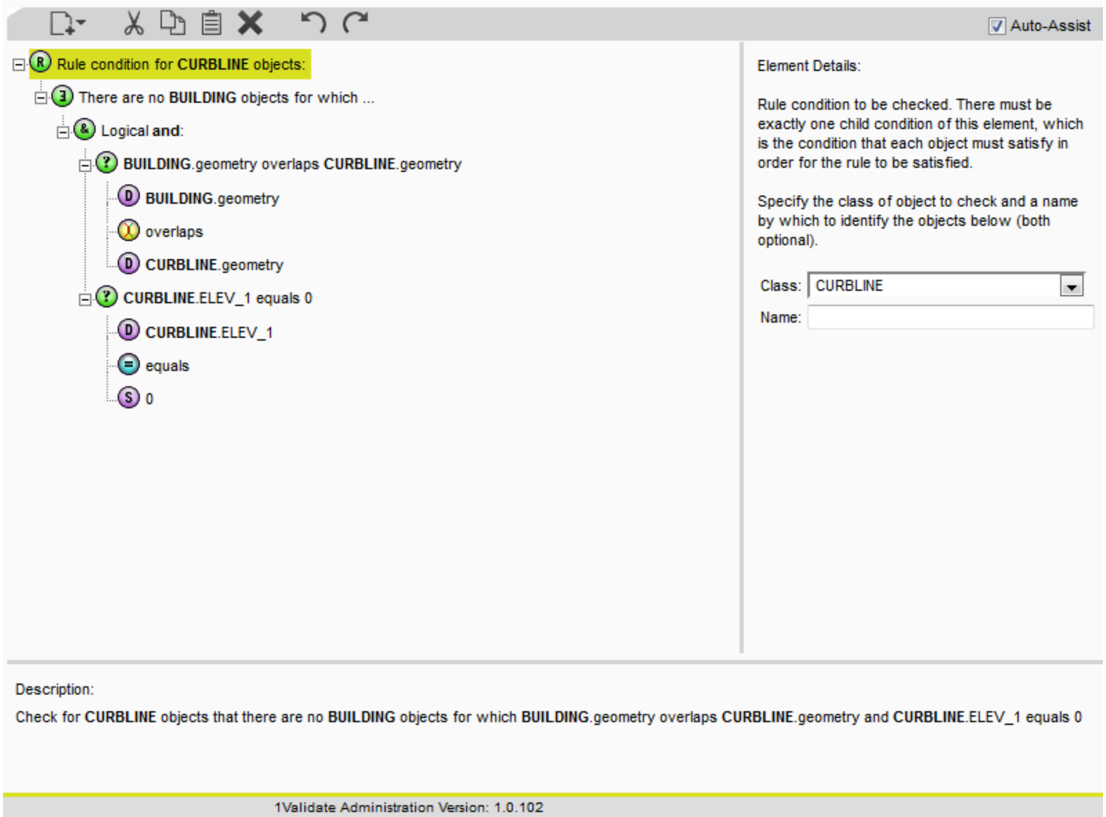


Figure 10-5: *Completed rule*

Run the rule check

1. Save the rule.
2. Add a **Check Rules** and a **Pause** task to your session.
3. Run the session.

Results

The check should complete successfully. View the results in the HTML report and note how many non-conforming objects there are.

Check that the CURBLINE object with an ID of 19850 is no longer being flagged as a non-conforming object.

Exercises

Exercise: Curbline 3.0

Go back to the Curbline 2.0 rule created earlier. Create a copy of this rule called `Curbline 3.0`. Insert an additional condition so that the rule also checks that the `BUILDING.FCODE` is `1810`. Run this rule.

How many non-conforming objects are there this time?

Your results

How do these results differ from the results from `Curbline 2.0`?

Can you see why the two sets of results are different?

Your results

Summary

In this chapter we saw how to refine a rule by adding a new clause and you were introduced to the AND condition.

In the next chapter you will learn to use an IF... THEN...ELSE structure.

11 IF...THEN...ELSE

In this chapter an IF…THEN...ELSE rule is created based on rules that have already been written.

In "Simple Attribute Check" on page 42 it was seen that many of the `BUILDING` objects have an `FCODE` of 1810. "Rule on two Classes" on page 55 and "Rule Refinement" on page 62 then checked to see if the `BUILDING` geometries overlapped the `CURBLINES` where the `CURBLINE.ELEV_1` attribute was 0. All of these conditions will be used in this chapter.

Objectives

By the end of this chapter you will be able to:

- Understand how the IF...THEN condition works (and the optional ELSE clause)
- Appreciate that rules can be written in different ways using the various conditions available in the rule builder

Key Concepts

IF...THEN...ELSE

The IF...THEN...ELSE condition 

creates a rule structure of two or three clauses. The third clause (ELSE) is optional.

The following diagrams show the flow between the clauses according to the values returned by each test.

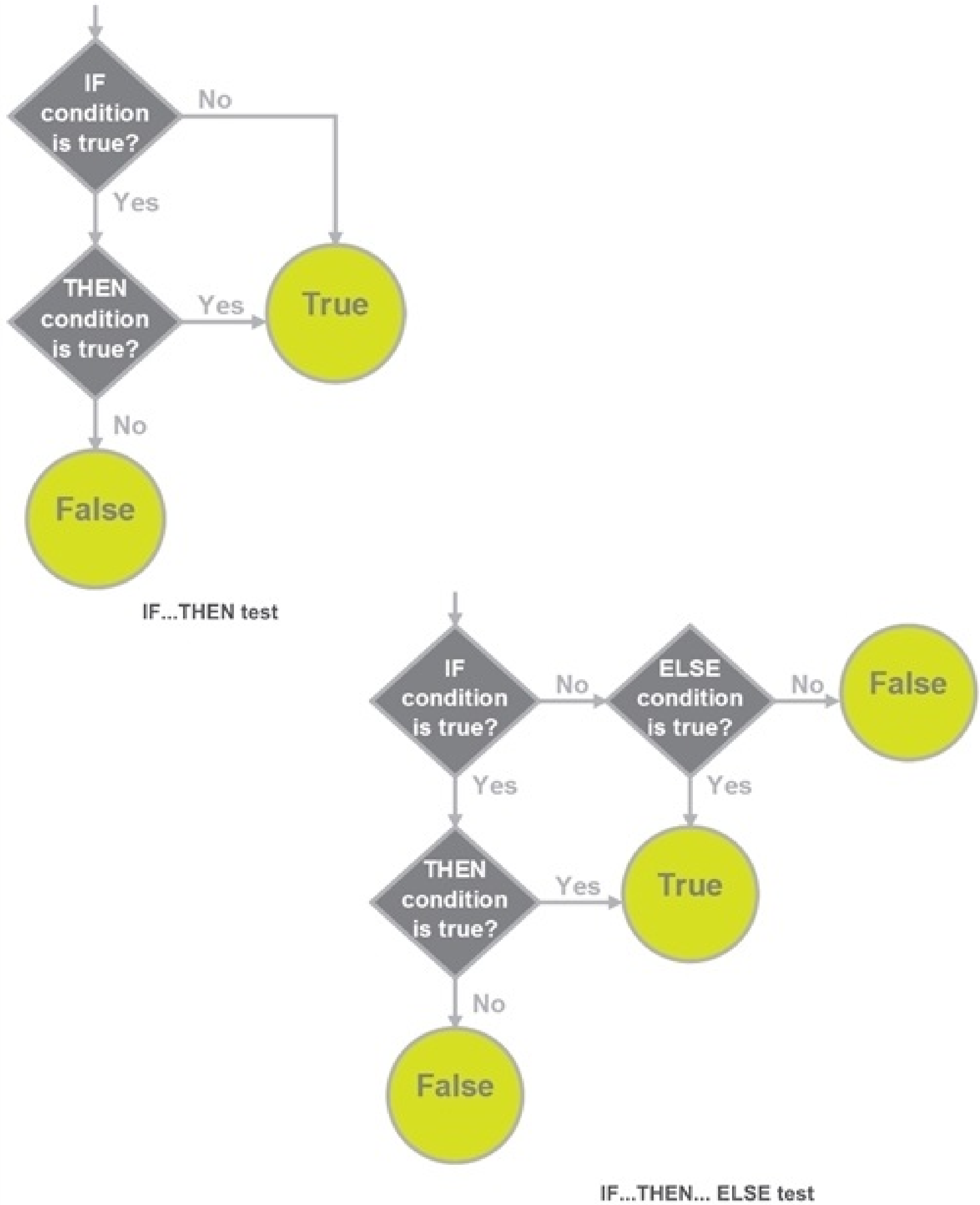


Figure 11-1: *IF...THEN and IF...THEN...ELSE tests*

Worked Example: IF THEN

The rule that will be created in this chapter will be:

`Check for BUILDING objects that if BUILDING.FCODE equals 1810 then there are no CURBLINE objects for which BUILDING.geometry overlaps CURBLINE.geometry and CURBLINE.ELEV_1 equals 0`

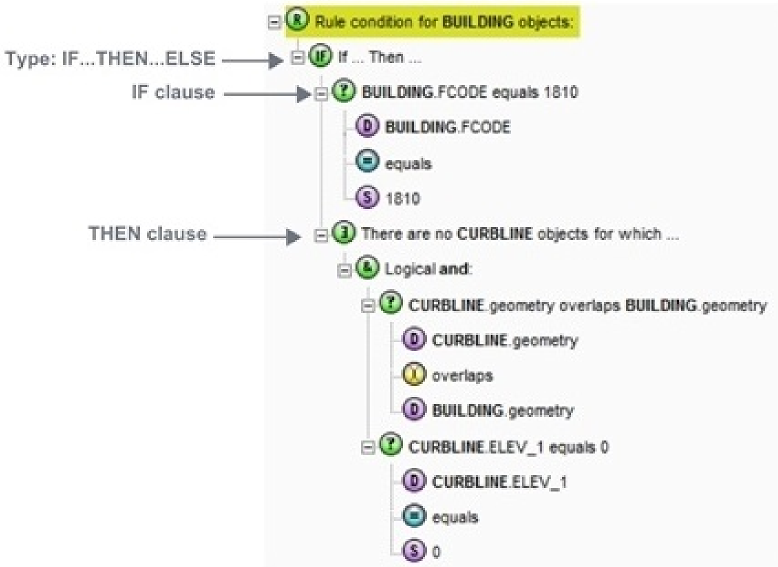


Figure 11-2: *A rule with an IF...THEN...ELSE condition*

Preparation

1. Select **Auto-assist** (see "Auto-assist" on page 34).
2. Select **Use schema** and the **Philly** schema (see "Context" on page 35).
3. Create a new rule with the name `IF_THEN check`.

Define the Rule Structure

1. Set up the BUILDING class on the root node.
2. Insert an **IF...THEN** condition.

Element details	Description
Type	IF...THEN...ELSE

Define the first clause in the IF…THEN condition

1. Insert a comparison on the upper **<<Condition>>** placeholder using the following values:

Element details	Description
Type	Comparison

2. Select the upper **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`BUILDING`
Attribute	`FCODE`

3. Select the **<<Relationship>>** placeholder and create a **Scalar Relationship** using the following values:

Element details	Description
Type	Scalar Relationship
Type	Equal

4. Select the lower **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Static Value
Datatype	Integer
Value	1810

Define the second clause in the IF…THEN condition

1. Insert an **Existence** condition on the remaining **<<Condition>>** placeholder using the following values:

Element details	Description
Type	Existence

Element details	Description
Qualifier	Exactly
Quantity	0
Class	`CURBLINE`

2. Insert a **Comparison** condition on the remaining **<<Condition>>** placeholder using the following values:

Element details	Description
Type	Comparison

3. Select the upper **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`BUILDING`
Attribute	`geometry`

4. Select the **<<Relationship>>** placeholder and create a **Spatial Relationship** using the following values:

Element details	Description
Type	Spatial Relationship
Type	Overlap

5. Select the lower **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`CURBLINE`
Attribute	`geometry`

Create a subclause within the second clause

1. Select the lower **Comparison** node and create an **AND** clause (see "Create an AND condition" on page 64).
2. Select the upper **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`CURBLINE`
Attribute	`ELEV_1`

3. Select the **<<Relationship>>** placeholder and create a **Scalar Relationship** using the following values:

Element details	Description
Type	Scalar Relationship
Type	Equal

4. Select the lower **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Static Value
Datatype	Integer
Value	`0`

The rule now matches the text in "Worked Example: IF THEN" on page 70.

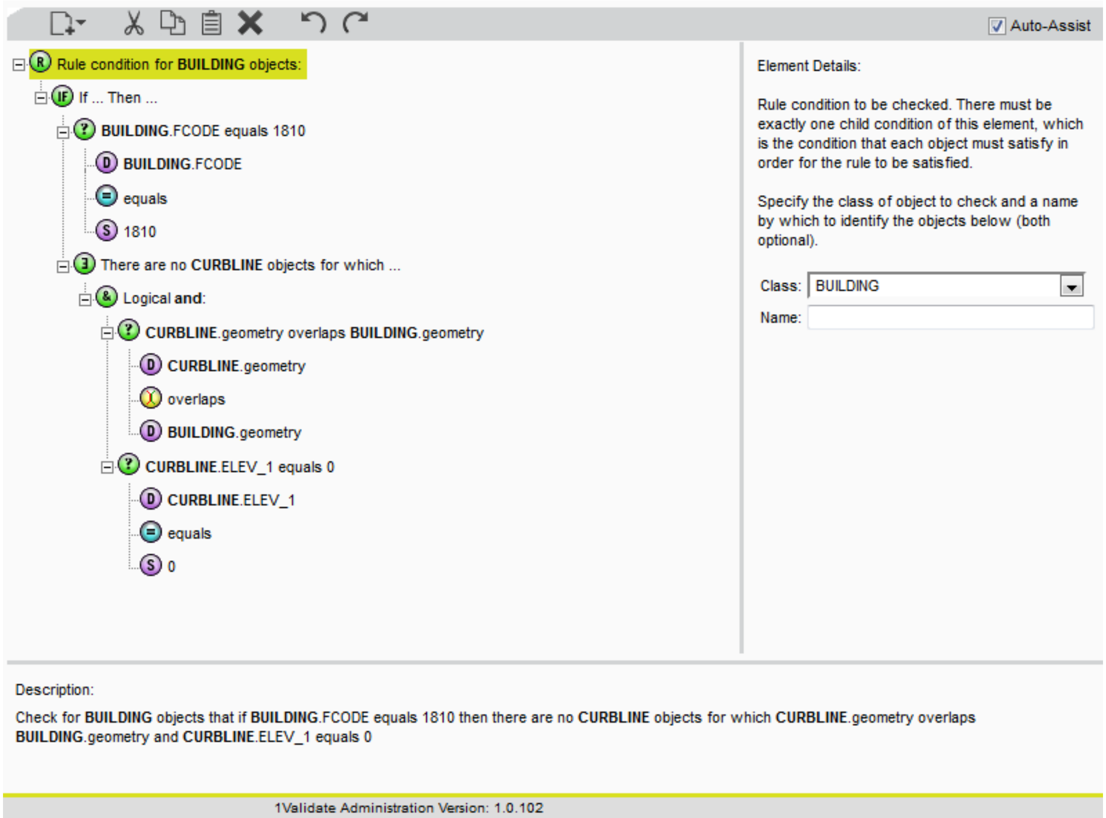


Figure 11-3: *Completed rule*

Run the rule check

1. Save the rule.
2. Add a **Check Rules** and a **Pause** task to your session.
3. Run the session.

Results

How many nonconforming objects are there?

Your results

Summary

This chapter has shown how to combine conditions set up in previous rules, using an IF...THEN structure.

12 Multiple Existence Conditions

This chapter shows how a rule can be built that uses more than two classes. In this chapter this is done by using multiple *Existence* conditions.

The scenario is that in the Philadelphia data we have FIRE_STATION points that should be located inside a building polygon. These FIRE_STATION points have detailed attribution, including an ADDRESS field giving the name of the street it is located on.

As the data store also contains STREETCENT objects, we can cross-check the FIRE_STATION.ADDRESS field with the STREETCENT.ST_NAME attribute to ensure the FIRE_STATION is in the correct location.

In summary, we can check the FIRE_STATION point is inside a building, and that the FIRE_STATION is within a certain distance of a STREETCENT object where the FIRE_STATION.ADDRESS field contains the STREETCENT.ST_NAME.

Objectives

By the end of this chapter you will be able to:

- Understand that rules that involve more than two classes can be created using multiple conditions
- Understand that complex rules can be developed that involve both spatial and attribute based conditions

Key Concepts

Contains

A geometry contains another geometry when one object’s boundary and interior are inside another.

> **Note:** Contains is the reciprocal relationship to Within (see "Relationships" on page 132).

The types of geometries which can contain another are:

- Area-to-area
- Area-to-line
- Area-to-point
- Line-to-line
- Point-to-point

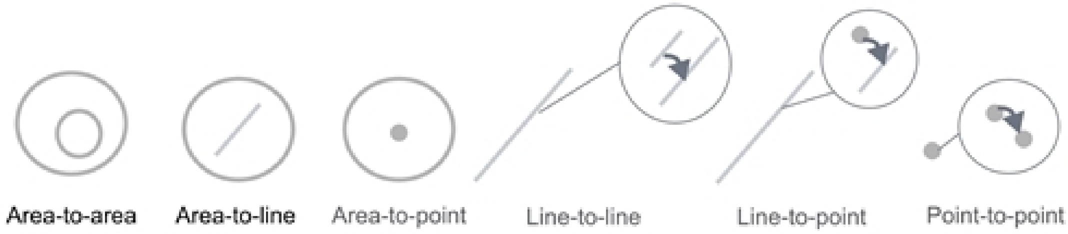


Figure 12-1: *Geometries that can contain other geometries*

Within Distance

Within Distance is a spatial relationship. The relationship checks that at the closest point, two geometries are within a specified in dataset units.

> **Note:** Dataset units are decimal degrees for latitude and longitude.

Worked Example: Firestation 1.0

The final rule will be:

```
Check for FIRE_STATION objects that (there is at least
1 BUILDING object for which BUILDING.geometry contains
FIRE_STATION.geometry) and (there is at least 1
STREETCENT object for which STREETCENT.geometry is
within a distance of 200 of FIRE_STATION.geometry and
FIRE_STATION.ADDRESS contains STREETCENT.ST_NAME)
```

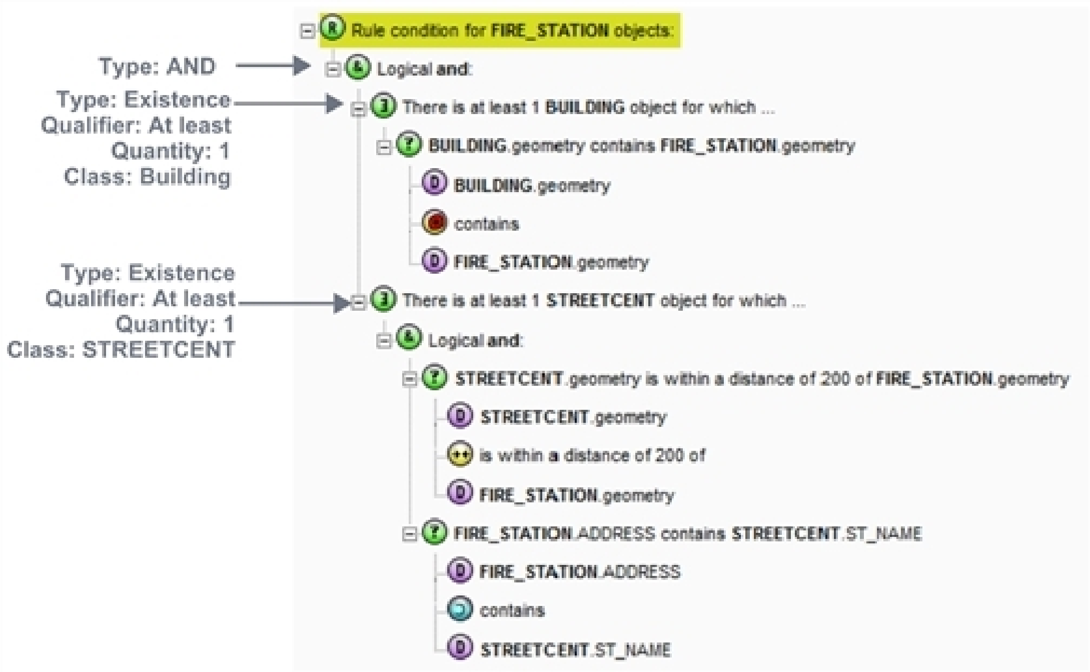


Figure 12-2: *Rule testing that FIRE_STATION point is inside a building, and that the FIRE_STATION is within a certain distance of the STREETCENT object referenced by the FIRE_STATION.ADDRESS attribute*

Preparation

1. Select **Auto-assist** (see "Auto-assist" on page 34).
2. Select **Use schema** and the **Philly** schema (see "Context" on page 35).
3. Create a new rule with the name `Firestation 1.0`.

Define the Rule Structure

1. Set up the FIRE_STATION class on the root node.
2. Create an **Existence** condition for the root node using the following values:

Element details	Description
Type	Existence
Qualifier	At least
Quantity	1
Class	`Building`

Define the Comparison

1. Select the **<<Condition>>** placeholder, and in **Type** select **Comparison**.

   The **<<Condition>>** placeholder is replaced by a **Comparison** node.

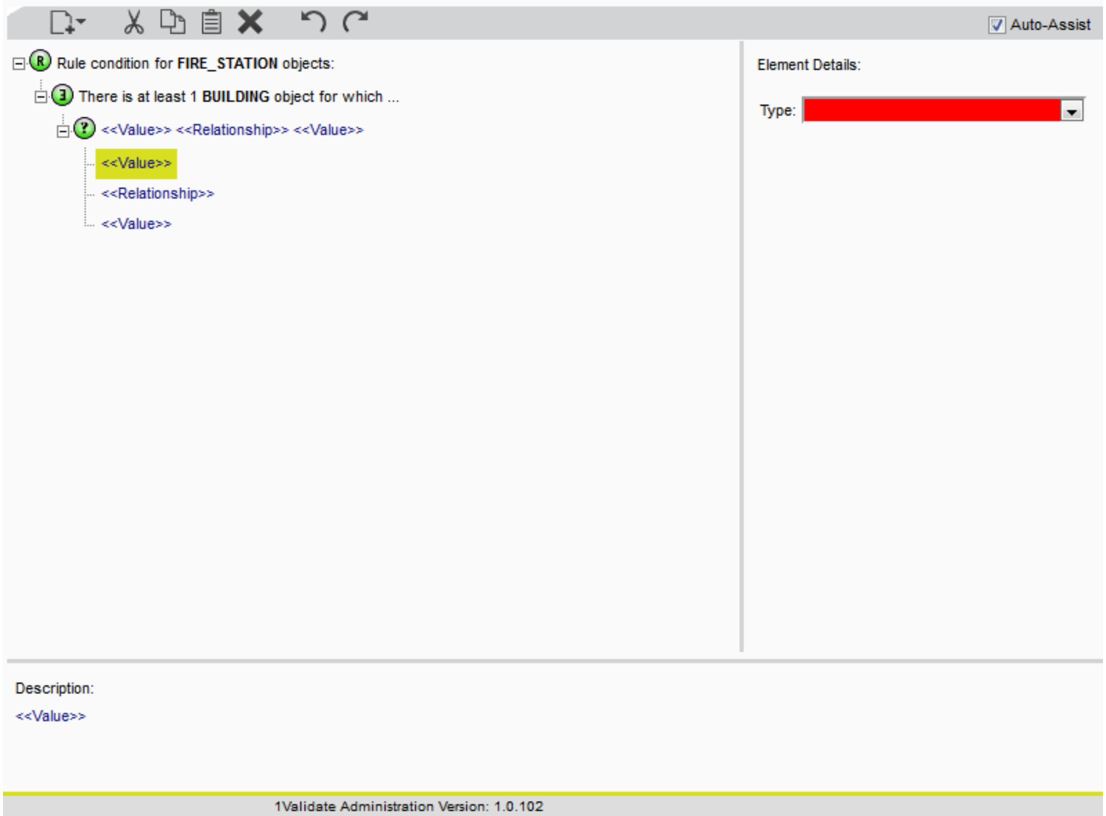


Figure 12-3: *Comparison node*

2. Select the upper **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	BUILDING
Attribute	geometry

Select the **<<Relationship>>** placeholder and create a **Spatial Relationship** using the following values:

Element details	Description
Type	Spatial Relationship
Type	Contains

Select the lower **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	FIRE_STATION
Attribute	geometry

Define a new clause

1. Select the root node and create a new **Existence** condition using the following values:

Element details	Description
Type	Existence
Qualifier	At least
Quantity	1
Class	STREETCENT

2. Select the **<<Condition>>** placeholder, and in **Type** select **Comparison**.

3. Select the upper **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`STREETCENT`
Attribute	`geometry`

Select the **<<Relationship>>** placeholder and create a **Spatial Relationship** using the following values:

Element details	Description
Type	Spatial Relationship
Type	Within Distance
Distance	200

Select the lower **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`FIRE_STATION`
Attribute	`geometry`

Define the third clause

1. Select the lower **Existence** node and create a new **Comparison** clause.
2. Select the upper **<<Value>>** and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`FIRE_STATION`
Attribute	`ADDRESS`

Select **<<Relationship>>** and create a **Scalar Relationship** using the following values:

Element details	Description
Type	Scalar Relationship
Type	Contains

Select the lower **<<Value>>** and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`STREETCENT`
Attribute	`ST_NAME`

The rule now matches the text in "Worked Example: Firestation 1.0" on page 77:

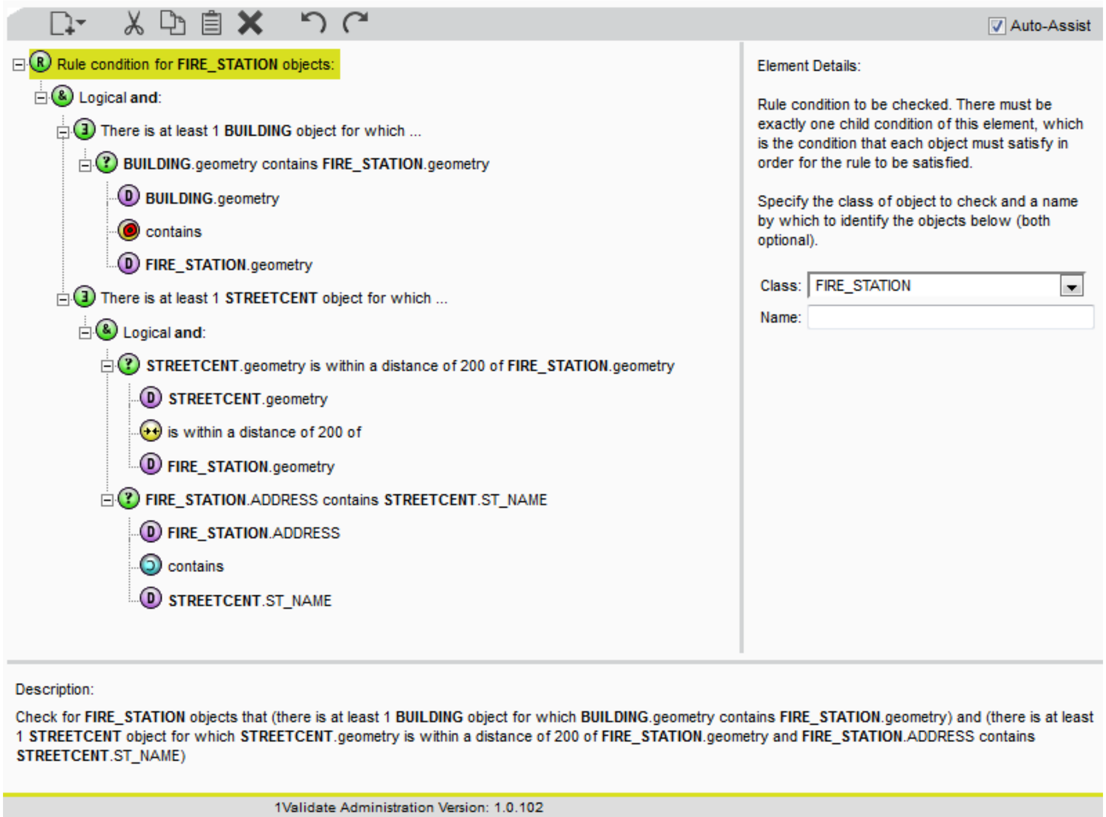


Figure 12-4: *Completed rule*

Run the rule check

1. Save the rule.
2. Add a **Check Rules** and a **Pause** task to your session.
3. Run the session.

Results

The check should complete successfully. View the results in the HTML report.

How many non-conforming objects are there?

Your results

Why are the objects non-conforming?

Your results

Summary

In this chapter we have seen how to set up a rule on two classes using multiple Existence conditions. We have also seen that rules can combine both spatial and attribute-based conditions.

13 For All Condition

In this chapter a check will be made using a For All condition to ensure that all FIRE_STATION geometries are located in the correct building. The BUILDING objects have a BUILDING_TYPE attribute.

Objectives

By the end of this chapter you will be able to:

- Understand how the FOR ALL condition works

Worked Example: For All Fire Station

The rule that will be created in this exercise will be:

```
Check for FIRE_STATION objects that for all BUILDING
objects for which BUILDING.geometry contains FIRE_
STATION.geometry check that BUILDING.BUILDING_TYPE
equals "FIRE STATION"
```

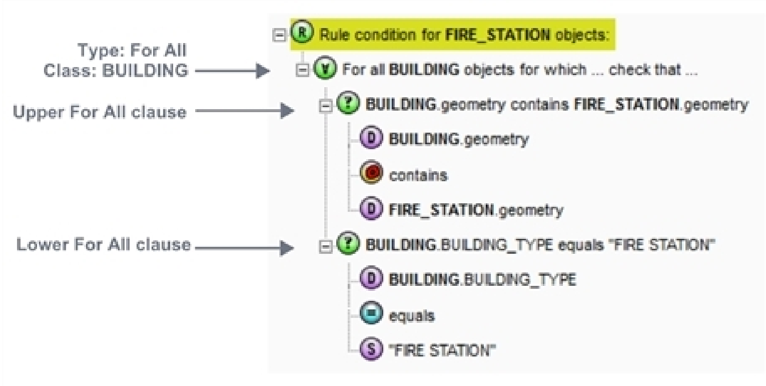


Figure 13-1: *A rule checking that all FIRE_STATION geometries are located in the correct building*

Preparation

1. Select **Auto-assist** (see "Auto-assist" on page 34).
2. Select **Use schema** and the **Philly** schema (see "Context" on page 35).
3. Create a new rule with the name `For All Fire Station check`.

Define the Rule Structure

1. Set up the FIRE_STATION class on the root node.
2. Create a **For All** condition on the root node.

For All has two clauses. The first clause finds a set of objects. The second clause then checks that all these objects satisfy a condition.

Define the first clause in the For All condition

1. Select the **<<Condition>>** placeholder, and in **Type** select **Comparison**.
2. Select the upper **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`BUILDING`
Attribute	`geometry`

3. Select the **<<Relationship>>** placeholder and create a **Spatial Relationship** using the following values:

Element details	Description
Type	Spatial Relationship
Type	Contains

4. Select the lower **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`FIRE_STATION`
Attribute	`geometry`

Define the second clause in the For All condition

1. Insert a comparison on the lower <<Condition>> placeholder using the following values:

Element details	Description
Type	Comparison

2. Select the upper <<Value>> placeholder and create a Dynamic Value using the following values:

Element details	Description
Type	Dynamic Value
Class	`BUILDING`
Attribute	`BUILDING_TYPE`

3. Select the <<Relationship>> placeholder and create a Scalar Relationship using the following values:

Element details	Description
Type	Scalar Relationship
Type	Equal

4. Select the lower <<Value>> placeholder and create a Static Value using the following values:

Element details	Description
Type	Static Value
Datatype	String
Value	`FIRE STATION`

The rule now matches the text in "Worked Example: For All Fire Station" on page 83:

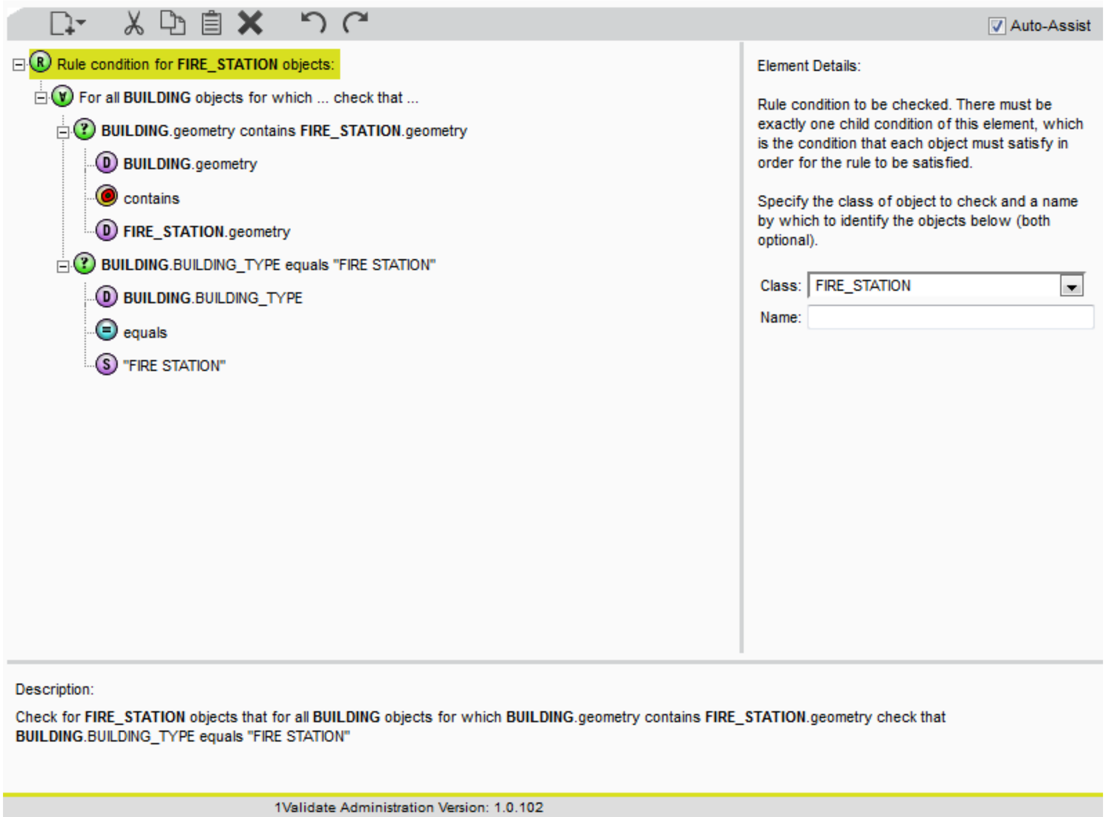


Figure 13-2: *Completed rule*

Run the rule check

1. Save the rule.
2. Add a **Check Rules** and a **Pause** task to your session.
3. Run the session.

Results

The check should complete successfully. View the results in the HTML report.

How many non-conforming objects are there?

Your results

Exercises

Exercise: Rewrite an IF...THEN...ELSE rule with a For All structure

Go back to the IF...THEN...ELSE check created in "IF...THEN...ELSE" on page 68.

Can you create a new rule that will return the same results but uses a For All condition?

Exercise: Add a clause to the Fire Station rule

Go back to the `Fire Station 1.0` rule created in "Multiple Existence Conditions" on page 75. Create a copy of this rule called `Fire Station 2.0`.

Now we know that the `BUILDINGS` have a `BUILDING_TYPE` attribute (as realised in this chapter); add a condition to the `Fire Station 1.0` rule that also checks that the `BUILDING_TYPE` is set to `FIRE STATION`.

Summary

This chapter has introduced the For All structure, the final condition to be described in this course.

Further information on all the conditions available in 1Integrate is found in "Conditions" on page 119.

14 Built-in Functions

This chapter will utilise several concepts discussed in this chapter, as well as introduce the use of built-in functions.

It will step through how to create a rule check to ensure that `STREETCENT` objects are connected to other `STREETCENT` objects (network connectivity.) In other words, the rule will verify that `STREETCENT` objects are not isolated, by checking that the start or end point of a `STREETCENT` object intersects (more specifically, touches) another `STREETCENT` object.

It will be necessary to use the object label in the rule builder to distinguish between the objects in the same class. It will also be necessary to distinguish between objects in the same class by inserting an additional condition to check that the IDs of objects that intersect each other are different.

Objectives

By the end of this chapter you will be able to:

- Appreciate that 1Integrate has a vast range of built-in functions that can be used within rules
- Understand that built-in functions require parameters
- Create a rule using some geometric built-in functions

Key Concepts

Object label

An object label uniquely identifies objects in the same class which are tested against each other in clauses and subclauses of a rule.

In the following example, the object labels A and B distinguish between the root object and objects found in the subclause, both of which have the same class label fire_station and come from the same class.

```
Check for fire_station objects A that there are at
least 5 fire_station objects B for which fire_
station:B.geometry is within 10000m of fire_
station:A.geometry
```

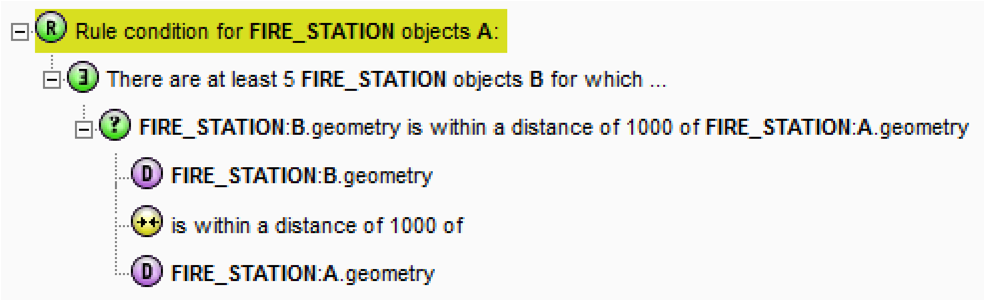


Figure 14-1: *Rule using object labels A and B*

Where the class label is omitted, all classes are considered.

In the following example, the subclause is executed over all objects except exclusion zones. The top level object has a class label of excl_zone and no object label. The object in the subclause has no class label and an object label of X.

```
Check for excl_zone objects that there are no objects X
for which X.geometry intersects excl_zone.geometry
```

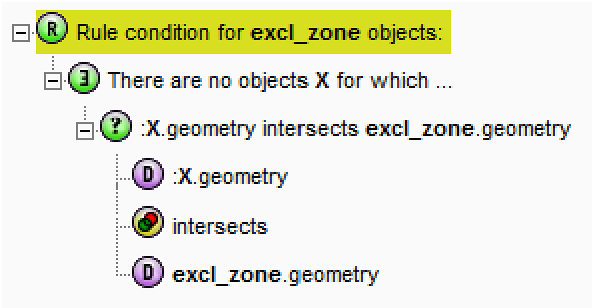


Figure 14-2: *Rule using object label X*

Built-in Functions

A built-in function returns a value calculated from one or more value node parameters. The context-sensitive help shows the number of parameters required for the function.

There are several types of built-in function which work with different types of data:

- Geometry
- Conversion
- Mathematical
- Bit manipulation
- String
- Collection
- Timestamp
- User-defined

start_of function

start_of returns a start point for a line geometry. The parameter is a simple line geometry.

end_of function

`end_of` returns the end point for a line geometry. The parameter is a simple line geometry.

Worked Example: STREETCENT Connectivity

The full rule will look like this:

```
Check for STREETCENT objects A that there is at least 1
STREETCENT object B for which (start_of
(STREETCENT:A.geometry) touches STREETCENT:B.geometry
or end_of(STREETCENT:A.geometry) touches
STREETCENT:B.geometry) and STREETCENT:A.ID does not
equal STREETCENT:B.ID
```

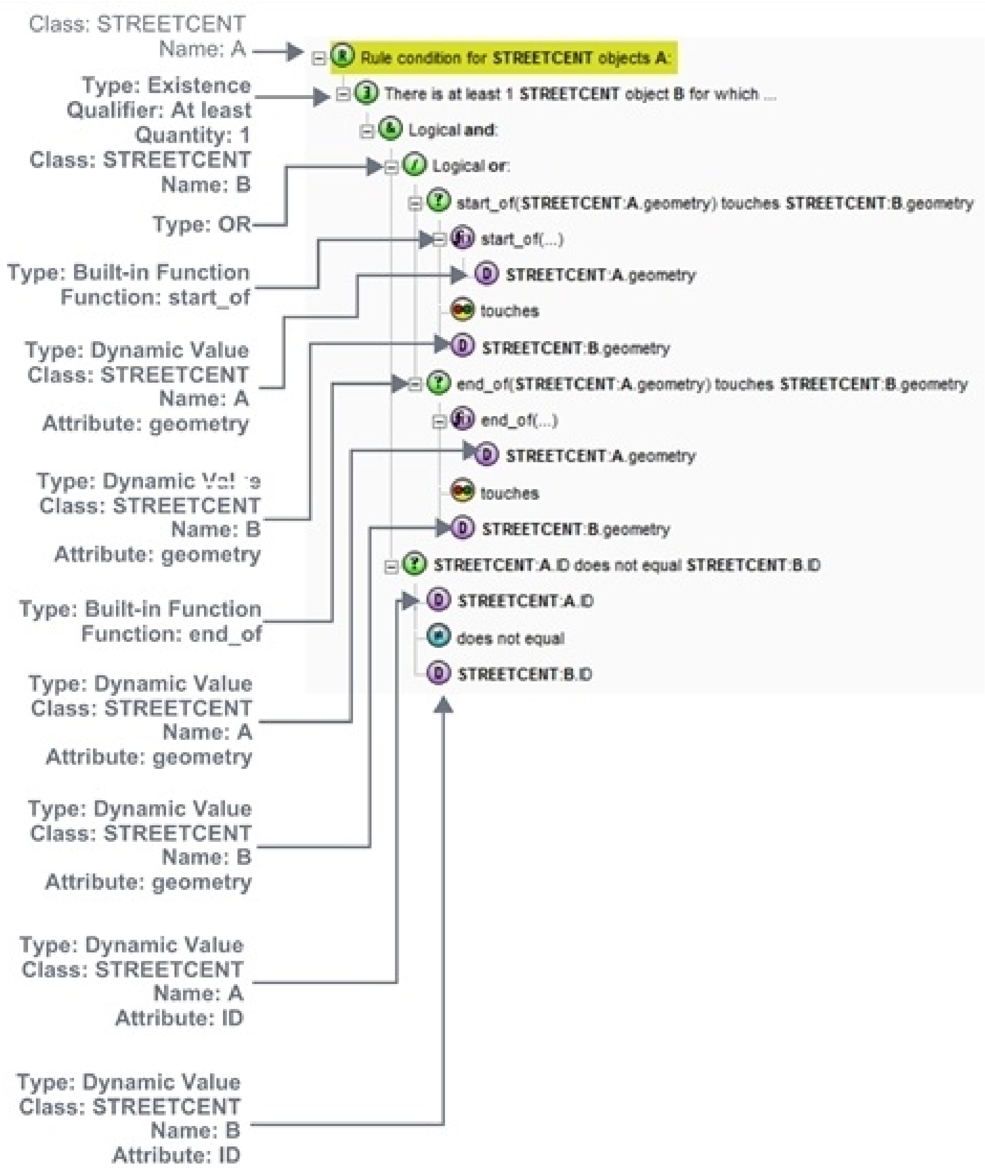


Figure 14-3: *Rule checking that that STREETCENT objects are connected to other STREETCENT objects*

Preparation

1. Select **Auto-assist** (see "Auto-assist" on page 34).
2. Select **Use schema** and the **Philly** schema (see "Context" on page 35).
3. Create a new rule with the name `STREETCENT connectivity`.

Define the Rule Structure

1. Select the root node.

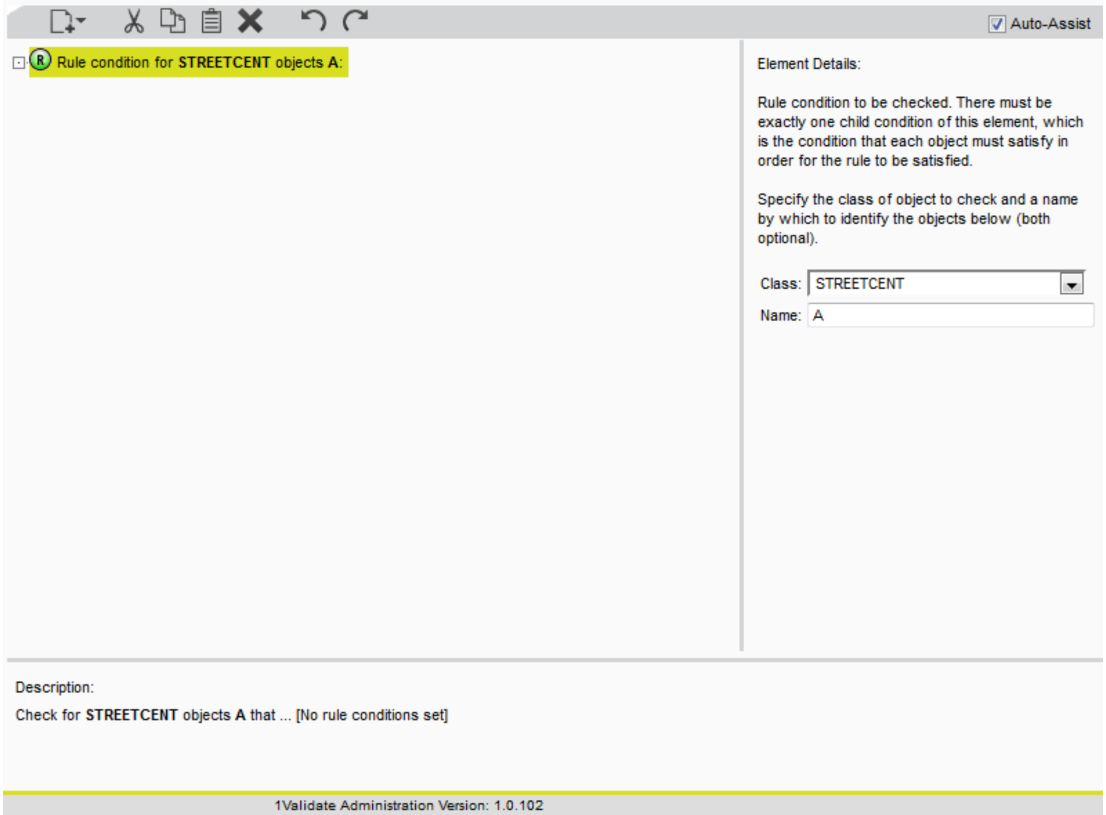


Figure 14-4: *root node of the rule*

2. In **Class** select **STREETCENT**.
3. In **Name** type `A`.
4. Insert an **Existence** condition using the following values:

Element details	Description
Type	Existence
Qualifier	At Least
Quantity	1

Element details	Description
Class	STREETCENT
Name	B

5. Insert an **AND** on the Existence **<<Condition>>** placeholder.
6. Insert an **OR** on the upper AND **<<Condition>>** placeholder.

Insert a Built-in Function

1. Select the upper **<<Condition>>** placeholder and create a **Comparison** condition.
2. Select the upper **<<Value>>** placeholder.

   In **Type** select **Built-in Function**.

   In **Function** select **start_of**.

   A **<<Value>>** placeholder is created for the **start_of** function.

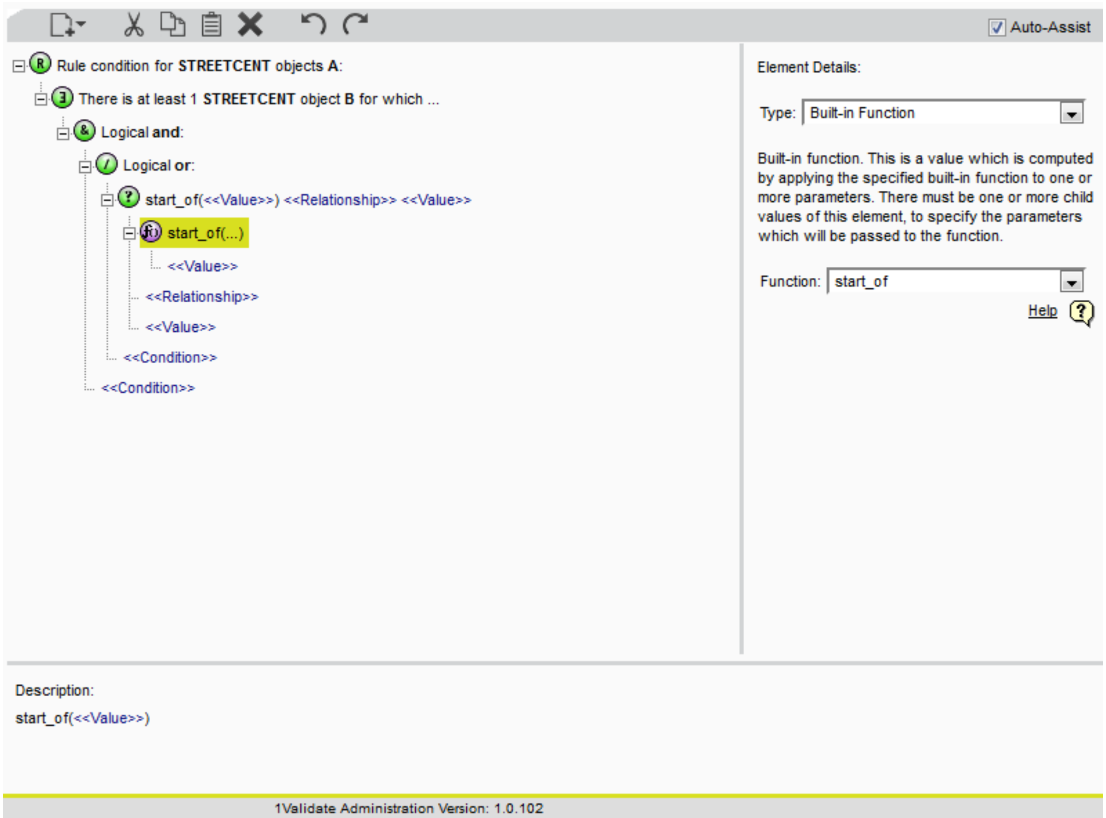


Figure 14-5: *start_of built-in function*

3. Select the built-in function's **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`STREETCENT`
Name	A
Attribute	`geometry`

4. Select the **<<Relationship>>** placeholder and create a **Spatial Relationship** using the following values:

Element details	Description
Type	Spatial Relationship
Type	Touch

5. Select the lower **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`STREETCENT`
Name	B
Attribute	`geometry`

Define the lower OR subclause

1. Select the remaining OR **<<Condition>>** placeholder and create a **Comparison** condition.
2. Select the upper **<<Value>>** placeholder and insert a **Built-in Function** using the following values:

Element details	Description
Type	Built-in Function
Function	`end_of`

3. Select the built-in function's **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`STREETCENT`
Name	A
Attribute	`geometry`

4. Select the **<<Relationship>>** placeholder and create a **Spatial Relationship** using the following values:

Element details	Description
Type	Spatial Relationship
Type	Touch

5. Select the lower **<<Value>>** placeholder and create a **Dynamic Value** using the following values:

Element details	Description
Type	Dynamic Value
Class	`STREETCENT`
Name	B
Attribute	`geometry`

Select the Logical OR node. The description should now read:

```
start_of(STREETCENT:A.geometry) touches
STREETCENT:B.geometry or end_of(STREETCENT:A.geometry)
touches STREETCENT:B.geometry
```

Define the lower AND subclause

1. Select the remaining **<<Condition>>** for the AND clause and create a **Comparison** condition.
2. Select the upper <<Value>> and create a Dynamic Value using the following values:

Element details	Description
Type	Dynamic value
Class	`STREETCENT`
Name	`A`
Attribute	`ID`

3. Select <<Relationship>> and create a scalar relationship using the following values:

Element details	Description
Type	Scalar Relationship
Type	Not Equal

4. Select the lower <<Value>> and create a Dynamic Value using the following values:

Element details	Description
Type	Dynamic Value
Class	`STREETCENT`
Name	B
Attribute	`ID`

The rule now matches the text in "Worked Example: STREETCENT Connectivity" on page 91.

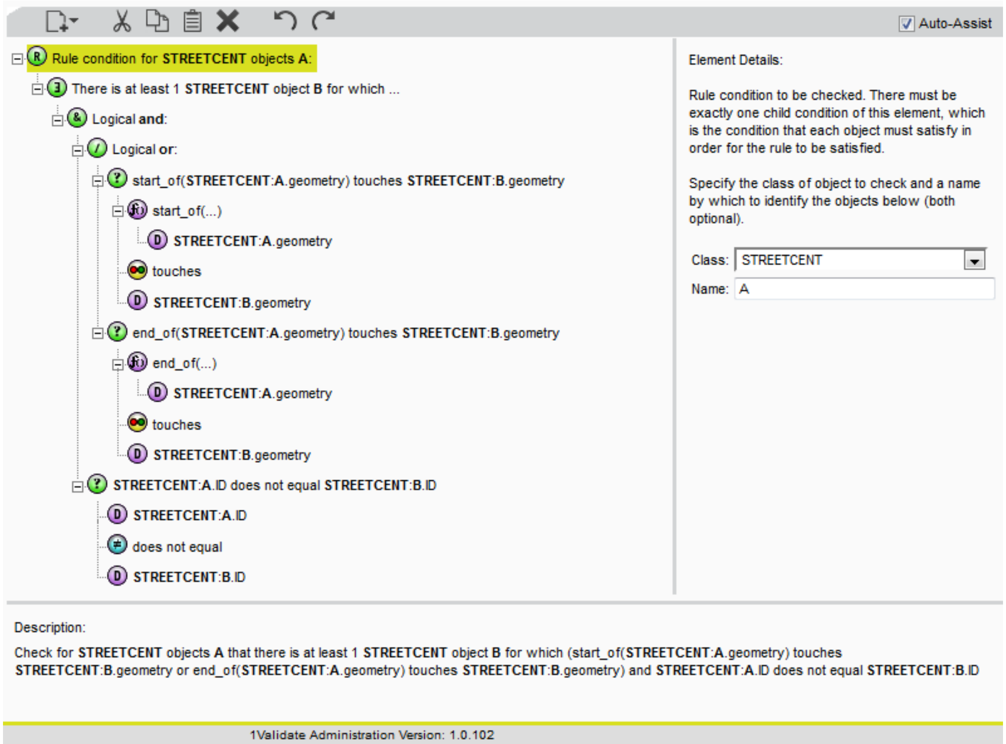
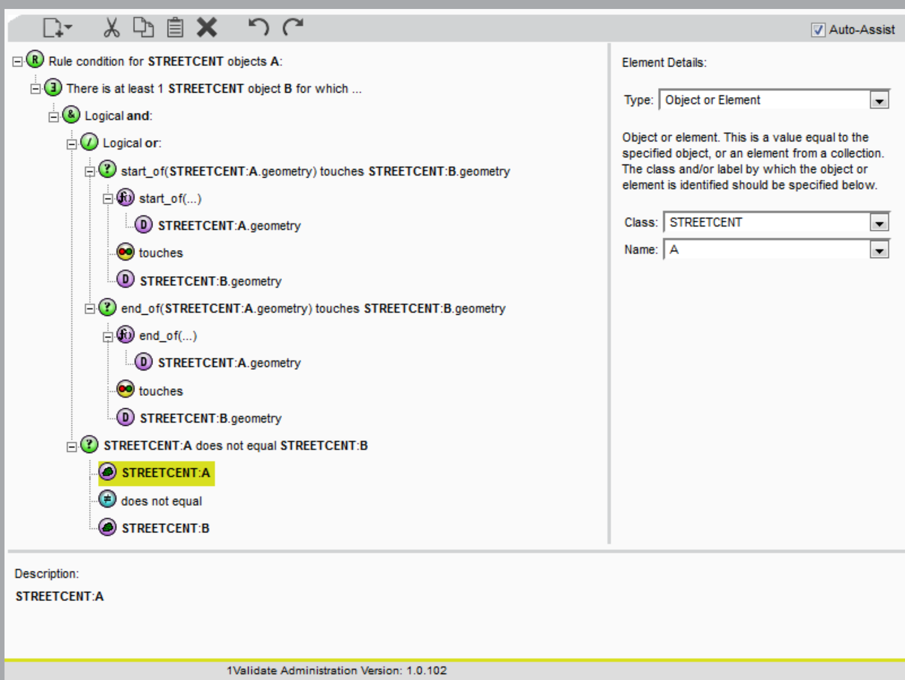


Figure 14-6: *Completed rule*

Note:
If a unique attribute is not present, the objects themselves can be compared using the Object or Element value node. For example:



Run the rule check

1. Save the rule.
2. Add a **Check Rules** and a **Pause** task to your session.
3. Run the session.

Results

How many non-conforming `STREETCENT` objects are there?

Your results

Summary

The complex rule created in this chapter used built-in functions to calculate geometric values. The chapter also introduced object labels.

This chapter completes the rule-building worked examples. The next chapter will show you how to reuse rule fragments using a template.

15 Rule Templates

In this chapter you will recreate the rule from the previous chapter (see "Worked Example: STREETCENT Connectivity" on page 91) by creating and using a template that defines the connectivity part of the rule.

Objectives

By the end of this chapter you will be able to:

- Understand the concept of templates, which are fragments of rules that can be reused
- Create a rule template and use it within a rule

Key Concepts

Rule template

A rule template allows fragments of rules to be created and stored and then reused by rules and other rule templates. If a modification is made, the change is picked up by all items referencing the template.

The output from the template could be a value, a condition or a relationship.

Rule templates can be referenced inside other templates so that trees of rule templates can be created with a rule at the root.

For example, a template could be used to define the relationship between road elements that make up a roundabout. The parameters would be the objects which define the roundabout, and the output would be a geometry value. This template could then be used in all rules which need to access roundabout geometries.

Once inserted into a rule, a template fragment is tailored to use the classes required in the rule check.

Note: Because changes are cascaded, you cannot delete a template when it is referenced.

Rule template parameters

Rule templates are created with objects as parameters. The object is defined in an input parameter to the rule template. A rule template can have zero to many parameters.

Parameters can be added, removed and changed even when the template is referenced.

Worked Example: STREETCENT Connectivity (Template)

`the start_of(STREETCENT:A.geometry) touches STREETCENT:B.geometry or end_of(STREETCENT:A.geometry) touches STREETCENT:B.geometry`

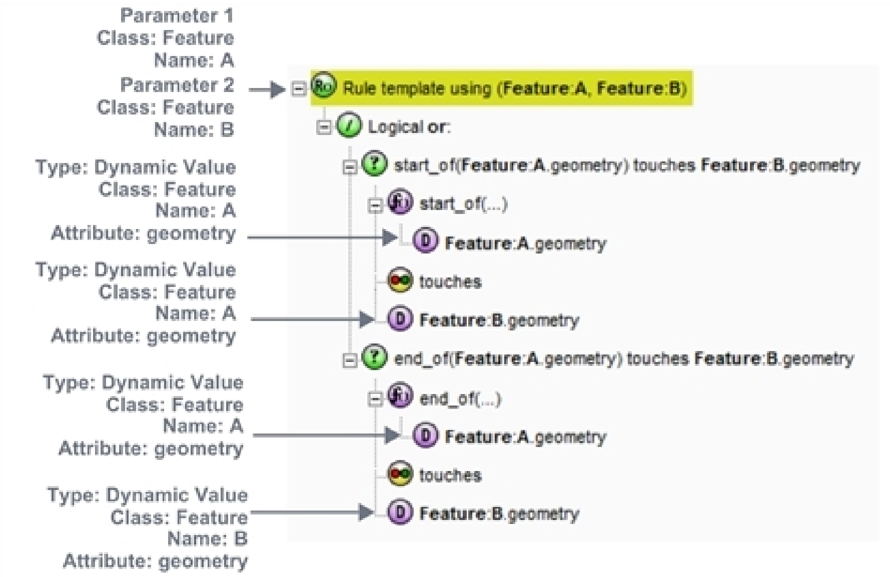


Figure 15-1: *Completed rule from "Worked Example: STREETCENT Connectivity" on page 91*

Preparation

1. Select **Auto-assist** (see "Auto-assist" on page 34).
2. Select **Use schema** and the **Philly** schema (see "Context" on page 35).
3. Create a new folder inside your rule folder called `Templates`.

Create a rule template page

1. Select the **Templates** folder.
2. Click **New** > **Rule Template**.

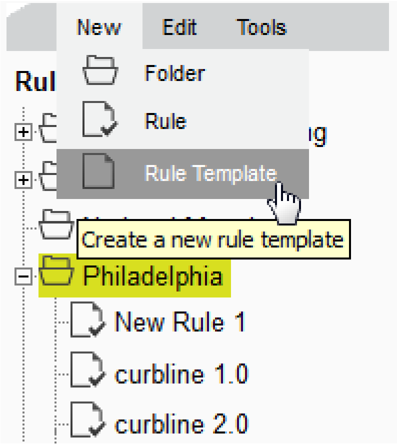


Figure 15-2: *New > Rule Template*

A rule template page is created in your Templates folder.

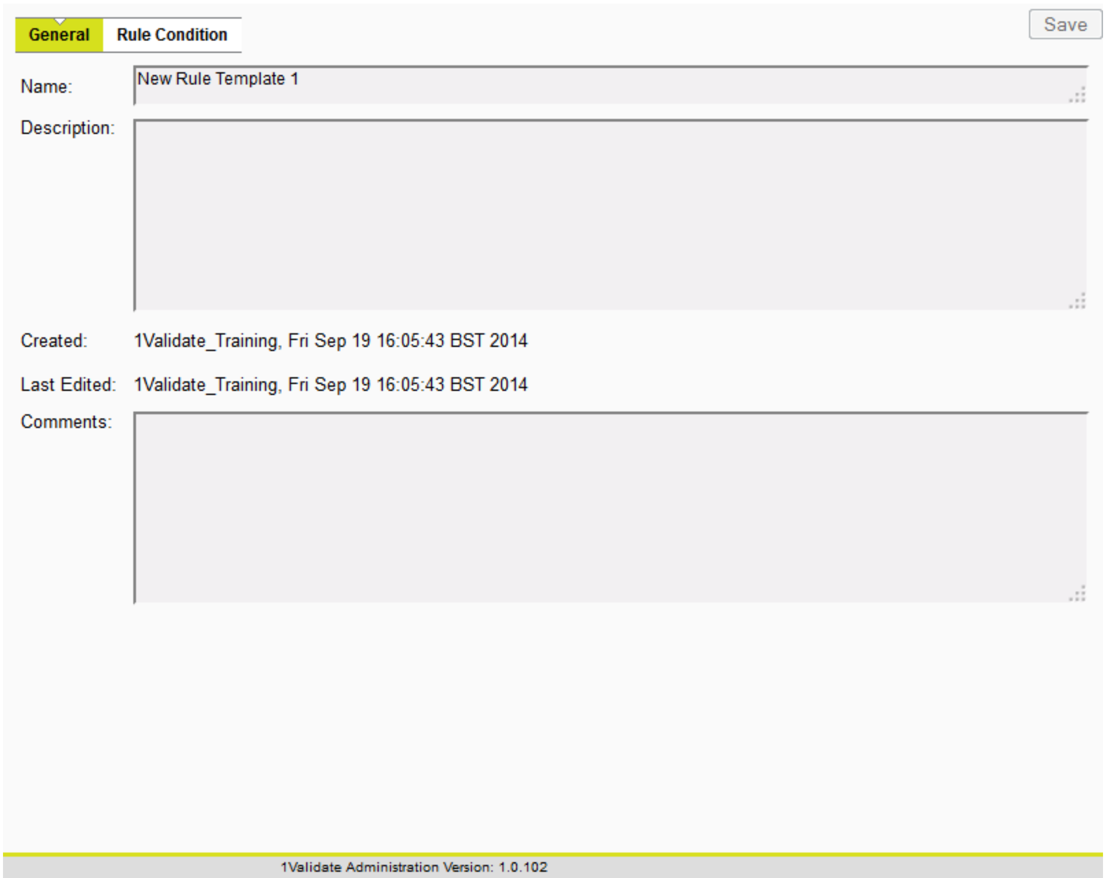


Figure 15-3: *A new rule template is created*

3. In **Name** type `Connectivity`.

Set up the template parameters

1. On the Rule Condition tab, click **Add parameter**.

   Entry fields for Parameter 1 are displayed.
2. In **Class** type `Feature`.
3. In **Name** type `A`.
4. Click **Add parameter**.

   Entry fields for Parameter 2 are displayed.
5. In **Class** type `Feature`.
6. In **Name** type `B`.

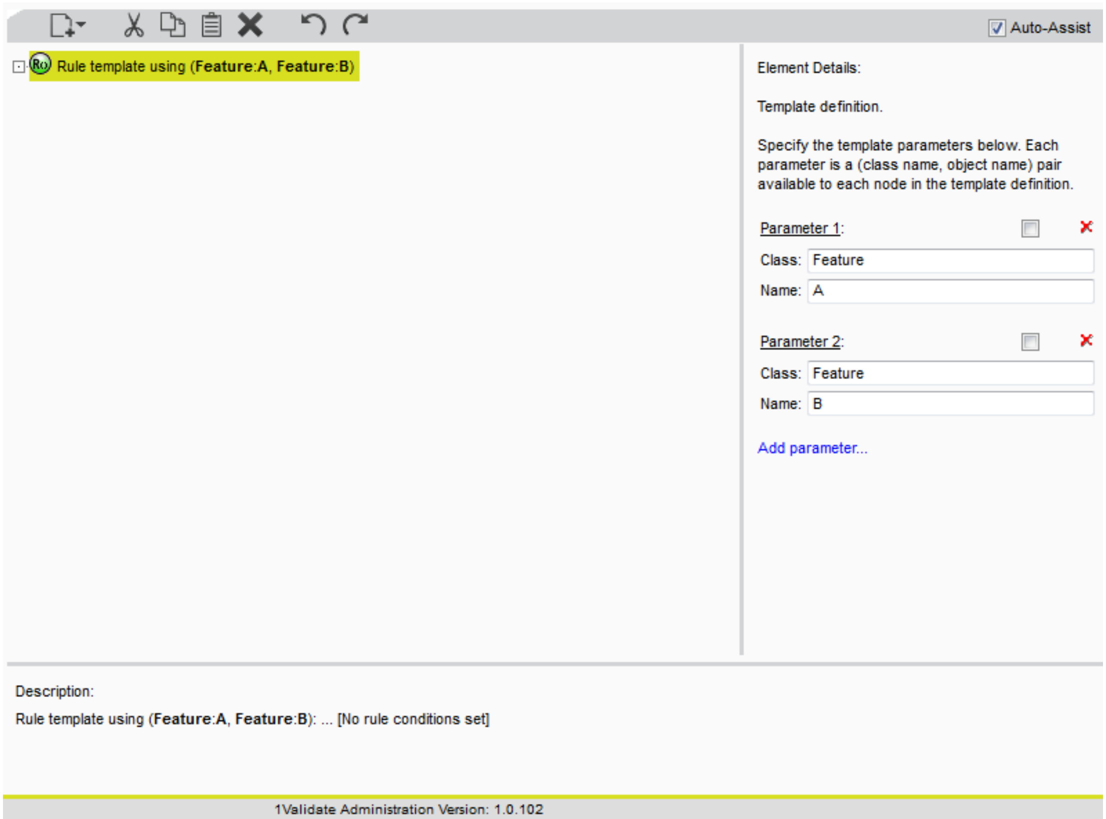


Figure 15-4: *Template parameters*

Set up the template conditions

Create the template to say `start_of(Feature:A.geometry) touches Feature:B.geometry or end_of(Feature:A.geometry) touches Feature:B.geometry`.

Because **Use Context** is cleared, you must type in the attribute geometry.

The rule now matches the text in "Worked Example: STREETCENT Connectivity (Template)" on page 100.

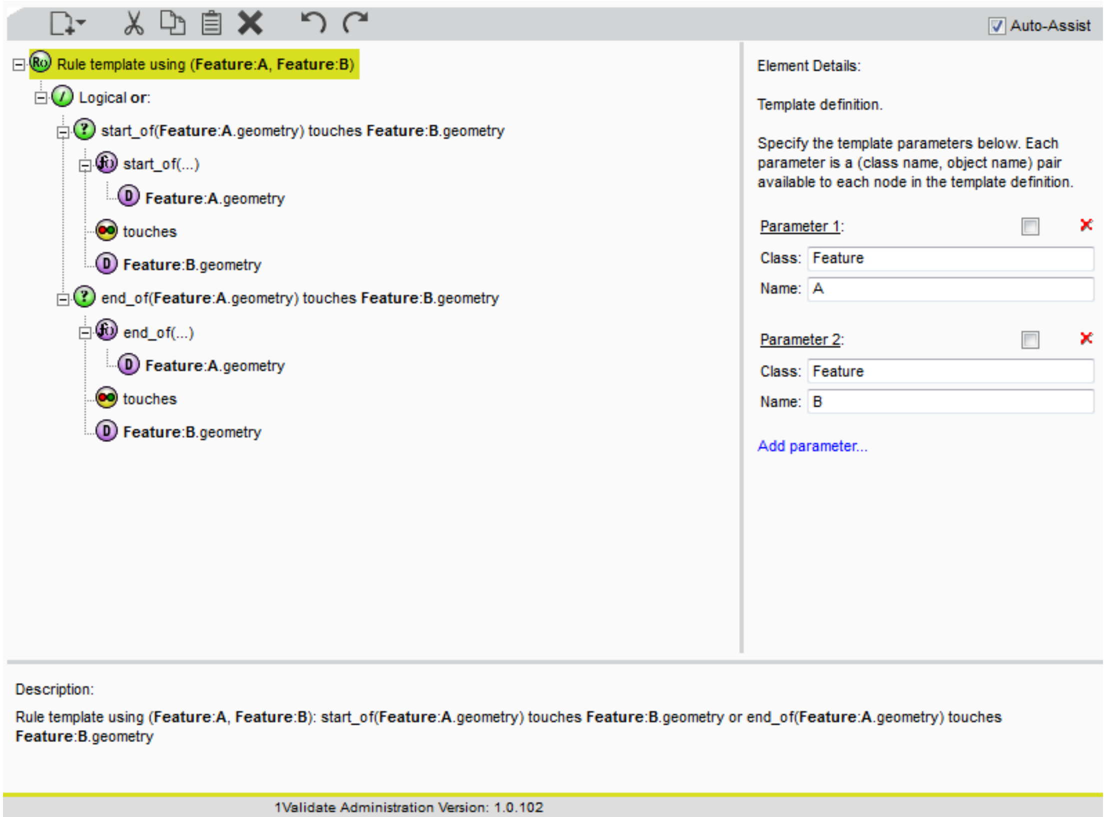


Figure 15-5: *Completed rule*

Save the rule template

Click **Save** to save the template.

This template can now be used in any rule that requires this kind of connectivity check.

Insert a rule template into a rule

Create a new rule with the name `STREETCENT connectivity 2`.

Recreate the STREETCENT connectivity rule with the Existence and AND clauses only. Leave the AND clauses empty. (See "Worked Example: STREETCENT Connectivity" on page 91 and "Refining an existing rule" on page 64.)

1. Select the upper AND clause.
2. Click 
3. Select **Rule Template**.

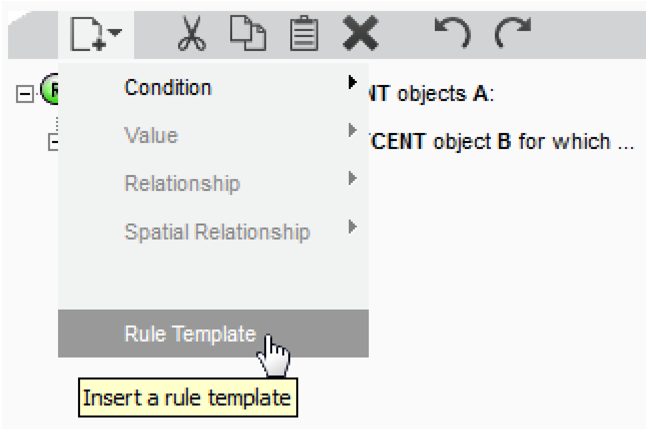


Figure 15-6: *Insert a rule template*

4. In Select Rule Template navigate to your **Templates** folder.
5. Select **Connectivity**.

   The Connectivity template is inserted under the upper AND clause.

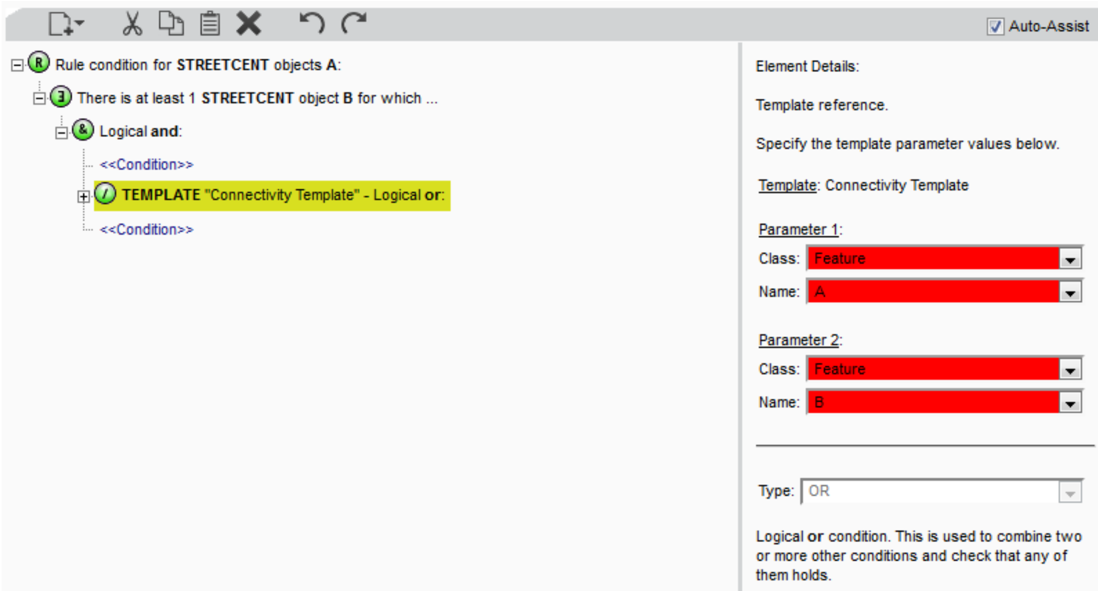


Figure 15-7: *Template is added*

The template nodes and parameters are greyed out so you cannot edit them. The parameters show Feature A and Feature B and in Description an error message is displayed.

6. Expand the Rule Template node. The Dynamic Value nodes show errors where the parameters are not completed.

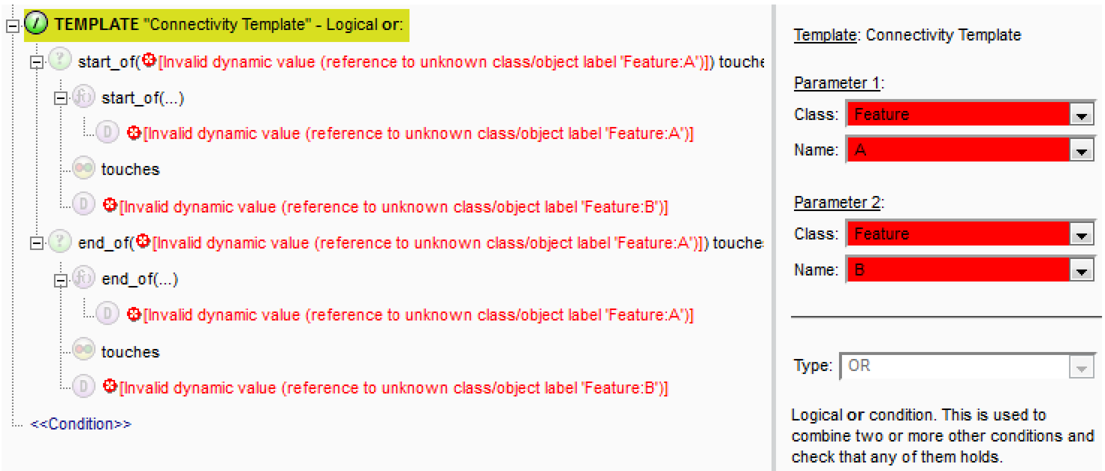


Figure 15-8: *Incomplete parameters*

7. Select the parameter drop-down menus and replace the **Class** "Feature" with STREETCENT for both parameters.

   The template is updated with the correct class names.

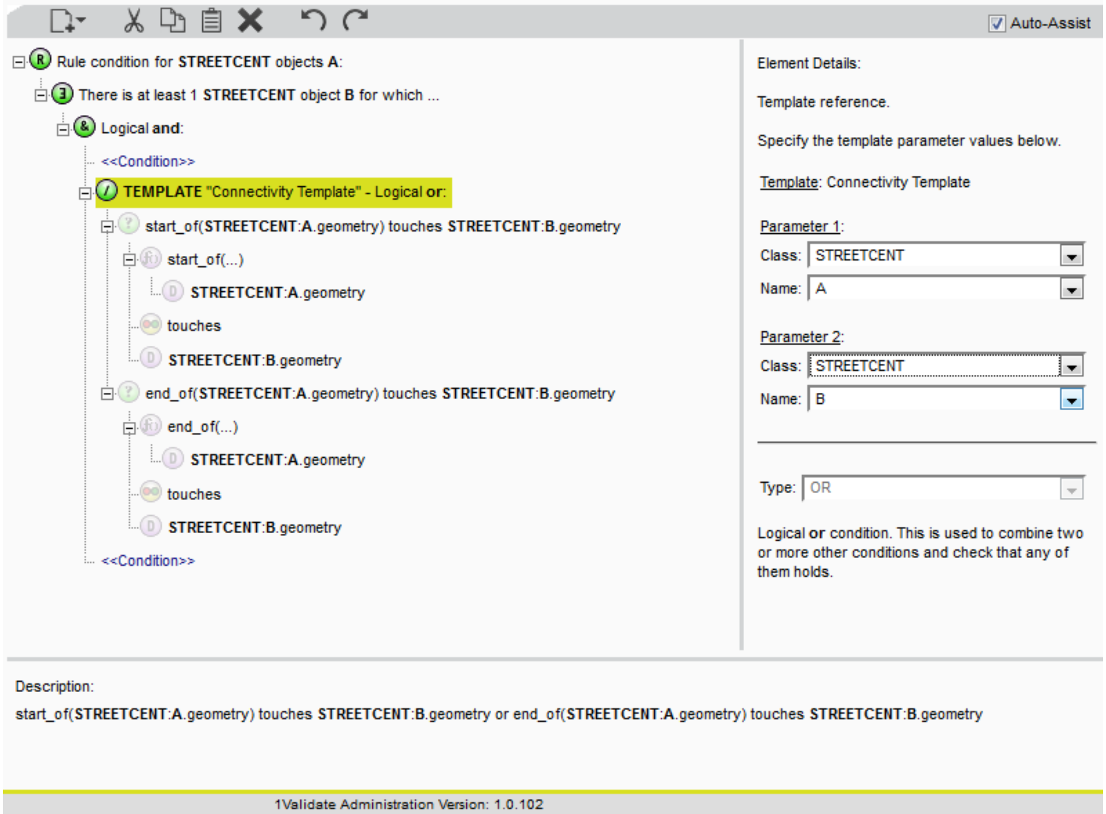
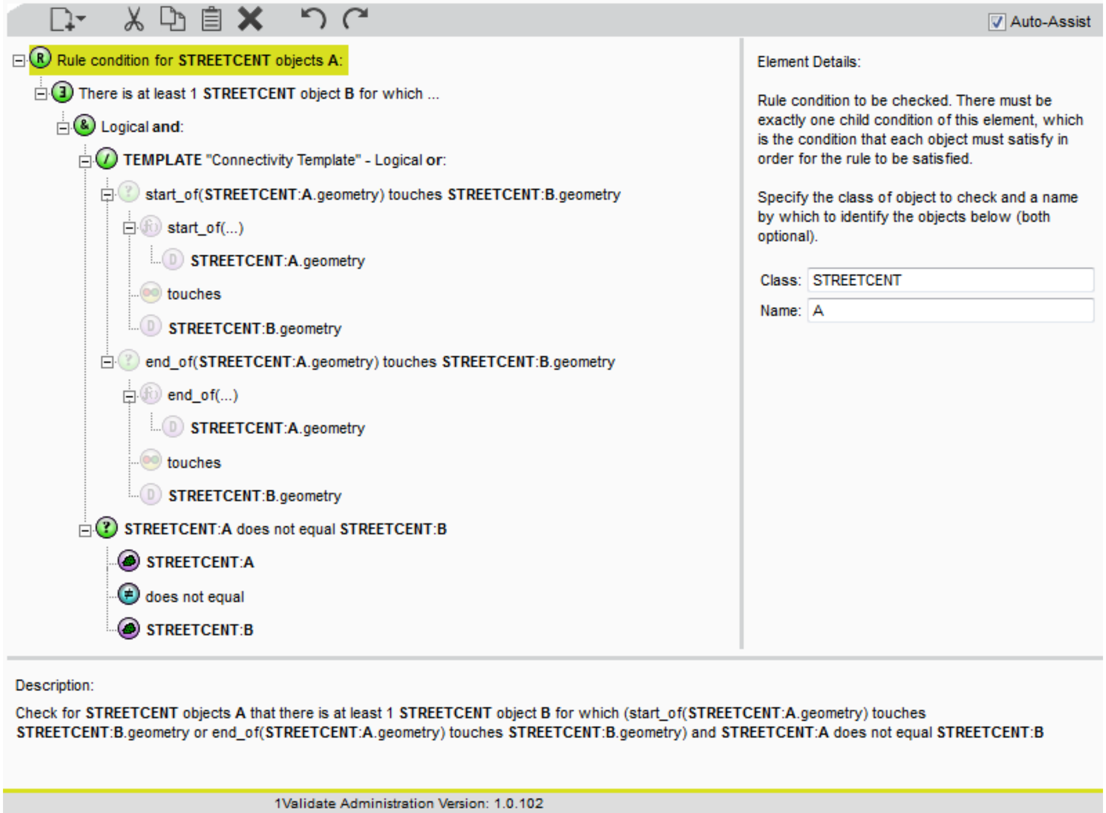


Figure 15-9: *Completed parameters*

8. Delete the upper AND clause.

9. Continue to create the rest of the rule (add in the comparison for the IDs) as described in "Worked Example: STREETCENT Connectivity" on page 91.



Run the rule check

1. Save the rule.
2. Add a **Check Rules** and a **Pause** task to your session.
3. Run the session.

Results

The rule should behave the same as the STREETCENT connectivity rule created in "Built-in Functions" on page 88.

Exercises

If you have time, create some more templates for use in rules you have written.

Summary

In this chapter you have seen how to work more efficiently by using a rule template to define rule fragments.

16 Transport Network Data

The following exercises use a new set of Oracle data that contains road and rail networks. Complete the following exercises, making a note of any results you find.

Most of these exercises require the use of built-in functions in the rule. When using these functions, be sure to check if any parameters should or could be added (use the tooltips).

Objectives

By the end of this chapter you will be able to:

- Further your understanding of the concepts presented in the course

Exercises

Exercise: Create a new datastore

Create a new datastore called `Transport Network`.

Select all classes and all attributes in the input mapping.

Create a new session and read the data into the 1Integrate cache.

Exercise: Road length

Create a rule called `Road Length` that checks that ROAD objects have a length longer than 5 m.

Note: You will need to use the **line_length** built-in function.

How many objects are less then 5m in length?

Your results

Exercise: Road spikes

Create a rule called `Road Spikes` that checks that ROAD geometries do not have any spikes.

> **Note:** When using the **has_spikes** function, use a second parameter of 0.05 to specify the sine of the angle in the spike.

How many objects have spikes?

Your results

Exercise: Road duplicates

Create a rule called `Road Duplicates` that checks that there are no duplicate ROAD geometries.

> **Note:** To create this rule, you will need to use a spatial relationship not previously used in other exercises. You will also need to distinguish between objects of the same class.

How many duplicate objects are there?

Your results

Exercise: Road connectivity

Create a rule called `Road Connectivity` that checks that if a ROAD object is less than 5m, make sure the object is connected to ROAD objects at both ends.

Note: You will need to ensure there is at least one other ROAD object at the start and at least one at the end.

How many objects are not connected at both ends?

Your results

Exercise: 3-D noding

Create a rule called `3-D noding` that for all ROAD objects that intersect another ROAD object, check that they intersect at common vertices (that is they share a node with the same x, y, z values).

Note: There is a function called **is_noded** that can be used for this. Read the tooltip to see what it requires.

How many objects do not intersect at common vertices?

Your results

Exercise: RAIL objects

If you have time, reuse some of these rules for the RAIL objects.

Do any RAIL objects fail any of the rules?

Your results

Summary

In this chapter you have worked through exercises using transport network data. You have loaded the data, and built rules using the concepts presented in this course.

"Utilities Data" on page 111 presents further exercises for you to work through.

17 Utilities Data

The following exercises use two sets of Oracle data. The first set consists of some utilities data, the second consists of some coincident OS MasterMap data. Because the data is coincident, it is possible to load both sets of data into the 1Integrate cache together so that rule assertion can be performed across the layers.

This is an example of a data aggregation exercise, where data is being brought together to assess quality. In this case, the OS MasterMap data is the gold standard layer, the benchmark by which all other layers are measured for quality. Once all the data is in the 1Integrate cache, business rules can be created and asserted to ensure the data conforms to those rules.

The following exercises are examples of real business rules created to determine the quality of utilities data against OS MasterMap.

The utilities data consists of CABLE and TERMINAL classes. Both of these classes have attributes to determine the name and type of the objects. These attributes will need to be used in the exercises.

The MasterMap data consists of a table called TOPOGRAPHICAREA. Attribution on this class determines the type of object (for example DESCRIPTIVETERM, and THEME). These will also be used in the exercises.

Complete the following exercises, making a note of any results you find. Most of these exercises require the use of built-in functions in the rule. When using these functions, be sure to check if any parameters should or could be added (use the tooltips).

Objectives

By the end of this chapter you will be able to:

- Further your understanding of the concepts presented in the course

Exercises

Exercise: Create a Utilities datastore

Create a new datastore called `Utilities` using the connection parameters provided.

Select all classes and all attributes in the Input Mapping.

Exercise: Create a MasterMap datastore

Create a new datastore called `MasterMap` using the connection parameters provided.

Select all classes and all attributes in the input mapping.

Create a new session and read both data stores into the 1Integrate cache.

Exercise: Cables in buildings

Create a rule called `Cables in Buildings`. The rule should satisfy the following business rule:

CABLE objects that have a NAME of MCC, PCC or MKS should not intersect buildings.

Note: You will need to identify TOPOGRAPHICAREA objects that are of type Building. Use the DESCRIPTIVEGROUP attribute to find the string `Building`. Use the scalar relationship Regular Expression to search for multiple string values in one check, delimited with a "|" (pipe symbol).

How many cables intersect buildings?

Your results

Terminals in buildings

Create a rule called `Terminals in Buildings`. The rule should satisfy the following business rule:

TERMINAL objects that have a TERM_TYPE of POLE , or a NAME of either CONNECTION_UNIT or TERMINATOR, should not be in buildings.

How many terminals intersect buildings?

Your results

Exercise: Terminals in water

Create a rule called `Terminals in Water`. The rule should satisfy the following business rule:

TERMINAL objects that have a TERM_TYPE of POLE, or a NAME of either CONNECTION_UNIT or TERMINATOR, should not intersect water objects.

Note: Use the TOPOGRAPHICAREA.THEME attribute to find the string `Water`.

How many terminals intersect water objects?

Your results

Exercise: Cable connectivity

Create a rule called `Cable Connectivity`. The rule should satisfy the following business rule:

CABLE objects must have another CABLE object attached at either the start or the end of it.

How many CABLE objects do not have another CABLE object attached to the start or the end of it?

Your results

Summary

In this chapter you have worked through exercises using utilities data. You have loaded the two sets of data, and read both into the 1Integrate cache. You have built rules using the two data stores.

18 MidCoast Water Case Study

Objectives

By the end of this chapter you will be able to:

- Consolidate your understanding of the concepts presented in the course

About MidCoast Water

MidCoast Water is a County Council, providing water and sewerage services to the Greater Taree and Great Lakes Council areas in Australia.

The authority was formed in 1997, taking on staff and assets from Greater Taree City Council, Great Lakes Council and NorthPower.

MidCoast Water provides services to 35,000 customers, over an area of some 7000 sq km. It operates four water supply schemes and 14 sewerage treatment plants and maintains a reticulation network including 34 reservoirs, 202 pumping stations, and 1820 km of pipelines.

MidCoast Water has continued to develop a unique geographic information system to help maintain their vital assets. Although this has greatly increased the quality of information available and the speed at which it can be transferred to maps, the system was lacking in regard to topological control.

Topology underpins the automated processing of geospatial datasets, so in order to move geometric objects successfully to accommodate accuracy improvements (matching GPS and non-GPS collected data), it is necessary to build a topological structure. This provides the framework to ensure that spatial data is error free and accessible in multiple GIS applications and business systems.

MidCoast Water have gained huge time, cost, and water savings through increased data accuracy and centralised our business spatial data with fit-and-forget technology. Combining Australia’s first installation of 1Integrate with Radius Topology, MidCoast Water has been able to build and manage rule-based processing workflows for spatial data.

Saving up to 80% in data editing and improving data quality to up to 300%, MidCoast Water has been able to utilise best of breed solutions in innovative ways.

The data

The data used in this training course can be broken down into a number of tables.

Table Name	Description
SP_ SEWGPIPE	A gravity-fed sewer pipe
SP_ SEWRPIPE	A rising main sewer pipe. (The pipe is pressurised, but pushed from the pump.)
SP_ SEWVPIPE	A vacuum main sewer pipe. (The pipe is pressurised, but pulled to a pump.)
SP_ SEWNODE	A sewer node object
SP_ SEWBASIN	Sewer catchment areas
SP_ WATPIPE	A water pipe
SP_ WATNODE	A water node object

The primary key on these tables is called `GID` and the geometry column is called `geometry`.

Exercise: Create a MidCoast Water datastore

Create a new datastore called `MidCoast` using the connection parameters provided.

Select all classes and all attributes in the Input Mapping.

Rules to be defined

There are a number of rules of varying complexity that can be applied to the data.

Sewer rules

Exercise: Sewer diameter

Check that the diameter of a gravity-fed sewer pipe contains a value greater than 0 (attribute `PIPE_DIA`).

Exercise: Sewer material

Check that the material of a gravity-fed sewer pipe contains a value (attribute `PIPE_MATRL`).

Exercise: Sewer node type

Check that if the `NODE_TYPE` of a sewer node is `ENDCAP` then it should be at the start or end of a gravity-fed sewer pipe.

Exercise: Isolated sewer nodes

Check that sewer nodes are not isolated (they can be attached to any type of sewer pipe).

Exercise: Sewer nodes in basin

Check that sewer nodes are in a catchment area (except if they have a `NODE_TYPE` of `JUNCTION`).

Water rules

Exercise: Water pipes

Check that where two water pipes join and their `PIPE_DIA` attributes are different that there is a water node object that is either `JUNCTION` or `REDUCER` (use `NODE_TYPE`).

Summary

This case study is the final chapter in the course. The case study consolidates your knowledge of rule building within 1Integrate and has allowed you to practise the techniques you have learned.

Having completed the course you should now:

- Be familiar with the 1Integrate interface for connecting to an external spatial data source
- Know how to assess the quality of the data against a set of defined business rules
- Have a thorough understanding of the 1Integrate rule builder for creating and refining rules

Appendices

A Conditions

Condition	Description	Parameters	Child Nodes
Comparison	A comparison compares two **values** using a **relationship** and returns true or false to indicate whether the values fulfil the relationship. The data types of values that can be used depends on the relationship.	None	▶ Value ▶ Relationship ▶ Value
Reference	Test for a reference between two objects. Specify the two objects and the name of the reference between them. The condition is satisfied if the first object refers to the second object using the given reference.	▶ Class label 1 ▶ Object label 1 (optional) ▶ Reference ▶ Class label 2 ▶ Object label 2 (optional)	None
Test in Range	Test that a value lies in the range between two other values. Specify whether or not the end points of the range should be included in the comparison (i.e. > or ≥). The values can be integers, reals or strings.	▶ Include upper endpoint (yes or no) ▶ Include lower endpoint (yes or no)	▶ Value (to test) ▶ Value (range minimum) ▶ Value (range maximum)

Condition	Description	Parameters	Child Nodes
AND	Logical AND condition. Used to combine two or more other conditions (supplied as child objects) and check that they are all true.	None	▶ Condition1 ▶ Condition2 ▶ Condition3 ▶ ... ▶ ConditionN
OR	Logical OR condition. Used to combine two or more other conditions (supplied as child objects) and check of any of them are true.	None	▶ Condition1 ▶ Condition2 ▶ Condition3 ▶ ... ▶ ConditionN
XOR	Logical XOR condition. Used to combine two or more conditions (as child nodes) and return true only if exactly one (but no more than one) of the conditions is true, or false otherwise.	None	▶ Condition1 ▶ Condition2 ▶ Condition3 ▶ ... ▶ ConditionN
NOT	Logical NOT condition. Used to invert the result of another condition (supplied as a child object). For example, if the child condition is true, this condition returns false and vice-versa.	None	▶ Condition

Condition	Description	Parameters	Child Nodes
IF... THEN... ELSE	This must have two or three child conditions. The first condition is checked and if it holds then the overall result is the result of checking the second condition. Otherwise, if the first condition does not hold, the overall result is the result of checking the third condition, or true if the third condition is omitted.	None	▶ Condition1 ▶ Condition2 ▶ Condition3 (optional)

Condition	Description	Parameters	Child Nodes
Existence	This may be used to check for the existence (or absence) of related objects satisfying a condition specified in a clause. The objects in the clause are identified by the class and object label pair. You specify a qualifier ("at least", "at most", or "exactly"), a number of objects and another class name. The condition checks that there exists at least one object for which the child condition holds. To check for the absence of objects satisfying a condition, look for exactly 0 objects.	▶ Qualifier ▶ Integer ▶ Class Label ▶ Object Label (optional)	▶ Condition
For All	Check that all related objects satisfy some other condition. This condition must have two child conditions. The first condition *finds* a set of objects, the second condition then *checks* that these objects meet certain requirements.	▶ Class label ▶ Object label (optional)	▶ Condition (find) ▶ Condition (check)

Condition	Description	Parameters	Child Nodes
Existence in Collection	Checks whether the required number of elements exist in an object collection or an array that match the condition in the clause.	▶ Qualifier ▶ Integer ▶ Class label ▶ Object label (optional)	▶ Value (collection name) ▶ Condition
For All in Collection	Checks all objects or elements in a collection satisfy a condition. The condition requires a value which should be an array or a collection of objects, and a condition to check. For a collection of simple types, or objects of unspecified class, the class label may be omitted. Otherwise the class label specifies the type of object in the collection.	▶ Class label ▶ Object label	▶ Value ▶ Condition

B Values

Value	Description	Parameters	Child Nodes
Aggregate Value	Calculates a single value by combining others, using functions such as Count, Sum, Average, max, min or geometric union. This value should have a condition to test and zero or more child values. This value is computed by aggregating the values over any objects that are satisfied by the condition. There are a number of different types of aggregate functions. Detailed information on the child values required are provided in a tool-tip within 1Integrate.	▶ Function	▶ Condition ▶ Values
Array Element	An indexed element in an array. The first value should be the array. The second value should be the index into the array, always starting from 0.		

Value	Description	Parameters	Child Nodes
Built-in Function Value	A value which is computed by applying the specified built-in function to one or more parameters. There are typically one or more child values of this element, to specify the parameters which will be passed to the function. When a function has been selected from the list of all possible built-in functions, a help icon provides a tool tip with information about the parameters required by this function.	▶ Function name	▶ Value 1 ▶ Value n (optional)
Class Name	Returns the name of the class of an object. The object is specified by the class label or object label pair.	▶ Class label ▶ Object label	None
Conditional Value	This value should have a condition to test and two child values to choose between. If the condition is true, the result is the first child value. If the condition is false, the result is the second child value.	None	▶ Condition ▶ Value 1 ▶ Value 2

Value	Description	Parameters	Child Nodes
Difference	This value should have two child values (integers or real numbers). The result is obtained by subtracting the second value from the first.	None	▶ Value 1 ▶ Value 2
Division	This value should have two child values (integers or real numbers). The result is obtained by dividing the first value by the second.	None	▶ Value 1 ▶ Value 2
Dynamic Value	An attribute from an object. An object's geometry is one of its attributes. You must specify the class of the object and the attribute to read. You may also need to select the name of an object identified in an earlier part of the rule, if it is necessary to distinguish between different objects of the same class.	▶ Class label ▶ Object label ▶ Attribute name	None

Value	Description	Parameters	Child Nodes
Named Constant	A named constant is a lookup value. Rule authors can specify a key word that equates to a value used when running rules or performing actions. Named constants are listed in a metadata store. When you insert a named constant, you must select a metadata store, a table of named constants within the metadata store, and a key word within the table. If the key word has an associated code value, when the rule or action is run it will use the code value instead of the key word. If there are no code values specified for this metadata table then the key is used, which allows for restricting a value to one from a fixed codelist.		
Negative	This value should have one child value (an integer or a real number), containing a sign that will be inverted. For example, `-1` becomes `+1` and vice-versa.	None	▶ Value

Value	Description	Parameters	Child Nodes
Nested Value	A value nested within a composite value. These are used in two ways: A. Some custom data stores or built-in functions allow attributes that are structures containing a number of sub-attributes. B. Some custom built-in functions return objects. A nested value allows you to access attributes of these objects, because they are not accessible via an attribute value as the object is not 'in scope'.		
Null Value	A Null value always evaluates as null. This can be used to verify if an object attribute value or the result of a calculation is null. **Note:** Searching on a null value is not valid for arrays.	None	None

Value	Description	Parameters	Child Nodes
Object or Element Value	An object or an element from a collection. These are used in two ways: A. To access a whole object directly, e.g. to compare objects to check whether they are or the same object or not, or to pass an object in to a built-in function or store it as a temporary value. In this case, users specify the class and, if required, the name alias of the object. B. To access an element when looping through a collection or array. One example is to use a "For All in Collection" or "Loop over a Collection" over a geometry in order to access each part of the geometry. Within the loop, access the element using this Object or Element value, leaving the class name blank but selecting the name specified by the loop.	▶ Class label ▶ Object label	None

Value	Description	Parameters	Child Nodes
Product	This value should have two child values (integers or real numbers). The result is obtained by multiplying together all child values.	None	▶ Value 1 ▶ Value 2
Remainder	This value should have two child values (integers or real numbers). The result is obtained by dividing the first value by the second, and taking the remainder.	None	▶ Value 1 ▶ Value 2
Static Value	A Static Value is fixed, and does not change. This can be used on either side of a comparison condition or as part of a more complicated expression. It can be a boolean, an integer, a real (floating point) or a string (text or timestamp). You must specify both the type of value, and the value itself.	▶ Datatype	None
Sum	This value should have at least two child values (integers or real numbers). The result is the sum of all child values. Alternatively, the values can be string types, in which case they are joined together.	None	▶ Value 1 ▶ Value 2 ▶ Value n (optional)

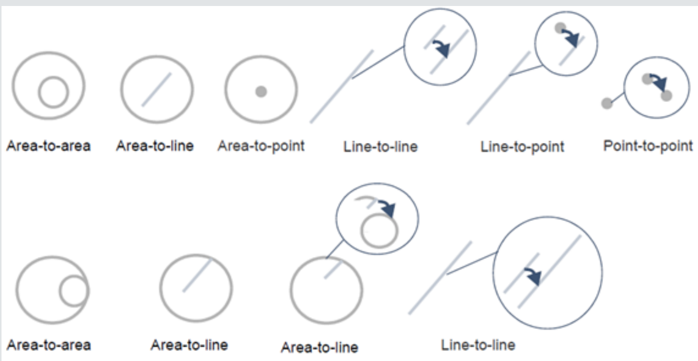
Value	Description	Parameters	Child Nodes
Temporary Value	A temporary value is initially null and can be used to hold temporary results during an action. **Note:** A temporary value can be used in an action but not in a rule. The value can have either local or global *scope*. If it has local scope, the value will be reset to null before the action is applied to each object. If it has global scope, the value will be reset to null only at the start of the whole task. A temporary value can be any type, but this type is determined by the value it is given.	- Scope - Value	None

C Relationships

Scalar Relationships

Scalar Relationship	Description
Equal	The two values are the same.
Not Equal	The two values are different.
Less Than	The first value is less than the second value.
Less Than or Equal	The first value is less than the second value, or the two values are the same.
Greater Than	The first value is greater than the second value.
Greater Than or Equal	The first value is greater than the second value, or the two values are the same.
Begins With	For string values, test whether the first string begins with the second string.
Ends With	For string values, test whether the first string ends with the second string.
Contains	For string values, test whether the first string contains the second string.
Regular Expression	For string values, check whether the first string matches the wild card string or Perl regular expression in the second string.

Spatial Relationships

Spatial Relationship	Description
Beyond	**Beyond** checks that the geometry is at least a certain distance away from another (specified in dataset units). The order of the geometries does not matter. Note: **Beyond** is the opposite of **Within Distance**.
Contains	**Contain** means that the other geometry is completely inside this line or polygon (one object's boundary and interior are inside another). Note: Their boundaries *are* allowed to intersect, so it includes the **Cover** and **Equal** relationships. For lines the boundary means the end points, and for polygons the boundary means all inner and outer rings. The types of geometries that can contain another are: ▶ Area *contains* area ▶ Area *contains*line ▶ Area *contains* point ▶ Line *contains* line ▶ Line *contains* point ▶ Point *contains* point  Figure C-1: *Contains geometries* Note: **Contains** is the reciprocal of **Within**.

Spatial Relationship	Description
Covered By	**Covered By** is a special case of **Within**. It is only true when this geometry is contained within another, *and* their boundaries intersect (so they share some boundary). For lines the boundary means the end points, and for polygons the boundary means all inner and outer boundary rings. The types of geometries which can be covered by another are: ▶ Area *covered by* area ▶ Line *covered by* area ▶ Line *covered by* line Note: **Covered by** is the reciprocal of **Covers**.
Covers	**Covers** is a special case of **Contains**. It is only true when the other geometry is contained within this one, *and* their boundaries intersect (so they share some boundary). For lines the boundary means the end points, and for polygons the boundary means all inner and outer boundary rings. The types of geometries which can cover another are: ▶ Area *covers* area ▶ Area *covers* line ▶ Line *covers* line Note: **Covers** is the reciprocal of **Covered By**.

Spatial Relationship	Description
Cross	Two geometries **Cross** when a line runs across the boundary of an area, or the interior of another line. **Note:** At least one of the geometries *must* be a line. The types of geometries which can cross are: - Line and line A line crosses another line if only their interiors intersect at single points (and at the same height). - Area and line A line crosses a polygon geometry if its interior intersects the polygon's interior and either of the line's endpoints are outside the polygon. The order of the geometries does not matter.  Figure C-2: *Cross geometries*
Disjoint	**Disjoint** means that the geometries do not intersect at all (neither the boundaries nor the interiors intersect). The order of the geometries does not matter. **Note:** **Disjoint** is the opposite of **Intersect**.

<table>
<tr><th>Spatial Relationship</th><th>Description</th></tr>
<tr><td>Equal</td><td>Two geometries are Equal when they are identical (they have the same interior, boundary and exterior).

Geometries do not need to have the same number of vertices (as long as they follow the same path, within geometric tolerance), and lines do not need to have the same direction.

Polygon boundaries do not have to use the same start/end vertex in order to be equal.

The order of the geometries does not matter.

The types of geometries which can be equal are:
▶ Area and area
▶ Line and line
▶ Point and point

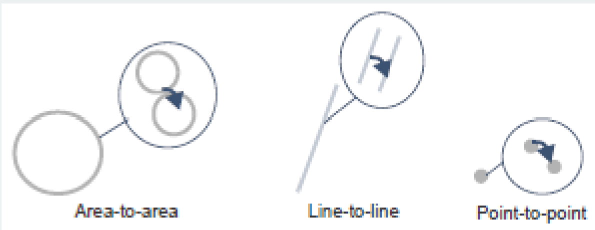


Figure C-3: Equal geometries</td></tr>
<tr><td>Intersect</td><td>Intersect means any sort of spatial interaction.

It checks that either the boundaries or interiors of two geometries intersect in any way, using any of the following relationships:
▶ Equal
▶ Touches
▶ Overlap
▶ Cross
▶ Within
▶ Contains

The order of the geometries does not matter.

Note: Intersect is the opposite of Disjoint.</td></tr>
</table>

<table>
<tr><th>Spatial Relationship</th><th>Description</th></tr>
<tr><td>Overlap</td><td>
Two geometries Overlap when two lines or two areas partly overlay each other.
Overlap means geometries are partly inside and partly outside each other (so does not include being contained or being equal).
The types of geometries that can overlap are:

Area and area
A polygon overlaps another polygon if it shares some but not all of its area.
Line and line
A line overlaps another line if it shares some but not all of its length.

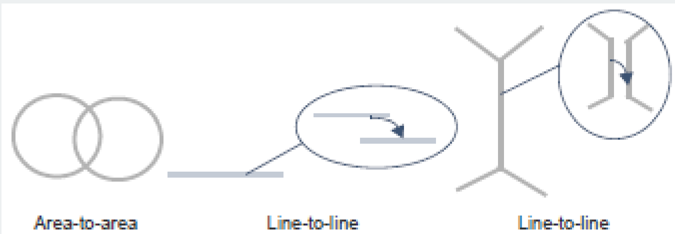
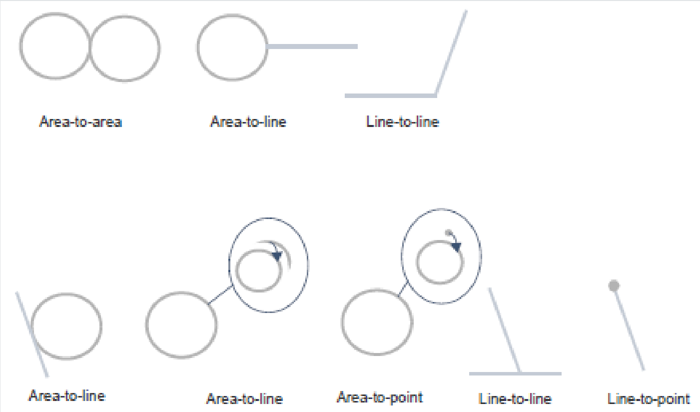
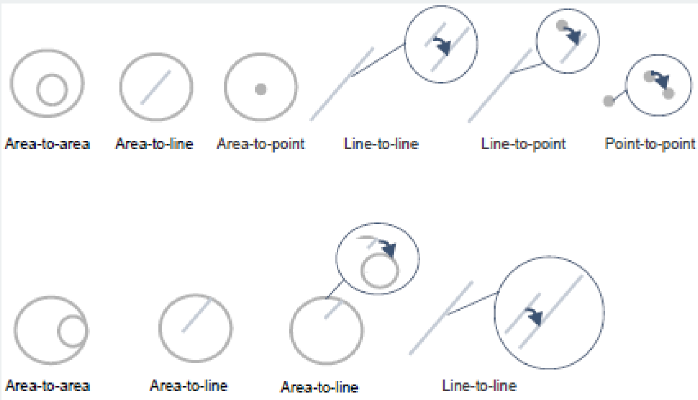
The order of the geometries does not matter.


Figure C-4: Overlap geometries
</td></tr>
</table>

Spatial Relationship	Description
Touches	**Touch** means that the boundaries intersect but the interiors do not. For lines the boundary means the end points, and for polygons the boundary means all inner and outer rings. Adjacent polygons or end-to-end lines touch. The types of geometries that can touch are: ▶ Area and area ▶ Area and line ▶ Area and point ▶ Line and line ▶ Line and point The order of the geometries does not matter.  Figure C-5: *Touch geometries*

Spatial Relationship	Description
Within	**Within** means this geometry is completely inside the other, which must be a line or polygon. Their boundaries *are* allowed to intersect, so it includes the **Covered by** and **Equal** relationships. For lines the boundary means the end points, and for polygons the boundary means all inner and outer rings. The types of geometries which can be within another are: ▶ Area *within* area ▶ Line *within* area ▶ Line *within* line ▶ Point *within* area ▶ Point *within* line ▶ Point *within* point  Figure C-6: *Within geometries* **Note: Within** is the reciprocal of **Contains**.
Within Distance	**Within Distance** checks that the geometries approach within a specified minimum distance of each other (specified in dataset units). The order of the geometries does not matter. **Note: Within Distance** is the opposite of **Beyond**.

D Aggregate Functions

Function	Description	Parameter(s)
all_distinct	Returns true if all the input values are distinct (different).	- Any value to test. - *(optional)* Any additional values to test.
all_same	Returns true if all the input values are identical.	- Any value to test. - *(optional)* Any additional values to test.
as_list	Returns a list containing all the values passed in. The list may contain duplicates.	- Any item to be added to the list. - *(optional)* Any additional items to add to the list.
avg	Returns the average (mean) of the input values.	- A numerical value.
count	Counts the number of objects or the number of non-null parameter values. If no parameters are passed, this function counts the number of objects traversed. Otherwise it counts the number of objects for which any of the parameters has a non-null value.	- *(optional)* Any values to count.
count_ distinct	Counts the number of distinct groups of the specified value or values.	- Any value to count. - *(optional)* Any additional values to count.
intersection	Returns the intersection of the input geometries.	- Any geometry.

Function	Description	Parameter(s)
max	Returns the maximum value of the parameter over each object traversed. The values may be either numerical values, booleans or strings. Boolean values are regarded as being equal to either 1 for true or 0 for false. Strings are compared lexicographically. If numbers are compared to strings, they are converted to strings before being compared.	▶ A numerical, boolean or string value.
mbr	Returns the MBR (minimum bounding rectangle) of a set of geometries. The final result is the smallest rectangle, with sides parallel to the X and Y axes, containing all the geometry values passed in.	▶ Any geometry.
min	Returns the minimum value of the parameter over each object traversed. The values may be either numerical values, booleans or strings. Boolean values are regarded as being equal to either 1 for true or 0 for false. Strings are compared lexicographically. If numbers are compared to strings, they are converted to strings before being compared.	▶ A numerical, boolean or string value.

Function	Description	Parameter(s)
sum	Returns the sum of the parameter value over each object traversed. The values may be either numerical values, booleans or strings. Boolean values are regarded as being equal to either 1 for true or 0 for false. Strings are added together using concatenation. If numbers are added to strings, they are converted to strings first.	▶ A numerical, boolean or string value.
union	Returns the union of the input geometries.	▶ Any geometry.