# Developing Business Rules

Version 2023.3
2024-05-16

InterSystems Corporation   1 Memorial Drive   Cambridge MA 02142   www.intersystems.com

*Developing Business Rules*
InterSystems IRIS Data Platform Version 2023.3 2024-05-16

# Table of Contents

**1 About Business Rules ..... 1**

- 1.1 Rules as Classes ..... 2

**2 Getting Started ..... 5**

- 2.1 Using the Old Zen Rule Editor ..... 5
- 2.2 About Rule Definitions ..... 6
  - 2.2.1 Exporting and Importing Rules ..... 6
- 2.3 About Rule Sets ..... 6

**3 Working with Rules ..... 9**

- 3.1 About If and Else Clauses ..... 9
- 3.2 About Actions ..... 10
  - 3.2.1 Using the foreach Action ..... 10
- 3.3 Defining Expressions ..... 11
  - 3.3.1 Editing the Condition Property of an if Clause ..... 12
  - 3.3.2 Expression Operators ..... 14
  - 3.3.3 Expression Functions ..... 15
  - 3.3.4 Expression Examples ..... 15
- 3.4 Defining Constraints ..... 16
- 3.5 Disabling a Rule ..... 17
- 3.6 Passing Data to a Data Transformation ..... 17
- 3.7 Adding Business Rule Notification ..... 18

**4 Debugging Routing Rules ..... 19**

- 4.1 Testing Routing Rules ..... 19
  - 4.1.1 Testing with Raw Text of a Message ..... 19
  - 4.1.2 Test Results ..... 20
  - 4.1.3 Security Requirements ..... 20
- 4.2 Strategies for Debugging Routing Rules ..... 20

**Appendix A: Utility Functions for Use in Productions ..... 25**

- A.1 Built-in Functions ..... 25
- A.2 Syntax to Invoke a Function ..... 29

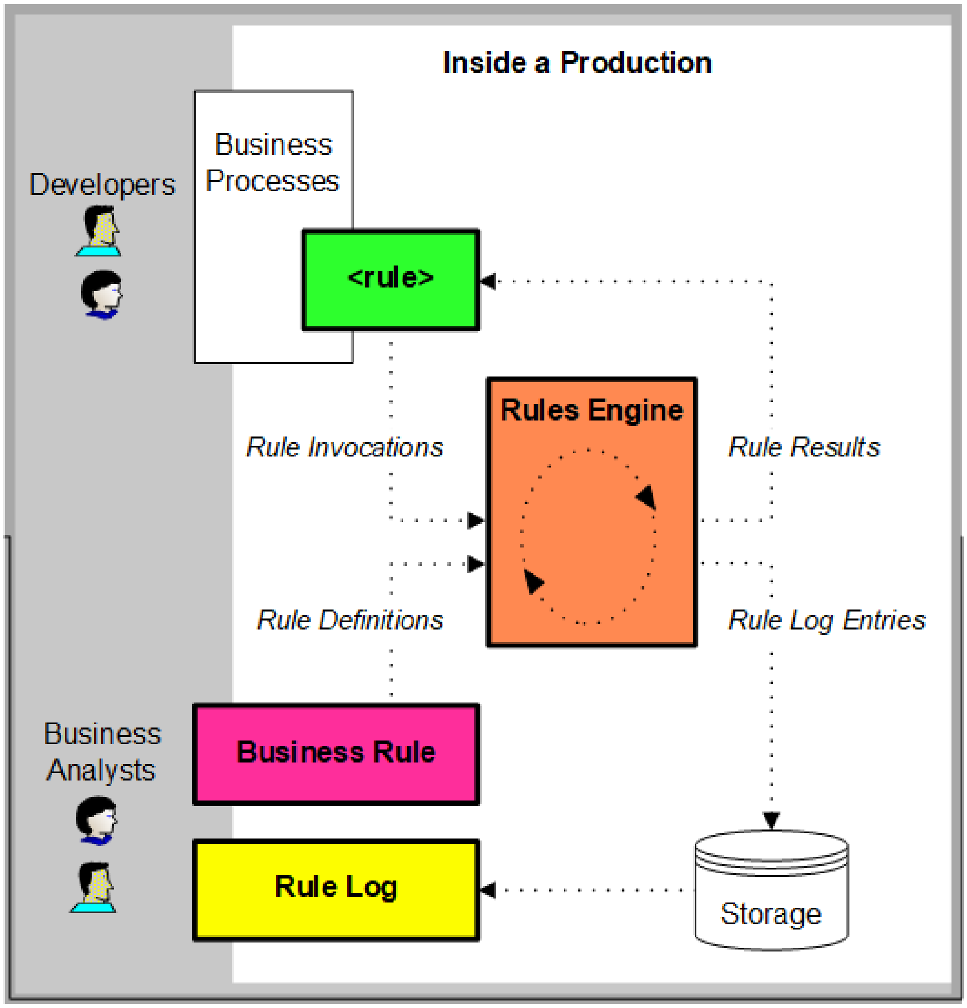
# List of Figures

Figure 4–1: Solving Problems with Routing Rules (Drawing A) ........ 21
Figure 4–2: Solving Problems with Routing Rules (Drawing B) ........ 22
Figure 4–3: Solving Problems with Routing Rules (Drawing C) ........ 23
Figure 4–4: Solving Problems with Routing Rules (Drawing D) ........ 24

# 1

# About Business Rules

*Business rules* allow nontechnical users to change the behavior of business processes at specific decision points. You can change the logic of the rule instantly, using the Rule Editor in the Management Portal. There is no need for programming or diagramming skills to change the rule, and there is no need to modify or compile production code for changes to take effect. The following figure shows how business rules work.



Suppose that an international enterprise runs a production that processes loan applications. The decision process is consistent worldwide. However, each bank in the enterprise has its own acceptance criteria, which may vary from country to country. Business rules support this division of responsibility as follows:

1. The developer of the business process identifies a decision point, by naming the business rule that will make the decision on behalf of the business process. The developer leaves a placeholder for that business rule in the Business Process Language (BPL) code by invoking the Business Process Language (BPL) element <rule>. The <rule> element specifies the business rule name, plus parameters to hold the result of the decision and (optionally) the reason for that result. Suppose we call this rule `LoanDecision`.
2. Wherever the <rule> element appears in a BPL business process, a corresponding rule definition must exist within the production. A user at the enterprise, typically a business analyst, may define the rule using the browser-based online form called the Rule Editor. This form prompts the user for the simple information required to define the business rule called `LoanDecision`. InterSystems IRIS® saves this information in its configuration database.

   Any enterprise user who is familiar with the Rule Editor and who has access to it in the Management Portal can modify the rule definition. Modifications are simply updates to the database and can be instantly applied to a production while it is running. Therefore, it is possible for business analysts at various regional locations to run the Rule Editor to modify their copies of the rule to provide different specific criteria appropriate to their locales.
3. At runtime, upon reaching the BPL <rule> statement the business process invokes the rule named `LoanDecision`. The rule retrieves its decision criteria from the configuration database, which may be different at different locales. Based on these criteria, the rule returns an answer to the business process. The business process redirects its execution path based on this answer.
4. For ongoing maintenance purposes, the business process developer need not be involved if a rule needs to change. Any rule definition is entirely separate from business process code. Rule definitions are stored in a configuration database as classes and are evaluated at runtime. Additionally, rule definitions can be exported and imported from one InterSystems IRIS installation to another.

   In this way, enterprise users such as business analysts can change the operation of the business process at the decision point, without needing the programming expertise that would be required to revise the BPL or class code for the business process.

A rule definition is a collection of one or more rule sets. A rule set is a collection of one or more rules. Each rule set has an effective (or beginning) date and time as well as an ending date and time. When a business process invokes a rule definition, one and only one rule set is executed.

You have several options for specifying a rule workflow using business processes, data transformations, and business rules. For a discussion of the options, see Comparison of Business Logic Tools.

These topics describe how to define business rules including how to create and use rule sets using the Rule Editor as well as how to invoke rules using BPL and using business process routing engines.

# 1.1 Rules as Classes

The **Interoperability** > **Build** > **Business Rules** (or **Rule Editor**) page enables enterprise business analysts to shape the logical decisions made by a business process in a structured way without requiring programming skills.

Additionally, Studio enables business process developers to work with business rule definitions as classes, for example:

### Class Definition

```
/// Business rule responsible for mapping an input location
///
Class Demo.ComplexMap.Rule.SemesterBatchRouting Extends Ens.Rule.Definition
{

Parameter RuleAssistClass = "EnsLib.MsgRouter.RuleAssist";

XData RuleDefinition [ XMLNamespace = "http://www.intersystems.com/rule" ]
{
```

```
<ruleDefinition alias="" context="EnsLib.MsgRouter.RoutingEngine"
production="Demo.ComplexMap.SemesterProduction">
<ruleSet name="" effectiveBegin="" effectiveEnd="">
<rule name="" disabled="false">
<constraint name="source" value="Semester_Data_FileService"></constraint>
<constraint name="msgClass" value="Demo.ComplexMap.Semester.Batch"></constraint>
<when condition="1">
<send transform="" target="Semester_Data_FileOperation"></send>
<send transform="Demo.ComplexMap.Transform.SemesterBatchToSemesterSummaryBatch"
target="Semester_Summary_FileOperation"></send>
<send transform="Demo.ComplexMap.Transform.SemesterBatchToFixedClassBatch"
target="Semester_FixedClassBatch_FileOperation"></send>
<send transform="Demo.ComplexMap.Transform.SemesterBatchToFixedStudentBatch"
target="Semester_FixedStudentBatch_FileOperation"></send>
<send transform="" target="Semester_FixedStudent_BatchCreator"></send>
<return></return>
</when>
</rule>
</ruleSet>
</ruleDefinition>
}

}
```

You can open a business rule as a class in Studio, edit the document, and save the changes. Changes saved in Studio are immediately available in the **Rule Editor** page. However, you may have to refresh the page to see them.

### Package Mapping Rule Classes

Given that rules are classes, you can map rules to other namespaces. If you do so, you must recompile all the mapped rule classes in each namespace where you use them to ensure that the local metadata is available in each namespace.

For details, see Package Mapping.

# 2

# Getting Started

Each business rule in InterSystems IRIS is part of a rule set. In turn, each rule set is part of a larger rule definition. Sometimes, the terms *rule* and *rule definition* are used interchangeably, but each individual rule is ultimately grouped under a rule definition.

Rule definitions, rules sets, and rules are edited using the Rule Editor, which is accessed in the Management Portal by navigating to **Interoperability** > **Build** > **Business Rules**.

You can open existing rules in the Rule Editor by navigating to **Interoperability** > **List** > **Business Rules**.

**Important:** The Rule Editor includes two web applications: `/ui/interop/rule-editor` and `/api/interop-editors`. The `/api/interop-editors/` web application can be configured to use the same authentication methods as the Management Portal. If an alternative authentication method is not configured for the web application, users will need to re-authenticate as an InterSystems IRIS user upon opening the Rule Editor. The `/ui/interop/rule-editor` web application should remain unauthenticated.

## 2.1 Using the Old Zen Rule Editor

The Zen based Rule Editor has been replaced with a new Rule Editor. The new Rule Editor maintains the same functionality, but with an updated user interface.

The old Rule Editor is still accessible from the new Rule Editor. To access it, open the new Rule Editor, click on the menu in the upper right (identified by the username of the current user and a profile icon), then click **Open in Zen Rule Editor**.

You can also set the old Rule Editor to be used as the default editor. To do this:

1. In the Management Portal, navigate to **System Administration** > **Security** > **Applications** > **Web Applications**.
2. Select the **/ui/interop/rule-editor** web application.
3. Clear the **Enable Application** checkbox.
4. Click **Save**.

Disabling this web application will cause the Management Portal link to open the old Rule Editor. If you disable the **/ui/interop/rule-editor** web application, in order to access the new Rule Editor, you will have to re-enable the application.

## 2.2 About Rule Definitions

Individual business rules are grouped under a rule definition, which is built and edited using the Rule Editor. A rule definition includes the following settings:

**Package**

Package for the rule definition class.

**Name**

Name of the rule definition class.

**Description**

User-specified description of the rule definition and its purpose.

**Rule Type**

Type of rule definition, which determines valid actions when defining rules that belong to the rule definition.

**Context Class**

Class that determines which object properties you can modify when you edit a rule. For general business rules, the context class is generated from the business process class associated with the BPL process and ends in .Context. For routing rules that are not associated with a BPL process, the context class is usually the business process class used by the routing engine.

When creating the rule definition, you can use the **Filters** options to shrink the list of classes that appears in the **Context Class** drop-down list.

**Production Name**

(Optional) For routing rules, provides the name of the production where the rule will be used so the Rule Editor can offer predefined options when editing the rule. For example, if you specify a production and then modify a constraint, the configuration items in the production appear as options for the **Source** field of the constraint.

The routing rule is not automatically used in the production unless you specify the rule when you configure the production.

### 2.2.1 Exporting and Importing Rules

A rule definition, including its rule sets and rules, can be exported by navigating to **Interoperability** > **List** > **Business Rules**, and selecting **Export**. You can import a previously exported rule definition using the **Import** option.

Alternatively, you can also export and import rule classes from the **System Explorer** > **Globals** page of the Management Portal or the **Tools** menu in Studio as well.

## 2.3 About Rule Sets

When you create a new rule definition, a new rule set is created automatically. To define the rule set name and effective date range, or to create new rules sets, select the ✎ icon next to the rule set name.

In general, there are two types of rule sets:

- *General business rule sets* — A list of rules that are evaluated sequentially until one of them is found to be true. The rule that is found to be true determines the next action of the business process that invoked the rule. If none of the rules are true, the rule set returns a default value. You invoke this type of rule set using the BPL <rule> element.
- *Routing rule sets* — A rule set for use in message routing productions. Based on the types and contents of incoming messages (which you specify as constraints), the routing rule set determines the correct destination for each message and how to transform the message contents prior to transmission. You use a routing engine business process to invoke this type of rule set.

All rule sets have two properties:

- **Rule Set Name** — Identifier for the rule set.
- **Effective Range** — Defines the time during which the rule set is effective, that is, when its rules will be executed.

Typically, a rule definition includes only one rule set that is always in effect. However, a rule definition can include multiple rule sets as long as they are in effect at different times. Each time a business process invokes a rule, one and only one rule set is executed.

# 3

# Working with Rules

A rule set contains one or more rules that you define to satisfy specific functions in a business process. Once you create a new rule definition in the Rule Editor (**Interoperability** > **Build** > **Business Rules**), you are ready to start adding rules to a rule set.

Though you can give each rule a name, it is not required. By default, InterSystems IRIS® names the rules in sequential order in the form rule#*n*. If you give the rule a user-defined name, it appears in the class definition and also appears in parentheses next to the internal rule name in the rule log. The value of *n* changes if you reorder the rules in a rule set.

## 3.1 About If and Else Clauses

A rule can contain one or more **if** clauses and an **else** clause. Each clause can include actions such as **assign** or **return**.

The logic in an **if** clause can be executed only if the condition property associated with the clause holds true. The logic in an **else** clause can be executed only if *none* of the condition properties associated with the preceding **if** clauses holds true. When a rule contains multiple **if** clauses, only the logic in the first **if** clause where the condition property associated with the clause holds true is executed. For more information about conditions, see Editing the Condition Property of an if Clause.

As you develop rules, keep the following points in mind:

- Once the execution through a rule set encounters a **return** action, the execution of the rule set ends and returns to the business process that invoked the rule definition class.
- You can control the execution of more than one rule in a rule set by omitting the returns. In other words, if you want to check all rules, do not provide a **return** action within any of the rule clauses. You may then provide a value in a **return** action at the end of the rule set for the case where no rule clauses evaluate to true.
- Each **if** clause has a condition property. A common design for a general business rule set is one that contains one rule with a series of **if** conditions and returning a value depending on which condition is true. If you want to return a default value if none of the conditions is true, you can use the **else** clause with a return.
- A common design for a routing rule set is one that contains several rules each with a different constraint defined and each with one **if** clause describing how and where to route the message that matches the constraint.
- You can access property paths in virtual documents using the syntax described in the syntax described in Virtual Property Path Basics.

# 3.2 About Actions

Each **if** or **else** clause in a rule can include actions, but they are not required. The actions in a clause are executed if and only if the condition associated with the clause holds true. The following actions are supported:

| Rule Set Type | Action | Description |
|---|---|---|
| All | **assign** | Assigns values to properties in the business process execution context. |
| All | **return** | Returns to the business process without further execution of the rule. For general rules it also returns the indicated value to the result location. |
| All | **trace** | Adds the information you enter into the Event Log when this specific part of the rule is executed. For details, see <trace>. |
| All | **debug** | Adds the expression text and value to the Rule Log when this specific part of the rule is executed. The **debug** action is executed only if the router business process RuleLogging property specifies the d flag, For details on the RuleLogging property, see Rule Logging. |
| Segmented Virtual Document Routing Rule or HL7 Message Routing Rule | **foreach** | Loops through a repeating segment. A segment may repeat if it is designated as a repeating segment, is in a repeating loop, or both. See Using the foreach Action for more details. |
| Routing Rule | **send** | When evaluated by a routing engine business process, this action sends the message to a particular target after optionally transforming it. For the ability to pass data to the data transformation, see Passing Data to a Data Transformation. |
| Routing Rule | **delete** | When evaluated by a routing engine business process, this action deletes the current message. |
| Routing Rule | **delegate** | When evaluated by a routing engine business process, this action delegates the message to a different rule. |

The **send**, **delete**, and **delegate** actions should not be used within a BPL <rule>. If you include them, the action will not be executed and instead a string value will be returned that includes the given action.

You must ensure that you construct rule sets such that they are logically sound and result in the rule set being executed as you intended. For example, while it might make sense to set a default return value if none of the rules in a rule set are executed, it does not make sense to do so if you have created the rule set such that one rule is always executed. Typically, most actions reside in the **if** clauses of rules.

## 3.2.1 Using the foreach Action

The **foreach** action allows you to loop through a repeating segment and reference any of the fields within the segment.

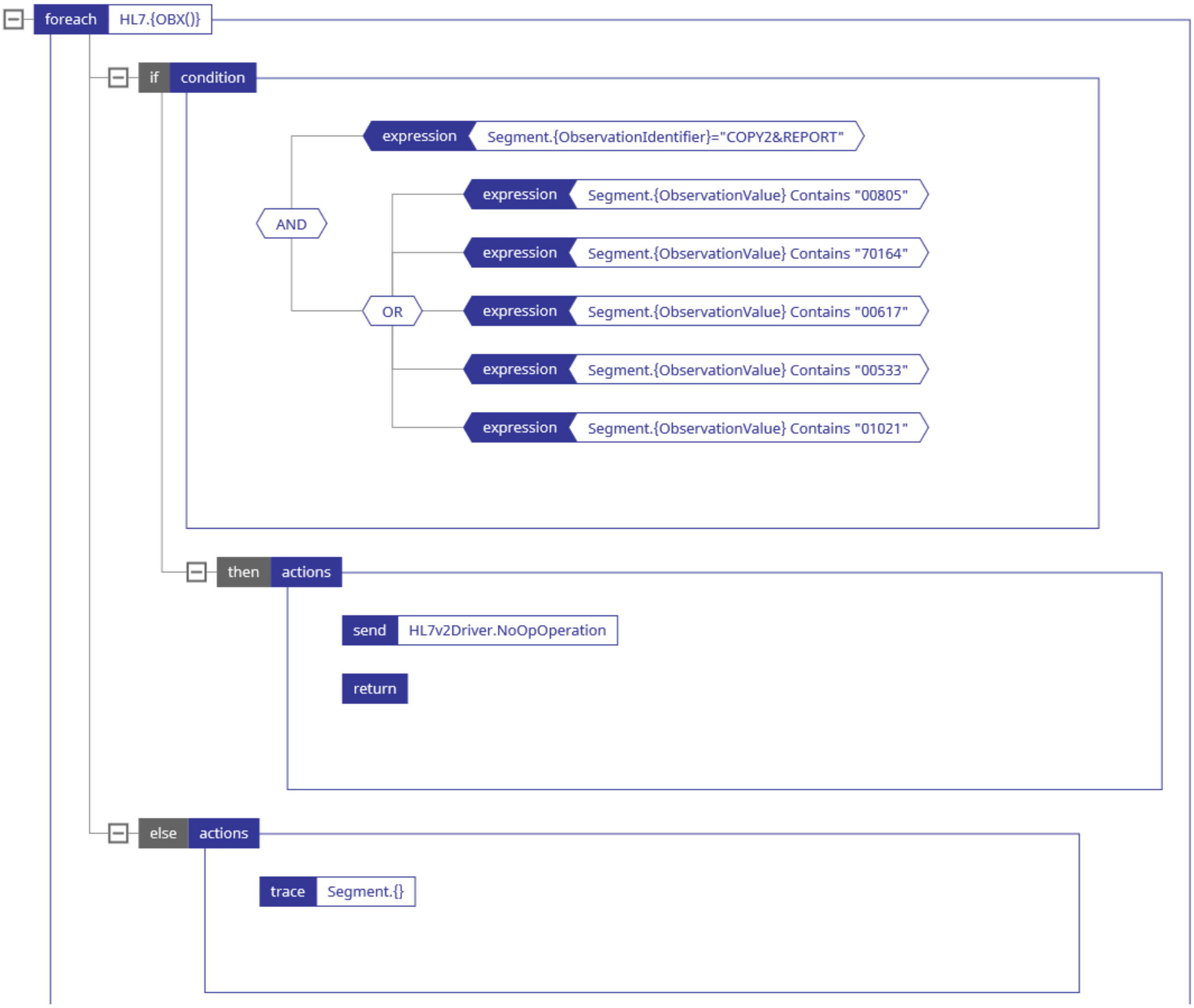
You specify the repeating segment in the **propertypath** property of the **foreach** action using the syntax described in Virtual Property Path Basics. For example, to access the `OBX` segments in the repeating `OBXgrp` of an HL7 document, you can specify `HL7.{OBXgrp().OBX}`, where the empty parentheses indicate the repeating group. A **foreach** action can contain one or more **if** clauses and an **else** clause. Within the clauses, you specify actions to execute when the conditions in the clauses hold true.

For example, you can use a **foreach** action to determine when a field in a repeating segment contains a particular value, and then specify a **send** action to route a message when the value is present. To reference the specific field, you can use `Segment.{<field-name>}` — for example: `Segment.{ObservationIdentifier}`.

The **if** and **else** clauses in a **foreach** action can contain one or more **rule** nodes. However, you cannot nest **foreach** actions.

When the rule executes a **return** action within a **foreach** loop, it exits the entire rule set, not just the loop or rule.

The following example shows the use of a **foreach** action within a business rule. The action iterates through a repeating `OBX` segment in an HL7 document to determine when the `ObservationIdentifier` field contains certain string values. When the values are found, the rule sends the document to a file operation. When the values are not found, the rule logs an entry in the Event Log using a trace action.


foreach
HL7.{OBX()}
if
condition
expression
Segment.{ObservationIdentifier}="COPY2&REPORT"
AND
OR
expression
Segment.{ObservationValue} Contains "00805"
expression
Segment.{ObservationValue} Contains "70164"
expression
Segment.{ObservationValue} Contains "00617"
expression
Segment.{ObservationValue} Contains "00533"
expression
Segment.{ObservationValue} Contains "01021"
then
actions
send
HL7v2Driver.NoOpOperation
return
else
actions
trace
Segment.{}

# 3.3 Defining Expressions

You can use expressions when modifying the values of four properties:

- condition property of an **if** clause— Specifies the condition for executing the logic in the **if** clause. For more information, see Editing the Condition Property of an if Clause.
- value property of an **assign** action — Specifies the value to assign
- value property of a **return** action in a general business rule — Specifies the value to return to the process that executed the rule set
- value property of a **trace** action — Specifies the text to include in a trace message. You can specify a literal text string or an expression to be evaluated. Expressions must use the scripting language specified in the *language* attribute of the corresponding <process> element.

You can set each property to one of the following supported values:

- A numeric value (integer or decimal), such as 1.1 or 23000.
- A string value enclosed in double quotes, for example:

  ```
  "NY"
  ```

  **Important:** The double quotes are *required*.

- A value of a context property. Recall that a BPL business process can contain a general-purpose, persistent variable called *context*. You define the *context* variable using the `<context>` and `<property>` elements of BPL. You can access the properties of the `context` object from anywhere in the business process. Therefore, if you invoke a rule from a business process using the `<rule>` element, you can access the context properties from within the rule.

  For context properties that contain collections such as lists and arrays, InterSystems IRIS supports several retrieval methods from within business rules, including **Count()**, **Find()**, **GetAt()**, **GetNext()**, **GetPrevious()**, **IsDefined()**, **Next()**, and **Previous()**. For more information, see Working with Collections.

  **Note:** Property names are case-sensitive and must *not* be enclosed in quotes, for example, PlaceOfBirth.

- An expression using supported operators, literal values, properties of the general-purpose, and persistent variable *context*for example:

  ```
  ((2+2)*5)/154.3
  "hello" & "world"
  Age * 4
  (((x=1) || (x=3)) && (y=2))
  ```

- A built-in function such as **Min()**, **Max()**, **Round(n,m)**, or **SubString()**. The function name must include parentheses. It must also include any input parameters, such as the numeric values *n* and *m* for **Round**. If there are no input values for the function, then the open and close parentheses must be present and empty.
- The *Document* variable, which represents the message object.
- Within a **foreach** action, a segment. For more information, see About Actions.

For examples, see Expression Examples.

## 3.3.1 Editing the Condition Property of an if Clause

In a rule definition, a *condition* consists of two values and a comparison operator between the values, for example:

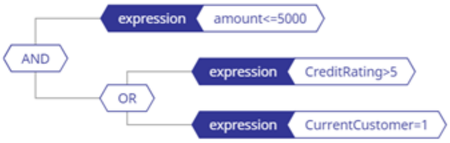
```
Amount  <=  5000
```

If a **condition** is not `true`, it is `false`. There are no other possible values for the **condition** property. A result that may be only true or false is called a *boolean* result. InterSystems IRIS stores boolean results as integer values, where `1` is true and `0` is false. In most cases, you do not need to use this internal representation. However, for a routing rule, you may want to execute the **if** clause that corresponds to the **condition** property any time the **constraint** for the rule holds true. In this case, you can set the **condition** property to `1`.

A **condition** property can contain more than one condition. InterSystems IRIS evaluates and compares all the conditions in the property before determining whether to execute the corresponding rule. The logic between conditions is determined by `AND` or `OR` operators. For example, consider a **condition** property with the following value:

```
IF  Amount  <=  5000
AND  CreditRating > 5
OR  CurrentCustomer = 1
```

The same value appears in the **Rule Editor** as follows



The value contains three conditions: Amount <= 5000, CreditRating > 5, CurrentCustomer = 1. Each condition could be true or false. InterSystems IRIS evaluates the conditions individually before evaluating the relationships between them defined by the `AND` and `OR` operators.

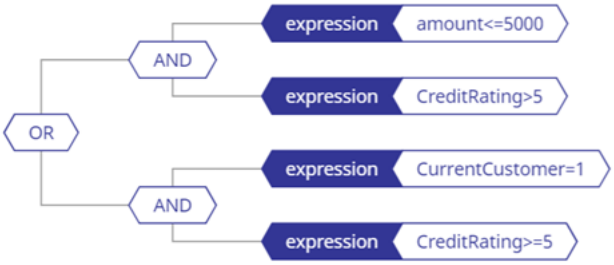
The `AND` and `OR` operators can operate only on true and false values. That is, the operators must be positioned between two boolean values and return a single boolean result as follows:

| Operator | Result is true when... |
|---|---|
| AND | Both values are true. |
| OR | At least one of the values is true, or both are true. If one of the values is false and the other is true, then the result (as a whole) is still true. |

If a **condition** property contains multiple `AND` or `OR` operators, the `AND` operators take precedence over the `OR` operators. Specifically, all the `AND` operations are performed first. Then, the `OR` operations are performed. For example, consider the following set of conditions:

```
IF  Amount  <=  5000
AND  CreditRating > 5
OR  CurrentCustomer = 1
AND  CreditRating >= 5
```

The same set of conditions appears in the **Rule Editor** as follows:



InterSystems IRIS evaluates the conditions as follows:

```
IF  (Amount  <=  5000  AND  CreditRating > 5)
OR  (CurrentCustomer = 1  AND  CreditRating >= 5)
```

That is, the whole set of conditions is true if either or both of the following statements is true:

- Someone requests an amount less than 5,000 *and* has a credit rating better than average.
- A current bank customer requests any amount *and* has a credit rating greater than or equal to the average.

If both statements are false, then the set of conditions (as a whole) is false.

To explain another way, InterSystems IRIS evaluates the set of conditions by taking the following steps:

1. Determine whether the result of the following `AND` expression is true or false:

   ```
   IF  Amount  <=  5000
   AND  CreditRating > 5
   ```

   Suppose this result is called “SafeBet.”

2. Determine whether the result of the following `AND` expression is true or false:

   ```
   IF  CurrentCustomer = 1
   AND  CreditRating >= 5
   ```

   Suppose this result is called “KnownEntity.”

3. Determine whether the result of the following `OR` expression is true or false:

   ```
   IF  SafeBet is true
   OR  KnownEntity is true
   ```

   If SafeBet is true and KnownEntity is false, then the set of conditions is true. Similarly, if SafeBet is false and KnownEntity is true, then the set of conditions is true. Lastly, if both SafeBet and KnownEntity are true, then the set of conditions is true.

## 3.3.2 Expression Operators

When defining an expression, you can select one of the follow arithmetic operators:

| Operator | Meaning |
|---|---|
| + | Plus (binary and unary) |
| – | Minus (binary and unary) |
| * | Times |
| / | Divide |

Additionally, the following logical operators are supported and return an integer value of 1 (true) or 0 (false):

| Operator | Meaning | Expression is true when... |
|---|---|---|
| AND (&&) | And | Both values are true. |
| OR (\|\|) | Or | At least one of the values is true. Both values may be true, or only one true. |
| ! | Not (unary) | The value is false. |
| = | Equals | The two values are equal. |
| != | Does not equal | The two values are *not* equal. |
| > | Is greater than | The value to the left of the operator is *greater* than the value to the right of the operator. |

| Operator | Meaning | Expression is true when... |
|---|---|---|
| < | Is less than | The value to the left is *less* than the value to the right. |
| >= | Is greater than or equal to | The value to the left is greater than the value to the right, *or* if the two values are equal. |
| <= | Is less than or equal to | The value to the left is less than the value to the right, *or* if the two values are equal. |
| [ | Contains | The string contains the substring to the right. Pattern matching for Contains is exact. If the value at left is “Hollywood, California” and the value at right is “od, Ca”, there is a match, but a value of “Wood” does not match. |

Lastly, you can use the following string operators:

| Operator | Meaning |
|---|---|
| & | Concatenation operator for strings. |
| _ | Binary concatenation to combine string literals, expressions, and variables. |

When more than one operator is found in an expression, the operators are evaluated in the following order of precedence, from first to last:

1. Any of the following logical operators: `! = != < > <= >= [`
2. Multiplication and division: `* /`
3. Addition and subtraction: `+ -`
4. String concatenation: `& _`
5. Logical `AND`: `&&`
6. Logical `OR`: `||`

## 3.3.3 Expression Functions

Within a rule definition, an expression can include a call to one of the InterSystems IRIS *utility functions*. These include mathematical or string processing functions similar to those that exist in other programming languages. When defining the expression, simply select a function from the drop-down list.

For a list of the available utility functions and the proper syntax for using them in business rules or DTL data transformations, see Utility Functions for Use in Productions.

## 3.3.4 Expression Examples

Within a rule definition, an *expression* is a formula for combining values and properties to return a value. The following table includes examples of expressions along with their computed values:

| Expression | Computed value |
|---|---|
| `((2+2)*5)/154.3` | `0.129617628` |
| `"hello" & "world"` | `"helloworld"` |

| Expression | Computed value |
|---|---|
| `Age * 4` | If `Age` is a *context* property (a property in the general-purpose, persistent *context* variable, which you can define using the **<context>** and **<property>** elements in BPL) and has the numeric value `30`, the value of this expression is `120`. |
| `1+2.5*2` | `6` |
| `2*5` | `10` |
| `Min(Age,80,Limit)` | This expression uses the built-in **Min()**function . If `Age` is a *context* property with the value `30` and `Limit` (likewise a property) has the value `65`, the value of this expression is `30`. |
| `Round(1/3,2)` | This expression uses the built-in **Round()** function. The result is `0.33`. |
| `x<65&&A="F"\|\|x>80` | This expression uses the operator precedence conventions that are described inExpression Operators). If `A` is a *context* property with the string value `F`, and `x` (likewise a property) has the integer value `38`, this expression has the integer value `1`. In InterSystems IRIS, an integer value of `1` is true and an integer value of `0` means false. |
| `Min(10,Max(X,Y))` | This expression uses the **Min()** and **Max()** functions. If `X` is a *context* property with the numeric value `9.125`, and `Y` (likewise a property) has the numeric value `6.875`, the value of this expression is `9.125`. |
| `(((x=1) \|\| (x=3)) && (y=2))` | This expression uses parentheses to clarify precedence in a complex logical relationship. |

When you select a property that takes an expression as its value, a blank text field appears at the top of the rule set diagram. You must ensure that you use the appropriate syntax for the property since the text field enables you to specify any string. Consider the following rules when you formulate an expression:

- An expression can include the values described in previous sections: numbers, strings, *context* properties, other expressions, functions, or any valid combination of these.
- White spaces in expressions are ignored.
- You can use any of the supported operators in an expression.
- If you want to override the default operator precedence, or if you want to make an expression easier to read, you can use parentheses to group parts of the expression and indicate precedence. For example, consider the following expression, which results in a value of `6`:

  ```
  1+2.5*2
  ```

  If you change the expression as follows, the result becomes `7`:

  ```
  (1+2.5)*2
  ```

- Business rules support parentheses to group complex logical expressions such as `(((x=1) || (x=3)) && (y=2))`.

## 3.4 Defining Constraints

If the rule set contains routing rules, you can define constraints such that when a message makes its way through the rule set, the rule logic is executed only if the message matches the defined constraints. Leaving a field blank will match all values. To set constraints for a rule, double-click the rule and define the following settings:

### Source

Configuration name of *one* of the following items:

- A business service (for a routing interface)
- A message routing process (if another rule chains to this routing rule set)

### Message Class

Identifies the production message object that is being routed by this rule. The value of this field depends on the routing rule type:

- *For a General Message Routing Rule*, you can click the ellipsis (**...**) next to the **Message Class** field to invoke the **Finder Dialog** and select the appropriate message class. You can choose the category of message class to narrow your choices.
- *For a Virtual Document Message Routing Rule*, you can choose from the list of defined virtual document classes.

### Schema Category

For virtual document routing rules, identifies the category of the message class and specifies its structure. You can choose from the list of category types defined for your chosen virtual document class. The types may be built-in or imported from a custom schema.

### Document Name

For virtual document routing rules, identifies the message structure. The acceptable values depend on the message class. You can choose from the list of category types defined for your chosen virtual document class. The types may be built-in or imported from a custom schema.

If you specify more than one value in the **Document Name** field, the rule matches any of the specified **Document Name** values and no others.

## 3.5 Disabling a Rule

If you would like to prevent a rule set from executing a rule, but do not want to delete the rule, you can disable it. Simply double-click the rule, and select **Disable**.

## 3.6 Passing Data to a Data Transformation

A Send action can invoke a data transformation before sending the message to a target within the production. This data transformation can use its `aux` variable to obtain information from the rule. Some of this data, for example the name of the rule and the reason the rule was fired, is available to the transformation without making changes to the rule class.

In order to pass additional information to the transformation, you need to edit the rule class in an IDE to assign values to properties of the class. A value assigned to the `RuleUserData` property of the rule class is available to the transformation if it accesses the `aux.RuleUserData` variable. A value assigned to the `RuleActionUserData` property of the rule class is available to the transformation as `aux.RuleActionUserData`.

For additional information about accessing the `aux` variable in a transformation, see the list of valid expressions in DTL Syntax Rules.

## 3.7 Adding Business Rule Notification

InterSystems IRIS enables you to set up rule notifications so that the system takes specific actions each time a rule is executed. Unlike most activities related to rules, setting up notifications requires programming. You must create a subclass of the Ens.Rule.Notification class and override the **%OnNotify** method of the subclass. The signature of **%OnNotify** method is as follows:

```
ClassMethod %OnNotify(pReason As %String,
                      pRule As Ens.Rule.RuleDefinition)
                      As %Status
```

Possible *pReason* values are:

- `BeforeSave`
- `AfterSave`
- `Delete`

At runtime, the production framework automatically finds your subclass of Ens.Rule.Notification and uses the code in **%OnNotify** to determine what to do upon executing a rule.

# 4

# Debugging Routing Rules

This topic describes how to test routing rules without sending the message through the entire production. It also contains flow diagrams that can help you debug problems in routing rules defined for EDI messages in a production.

## 4.1 Testing Routing Rules

Using the **Test** button of the Rule Editor, you can see whether a message triggers any of the routing rules without having to send the message through the entire production. Running this test does not transform or send the message, but any functions in the condition are executed as if the message ran through the production.

If you want to test a rule's constraint that is based on the source of the message, use the **Production Source** field to specify the business host in the production that is sending the message. You can use the drop down menu to choose the business host from a list.

You can use the **Context** field to specify the contents of the message in one of three ways:

- Specify User Input and then click next to paste the raw text of a message.
- Specify the Document Body ID of an existing message. You can find the Document Body ID for a message by looking at the **<Object Id>** field on the **Body** tab of the Message Viewer.
- Specify the Message Header ID of an existing message. If the **Production Source** field is blank, the Source Config Name in the message header is used as the source. You can find the Message Header ID for a message by looking at the **<Object Id>** field on the **Header** tab of the Message Viewer.

### 4.1.1 Testing with Raw Text of a Message

You can test a routing rule by pasting the raw text of a message into the test tool. To do so:

1. Optionally, in the **Production Source** field, enter the business host that is sending the message.
2. Choose **User Input** from the **Context** drop-down list.
3. Click the **Next** button.
4. Paste the raw text of the message in the **HL7 Document Content** text field.
5. Enter any other constraints that you want to test by entering information in the **DocType** field or by selecting from the **Category** or **Name** drop downs.
6. Click **Submit**.

### 4.1.2 Test Results

The test results will tell you whether the message met a rule’s constraint and whether an if or else clause was triggered.

### 4.1.3 Security Requirements

A user must have the correct security privileges to test routing rules. They must have USE permissions for `%Ens_RuleLog` and `%Ens_TestingService`. In addition, they must have Select SQL privileges on `Ens_Rule.log` and `Ens_Rule.DebugLog` tables.

## 4.2 Strategies for Debugging Routing Rules

This section describes strategies for debugging the routing rules in an EDI message routing production.

The primary symptom for problems in routing rules is that the message does not reach its destination. Perhaps the message reaches a point along the way, such as a business operation or routing process within the routing production, but it does not reach its target destination, which is generally an application server outside InterSystems IRIS®. In that case you can follow the problem-solving sequence captured in the next four drawings: “Solving Problems with Routing Rules,” Drawings A, B, C, and D.

***Figure 4–1: Solving Problems with Routing Rules (Drawing A)***

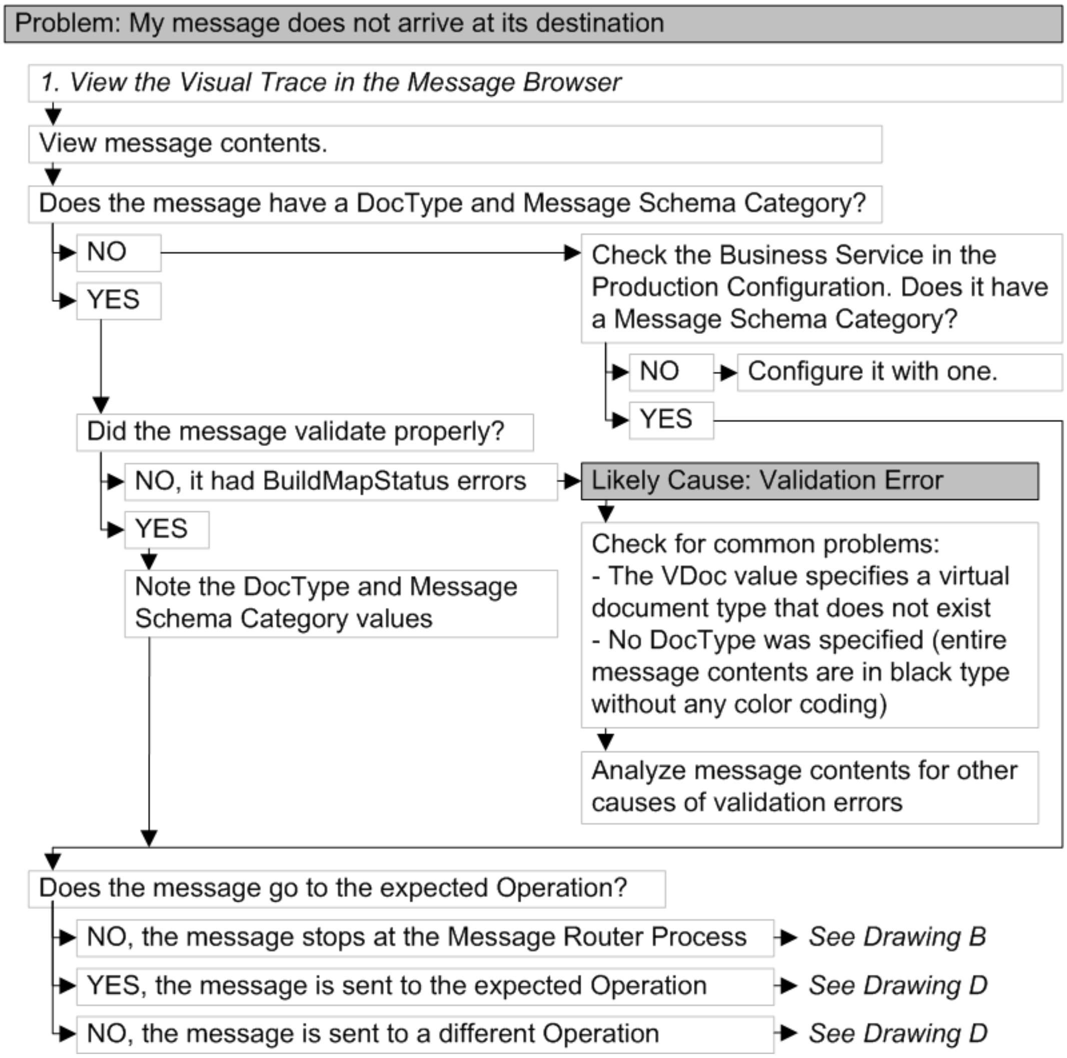
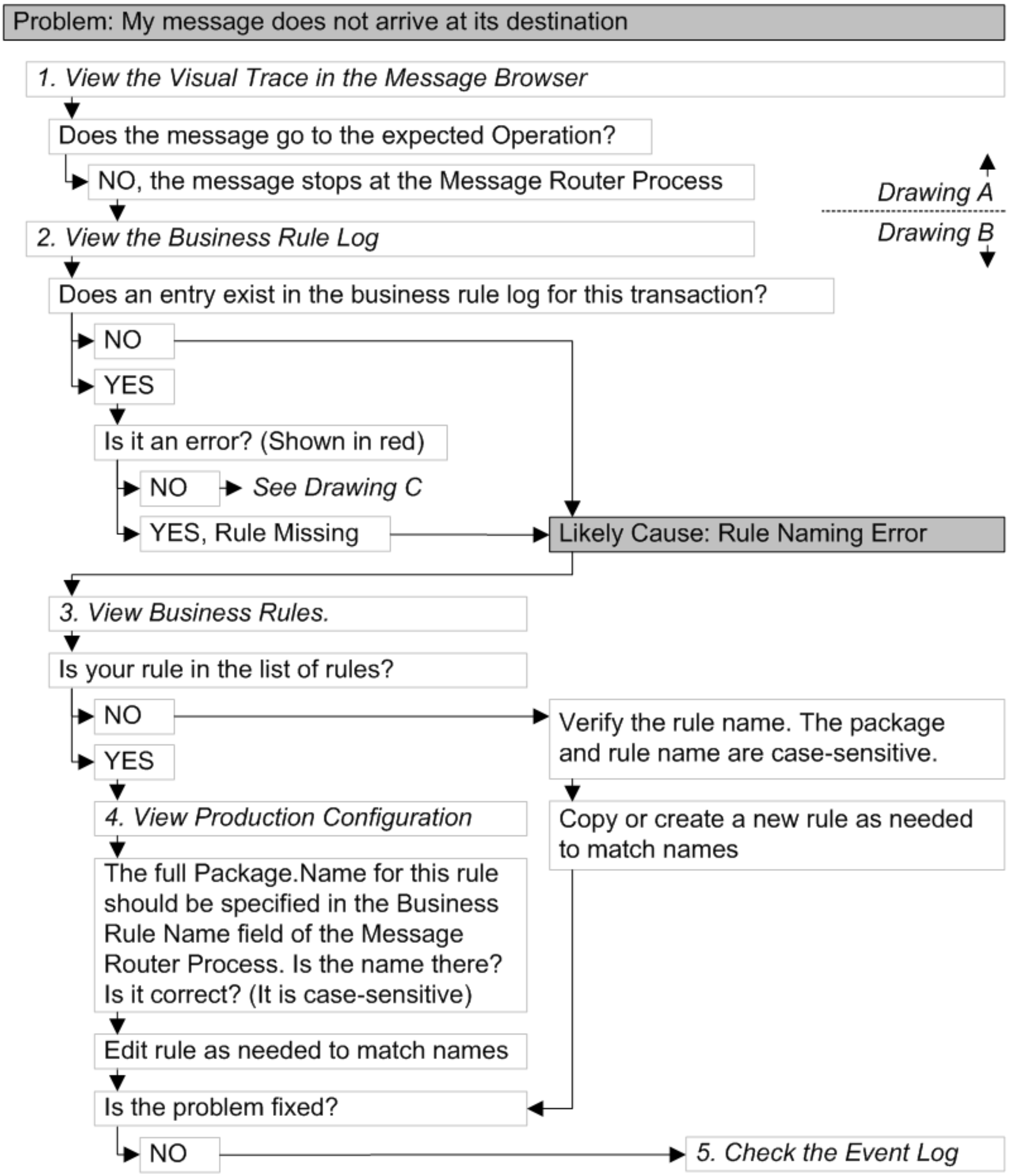

Problem: My message does not arrive at its destination
1. View the Visual Trace in the Message Browser
View message contents.
Does the message have a DocType and Message Schema Category?
NO
YES
Check the Business Service in the Production Configuration. Does it have a Message Schema Category?
NO
Configure it with one.
YES
Did the message validate properly?
NO, it had BuildMapStatus errors
Likely Cause: Validation Error
YES
Note the DocType and Message Schema Category values
Check for common problems:
- The VDoc value specifies a virtual document type that does not exist
- No DocType was specified (entire message contents are in black type without any color coding)
Analyze message contents for other causes of validation errors
Does the message go to the expected Operation?
NO, the message stops at the Message Router Process
See Drawing B
YES, the message is sent to the expected Operation
See Drawing D
NO, the message is sent to a different Operation
See Drawing D

*Figure 4–2: Solving Problems with Routing Rules (Drawing B)*



Problem: My message does not arrive at its destination

*1. View the Visual Trace in the Message Browser*

Does the message go to the expected Operation?

NO, the message stops at the Message Router Process

*2. View the Business Rule Log*

Does an entry exist in the business rule log for this transaction?

NO

YES

Is it an error? (Shown in red)

NO

*See Drawing C*

YES, Rule Missing

Likely Cause: Rule Naming Error

*3. View Business Rules.*

Is your rule in the list of rules?

NO

YES

Verify the rule name. The package and rule name are case-sensitive.

Copy or create a new rule as needed to match names

*4. View Production Configuration*

The full Package.Name for this rule should be specified in the Business Rule Name field of the Message Router Process. Is the name there? Is it correct? (It is case-sensitive)

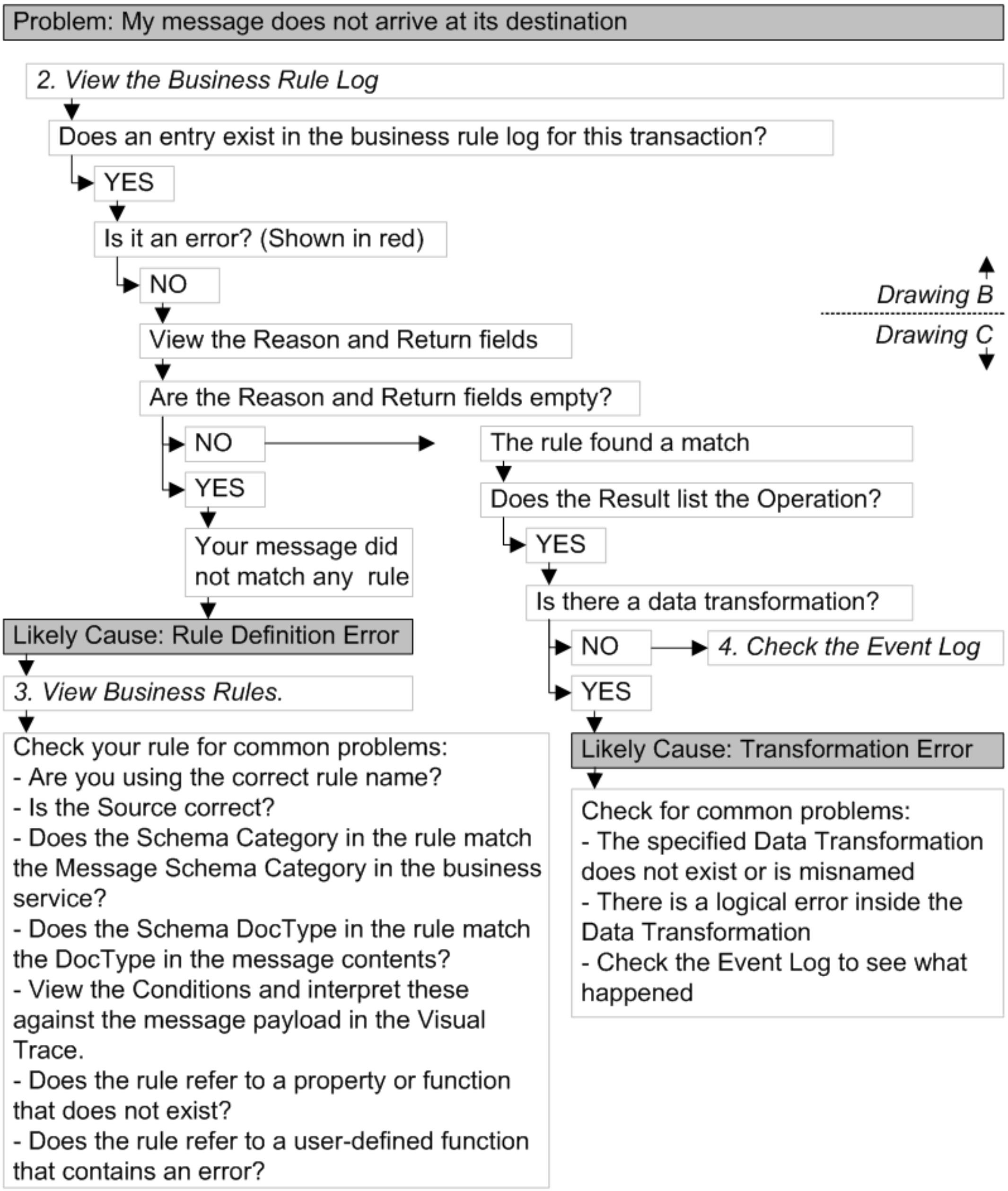
Edit rule as needed to match names

Is the problem fixed?

NO

*5. Check the Event Log*

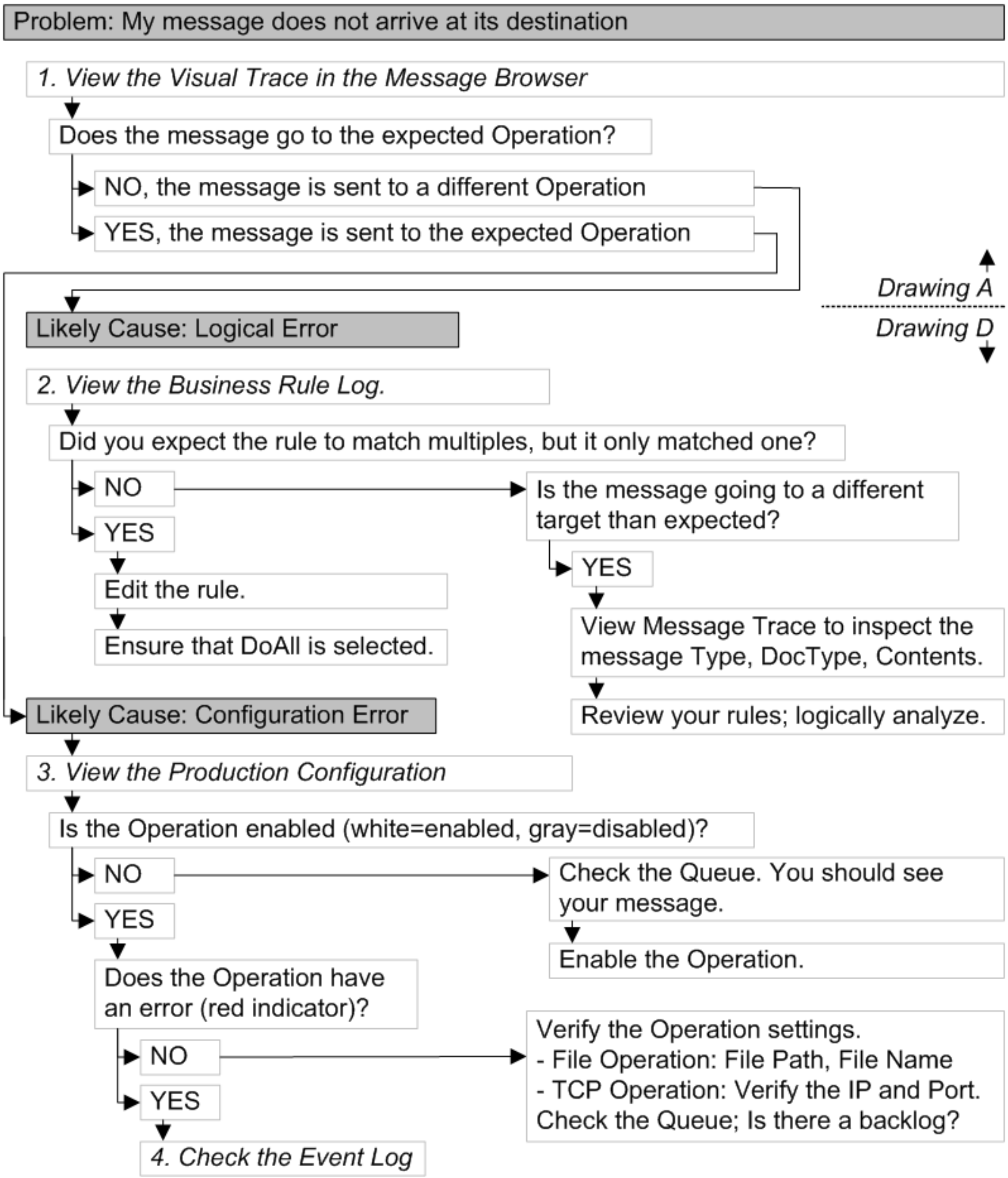
*Figure 4–3: Solving Problems with Routing Rules (Drawing C)*


Problem: My message does not arrive at its destination
2. View the Business Rule Log
Does an entry exist in the business rule log for this transaction?
YES
Is it an error? (Shown in red)
NO
Drawing B
Drawing C
View the Reason and Return fields
Are the Reason and Return fields empty?
NO
The rule found a match
YES
Does the Result list the Operation?
Your message did not match any rule
YES
Is there a data transformation?
Likely Cause: Rule Definition Error
NO
4. Check the Event Log
3. View Business Rules.
YES
Likely Cause: Transformation Error

Check your rule for common problems:
- Are you using the correct rule name?
- Is the Source correct?
- Does the Schema Category in the rule match the Message Schema Category in the business service?
- Does the Schema DocType in the rule match the DocType in the message contents?
- View the Conditions and interpret these against the message payload in the Visual Trace.
- Does the rule refer to a property or function that does not exist?
- Does the rule refer to a user-defined function that contains an error?

Check for common problems:
- The specified Data Transformation does not exist or is misnamed
- There is a logical error inside the Data Transformation
- Check the Event Log to see what happened

***Figure 4–4: Solving Problems with Routing Rules (Drawing D)***


Problem: My message does not arrive at its destination
1. View the Visual Trace in the Message Browser
Does the message go to the expected Operation?
NO, the message is sent to a different Operation
YES, the message is sent to the expected Operation
Drawing A
Drawing D
Likely Cause: Logical Error
2. View the Business Rule Log.
Did you expect the rule to match multiples, but it only matched one?
NO
YES
Is the message going to a different target than expected?
YES
Edit the rule.
Ensure that DoAll is selected.
View Message Trace to inspect the message Type, DocType, Contents.
Review your rules; logically analyze.
Likely Cause: Configuration Error
3. View the Production Configuration
Is the Operation enabled (white=enabled, gray=disabled)?
NO
YES
Check the Queue. You should see your message.
Enable the Operation.
Does the Operation have an error (red indicator)?
NO
YES
Verify the Operation settings.
- File Operation: File Path, File Name
- TCP Operation: Verify the IP and Port.
Check the Queue; Is there a backlog?
4. Check the Event Log

# A

# Utility Functions for Use in Productions

This topic describes the InterSystems IRIS® *utility functions* that you can use in business rules and DTL data transformations. These include mathematical or string processing functions such as you may be accustomed to using in other programming languages.

To define your own functions, see Defining Custom Utility Functions.

## A.1 Built-in Functions

The following lists the utility functions built into InterSystems IRIS.

**Note:** For boolean values, 1 indicates true and 0 indicates false.

### Contains(*val*,*str*)

Returns 1 (true) if *val* contains the substring *str*; otherwise 0 (false).

### ConvertDateTime (*val*,*in*,*out*,*file*)

Reads the input string *val* as a time stamp in *in* format, and returns the same value converted to a time stamp in *out* format. See Time Stamp Specifications for Filenames.

The default for *in* and *out* is %Q. Any %f elements in the *out* argument are replaced with the *file* string. If *val* does not match the *in* format, *out* is ignored and *val* is returned unchanged.

### CurrentDateTime(*format*)

Returns a string representing a date/time value in the given format. For a list of possible formats, see the Date and Time Expansion section of the class reference for the **FormatDateTime** method. For example, `CurrentDateTime("%H")` returns the current hour in 24–hour format as a 2–digit number. The default format is ODBC format (%Q) in the server's local timezone.

### DoesNotContain(*val*,*str*)

Returns 1 (true) if *val* does not contain the substring *str*.

### DoesNotIntersectList(*val*,*items*,*srcsep*,*targetsep*)

Returns 1 (true) if no item in the given source list (*val*) appears in the target list (*items*). For details on the arguments, see IntersectsList.

### DoesNotMatch(*val*,*pat*)

Returns 1 (true) if *val* does not match the pattern specified by *pat*. *pat* must be a string that uses syntax suitable for the ? pattern matching operator in ObjectScript. For details, see Pattern Match Operator.

### DoesNotStartWith(*val*,*str*)

Returns 1 (true) if *val* does not start with the substring *str*.

### Exists(*tab*,*val*)

The Exists function provides a way to predict the results of the Lookup function. Exists returns 1 (true) if *val* is a key defined within the table identified by *tab*; otherwise it returns 0 (false).

The *tab* value must be enclosed in double quotes, for example:

```
Exists("Alert","Priority_FileOperation")
```

### If(*val*,*true*,*false*)

If the argument *val* evaluates to 1 (true), the If function returns the string value of its *true* argument; otherwise it returns the string value of its *false* argument.

### In(*val*,*items*)

Returns 1 (true) if *val* is found in the comma-delimited string *items*.

### InFile(*val*,*file*)

Returns 1 (true) if *val* is found in the identified *file*.

### InFileColumn(*...*)

The function InFileColumn can have as many as 8 arguments. The full function signature is:

*InFileColumn(val*, *file*, *columnId*, *rowSeparator*, *columnSeparator*, *columnWidth*, *lineComment*, *stripPadChars*)

InFileColumn returns 1 (true) if *val* is in the specified column in a table-formatted text *file*. Arguments are as follows:

- *val* (required) is the value.
- *file* (required) is the text file.
- Default *columnId* is 1.
- Default *rowSeparator* is ASCII 10. A negative *rowSeparator* value indicates the row length.
- Default *columnSeparator* is ASCII 9. If *columnSeparator* is 0, the format of the file is said to be “positional.” In this case *columnId* means character position and *columnWidth* means character count.
- Default *columnWidth* is 0.
- Default *lineComment* is an empty string.
- Default *stripPadChars* consists of a blank space followed by ASCII 9.

### IntersectsList(*val*,*items*,*srcsep*,*targetsep*)

Returns 1 (true) if any item in the given source list (*val*) appears in the target list (*items*). The arguments *srcsep* and *targetsep* specify the list separators in the source and target lists respectively; for each of these, the default is `"><"`, which means that the lists are assumed to have the form `"<item1><item2><item3>"`.

The `IntersectsList` utility works well with the square bracket [ ] syntax to match values of a virtual document property. If there is more than one instance of the segment type in a message, the square bracket syntax returns the multiple values in a string like `<ValueA><ValueB><ValueC>`.

If the target list has only a single item, this function is essentially the same as the Contains function. If the source list has only a single item, this function is essentially the same as the In function.

### Length(*string*,*delimiter*)

Returns the length of the given string. If you specify *delimiter*, this function returns the number of substrings based on this delimiter.

### Like(*string*,*pattern*)

Returns 1 (true) if the given value (*string*) satisfies a SQL Like comparison with the given pattern string (*pattern*). In SQL Like patterns, % matches 0 or more characters, and _ matches any single character. Note that an escape character can be specified by appending "%%" to the pattern, e.g. "#%SYSVAR_#_%%%#" to match any value string that starts with "%SYSVAR" followed by any single character, an underscore, and anything else.

### Lookup(*table*,*keyvalue*,*default*, *defaultOnEmptyInput*)

The **Lookup()** function searches for the key value specified by *keyvalue* in the table specified by *table* and returns its associated value. This returned value is equivalent to the following global:

`^Ens.LookupTable(`*table*`,`*keyvalue*`)`

The *table* value must be enclosed in double quotes, for example:

```
Lookup("Gender",source.{PID:Sex},,"U")
```

If the key is not found in the table, the Lookup function returns the default value specified by the *default* parameter. The *default* parameter is optional, so if it is not specified and Lookup does not find a matching key, it returns an empty string. An exception is that if either the key value or the lookup table is empty, the **Lookup()** function returns either the default value or the empty string depending on the value of the *defaultOnEmptyInput* parameter as described in the following table. The default value of the *defaultOnEmptyInput* parameter is 0.

| defaultOnEmptyInput Value | key value and lookup table | Lookup() returns |
|---|---|---|
| 0 | either key value or lookup table is empty | empty string |
| 1 | key value is empty | empty string |
| | lookup table is empty but key value is not | default value |
| 2 | lookup table is empty | empty string |
| | key value is empty but lookup table is not | default value |
| 3 | either key value or lookup table is empty | default value |

The **Exists()** function returns true if a **Lookup()** function with the same parameters would find the key value in the lookup table.

### Matches(*val*,*pat*)

Returns 1 (true) if *val* matches the pattern specified by *pat*. *pat* must be a string that uses syntax suitable for the ? pattern matching operator in ObjectScript. For details, see Pattern Match Operator.

### Max(*...*)

Returns the largest of a list of up to 8 values. List entries are separated by commas.

### Min(*...*)

Returns the smallest of a list of up to 8 values. List entries are separated by commas.

### Not(*val*)

Returns 0 (false) if *val* is 1 (true); 1 (true) if *val* is 0 (false).

### NotIn(*val*,*items*)

Returns 1 (true) if *val* is not found in the comma-delimited string *items*.

### NotInFile(*val*,*file*)

Returns 1 (true) if *val* is not found in the identified *file*.

### NotLike(*string*,*pattern*)

Returns 1 (true) if the given value (*string*) does not satisfy a SQL Like comparison with the given pattern string (*pattern*). See Like.

### Pad(*val*,*width*,*char*)

Reads the input string *val*. Adds enough instances of *char* to widen the string to *width* characters. If *width* is a positive value, the padding is appended to the right-hand side of the *val* string. If *width* is a negative value, the padding is prepended to the left-hand side of the *val* string.

### Piece(*val*,*char*,*from*,*to*)

If the delimiter character *char* is present in the string *val*, this separates the string into pieces. If there are multiple pieces in the string, *from* and *to* specify which range of these pieces to return, starting at 1. If multiple pieces are returned, the delimiter in the return string is the same as the delimiter in the input string. For example:

*Piece("A,B,C,D,E,F")* returns "A"

*Piece("A!B!C!D!E!F","!",2,4)* returns "B!C!D"

The default *char* is a comma, the default *from* is 1, and the default *to* is *from* (return one piece). For details, see $PIECE.

### ReplaceStr(*val*,*find*,*repl*)

Reads the input string *val*. Replaces any occurrences of string *find* with the string *repl*, and returns the resulting string.

**Note:** Use ReplaceStr instead of the Replace function, which has been deprecated.

### RegexMatch(*string*,*regex*)

Given an input string *val* and a regular expression *regex*, returns 1 if *val* matches the regular expression; returns 0 otherwise.

**Round(*val*,*n*)**

Returns *val* rounded off to *n* digits after the decimal point. If *n* is not provided (that is, *Round(val)*) the function drops the fractional portion of the number and rounds it to the decimal point, producing an integer.

**Rule(*rulename*,*context*,*activity*)**

Evaluates the rule specified in the *rulename* with the given *context* object and the given *activity* label for the Rule Log and returns the value.

**Schedule(*ScheduleSpec*, *ODBCDateTime*)**

Evaluates the state of the given *ScheduleSpec* string, named Schedule or Rule at the moment given by *ODBCDateTime*. If *ScheduleSpec* begins with '@' it is a Schedule name or Rule name, otherwise a raw Schedule string. If *ODBCDateTime* is blank, the evaluation is done for the current time.

**StartsWith(*val*,*str*)**

Returns 1 (true) if *val* starts with the substring *str*.

**Strip(*val*,*act*,*rem*,*keep*)**

Reads the input string *val*. Removes any characters matching the categories specified in the *act* template and the *rem* string, while retaining any characters found in the *keep* string. Returns the resulting string. For details and examples, see $ZSTRIP.

**SubString(*str*,*n*,*m*)**

Returns a substring of a string *str*, starting at numeric position *n* and continuing until numeric position *m*. The number 1 indicates the first character in the string. If *m* is not provided (that is, *SubString(str,n)*) the function returns the substring from position *n* to the end of the string.

**ToLower(*str*)**

Returns the string *str* converted to lowercase.

**ToUpper(*str*)**

Returns the string *str* converted to uppercase.

**Translate(*val*,*in*,*out*)**

Reads the input string *val*. Translates each occurrence of a character in string *in* to the character at the corresponding position in string *out*, and returns the resulting string.

**Note:** These functions are defined by methods in the class Ens.Util.FunctionSet.

## A.2 Syntax to Invoke a Function

When you reference a function in a business rule or a DTL data transformation, the syntax must include parentheses. It must also include any input parameters, such as the numeric values for the mathematical functions Min, Max, or Round. If there are no input values for the function, then the open and close parentheses must be present, but empty.

The following are examples of valid function syntax:

| Expression | Computed Value |
|---|---|
| `Min(Age,80,Limit)` | When `Age` is a property with the value `30` and `Limit` (likewise a property) has the value `65`, the value of this expression is `30`. |
| `Round(1/3,2)` | `0.33` |
| `Min(10,Max(X,Y))` | When `X` is a property with the numeric value `9.125`, and `Y` (likewise a property) has the numeric value `6.875`, the value of this expression is `9.125`. |

If the value input to any function is a string that starts with a number, nonnumeric characters in the string are dropped and the numeric portion is used. The string `"3a"` is treated like the number `3`, so the function `Min("3a","2OfThem")` returns the value `2`. A string that begins with a nonnumeric character such as `"a123"` has the numeric value `0`.

The business rule syntax for utility functions differs from the DTL syntax in the following significant way:

- Business rules reference the utility functions simply by name:

  `ToUpper(value)`

- DTL uses two leading dots immediately before the function name, as if the function were a method:

  `..ToUpper(value)`

The following DTL data transformation uses a utility function called **ToUpper()** to convert a string to all uppercase characters. The <assign> statement references **ToUpper()** using double-dot syntax, as if it were a method in the class:

### Class Definition

```
Class User.NewDTL1 Extends Ens.DataTransformDTL
{

XData DTL
{
<?xml version="1.0" ?>
<transform targetClass='Demo.Loan.Msg.Approval'
           sourceClass='Demo.Loan.Msg.Approval'
           create='new' language='objectscript'>
<assign property='target.BankName'
        value='..ToUpper(source.BankName)' action='set'/>
<trace value='Changed all lowercase letters to uppercase!' />
</transform>
}

}
```