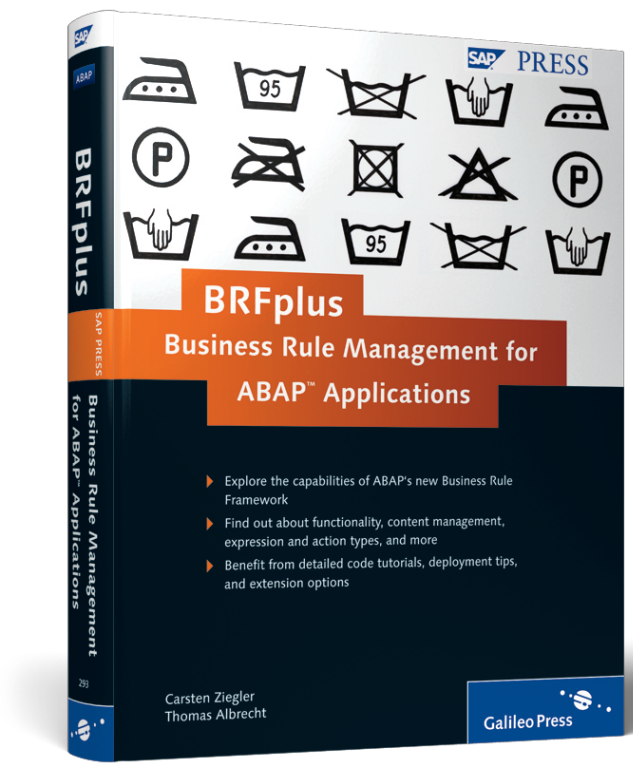
SAP PRESS

Carsten Ziegler and Thomas Albrecht

BRFplus—Business Rule Management for ABAP™ Applications



www.sap-press.com

Galileo Press

Bonn • Boston

Contents at a Glance

1 About Business Rules 21

2 BRFplus—a Brief Walk-Through 39

3 Tutorials 77

4 Object Management 121

5 Objects 169

6 Tools and Administration 297

7 Advanced 337

8 Deployment 399

9 Methodology 399

A Formula Functions 425

B The Authors 431

Contents

Foreword 13
Introduction 17

1 About Business Rules 21

1.1 Definition 21
1.2 Ubiquity of Business Rules 22
1.3 Business Rule Management Systems 23
1.3.1 Rule Authoring Environment 24
1.3.2 Rule Repository 25
1.3.3 Rule Engine 26
1.4 Rule Representation 26
1.4.1 Rules and Rulesets 26
1.4.2 Rule Flows 27
1.4.3 Decision Tables 27
1.4.4 Decision Trees 28
1.4.5 Formulas 28
1.4.6 Scorecards 29
1.5 Application Architecture 29
1.5.1 Traditional Application Design 29
1.5.2 Application Design with Business Rules Management 30
1.5.3 Usage Patterns 31
1.6 Business Rules at SAP 32
1.6.1 The Origin of BRFplus 33
1.6.2 BRFplus Rules Authoring—Features 33
1.6.3 BRFplus Rules Engine—Features 34
1.6.4 BRFplus Rules Repository—Features 34
1.6.5 SAP's Business Rules Strategy and Recommendations 35
1.6.6 Outlook 37

2 BRFplus—a Brief Walk-Through 39

2.1 BRFplus Workbench 39
2.1.1 Prerequisites 39
2.1.2 Starting the BRFplus Workbench 40

2.1.3 UI Execution API 42
2.1.4 Layout 43
2.1.5 Help 46
2.1.6 Personalization 49
2.2 Workflow 53
2.3 The Example 53
2.4 Application—Container for BRFplus Objects 55
2.5 Function—Interface Between Code and Rules 57
2.6 Data Object—Data Carriers 59
2.7 Ruleset—Collection of Rules 61
2.8 Rule—Central Entity 63
2.9 Expressions—Computational Units 64
2.10 Actions—Performing Tasks Outside of BRFplus 67
2.11 Catalog—Visualization and Navigation Help 68
2.12 Simulating and Testing Rules 69
2.13 Function Processing and Rules Evaluation 72
2.14 BRFplus Releases 73

3 Tutorials 77

3.1 Creation of a Pricing Application with the BRFplus Workbench ... 77
3.1.1 Application 77
3.1.2 Function 79
3.1.3 Ruleset 86
3.1.4 Rule 91
3.1.5 Formula Expressions 94
3.1.6 Decision Tables 97
3.1.7 Activation 102
3.1.8 Simulation 104
3.2 Creation of a Pricing Application with the API 106
3.2.1 Application 107
3.2.2 Function 108
3.2.3 Ruleset 111
3.2.4 Ruleset Variables 112
3.2.5 Decision Tables as Initialization Expressions 113
3.2.6 Rule 116
3.2.7 Formula Expressions 118
3.2.8 Embedding the Function into ABAP Code 119

4 Object Management 121

4.1 Basic Object Handling 122
4.1.1 Object Creation 122
4.1.2 Object Selection 123
4.1.3 Object Maintenance 126
4.2 Basic Object API 130
4.2.1 Naming Conventions 130
4.2.2 Interface Hierarchy 132
4.2.3 Types, Constants, and Messages 133
4.2.4 The Factory Class: Central Point of Access 134
4.2.5 Transactional Behavior 135
4.3 Common Object Settings 142
4.3.1 Attributes 142
4.3.2 Names 145
4.3.3 Versioning 147
4.3.4 Texts and Documentation 151
4.3.5 Access Level 155
4.4 Applications 157
4.4.1 Application Creation 157
4.4.2 Application Properties 159
4.4.3 Application API 161
4.5 Catalogs 164

5 Objects 169

5.1 Functions 169
5.1.1 Mode of Operation 170
5.1.2 Signature 171
5.1.3 Simulation, Web Services, Code Generation, and Tracing 172
5.1.4 Function API 173
5.2 Data Objects 180
5.2.1 Data Objects API 182
5.2.2 Elements 185
5.2.3 Structures 198
5.2.4 Table 201
5.3 Rulesets 202
5.3.1 Ruleset Header 202

5.3.2 Rules in the Ruleset 205
5.3.3 Deferred Ruleset Processing 208
5.3.4 Ruleset API 209
5.4 Rules 211
5.5 Expressions 215
5.5.1 Expression API 217
5.5.2 Constant Expressions 219
5.5.3 Value Range Expressions 220
5.5.4 Direct Values and Value Ranges 223
5.5.5 Boolean Expressions 225
5.5.6 BRMS Connector Expression 229
5.5.7 Case Expressions 234
5.5.8 DB Lookup Expressions 236
5.5.9 Decision Table Expressions 241
5.5.10 Decision Tree Expressions 249
5.5.11 Dynamic Expressions 251
5.5.12 Formula Expressions 253
5.5.13 Function Call Expressions 258
5.5.14 Loop Expressions 260
5.5.15 Procedure Call Expressions 263
5.5.16 Random Number Expressions 269
5.5.17 Search Tree Expressions 270
5.5.18 Table Operation Expressions 275
5.5.19 XSL Transformation Expressions 279
5.6 Actions 281
5.6.1 Actions API 283
5.6.2 Log Message Action 283
5.6.3 Procedure Call Actions 285
5.6.4 Send Email Action 286
5.6.5 Start Workflow Actions 287
5.6.6 Workflow Event Actions 293

6 Tools and Administration 297

6.1 Simulation Tool 299
6.2 XML Export and Import 304
6.2.1 XML Export Tool 305
6.2.2 XML Import Tool 306
6.2.3 Tracking of Imported Object Versions 307

6.2.4 XML Format 308
6.2.5 XML Export and Import API 311
6.3 Web Service Generation Tool 314
6.4 Application Usage Tool 318
6.5 Application Administration Tool 321
6.5.1 Deletion of Unused Objects 323
6.5.2 Deletion of Marked Objects 324
6.5.3 Discarding of Object Versions 325
6.5.4 Cleanup Database 327
6.5.5 Reorganization of Objects 329
6.6 Transport Analysis Tool 330
6.6.1 Transport in Detail 330
6.6.2 Usage of the Tool 333

7 Advanced 337

7.1 Performance 337
7.1.1 Performance Relevant Factors 338
7.1.2 Performance Measurements 343
7.2 Tracing 347
7.2.1 Technical Trace 349
7.2.2 Lean Trace 353
7.3 Extending BRFplus 369
7.3.1 Application Exits 370
7.3.2 Custom Formula Functions 378
7.4 User Interface Integration 385
7.4.1 Object Manager 386
7.4.2 Embedding the UI 389
7.4.3 Object Manager Configuration 392
7.4.4 Object Changes and Events 394

8 Deployment 399

8.1 Change and Transport System 399
8.2 Local Scenarios 401
8.2.1 Application Exit for Object Changeability 402
8.2.2 Combination of Objects with Different Storage Types 404
8.2.3 Creation of Objects with the API 405
8.3 Remote Scenarios 405

9 Methodology 409

9.1 Classic Approach to Software Development 409
9.2 Classification of System Building Blocks 411
9.3 Model for Business Rules 412
9.4 Action Plan 413
9.4.1 Team Setup 414
9.4.2 Creation of Sketch 414
9.4.3 Definition of Business Rule Service 415
9.4.4 Education in BRFplus 417
9.4.5 Creation of Draft 417
9.4.6 Refinement of Business Rule Service 420
9.4.7 Analysis for Extensions 420
9.4.8 Creation of First Version 422
9.4.9 Implementation of Business Rule Service in Business Process 422
9.4.10 Testing 423
9.5 Effort Estimation 423

Appendices 425

A Formula Functions 425
B The Authors 431

Index 433

In Simulation Mode section we select the option Show also Results of Intermediate Steps. As depicted in Figure 3.55 we can start the simulation by clicking on the Run Simulation button.

For the given test values, our function should compute a final price of 1.73. The result screen of the simulation tool should look similar to Figure 3.56. It includes a detailed description of the processed steps to make the determination of the result value comprehensible.

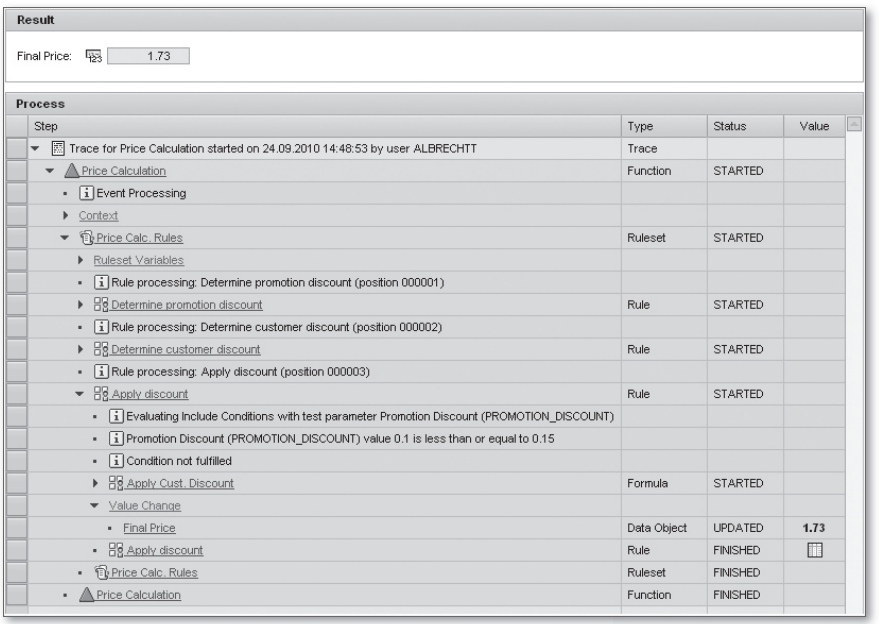


Figure 3.56 Simulation Result

3.2 Creation of a Pricing Application with the API

The previous sections showed the idea of the pricing application and how it can be created with help of the user interface, the BRFplus Workbench. This section shows how the same application can be created programmatically with the BRFplus API.

Package SFDT_DEMO_OBJECTS

Package SFDT_DEMO_OBJECTS is part of the BRFplus shipment. It contains many other example and tutorial programs. The programs are a good source to copy code and development patterns.

3.2.1 Application

The approach with the API is very similar to that of the UI. First, an application has to be created as a container for all other BRFplus artifacts. Listing 3.1 shows the code needed to create the application.

```
DATA: lo_factory          TYPE REF TO if_fdt_factory,
      lo_application      TYPE REF TO if_fdt_application,
      lt_message          TYPE if_fdt_types=>t_message,
      lv_boolean          TYPE abap_bool.

FIELD-SYMBOLS: <ls_message> TYPE if_fdt_types=>s_message.

lo_factory = cl_fdt_factory=>if_fdt_factory~get_instance( ).
lo_application = lo_factory->get_application( ).
lo_application->if_fdt_transaction~enqueue( ).
lo_application->set_development_package( '$TMP' ).
lo_application->if_fdt_admin_data~set_name( 'PRICING' ).
lo_application->if_fdt_admin_data~set_texts(
  iv_short_text = 'Pricing' ).
lo_application->if_fdt_transaction~activate(
           IMPORTING et_message           = lt_message
                     ev_activation_failed = lv_boolean ).
write_errors lt_message. "macro, which exits in case of error
lo_application->if_fdt_transaction~save( ).
lo_application->if_fdt_transaction~dequeue( ).
"Get an application specific factory instance
lo_factory = cl_fdt_factory=>if_fdt_factory~get_instance(
  iv_application_id = lo_application->mv_id ).
```

Listing 3.1 Creation of an Application

Any new object is created with help of the BRFplus factory. The object variables are always typed with an appropriate interface and not with classes. The classes that implement the interfaces are not published for usage and may change in an incompatible way in future releases. In the case of the application above, the interface is `IF_FDT_APPLICATION`.

All object interfaces embed the interfaces `IF_FDT_TRANSACTION` and `IF_FDT_ADMIN_DATA`. They allow performing common transactional actions and respectively changing common object properties.

Objects need to be locked when changes are saved. As for the application of our example, it is best done right after the creation with method `ENQUEUE`.

By default a created application has the storage type customizing. We thus only have to set local development package $TMP to ensure a local customizing application. We can omit an explicit call of method CREATE_LOCAL_APPLICATION in this way. After activating, saving and unlocking the application, a new, application-specific factory instance is created. This will make sure that all objects created with help of this factory are automatically assigned to the newly created application.

The handling of errors has been simplified in the sample code of this tutorial. The statement WRITE_ERRORS is actually a macro that writes an error message to the output screen and stops the processing of the program. Listing 3.2 shows the code for the macro.

```
DEFINE write_errors.
  IF &1 IS NOT INITIAL.
    LOOP AT &1 ASSIGNING <ls_message>.
      WRITE: <ls_message>-text.
    ENDLOOP.
    RETURN. ">>>
  ENDIF.
END-OF-DEFINITION.
```

Listing 3.2 Macro for Message Handling

In productive code it is not sufficient to take only error messages into account that are returned as method parameters. The code should also be embedded into an appropriate TRY-CATCH block, because many methods of the BRFplus API may throw exceptions. All exception classes of BRFplus inherit from class CX_FDT. They therefore share the common attribute MT_MESSAGE, which stores messages with detailed information about the problem that led to the exception.

3.2.2 Function

Similar to the UI tutorial, the next step is the creation of a function. Listing 3.3 shows how to do it.

```
 DATA: lo_function    TYPE REF TO if_fdt_function,
       lts_context_id TYPE if_fdt_types=>ts_object_id,
       lv_result_id   TYPE if_fdt_types=>id.

lo_function ?= lo_factory->get_function( ).
lo_function->if_fdt_transaction~enqueue( ).
lo_function->if_fdt_admin_data~set_name( 'PRICE_CALCULATION' ).
lo_function->if_fdt_admin_data~set_texts(
  iv_short_text = 'Price Calculation' ).
```

```
lo_function->set_function_mode(
  if_fdt_function=>gc_mode_event ).

* code for context creation (lts_context_id) and result
* creation (lv_result_id) to be inserted here
lo_function->set_context_data_objects( lts_context_id ).
lo_function->set_result_data_object( lv_result_id ).

lo_function->if_fdt_transaction~activate(
  EXPORTING iv_deep              = abap_true
  IMPORTING et_message           = lt_message
            ev_activation_failed = lv_boolean ).
write_errors lt_message. ">>> exit in case of error
lo_function->if_fdt_transaction~save(
  EXPORTING iv_deep = abap_true ).
lo_function->if_fdt_transaction~dequeue(
  EXPORTING iv_deep = abap_true ).
```

Listing 3.3 Creation of a Function

New compared to the application are the function-specific method calls such as `SET_UNCTION_MODE`. At the place of the comment (`*code for...to be inserted here`) the creation of the result and context data objects still has to be inserted. This part will be covered later. This substitution pattern is followed for the rest of this tutorial.

In the additional coding there is a slight difference in the way the function is activated, saved, and unlocked. Those operations are all done "deeply" by handing over parameter `IV_DEEP`. The methods will therefore include all referenced objects of the function, such as context and result data objects.

Types and Constants

Type and constant definitions are often included in the object-type-specific interface, for example, `IF_FDT_FUNCTIONS`. Common types are, however, included in the interface `IF_FDT_TYPES` and common constants can be found in the interface `IF_FDT_CONSTANTS`.

The creation of data objects for the function signature, which has been omitted from Listing 3.3, follows now. Listing 3.4 illustrates the creation of one data object for the context. The other context data objects have to be created in the same way.

```
DATA lo_element TYPE REF TO if_fdt_element.
lo_element ?= lo_factory->get_data_object(
    iv_data_object_type = if_fdt_constants=>gc_data_object_type_element ).
```

```
lo_element->if_fdt_transaction~enqueue( ).
lo_element->if_fdt_admin_data~set_name( 'CUSTOMER' ).
lo_element->if_fdt_admin_data~set_texts(
  iv_short_text = 'Customer' ).
lo_element->set_element_type(
  if_fdt_constants=>gc_element_type_text ).
lo_element->set_element_type_attributes( iv_length = 30 ).
INSERT lo_element->mv_id INTO TABLE lts_context_id.
```

Listing 3.4 Creation of a Context Data Object

For the creation of the other data objects, only a few changes are needed to pass different values. The values are listed in Table 3.3.

Name	Text	Element Type	Type Attributes
CUSTOMER	Customer	Text	length = 30
ITEM	Item	Text	length = 30
PROMOTION	Promotion	Text	length = 20
SHELF_PRICE	Shelf Price	Number	length = 7, decimals = 2, only positive = true

Table 3.3 API Tutorial—Context Data Objects

The creation of domain values for the elementary data objects is optional. To keep the tutorial simple, they have been left out. Information on this topic is available in Sections 5.2.2 and 5.5.2.

The function also requires a result data object. Creating a result data object is no different than creating the context data objects. Listing 3.5 shows how to do it.

```
lo_element ?= lo_factory->get_data_object(
    iv_data_object_type = if_fdt_constants=>gc_data_object_type_element ).
lo_element->if_fdt_transaction~enqueue( ).
lo_element->if_fdt_admin_data~set_name( 'FINAL_PRICE' ).
lo_element->if_fdt_admin_data~set_texts(
  iv_short_text = 'Final Price' ).
lo_element->set_element_type( if_fdt_constants=>gc_element_type_number ).
lo_element->set_element_type_attributes(
  iv_length        = 7
  iv_decimals      = 2
  iv_only_positive = abap_true ).
lv_result_id = lo_element->mv_id.
```

Listing 3.5 Creation of a Result Data Object

3.2.3 Ruleset

The next step is the creation of the ruleset. In this tutorial we use a second report for this purpose. The new report also includes the macro for error handling in Listing 3.2. The objects created so far are referred to by their technical ID, which is assumed to be known. In the Workbench the ID can be found in the GENERAL section of the editor. Listing 3.6 shows how to create the ruleset.

```
DATA: lo_ruleset      TYPE REF TO if_fdt_ruleset,
      lts_rule        TYPE if_fdt_ruleset=>ts_rule,
      lts_variable    TYPE if_fdt_ruleset=>ts_variable,
      lts_expression  TYPE if_fdt_ruleset=>ts_init_expr.

lo_factory = cl_fdt_factory=>if_fdt_factory~get_instance(
  iv_application_id = '00505683359D02EE98FC41ECA7650AA3' ).

lo_ruleset ?= lo_factory->get_ruleset( ).
lo_ruleset->if_fdt_transaction~enqueue( ).
lo_ruleset->set_ruleset_switch(
  iv_switch = if_fdt_ruleset=>gc_switch_on ).
lo_ruleset->if_fdt_admin_data~set_name(
  'PRICE_CALCULATION_RULES' ).
lo_ruleset->if_fdt_admin_data~set_texts(
  iv_short_text = 'Price Calc. Rules'
  iv_text       = 'Price Calculation Rules' ).
lo_ruleset->set_function_restriction(
  iv_function_id = '00505683359D02EE98FC41EE6215CAA4' ).

* code for ruleset variables (lts_variable) to be inserted here
lo_ruleset->set_ruleset_variables( lts_variable ).

* code for ruleset initializations (lts_expression) to be
* inserted here
lo_ruleset->set_ruleset_initializations( lts_expression ).

* code for rules (lts_rule) to be inserted here
lo_ruleset->set_rules( lts_rule ).

lo_ruleset->if_fdt_transaction~activate(
  EXPORTING iv_deep              = abap_true
  IMPORTING et_message           = lt_message
            ev_activation_failed = lv_boolean ).
write_errors lt_message. ">>> exit in case of error
lo_ruleset->if_fdt_transaction~save(
  EXPORTING iv_deep = abap_true ).
```

```
lo_ruleset->if_fdt_transaction~dequeue(
  EXPORTING iv_deep = abap_true ).
```

Listing 3.6 Creation of a Ruleset

As you can see, the function does not hold any reference to the ruleset but vice versa. The ruleset makes the execution of its rules dependent on the function with method `IF_FDT_RULESET~SET_FUNCTION_RESTRICTION`.

A common mistake is forgetting to enable the ruleset for processing by switching it on with method `IF_FDT_RULESET~SET_RULESET_SWITCH`.

Some Tips and Tricks when Working with the BRFplus API

- Intermediate calls of method `IF_FDT_TRANSACTION~CHECK` can help to identify problems with newly created objects early. Later those checks can be removed.
- In messages of type `IF_FDT_TYPES=>S_MESSAGE` there can be information about the source, where exactly the message was created.

Three important steps have been left out in Listing 3.6. They are replaced by comments, which indicate what has to be implemented at the specified places:

1. Definition of ruleset variables—`LTS_VARIABLE`
2. Definition of initialization expressions—`LTS_EXPRESSION`
3. Definition of a rule—`LTS_RULE`

3.2.4 Ruleset Variables

Ruleset variables extend the function context, but within the scope of the ruleset only. They are also based on data objects. Listing 3.7 shows how to create the two ruleset variables for our example. To conveniently pass them to the ruleset, they are stored in a table `LTS_VARIABLE` at the end.

```
DATA: lo_element      TYPE REF TO if_fdt_element,
      ls_variable     TYPE if_fdt_ruleset=>s_variable,
      lv_pro_discount TYPE if_fdt_types=>id,
      lv_cus_discount TYPE if_fdt_types=>id.

lo_element ?=
  lo_factory->get_data_object( iv_data_object_type =
    if_fdt_constants=>gc_data_object_type_element ).
lo_element->if_fdt_transaction~enqueue( ).
lo_element->if_fdt_admin_data~set_name( 'CUSTOMER_DISCOUNT' ).
lo_element->if_fdt_admin_data~set_texts(
```

```
    iv_short_text = 'Customer Discount' ).
lo_element->set_element_type(
  if_fdt_constants=>gc_element_type_number ).
lo_element->set_element_type_attributes(
  iv_length        = 3
  iv_decimals      = 2
  iv_only_positive = abap_true ).
ls_variable-position       = 1.
ls_variable-data_object_id = lo_element->mv_id.
INSERT ls_variable INTO TABLE lts_variable.
lv_cus_discount = lo_element->mv_id.

lo_element ?= lo_factory->get_data_object(
    iv_data_object_type = if_fdt_constants=>gc_data_object_type_element ).
lo_element->if_fdt_transaction~enqueue( ).
lo_element->if_fdt_admin_data~set_name( 'PROMOTION_DISCOUNT' ).
lo_element->if_fdt_admin_data~set_texts(
    iv_short_text = 'Promotion Discount' ).
lo_element->set_element_type( if_fdt_constants=>gc_element_type_number ).
lo_element->set_element_type_attributes(
  iv_length        = 3
  iv_decimals      = 2
  iv_only_positive = abap_true ).
ls_variable-position       = 2.
ls_variable-data_object_id = lo_element->mv_id.
INSERT ls_variable INTO TABLE lts_variable.
lv_pro_discount = lo_element->mv_id.
```

Listing 3.7 Creation of Ruleset Variables

The variables `LV_CUS_DISCOUNT` and `LV_PRO_DISCOUNT` store the technical IDs of the two ruleset variables. They are needed for the creation of decision tables, formulas, and the rule below.

3.2.5 Decision Tables as Initialization Expressions

Next the two Decision Table expressions that initialize the ruleset variables have to be created. The following listings show only the creation of decision table "Customer Discount," which is based on the data objects "Customer" and "Item." The second decision table DT_PROMOTION_DISCOUNT can be created similarly with only a few modifications.

Listing 3.8 contains only the basic steps to create a decision table in general. The detailed column definition and the table data are transferred into subsequent listings.

```
DATA: lo_decision_table TYPE REF TO if_fdt_decision_table,
      lts_column     TYPE if_fdt_decision_table=>ts_column,
      lts_table_data TYPE if_fdt_decision_table=>ts_table_data,
      ls_expression  TYPE if_fdt_ruleset=>s_init_expr.

lo_decision_table ?=
  lo_factory->get_expression( iv_expression_type_id =
    if_fdt_constants=>gc_exty_decision_table ).
lo_decision_table->if_fdt_transaction~enqueue( ).
lo_decision_table->if_fdt_admin_data~set_name(
  'CALC_CUSTOMER_DISCOUNT' ).
lo_decision_table->if_fdt_admin_data~set_texts(
    iv_short_text = 'Calc. Cust. Discount'
    iv_text       = 'Calculate Customer Discount' ).

* code for table definition (lts_column) to be inserted here
lo_decision_table->set_columns( lts_column ).
lo_decision_table->if_fdt_expression~set_result_data_object(
  lv_cus_discount ).   "customer discount

* code for table content (lts_table_data) to be inserted here
lo_decision_table->set_table_data( lts_table_data ).

ls_expression-position    = 1.
ls_expression-id          = lo_decision_table->mv_id.
ls_expression-change_mode =
  if_fdt_ruleset=>gc_change_mode_update.
INSERT ls_expression INTO TABLE lts_expression.
```

Listing 3.8 Ruleset Variable Initialization with Decision Tables

Listing 3.9 shows how the decision table columns are defined. The code has to be inserted at comment `* code for table definition ...` into Listing 3.8.

```
DATA: ls_column TYPE if_fdt_decision_table=>s_column.

ls_column-col_no    = 1.                    "customer
ls_column-object_id = '00505683359D02EE98FC41EEC610CAA5'.
ls_column-is_result =  abap_false.
INSERT ls_column INTO TABLE lts_column.
ls_column-col_no    = 2.                    "item
ls_column-object_id = '00505683359D02EE98FC41EEF243CAA6'.
```

```
ls_column-is_result =  abap_false.
INSERT ls_column INTO TABLE lts_column.
ls_column-col_no    = 3.                  "customer discount
ls_column-object_id = lv_cus_discount.
ls_column-is_result =  abap_true.
INSERT ls_column INTO TABLE lts_column.
```

Listing 3.9 Decision Table Column Definition

Listing 3.10 shows the code for the creation of the decision table cells. It has to be inserted at the comment `* code for table content ...` into Listing 3.8.

```
DATA: ls_range       TYPE if_fdt_range=>s_range,
      ls_table_data  TYPE if_fdt_decision_table=>s_table_data.

FIELD-SYMBOLS <lv_number> TYPE if_fdt_types=>element_number.

ls_range-position = 1.
ls_range-sign     = if_fdt_range=>gc_sign_include.
ls_range-option   = if_fdt_range=>gc_option_equal.

ls_table_data-col_no = 1.                     "start of row 1
ls_table_data-row_no = 1.
GET REFERENCE OF 'SAP' INTO ls_range-r_low_value.
INSERT ls_range INTO TABLE ls_table_data-ts_range.
INSERT ls_table_data INTO TABLE lts_table_data.
CLEAR ls_table_data-ts_range.
ls_table_data-col_no = 2.
ls_table_data-row_no = 1.
GET REFERENCE OF 'Ballpen' INTO ls_range-r_low_value.
INSERT ls_range INTO TABLE ls_table_data-ts_range.
INSERT ls_table_data INTO TABLE lts_table_data.
CLEAR ls_table_data-ts_range.
ls_table_data-col_no = 3.
ls_table_data-row_no = 1.
CREATE DATA ls_table_data-r_value TYPE
  if_fdt_types=>element_number.
ASSIGN ls_table_data-r_value->* TO <lv_number>.
<lv_number> = '0.15'.
INSERT ls_table_data INTO TABLE lts_table_data.
CLEAR ls_table_data-r_value.

ls_table_data-col_no = 1.                    "start of row 2
ls_table_data-row_no = 2.
GET REFERENCE OF 'SAP' INTO ls_range-r_low_value.
```

```
INSERT ls_range INTO TABLE ls_table_data-ts_range.
INSERT ls_table_data INTO TABLE lts_table_data.
CLEAR ls_table_data-ts_range.
ls_table_data-col_no = 2.
ls_table_data-row_no = 2.
GET REFERENCE OF 'Pencil' INTO ls_range-r_low_value.
INSERT ls_range INTO TABLE ls_table_data-ts_range.
INSERT ls_table_data INTO TABLE lts_table_data.
CLEAR ls_table_data-ts_range.
ls_table_data-col_no = 3.
ls_table_data-row_no = 2.
CREATE DATA ls_table_data-r_value TYPE
  if_fdt_types=>element_number.
ASSIGN ls_table_data-r_value->* TO <lv_number>.
<lv_number> = '0.12'.
INSERT ls_table_data INTO TABLE lts_table_data.
CLEAR ls_table_data-r_value.

ls_table_data-col_no = 3.               "start of row 3
ls_table_data-row_no = 3.
CREATE DATA ls_table_data-r_value TYPE
  if_fdt_types=>element_number.
ASSIGN ls_table_data-r_value->* TO <lv_number>.
<lv_number> = '0'.
INSERT ls_table_data INTO TABLE lts_table_data.
```

Listing 3.10 Decision Table Cell Data

Values for result columns are handed over in the reference variable `LS_TABLE_DATA-R_VALUE`. It is necessary to create new variables with the ABAP statement `CREATE DATA` each time. Otherwise all lines would point to the same data in the memory. This is a very common mistake.

3.2.6 Rule

In the ruleset sample code of Listing 3.6 there is the comment `* code for rules...` as a replacement for the actual rule creation coding. Listing 3.11 shows how the rule of our example application can be created. It is submitted with parameter `LTS_RULE` to the ruleset afterwards with method `IF_FDT_RULESET~SET_RULES`.

In this tutorial an unnamed rule is created, which is only valid within the scope of the ruleset it is assigned to. The rule can be identified by the text description setting within the ruleset.

```
DATA: lo_rule             TYPE REF TO if_fdt_rule,
      ls_rule             TYPE if_fdt_ruleset=>s_rule,
      ls_cond_range       TYPE if_fdt_range=>s_param_range,
      ls_rule_expr        TYPE if_fdt_rule=>s_expression,
      lt_rule_expr        TYPE if_fdt_rule=>t_expression,
      lv_formula_cus_disc TYPE if_fdt_types=>id,
      lv_formula_pro_disc TYPE if_fdt_types=>id.

lo_rule ?= lo_factory->get_expression(
  iv_expression_type_id = if_fdt_constants=>gc_exty_rule ).
lo_rule->if_fdt_transaction~enqueue( ).
lo_rule->if_fdt_admin_data~set_texts(
  iv_text = 'Apply Discount' ).

* promotion discount > cusomter discount
CLEAR ls_range.
ls_cond_range-parameter_id = lv_pro_discount.
ls_range-position          = 1.
ls_range-sign              = if_fdt_range=>gc_sign_include.
ls_range-option            = if_fdt_range=>gc_option_greater.
ls_range-low               = lv_cus_discount.
INSERT ls_range INTO TABLE ls_cond_range-ts_range.
lo_rule->set_condition_range( ls_cond_range ).

* define formulas, insert missing code later

ls_rule_expr-position    = 1.
ls_rule_expr-change_mode = if_fdt_rule=>gc_change_mode_update.
ls_rule_expr-expression  = lv_formula_pro_disc.
INSERT ls_rule_expr INTO TABLE lt_rule_expr.
lo_rule->set_true_action_extended(
  it_expression = lt_rule_expr ).

CLEAR lt_rule_expr.
ls_rule_expr-expression  = lv_formula_cus_disc.
INSERT ls_rule_expr INTO TABLE lt_rule_expr.
lo_rule->set_false_action_extended(
  it_expression = lt_rule_expr ).

ls_rule-position = 1.
ls_rule-rule_id  = lo_rule->mv_id.
ls_rule-switch   = if_fdt_ruleset=>gc_switch_on.
INSERT ls_rule INTO TABLE lts_rule.
```

Listing 3.11 Creation of a Rule

The rule uses the "Promotion Discount" data object as its test parameter and compares it to the "Customer Discount" data object. The comparison happens in form of a value range condition (method SET_CONDITION_RANGE). However, it would also be possible to set any expression or data object of Boolean type as a condition. In such a case method SET_CONDITION would be used.

Like the ruleset, also the rule needs to be explicitly enabled for processing. This is achieved with the component SWITCH in the rule table LTS_RULE.

Rule operations are referred to as actions in the API. To specify the rule actions there are two methods available in interface IF_FDT_RULE, SET_TRUE_ACTION_EXTENDED and SET_FALSE_ACTION_EXTENDED. They have exactly the same parameters.

Listing 3.11 shows how an expression can update the value of a context data object. In a similar way it would also be possible to set actions or define updates between context data objects. You can find more explanations of the rule capabilities in Section 5.4.

The creation of the Formula expressions had been omitted so far, to keep the focus on the rule. The next section shows how to create the Formula expressions. The variables LV_FORMULA_CUS_DISC and LV_FORMULA_PRO_DISC will store the respective IDs afterwards.

3.2.7 Formula Expressions

The creation of the Formula expressions is the last required step in this API tutorial. Listing 3.12 contains the missing piece of code that has to be inserted into Listing 3.11 instead of comment * define formulas....

```
DATA: lo_formula TYPE REF TO if_fdt_formula,
      lv_formula TYPE string.

lo_formula ?= lo_factory->get_expression(
  iv_expression_type_id = if_fdt_constants=>gc_exty_formula ).
lo_formula->if_fdt_transaction~enqueue( ).
lo_formula->if_fdt_admin_data~set_name(
  'APPLY_PROMOTION_DISCOUNT' ).
lo_formula->if_fdt_admin_data~set_texts(
    iv_short_text = 'Apply Prom. Discount'
    iv_text       = 'Apply Promotion Discount' ).
lo_formula->if_fdt_expression~set_result_data_object(
  '00505683359D02EE98FC41EF79D24AA9' ). "Final Price
* Final Price = Shelf Price / ( 1 + Promotion Discount )
lv_formula = '00505683359D02EE98FC41EF3591CAA8' &&
             ` / ( 1 + ` && lv_pro_discount &&  ` )`.
```

```
lo_formula->set_formula( lv_formula ).
lv_formula_pro_disc = lo_formula->mv_id.
```

Listing 3.12 Creation of a Formula Expression

Formulas can simply be defined as a text string, which is passed to method `IF_FDT_FORMULA->SET_FORMULA`. When (context) data objects are used as operands, they are represented by their technical ID in the formula string. In the listing the variable `LV_PRO_DISCOUNT` contains the ID of the "Promotion Discount" data object, while the ID of the "Shelf Price" data object is statically declared.

The second formula, which will calculate the customer discount, can be built in exactly the same way as the first one. Its technical ID has to be stored in variable `LV_FORMULA_CUS_DISC`. Only the formula string has to differ slightly. The "Customer Discount" data object has to be used instead of the "Promotion Discount" data object. Listing 3.13 shows how to do this.

```
* Final Price = Shelf Price / ( 1 + Customer Discount )
lv_formula = '00505683359D02EE98FC41EF3591CAA8' &&
             ` / ( 1 + ` && lv_cus_discount &&  ` )`.
```

Listing 3.13 Formula String

3.2.8 Embedding the Function into ABAP Code

The design time part is now complete. The last step in this tutorial is the creation of an ABAP program that will execute the created function. The code of that program is provided in Listing 3.14.

```
DATA: lo_function TYPE REF TO if_fdt_function,
      lo_context  TYPE REF TO if_fdt_context,
      lo_result   TYPE REF TO if_fdt_result,
      lx_fdt      TYPE REF TO cx_fdt,
      lv_string   TYPE string.

FIELD-SYMBOLS <ls_message> TYPE if_fdt_types=>s_message.

lo_function =
  cl_fdt_factory=>if_fdt_factory~get_instance(
 )->get_function( iv_id = '00505683359D02EE98FC41EE6215CAA4' ).

TRY.
    lo_context = lo_function->get_process_context( ).
    lo_context->set_value( iv_name  = 'CUSTOMER'
                           ia_value = 'SAP' ).
    lo_context->set_value( iv_name  = 'ITEM'
```

```
                          ia_value = 'Pencil' ).
    lo_context->set_value( iv_name  = 'PROMOTION'
                          ia_value = ' ' ).
    lo_context->set_value( iv_name  = 'SHELF_PRICE'
                          ia_value = '10' ).
    lo_function->process( EXPORTING io_context = lo_context
                         IMPORTING eo_result  = lo_result ).
    lo_result->get_value( IMPORTING ea_value = lv_string ).
    WRITE lv_string .
  CATCH cx_fdt INTO lx_fdt.
    LOOP AT lx_fdt->mt_message ASSIGNING <ls_message>.
      WRITE / <ls_message>-text.
    ENDLOOP.
ENDTRY.
```

Listing 3.14 Function Processing

The function is processed with help of the function object's `PROCESS` method. BRFplus provides also a static `PROCESS` method in class `CL_FDT_FUNCTION_PROSSESS` for the same purpose. The static call is less elegant but optimized for better performance. More information about performance in BRFplus can be found in Section 7.1.

This chapter provides a description of all the different object types and sub types that BRFplus provides for the assembly of rules. The focus for the definition and usage of the objects lies in the UI provided by the BRFplus Workbench. But the most important aspects of the respective programming API are also included.

5 Objects

Application and catalog objects have already been introduced in Sections 4.4 and 4.5. They are the foundation for deploying and managing the vast majority of all the other object types that BRFplus has to offer for the definition of rules. This chapter shall illuminate the purpose, definition, and usage of the different object types. Especially when it comes to expressions, there may exist multiple approaches to define a rule in a more or less elegant way. Thus it is recommended that you get at least a rough overview of all the object types.

The description of the different object types in this chapter follows the recommended way of defining business rules with BRFplus. It starts with the fundamental function and data objects, continues with rulesets and rules, and eventually addresses the huge variety of standard expression and actions types. As in the previous chapter, for each object type the description of the respective API is condensed into a separate section that may be skipped for later reading.

5.1 Functions

The glue between application code and BRFplus rules is the function object. It is the starting point for triggering the processing of BRFplus rules. Within their application, the function objects must have a unique name.

To ensure fast rule processing, BRFplus generates and executes program code in the form of an ABAP class, the first time a function is executed.

Function objects incorporate a so-called *context*, a set of *data objects* to pass data. It can roughly be compared to the list of importing or changing parameters of a function module. Data objects are the equivalent of a variable in programming

languages, combined with a specific type. An additional, single *result data object* allows for returning the outcome of the rule processing.

If you want to define new rules for your application, you typically start with the definition of a function object. Here, you first need to think about the required mode of operation.

5.1.1 Mode of Operation

Three different modes of operation exist and need to be considered when designing new rules

- **Functional Mode**
  A single, top-level expression can be processed and its result will be returned in the result data object. Even though the nesting of expressions into each other allows for some quite sophisticated logic, this mode is especially suitable for rather simple use cases. If you intend to use condition-based and sequential processing of multiple expressions, you should use the event mode.
- **Event Mode**
  The actual rule logic is not directly determined by the function. Rather, the logic is contained in one or several rulesets that can subscribe to the function. When the function is processed, all subscribed rulesets are executed. Thus the function becomes extensible. The order of the ruleset execution can depend on a priority setting in the ruleset. Each ruleset can incorporate rules that may change the state of the context or trigger the execution of external activities.

  By default, the result data object becomes a table that collects the IDs of all executed actions. But any other data object can also be set as a result. Within rule definitions, the result can be used just like any other data object of the function context. Rulesets can also have an additional, temporary context that allows more flexibility in the rule design. In short, the event mode is the preferred way to unleash the full potential of BRFplus rules design.
- **Functional and Event Mode**
  The third mode is actually a mixture of the other two modes. First a top-level expression is evaluated to prefill the result data object. Then a list of subscribed rulesets is optionally processed. The mode exists largely for historic reasons. With the current release, the same logic can be designed with rules in the event mode meanwhile.

In the BRFplus Workbench, the function settings are distributed across several tabs. The Mode is set on the first tab, the Properties tab which is shown in Figure 5.1. If the mode requires a top-level expression, it can be specified directly beneath

as Top Expression. For the pure functional mode, the result data object needs to be defined on the Signature tab because there is no default available.

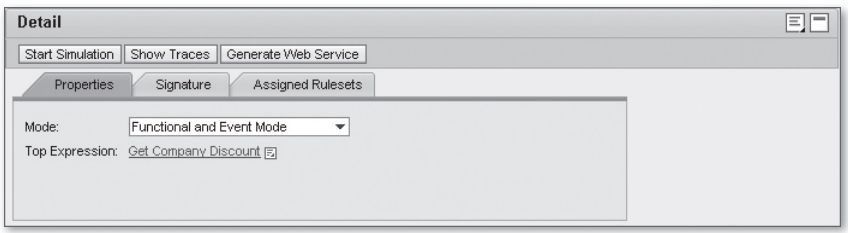


Figure 5.1 Function Properties

If the mode allows the subscription of rulesets, an additional tab, Assigned Rulesets, is displayed. Here you will see the list of all subscribed rulesets together with their priority setting. Rulesets of priority 0 (zero) may be executed in an arbitrary order. Priority 1 is the highest priority. Clicking on the button Create Ruleset will create a new ruleset object that automatically subscribes to the function.

Tip

Before you continue with the detailed design of the connected top expression or ruleset, it is recommended that you define the function's signature first. Data objects defined in the signature can then easily be accessed as context parameters in the UI.

5.1.2 Signature

The signature of a BRFplus function consists of the *context* and a *result data object*. The data objects in the context serve as a list of primarily optional parameters. Parameters that are not passed get an initial value. The context is mainly intended to pass required data the rule logic is based on. But it is also possible to read values from the context, after the function processing has been completed. Especially in the event mode, rules may be used to alter the values in the context.

In the BRFplus Workbench, you can assign existing data objects to the context on the Signature tab with the button Add Existing Data Objects (see Figure 5.2). For simple cases, you may reuse the standard data objects of application `FDT_SYSTEM`. The button Add New Data Object offers two options to create and assign new data objects to the context. There is the "classical" way of creating a single new data object of any type. Multiple elementary data objects can be created at once in a rapid way, using option Add Multiple Elements. For details see Section 5.2.2.

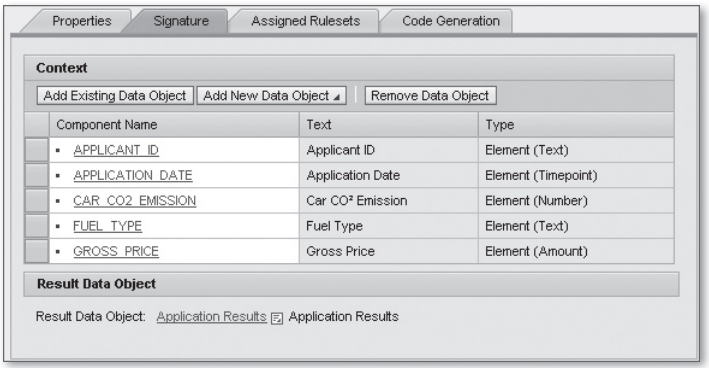


Figure 5.2 Function Signature

The result data object will basically contain the rule evaluation result. Using a structure or table type data object allows you to pass back multiple values at once.

When the function is set to event mode, the standard data object actions of application `FDT_SYSTEM` is set by default. This is a table that will collect the IDs of all action objects that are actually processed. You may, however, replace the actions data object or include it in a custom-defined, structured result data object.

A data object must be used only one time in the signature of the function. For example, it is not possible to use the same data object in the context and also as the result data object. When using structures and tables, you must also take care that the data objects they reference as components are only used once. In general, it is recommended that different structures and tables should not share the same data objects.

The usage of a table in the function signature will implicitly also make the table's structure and/or elements accessible within the connected rule definitions. Expressions can easily retrieve or change data of a single table line by using the Table Operation expression, which is described in detail in Section 5.5.17.

5.1.3 Simulation, Web Services, Code Generation, and Tracing

The execution of BRFplus functions is usually triggered from within application code. To be executable, an active version of the function and its referenced objects must exist.

In the BRFplus Workbench, a simulation tool exists that allows you to analyze the processing and outcome of an active function with test values. The tool can be

invoked either for a currently displayed function with the button Start Simulation, or for any function via the menu entry Tools • Simulation.

If the Display Technical Aspects setting is activated in the user's personalization, an additional tool is available to generate a Web service for a BRFplus function. In this way the BRFplus function can easily be consumed from external systems as well. Similar to the simulation tool, the Web Service generation tool is available for the currently displayed function via a button Generate Web Service, or for any function via the menu entry Tools • Web Service Generation. To complete the picture of the BRFplus function, we will now take a quick glance at some more technical aspects.

When a function is processed, BRFplus will try to execute generated ABAP code to boost the speed of evaluation. The code is automatically generated when the function is executed the very first time. It has to be regenerated when a used object of the function was changed. As a fallback, there always exists a slower interpretation mode. You can switch to this mode, if, for example, an object type does not support code generation.

The code generated for a function is encapsulated into ABAP classes. If one or several used objects have multiple versions, also multiple classes may be required to cover different time intervals.

If you have switched on the Display of Technical Attributes setting in the user personalization, you can see all generated classes of a function on the tab Code Generation. It lists the class names and the time intervals of validity. A flag indicates whether the generated code supports lean tracing.

Tracing is actually available in two different flavors: the *technical trace* and the *lean trace*. The technical trace shows the full details of what happens during function processing. It works only in interpretation mode and is mainly used for debugging purposes, for example, by the simulation tool. When the simulation tool is used, no ABAP class will therefore be generated.

The lean trace addresses the need to keep a record of all executions of a function, also for mass processing. It is integrated into the code generation and will store only a subset of the data to keep a high performance. More details about this topic are provided in Section 7.2.

5.1.4 Function API

The function object type is technically reflected by interface `IF_FDT_FUNCTION`. It provides the methods listed in Table 5.1.

Method	Description
SET_FUNCTION_MODE	Sets the mode of operation.
SET_EXPRESSION	Sets a top expression for the functional mode.
SET_CONTEXT_DATA_OBJECTS	Defines the function context.
SET_RESULT_DATA_OBJECT	Defines the function result.
PROCESS	Processes the function.
SET_CONTEXT_CLASS	Obsolete
SET_FILTER	Obsolete
SET_TRACE_MODE	Obsolete

Table 5.1 Methods of Interface IF_FDT_FUNCTION

For a new function you usually first set the mode of operation with method SET_FUNCTION_MODE. It has a single parameter, which accepts one of the following values that are defined in the function interface:

- GC_MODE_FUNCTIONAL: for the functional mode
- GC_MODE_EVENT: for the event mode
- GC_MODE_FUNCTIONAL_AND_EVENT: for functional and event mode

For the functional mode, a top expression has to be specified with method SET_EXPRESSION. The subscription of rulesets for the event mode is not maintained in the function, but in the individual rulesets.

The function context can be specified or changed by passing a list of data object IDs with method SET_CONTEXT_DATA_OBJECTS. The result data object can be set independently, passing a data object ID in method SET_RESULT_DATA_OBJECT.

In the current release of BRFplus, there is a filter setting for functions under evaluation. However, you actually should not use the related methods SET_FILTER and GET_FILTER. The methods SET_TRACE_MODE and SET_CONTEXT_CLASS are obsolete as well and must not be used any more.

Function Processing API

What mainly differentiates the API of the function from the API of all other object types is its method IF_FDT_FUNCTION~PROCESS—or, in other words, the possibility of triggering the execution of BRFplus rules. The method signature allows to initiate the processing in different ways. Table 5.2 summarizes its parameters.

Parameter	Description
IO_CONTEXT	A runtime context instance of type IF_FDT_CONTEXT.
ITS_ID_VALUE	An alternative to parameter IO_CONTEXT. One can use this list to assign values to the context parameters.
IV_TIMESTAMP	Allows function processing for a previous point in time (time travelling).
IV_TRACE_MODE	Activates tracing during function processing in the specified mode.
EO_RESULT	Provides a handle to the actual function result after the processing is completed.
EO_TRACE	A trace instance to read out the traced information.

Table 5.2 Parameters of Method IF_FDT_FUNCTION~PROCESS

The runtime context for the function can be provided by a separate object in parameter IO_CONTEXT, which has to implement interface IF_FDT_CONTEXT. The interface allows a wide range of context manipulation. It defines the methods that are listed in Table 5.3.

Method	Description
GET_INSTANCE	Returns a new instance of the context interface implementation for a specific timestamp.
COPY	Returns a copy of this context instance.
SET_VALUE	Sets a value for a specific data object in the context. The data object may be referred to by its ID or name. Attribute MV_TIMESTAMP_VALUE_CHANGE is updated.
REMOVE_VALUE	Removes the value for a specific data object in the context. The data object may be referred to by its ID or name. Attribute MV_TIMESTAMP_VALUE_CHANGE is updated.
GET_VALUE	Gets the currently set value for a specific data object in the context. The data object may be referred to by its ID or name.
HAS_VALUE	Checks if a value has been set for a specific data object in the context. The data object may be referred to by its ID or name.
INSERT_DATA_OBJECT	Adds a data object to the context
REMOVE_DATA_OBJECT	Removes a specific data object from the context
GET_DATA_OBJECTS	Returns a list of all context data objects

Table 5.3 Methods of Interface IF_FDT_CONTEXT

With method GET_PROCESS_CONTEXT a default instance with a preset list of context data objects can be retrieved. The context instance is mainly used to set con-

text values for the function processing. But it also allows you to retrieve possibly changed context values afterwards.

To begin, the context interface defines the following two read-only attributes:

- MV_TIMESTAMP contains the processing timestamp the context instance is created for
- MV_TIMESTAMP_VALUE_CHANGE contains the timestamp when a context value has last been changed.

Passing a context instance via parameter IO_CONTEXT to method PROCESS is one option for setting context values. A faster approach with respect to performance is using table parameter ITS_ID_VALUE instead. The creation of an explicit context object instance is then not required. The structure type IF_FDT_TYPES=>S_ID_VALUE of the parameter defines the mapping of a context data object ID to a value by data reference. On the one hand, it is therefore required to know the IDs of the involved data objects; on the other hand, name changes to the data objects do not matter.

Processing of the function can be triggered for a specific timestamp, which becomes relevant when versioned objects are part of the underlying rule definition. The timestamp needs either to be passed upon creation of the context instance in method GET_PROCESS_CONTEXT, or directly in the PROCESS method via parameter IV_TIMESTAMP. If no timestamp is passed, the latest active version of the processed BRFplus objects will be used.

If you want to trace the function processing, you need to request a trace object with parameter EO_TRACE. Tracing can be requested in different modes by using parameter IV_TRACE_MODE. The following constant values of interface IF_FDT_CONSTANTS are supported:

- GC_TRACE_MODE_TECHNICAL
  Activates the technical trace which enforces processing in interpretation mode.
- GC_TRACE_MODE_LEAN
  Activates the lean trace, if possible.
- GC_TRACE_MODE_LEAN_REQUIRED
  Enforces the lean trace. If lean tracing is blocked, for example because code generation is not possible, an exception occurs.

If parameter IV_TRACE_MODE is not passed, the technical trace is used by default. Details on tracing and how to use the provided trace object are given in Section 7.2.

The result of the function processing can be accessed via paramater EO_RESULT of type IF_FDT_RESULT. Like the context interface, the result interface contains the two read-only attributes MV_TIMESTAMP and MV_TIMESTAMP_VALUE_CHANGE. Most important is method GET_VALUE, which provides the value of the result data object. When nested data objects like structures and tables are used, it is possible to retrieve only specific values by passing the name or ID of the corresponding data object in parameter IV_ID or respectively IV_NAME. The value can be requested either by direct value assignment with the exporting parameter EA_VALUE, or by data reference using parameter ER_VALUE. To ensure that the passed variables are of compatible types, you may dynamically create them using methods of the data object interface. See Section 5.2.1 for details. In Table 5.4 you find a complete overview of the methods defined in interface IF_FDT_RESULT.

Method	Description
GET_VALUE	Retrieves the value of the result data object or a part of it.
GET_DATA_OBJECT	Returns an instance of the result data object.
SET_VALUE	Sets the value for the result data object or a part of it.
SET_DATA_OBJECT	Sets the result data object.
COPY	Returns a copy of the this result object instance.

Table 5.4 Methods of Interface IF_FDT_RESULT

The execution of a function may fail with an exception. It is therefore recommended that you encapsulate the method call into a TRY – CATCH – ENDTRY block. Besides exceptions of type CX_FDT_INPUT or CX_FDT_SYSTEM, which typically occur due to a false method call or some general system problem, a CX_FDT_PROCESSING exception may occur. Processing errors can therewith be handled in a separate way.

As a synopsis let us look at a typical piece of code to call a BRFplus function.

```
DATA: lo_function      TYPE REF TO if_fdt_function,
      lo_context       TYPE REF TO if_fdt_context,
      lo_result        TYPE REF TO if_fdt_result,
      lv_result_string TYPE string,
      lx_fdt           TYPE REF TO cx_fdt.
FIELD-SYMBOLS: <ls_message> TYPE if_fdt_types=>s_message.
CONSTANTS: lc_fct_id TYPE if_fdt_types=>id
VALUE '00145EF41CBA02DDA8819E3B1652013A'.
TRY.
  lo_function = cl_fdt_factory=>get_instance( )->get_function(
iv_id = lc_fct_id ).
  lo_context = lo_function->get_process_context( ).
```

```
  lo_context->set_value( iv_name = 'INPUT'
                         ia_value = 'Hello World' ).
  lo_function->process( EXPORTING io_context = lo_context
                        IMPORTING eo_result = lo_result ).
  lo_result->get_value( IMPORTING ea_value =
                        lv_result_string ).
 CATCH cx_fdt_processing.
  lv_result_string = 'PROCESSING ERROR!'.
 CATCH cx_fdt INTO lx_fdt.
  LOOP AT lx_fdt->mt_message ASSIGNING <ls_message>.
    WRITE: / <ls_message>-text.
  ENDLOOP.
ENDTRY.
WRITE:/ 'The result is : ' , lv_result_string.
```

Listing 5.1 Function Call Example

First the function object with a specific ID is retrieved. A context object for the function is created afterwards, and the value "Hello World" is set for context parameter `INPUT`. Next the function is processed and the result value of type string is retrieved. In case a processing error should occur, the result string is set to "PROCESSING ERROR!" If any other exception occurs, the error messages are output. Finally, the determined result string is simply output.

To support function processing with even higher performance, the BRFplus API includes the extra class `CL_FDT_FUNCTION_PROCESS`. It allows you to trigger function processing statically, without the need to instantiate a function object.

Static Function Processing API

Processing a function with method `IF_FDT_FUNCTION~PROCESS` is quite intuitive. But it requires the creation of multiple BRFplus object instances with according database selections. These are rather costly actions in terms of the speed of execution. For use cases that are performance critical and/or call many functions, BRFplus offers an additional approach. Class `CL_FDT_FUNCTION_PROCESS` contains a static `PROCESS` method that can be used as an alternative to its counterpart in interface `IF_FDT_FUNCTION`. At first glance its signature looks quite similar, but it has the following deviations:

- `IV_FUNCTION_ID`
  ID of the function to be processed. No function object instance is required.
- `CT_NAME_VALUE`
  A table of name/value pairs. It depicts another option for setting and retrieving context parameter values.

- EA_RESULT
  It replaces the parameter EO_RESULT of type IF_FDT_RESULT. The result value is directly stored into the passed variable so no intermediate result object is required.

All other parameters that are described in Section 5.1.4 exist as well and have the same meaning.

The table parameter CT_NAME_VALUE constitutes a further option for setting the values for context parameters. Its structure type ABAP_PARMBIND maps a context parameter name to a value by data reference. Because CT_NAME_VALUE is a changing parameter, it also contains the latest context values after the processing.

The use of a static method without any function, context, or result objects can noticeably improve the execution speed. The ultimate optimization measure with respect to performance is, however, not obvious. Context data that is passed to the function usually needs to be converted into a BRFplus internal format. The data conversion requires an instantiation of the corresponding context data objects, and in turn several accesses to the database. The data conversion can be avoided by providing the context values in the BRFplus internal format. For elementary context data objects, you may simply use variables that are of the corresponding type in interface IF_FDT_TYPES:

- ELEMENT_TEXT
- ELEMENT_NUMBER
- ELEMENT_BOOLEAN
- ELEMENT_AMOUNT
- ELEMENT_QUANTITY
- ELEMENT_TIMEPOINT

For structure and table types, you can either define a matching type yourself, or dynamically get a suitable data reference with the static method GET_DATA_OBJECT_REFERENCE of class CL_FDT_FUNCTION_PROCESS.

An optimized way to invoke function processing could look like the following simple example. It assumes that the function uses a structure NUMBER_DESCRIPTION in its context. The structure has the two components NUMBER of type number and DESCRIPTION of type text.

```
DATA: lt_name_value TYPE abap_parambind_tab,
      ls_name_value LIKE LINE OF lt_name_value,
      lv_result_string TYPE string.
CONSTANTS lc_fct_id TYPE if_fdt_types=>id VALUE
```

```
'00145EF41CBA02DDA8819E3D1652013A'.
FIELD-SYMBOLS <la_data> TYPE ANY DATA.
TRY.
  ls_name_value-name = 'NUMBER_DESCRIPTION'.
  cl_fdt_function_process=>get_data_object_reference(
    EXPORTING iv_function_id = lc_fdt_id
              iv_data_object = ls_name_value-name
    IMPORTING er_data = ls_name_value-value ).
  ASSIGN COMPONENT 'NUMBER' OF STRUCTURE ls_name_value-value->*
                                         TO <la_data>.
  <la_data> = '3.14159'. "Set the number value
  ASSIGN COMPONENT 'DESCRIPTION'
   OF STRUCTURE ls_name_value-value->* TO <la_data>.
  <la_data> = 'PI'. "Set the text value
  APPEND ls_name_value to lt_name_value.
  cl_fdt_function_process=>process(
     EXPORTING iv_function_id = lc_fdt_id
     CHANGING  ct_name_value = lt_name_value
     IMPORTING ea_result = lv_result_string ).
  WRITE: / 'The result is : ' , lv_result_string.
 CATCH cx_fdt.
  WRITE / 'Error'.
ENDTRY.
```

Listing 5.2 High-Performance Function Call Example

5.2 Data Objects

Data objects are the carriers of values in BRFplus. They combine the aspects of a variable and a data type definition. Data objects incorporate a type with individual characteristics. They may additionally impose usage restrictions, for example, for allowed comparisons. At runtime the data object represents a single parameter or variable instance.

Three different sorts of data objects exist in BRFplus: elements, structures, and tables. They have in common that they can be bound to existing data type definitions in the *Data Dictionary* (DDIC). When you bind a data object, it is by default named like the underlying type definition. Because names do not need to be unique in BRFplus, you may create multiple data objects with the same name, potentially the same binding, but different IDs. For equally named data objects, it is recommended that you define different short text descriptions.

In the case that a data object is bound, all settings that can possibly be derived from the existing type definition are automatically set and become unchangeable.

This partly affects also the text and documentation properties. The dependency types are automatically set to be language- but not version-dependent. By default the bound data object has no own texts set. Instead the texts are dynamically retrieved from the corresponding definition in the data dictionary. It is, however, still possible to change the texts and thereby overwrite the usage of the data dictionary definitions.

Even though the binding is a common feature for all data object types, it has different, type-specific consequences. For example, the binding for a table type implies the creation of a bound structure type. Binding for a structure type requires the creation of appropriately bound data objects that constitute the components of the structure. Because tables and structures may be deeply nested, a binding can result in the creation of an arbitrary number of required data objects.

The context of a function should contain only data objects that are actually required for the rules processing. It is thus recommended that you use binding against complex data structures only, if all contained elements are actually needed.

If a bound DDIC-type has been changed, the binding can be refreshed. For elementary types, this refreshing will update all derived settings. For structures or tables, it may also result in the creation of new data objects for added components.

Caution

If you refresh the DDIC-binding for a structure that has been extended, new data objects are created only for the added components. But when you first remove the DDIC-binding and then bind against the same structure again, new data objects will be created for *all* components of the structure! Rules may already reference the former data objects and thus become invalid.

At runtime, values are assigned to the used data objects according to their type. A value transfer among data objects is possible only if the value is convertible. For data objects of type element, any value can be converted into a text string, for example, but a text must consist only of numeric digits to be convertible into a number.

As a precondition of being convertible, the involved data objects need to match structurally. If the source data object is a table or structure, the target data object needs to provide an equally named and matching counterpart for each structure component. In addition, a table can be converted only into a flat structure or element at runtime if it has exactly one line of entry.

For very simple use cases you will not need to create any data object yourself. For each element type the following default data objects in Table 5.5 of standard application FDT_SYSTEM are available for reuse.

Data Object	Description
TEXT	Text type without restrictions
NUMBER	Number type without restrictions
BOOLEAN	Boolean type without restrictions
AMOUNT	Amount type without restrictions
QUANTITY	Quantity type without restrictions
TIMEPOINT	Timepoint type without restrictions
TIMESTAMP	Timepoint of type Universal Time Coordinated (UTC) timestamp, including date and time
DATE_TIME	Timepoint of type date and time (local)
DATE	Timpoint of type date
TIME	Timepoint of type time
USER_NAME	Text type, bound to DDIC type SYUNAME
BACKGROUND	Boolean type, bound to DDIC type SYBATCH
ACTION	Text type, intended to store IDs of actions

Table 5.5 Default Data Objects

5.2.1 Data Objects API

The common parts of data objects are reflected by interface IF_FDT_DATA_OBJECT. The defined methods are supported by all data object types, but mostly implemented in a special way for each type.

The type of a data object is available in attribute MV_DATA_OBJECT_TYPE. Alternatively, method GET_DATA_OBJECT_TYPE can be used for that purpose. The three possible constant values are defined in interface IF_FDT_CONSTANTS:

- GC_DATA_OBJECT_TYPE_ELEMENT: for elements
- GC_DATA_OBJECT_TYPE_STRUCTURE: for structures
- GC_DATA_OBJECT_TYPE_TABLE: for tables

Method IS_DEEP returns a Boolean value that indicates whether the data object deeply nests other data objects. Deep nesting is in fact only possible for structure or table types. It means that an involved (table) structure embeds another structure or table, which may again be deeply nested.

Binding

With method SET_DDIC_BINDING the data object can be bound to a specific data dictionary type. You simply have to pass the desired type name. To refresh the binding, you can simply call the method with the currently set type name again. If you want to remove an existing DDIC-binding, you have to pass an empty type name. Settings that have been derived from the DDIC-binding will, however, not be cleared.

Binding for Global Data Types

Similar to DDIC-binding, method SET_GDT_BINDING can be used to bind a data object against a global data type in *SAP Business ByDesign* systems. The method accepts a path definition in parameter IV_PROXY_BINDING_PATH that points to the attribute of a business object's *node*. The attribute's global data type will be used for the binding.[1] The format of the proxy path is <bo_name>.~BO_NODE_NAME.<bo_node>[.<bo_node>]*.~ATTRIBUTE.<attr_name>[.<attr_name>]*.

<bo_name> is the proxy name of the business object, <bo_node> is a contained node, and <attr_name> a node attribute.

Since only one type of binding is possible at a time, calling a binding method will invalidate any previously set, different binding. The current type of binding can be retrieved with method GET_BINDING_TYPE. It will return one of the following constant values:

- IF_FDT_DATA_OBJECT=>GC_BINDING_TYPE_NONE: no binding
- IF_FDT_DATA_OBJECT=>GC_BINDING_TYPE_DDIC: binding against a data dictionary type
- IF_FDT_DATA_OBJECT=>GC_BINDING_TYPE_GDT: binding against a global data type
- IF_FDT_ELEMENT=>GC_BINDING_TYPE_REF_ELEMENT: binding against another BRFplus element; this is possible only for elementary data objects.

Unlike other properties, the setting of a binding property may result in the creation of an arbitrary number of required data objects for structures and tables. The IDs of the created data objects are listed in the exporting table parameter ETS_OBJECT_ID.

1 The alternative binding parameter IV_ESR_BINDING_PATH has become obsolete. The related utility-method CREATE_ESR_BINDING_PATH is therefore also of no importance any more.

Precaution

Structures and tables should be activated and saved only in deep mode to prevent inconsistencies. Required components may otherwise be missing or may not reflect the latest state of the origin type.

Derivation

Method `SET_DDIC_BINDING` implicitly reuses method `DERIVE_DATA_OBJECT`, that allows to derive settings from *any* ABAP data type. The data type can either be explicitly passed by the type's name in parameter `IV_TYPENAME` or dynamically be determined from a variable passed through parameter `IA_DATA`. With the Boolean parameter `IV_INCL_TEXTS` the adaption of text settings can be included or excluded. Method `DERIVE_DATA_OBJECT` will also create additionally required data objects for structure or table types. The corresponding object IDs are available in parameter `ETS_OBJECT_ID`. In contrast to method `SET_DDIC_BINDING`, the passed ABAP type is not stored and all settings remain changeable.

Data Creation

The data object itself does not store any runtime values. Rather, this is the task of the runtime context and result. But the data object allows you to dynamically create a reference to an appropriate variable. Method `CREATE_DATA_REFERENCE` provides the variable via a suitable reference in parameter `ER_DATA`. Information about the created data field can be retrieved in parallel, using parameter `EO_DATADESCR`. It provides a matching instance of the *ABAP Runtime Type Services*.

Data Conversion

Method `IS_CONVERTIBLE_TO` allows you to check whether data of one data object can in principle be transferred to another data object that is specified in parameter `IO_TARGET_DATA_OBJECT`. The method can return three different values:

- `GC_CONVERSION_POSSIBLE`
  Data can always be transferred.
- `GC_CONVERSION_DATA_DEPENDENT`
  Data can be transferred only if it fits the type and characteristics of the target data object. A text can, for example, be transferred into a number, if it consists only of numeric digits.
- `GC_CONVERSION_IMPOSSIBLE`
  Data cannot be transferred to the target data object.

The actual conversion of a value from one data object to another can be achieved with method CONVERT_TO. It requires the parameters listed in Table 5.6.

Parameter	Description
IV_TIMESTAMP	Timestamp to determine the active version for the involved data objects
IA_SOURCE_DATA	Value that is compatible with the source data object
IO_TARGET_DATA_OBJECT	The target data object that determines the output format
CA_TARGET_DATA	The target value that is adapted

Table 5.6 Parameters of Method IF_FDT_DATA_OBJECT~CONVERT_TO

If the target data object is of type table, the source data will be inserted into the provided target table data. For structures, only the common components that occur in the source and target data object are filled. In case the conversion is not possible, a CX_FDT_CONVERSION exception is raised.

5.2.2 Elements

Elements—or rather data objects of type element—define simple value entities of a specific type. The element definition consists of a basic data type such as a number or text, together with some optional properties. A list of valid domain values can be provided too.

Creation

For the creation of elements, the BRFplus Workbench offers two different approaches—the standard approach for a single element and also a mass creation tool. The object creation popup for elements shown in Figure 5.3 includes an area with the most important properties.

In the Define Data Binding section, you may directly bind the element against a DDIC type and thereby derive all basic settings. Through binding even the name is defaulted, if you have not yet entered one. Without binding you need to specify or rather change the Element Type (the default is "Text"). Some corresponding Element Attributes can also be set. To adjust further, more special properties, you need to navigate to the element object.

Another way to swiftly create multiple elements at once is the mass creation popup. In the Repository View you may access it within the context menu of an Application, Data Object or Element node by selecting menu entry Create Data Object • Elements (Mass creation).

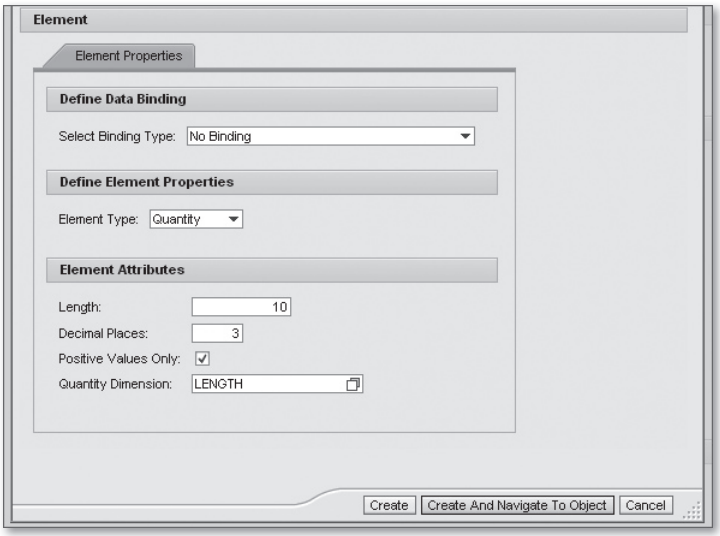


Figure 5.3 Element Creation Popup

In the popup you can enter a list of up to ten new elements to be created. For each line you simply need to specify the Element Type, Name, and a Short Text. It is also possible to define a binding against a DDIC-type or other BRFplus element by choosing the corresponding option in the Type column. The element type will then automatically be derived from the reference type or element, which you enter into column Reference. Figure 5.4 shows how the popup is used.

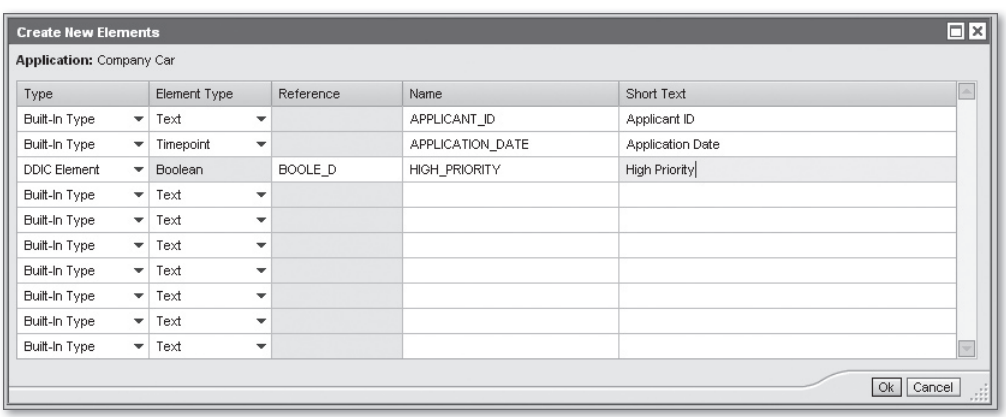


Figure 5.4 Element Mass Creation Popup

Creating the intended elements in this way allows you to immediately continue with the rules definition, and to adjust the element details at a later point in time. We will now have a closer look at the details.

Element Types and Attributes

Each element supports only a specific type of values. The following element types are available:

- **Text**
  Text strings can be used for any kind of character-like data. Texts stored by BRFplus are limited to a length of 255 characters. At runtime there is no such limitation. Strings retrieved from external sources or string concatenations during processing may have any length.
- **Number**
  This element type can be used for any numeric data. With SAP NetWeaver 7.02 BRFplus uses the new base type `Decfloat34`. It allows you to handle a wide range of numbers regarding length and precision. Numbers stored by BRFplus are, however, limited to a maximum length of 31 digits and a precision of 10 decimals.
- **Boolean**
  A Boolean element supports only two values, representing a logical "True" or "False." It is mainly used to store the result of a condition evaluation. BRFplus takes over the ABAP convention to store Boolean values as a character of length 1, which may have the value "X" for "True" and " " (space) for "False."
- **Amount**
  This type is intended for monetary amounts. In BRFplus an amount always consists of two values: a number and a currency value. At design time, the decimals of the number value must not exceed the currency's decimals setting, that has been customized in table TCURX for the current client. At runtime you should take care to pass amount values in the right format. Outside BRFplus, amounts of ABAP type `CURR` are sometimes shifted to be stored with 2 decimals on the database, independent of the used currency. They might need to be shifted back to the actual decimals, by using an appropriate conversion routine.
- **Quantity**
  This type can be used for unit-dependent quantity values. Analogous to amounts, a quantity always consists of two values in BRFplus: a number and a unit value. A quantity element can be restricted to allow only units of a specific dimension, such as "length," "mass," etc. For units BRFplus uses the standard data type `UNIT`. Consequently, the valid units are customized for each client in

table T006, field MSEHI. Because data type `UNIT` uses a conversion routine, the external representation MSEH3 of table T006A is displayed the UI and automatically transformed into the corresponding MSEHI value.

- **Timepoint**
  This type can be used for time- and date-related values. The timepoint element supports several timepoint types. Depending on the type, a time and/or date value can be entered. The type may also impose different semantic meaning. For example, a timestamp may refer to local time or to universal time coordinated (UTC).

Client Dependency

Units and currencies are customizing data and thus are stored client-dependent. In case you define amount or quantity values within system objects that are stored client-independent, they might not be valid in every client. Also elements of type quantity that are defined on system level and imply a dimension restriction might not be usable everywhere.

Depending on the element type, the following attributes can be maintained to restrict the allowed values.

- **Length**
  Defines the allowed, total number of characters within a text string or digits within a numeric value. Length 0 has the meaning of "no restriction." It corresponds, however, to the maximal supported length.
- **Decimals**
  Defines the allowed decimal digits within a numeric value.
- **Only Positive**
  A flag that indicates that only positive numeric values are allowed.
- **Dimension**
  Restricts the usage of units within a quantity. To be convertible, the units of two quantities usually must be of the same dimension, such as "length," "mass," and so on. Setting a dimension can reduce the risk of creating rules with incompatible quantity data.
- **Timepoint Type**
  Restricts a timepoint element to a specific timepoint type. The available timepoint types are: *time*, *date*, *timestamp* and *UTC timestamp*.

Table 5.7 summarizes the element types together with their technical type and the applicable attributes.

Type	Length (max.)	Decimals (max.)	Only Positive	Dimension	BRFplus Type in IF_FDT_TYPES	ABAP Base Types
Text	255	–	–	–	ELEMENT_TEXT	SSTRING
Number	31	10	x	–	ELEMENT_NUMBER	DF34_RAW
Boolean	–	–	–	–	ELEMENT_BOOLEAN	CHAR1
Amount	31	–	x	–	ELEMENT_AMOUNT	DF34_RAW, CHAR5
Quantity	31	10	x	x	ELEMENT_QUANTITY	DF34_RAW, CHAR3
Timepoint	–	–	–	–	ELEMENT_TIMEPOINT	DATS,TIMS, TIMESTAMP

Table 5.7 Element Types and Attributes

Binding

If you bind an element against a DDIC type, all element attributes that can be derived are automatically set and become unchangeable. The used DDIC type must however match to the element type. The following Table 5.8 shows the compatibility matrix.

ABAP Type Kind	BRFplus Element Type	Max Length	Max Number of Decimals	Only Positive Values	Timepoint Type
Char	Text	1–255	n/a	n/a	n/a
String	Text	255	n/a	n/a	n/a
Numc	Text	1–255	n/a	n/a	n/a
Int1	Number	1–3	0	Yes	n/a
Int2	Number	5	0	No	n/a
Int, Int4	Number	10	0	No	n/a
Packed	Number	1–31	0–10	No	n/a
Float	Number	16	0–10	No	n/a
Decfloat16	Number	16	0–10	No	n/a
Decfloat34	Number	31	0–10	No	n/a
CURR (DDIC)	Amount	1–31	n/a	No	n/a

Table 5.8 Element Types Compatibility Matrix

ABAP Type Kind	BRFplus Element Type	Max Length	Max Number of Decimals	Only Positive Values	Timepoint Type
QUAN (DDIC)	Quantity	1–31	0-10	No	n/a
Date	Timepoint	n/a	n/a	n/a	Date
Time	Timepoint	n/a	n/a	n/a	Time
Timestamp	Timepoint	n/a	n/a	n/a	UTC Timestamp

Table 5.8 Element Types Compatibility Matrix (Cont.)

Text elements cover any string or character-like type, including numeric characters. Hexadecimal types of length 16 that may represent a 32 characters UUID are exceptionally supported too.

All kind of numeric types are supported by the number element type.

Boolean types must be characters of length 1 and have exactly the two domain values: "X" and " " (space).

Amounts and quantities may only be bound against their special DDIC type counterparts. When used within a function signature, you should not forget also to pass a currency or a unit value, which are always part of the BRFplus representation. In general it is recommended that you use DDIC structures that contain both a value field and a reference field for the currency or unit.

Eventually date and time types can be mapped to a timepoint element of the corresponding type. For timestamp types the mapping is ambiguous because they are actually packed number types. Currently a packed number type is recognized as a timestamp when it fulfills the following constraints, which are still subject to change:

- The type must use domain `TZNTSTMPS` or its name must include the key word "timestamp"
- The used decimal number must be of length 15 and with no decimals.

Elements can be bound not only to DDIC-types, like all data objects. They additionally support binding against another BRFplus element of the same element type. In this case all attributes of the bound element type become directly valid also for the referencing element. This feature is useful if, for example, you need two element instances of exactly the same type within a function or ruleset context.

Comparisons

Within business rules, the values of context parameters or expression results are often compared against each other or against fixed value ranges. This allows you to discriminate between different processing paths or to trigger some conditional action. BRFplus therefore supports a wide range of comparison operations and operators.

The operations can semantically be categorized into three groups: ordinal comparisons such as "is greater than," string comparisons such as "matches pattern," and implicit comparisons, such as "is initial."

Table 5.9 lists all comparison operations available in BRFplus. The ID is the technical representation of the operator. It corresponds to its ABAP counterpart when feasible.

Category	ID	Description	Comment
Ordinal comparisons	EQ	Is equal to	
	NE	Is not equal to	
	LT	Is less than	
	LE	Is less than or equal to	
	GT	Is greater than	
	GE	is greater than or equal to	
	BT	Is between	The values defining the upper and lower limit of the comparison interval are included.
	NB	Is not between	The logical inverse to "Is between"
String comparisons	CA	Contains any	The sequence of characters does not matter
	NA	Does not contain any	The sequence of characters does not matter
	CO	Contains only	The sequence of characters does not matter
	CN	Does not contain only	The sequence of characters does not matter
	CS	Contains string	The sequence of characters must be identical in the test and comparison parameter
	NS	Does not contain string	Logical inverse of "Contains string"

Table 5.9 Comparison Operations

Category	ID	Description	Comment
	CP	Matches pattern	The following wildcards are supported: + – matches exactly one character * – matches any number of characters Example: "Hello" matches "He*o" but not "He+o". Hero matches "He*o" and "He+o" as well.
	NP	Does not match pattern	Logical inverse of "Matches Pattern"
Implicit comparisons	I1	Is initial	An initial string is empty or contains only whitespace. An initial number is zero. An initial Boolean parameter equals "false." Initial amounts and quantities are zero and have no currency or unit. An initial time point has either no time point type or the respective fields are initial. Initial time and date fields are empty or consist only of zeros.
	I2	Is not initial	Logical inverse of "Is initial"

Table 5.9 Comparison Operations (Cont.)

For ordinal comparisons of texts, case-sensitivity is not taken into account. For example, *a is less than B* is evaluated as true, although `a` has a higher ASCII code value than `B`.

For string comparisons, leading and trailing whitespace is removed from the test parameter; but whitespace inside of a string is retained. Pattern matching comparisons are always performed case-insensitive. Otherwise, case-sensitivity can only be explicitly be set in range or Case expressions. See also Sections 5.5.3 and 5.5.6 for details.

All operations involve an incoming test parameter and—except for implicit comparisons—a comparison parameter. Whether a particular operator can be used or not depends on the operator category and the type of the parameters involved. Implicit comparisons are always possible because no comparison parameter is required.

If the test and comparison parameters are of the same type, all ordinal comparisons are applicable in general. The only exceptions are Boolean parameters that support only the two comparisons "Equal to" and "Not equal to." If the test and comparison parameters are of different types, ordinal comparisons can be performed

only among texts and numbers. If possible, the text parameter is converted into a number and the parameters are compared numerically. Otherwise, the numeric parameter is converted to text and an alphanumeric comparison is performed.

String comparisons are possible for all but Boolean test parameters if the comparison parameter is either of type text or number. All non-textual parameters are translated into text in these cases.

Normalization

The way an operator actually works on the involved parameter types may require some internal conversion into a normalized format first. For alphanumeric comparisons, text parameters are internally converted into a byte sequence, according to the ABAP command `CONVERT TEXT`. On the one hand, this kind of normalization avoids inconsistencies when characters belong to different codepages; on the other hand, the behavior may sometimes differ from the expected outcome.

Parameters of type amount or quantity may have different currencies or units. In case the values are not zero, a conversion into one of the involved currencies or units is implicitly executed. The values are then numerically compared.

Date and time fields are compared numerically to each other. Currently only parameters of the same timepoint type can be compared.

Allowed Comparisons

In addition to technical restrictions, it is possible to further limit the allowed comparisons. In the Workbench, you may change the setting for Allowed Comparisons within the Element Properties tab (see Figure 5.5):

- **No Restrictions**
  This is the default. All supported comparison operations are allowed.
- **Equals**
  It is only allowed to use operator "Is equal to" against a single value.
- **Single Value**
  It is allowed to use the operators "Is equal to" or "Is not equal to" against a single value.
- **Value List**
  It is allowed to use the operators "Is equal to" or "Is not equal to" against one or multiple values.
- **Non-textual**
  It is allowed to use all but string comparisons against one or multiple values.
- **User-defined**
  An explicit list of allowed operators can be specified.

When "user-defined" comparisons are chosen, some additional settings are available in the emerging section User-Defined Properties, as depicted in Figure 5.5.

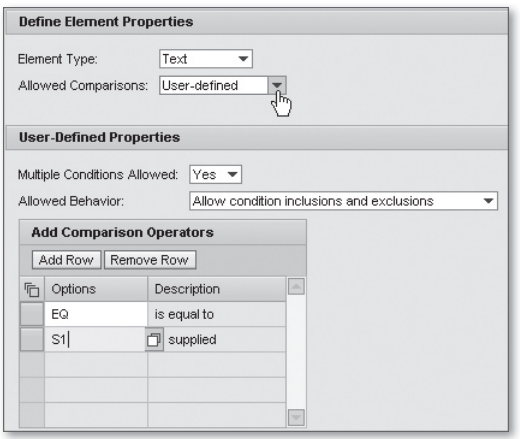


Figure 5.5 User-Defined Comparison Settings for Elements

With Multiple Conditions Allowed you may specify whether the comparison consists only of a single condition, or whether multiple conditions are allowed that will be semantically concatenated with a logical `OR`.

With the Allowed Behavior setting, the comparisons can be restricted to consist of:

- **Condition Inclusions**
  Conditions that need to be fulfilled.
- **Condition Exclusions**
  Conditions that must not be fulfilled.

If both types are allowed, there is no restriction in this respect.

Eventually it is possible to define an explicit list of comparison operators that shall be available when the data object is used for a comparison definition.

Checking an expression or action object that defines a comparison that is not allowed by the used element results in an error message by default. The severity of such a violation may, however, be altered on the tab Additional Properties. With the field Comparison Check Messages you may specify whether an error message, warning message or no message at all will be raised in such a case.

Domain Values

An elementary data object may be intended to support only values within a specific domain. These values are listed on the tab Domain Values.

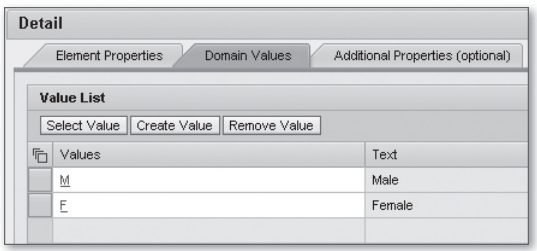


Figure 5.6 Element Domain Values

In case the element is bound against some reference type, the domain values are automatically derived from that type. In the case of DDIC-binding to a type that is using a DDIC domain, the allowed list of values is retrieved from the domain's Value Range (see Transaction SE11). BRFplus currently supports only Single Value or Value Table definitions in DDIC domains. Intervals of values are ignored.

If the elementary data object is not bound, it is possible to use a list of constant definitions as domain values. Constants are the most primitive type of expression objects. It is possible to reuse named constant objects with the button Select Value. New constants can be created by clicking on the button Create Value. You should consider creating unnamed constants if the constant values are only required for the element.

By default only a warning message will be displayed if a value is defined within some action or expression that is not contained in the list of domain values in the underlying element. To enforce valid values, you can increase the severity of such messages to type error on the tab Additional Properties in the field Existence Check Messages.

Element API

For each element type, BRFplus uses a corresponding ABAP type that is defined in interface `IF_FDT_TYPES`. The underlying base types impose the technical limitations for element values. All element attributes, such as length or decimals, are within the range of the base type. The types and ABAP base types are listed together in Table 5.10.

Type	BRFplus Type	ABAP Base Types
Text	ELEMENT_TEXT	SSTRING
Number	ELEMENT_NUMBER	DF34_RAW
Boolean	ELEMENT_BOOLEAN	CHAR1
Amount	ELEMENT_AMOUNT	DF34_RAW,CHAR5
Quantity	ELEMENT_QUANTITY	DF34_RAW,CHAR3
Timepoint	ELEMENT_TIMEPOINT	DATS,TIMS,TIMESTAMP

Table 5.10 ABAP Types for Element Types

The characteristics of elements mentioned above are reflected programmatically by interface IF_FDT_ELEMENT. The provided methods are summarized in Table 5.11.

Method	Description
SET_ELEMENT_TYPE	Sets the element type.
SET_REFERENCED_ELEMENT	Binds the element to another element.
SET_ELEMENT_TYPE_ATTRIBUTES	Specifies additional characteristics, according to the element type.
SET_ALLOWED_COMPARISONS	Defines which comparisons are allowed in connection with the element.
SET_VALUE_LIST	Sets a list of domain values, if the object is not bound.
SET_MSG_SEVERITY	Adjusts the severity of messages for some checks.

Table 5.11 Methods of Interface IF_FDT_ELEMENT

Most important is the specification of the element type with method SET_ELEMENT_TYPE, if it has not already been derived through DDIC-binding. Binding to another BRFplus element can be achieved with method SET_REFERENCED_ELEMENT, which simply requires a valid element ID. To revoke an existing element binding, an initial ID needs to be passed.

Element type specific attributes can be set with the corresponding importing parameters in method SET_ELEMENT_TYPE_ATTRIBUTES. If attributes are passed that are not supported by the element type, a CX_FDT_INPUT exception is raised. A dimension can, for example, be set only for elements of type quantity, and a timepoint type only for elements of type timepoint.

Allowed timepoint types are available as constants in interface IF_FDT_CONSTANTS:

- GC_TP_DATE: local date
- GC_TP_TIME: local time

- GC_TP_DATETIME: local timestamp
- GC_TP_TIMESTAMP_UTC: UTC timestamp
- GC_TP_DATETIME_OFFSET_UTC: UTC timestamp with offset (only available in *SAP Business ByDesign* systems)

For the length and decimals attributes, the maximal allowed values are available as constants directly in interface IF_FDT_ELEMENT:

- GC_MAXIMUM_TEXT_LENGHT: maximal number of characters for textual values
- GC_MAXIMUM_NUMBER_LENGTH: maximal number of digits for numeric values
- GC_MAXIMUM_NUMBER_OF_DECIMALS: maximal number of decimals for numeric values

To refine the allowed comparisons for an element, method SET_ALLOWED_COMPARISONS provides the following parameters:

- IV_ALLOWED_COMPARISONS
  Defines the type of comparison restrictions. Possible values are defined as constants GC_COMPARISONS_*. The passed type also limits the allowed values for the other parameters. Only if the type is set to GC_COMPARISONS_USER_DEFINED is any combination of restrictions allowed.
- IV_MULTIPLE
  Allows or forbids the definition of multiple conditions within a comparison.
- ITS_SIGN
  Lists the allowed condition types. The table may contain any combination of the two constant values GC_SIGN_INCLUDE for including conditions and GC_SIGN_EXCLUDE for excluding conditions.
- ITS_OPTION
  Lists the comparison operations that are allowed. Possible values are defined as constants GC_OPTION_*.

If the element is not bound, a list of valid domain values can be specified with method SET_VALUE_LIST. The values need to be created as Constant expression objects beforehand. Their IDs can then be passed in the table parameter ITS_CONSTANT_ID.

Eventually it is possible to specify the type of messages that are issued, when objects that define inconsistent values or comparisons for the element are checked

for consistency. Method SET_MSG_SEVERITY has two parameters that support the following options, defined in interface IF_FDT_CONSTANTS:

- GC_MSG_SEVERITY_ERROR: always issue an error message
- GC_MSG_SEVERITY_WARNING: always issue a warning message
- GC_MSG_SEVERITY_NONE: issue no message at all
- GC_MSG_SEVERITY_UNDEFINED: default behavior which can be a mixture of all the above for different messages

5.2.3 Structures

A structure in BRFplus is simply an ordered collection of data objects. This implies that the components of a structure are self-contained data objects with their own UUID, which can also be used independently. Access to the component of a structure simply means access to the corresponding data object.

Because data objects are inherently reusable, they may in principle also be part of several structures at once. You should, however, consider that two structures using the same data object cannot be used in a function context together.

Structures can contain not only elements but also other structures and tables, which then leads to a deeply nested data object. However, a recursive nesting where structure A includes structure B that again includes structure A is not allowed.

The names of the data objects used within a structure are important with respect to data conversion. Moving data from one structure to another is only possible if each component in the source structure has an equally named, compatible counterpart in the target structure. When data objects with the same name are created for such use cases, it is recommended that you give them at least different short text descriptions to make them easily distinguishable.

Creation

A newly created structure object initially has no components. In the BRFplus Workbench, you may assemble the structure either by adding already existing data objects or by creating new ones. The corresponding buttons are depicted in Figure 5.7.

Another way to create the structure components is using binding to a DDIC structure. If the components cannot directly be altered any more, they can only be refreshed after the DDIC structure has been modified.

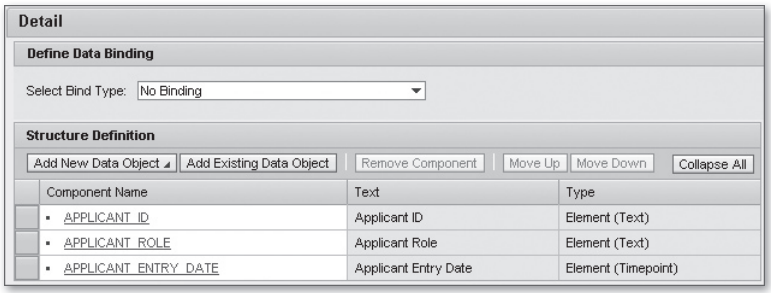


Figure 5.7 Structure Definition

Binding

When you bind a structure object to a DDIC structure, a data object is automatically generated for each structure component and is bound to the corresponding DDIC type as well. Fields of type `CURR` for monetary amounts and `QUAN` for quantities are treated specially. In DDIC structures these fields need to reference a currency field of type `CUKY` or unit field of type `UNIT` respectively. Three cases are distinguished by means of amounts:

1. Both fields, the amount field and the currency field, are contained within the bound DDIC structure. In the BRFplus structure they will be merged into a single element of type amount, which inherently consists of a corresponding number and currency field. At runtime BRFplus will recognize the correlation when values are exchanged using the DDIC structure type, for example, to and from the function context.
2. Multiple amount fields reference the same currency field within the bound DDIC structure. For each amount field, a corresponding element of type amount will be created in the bound BRFplus structure. The currency field is omitted and not explicitly available any more. Instead, each amount can theoretically have its own currency. At runtime BRFplus will recognize the one-to-many correlation for the currency field. Exporting data via the DDIC structure to BRFplus, the currency value will be distributed to all the amount fields in the BRFplus structure. Importing values from the BRFplus structure into the DDIC structure is, however, possible only if either all amounts share the same currency or if the amounts with a deviating currency are zero. Rules should therefore take care to set or alter currencies consistently. Figure 5.8 illustrates the different scenarios.

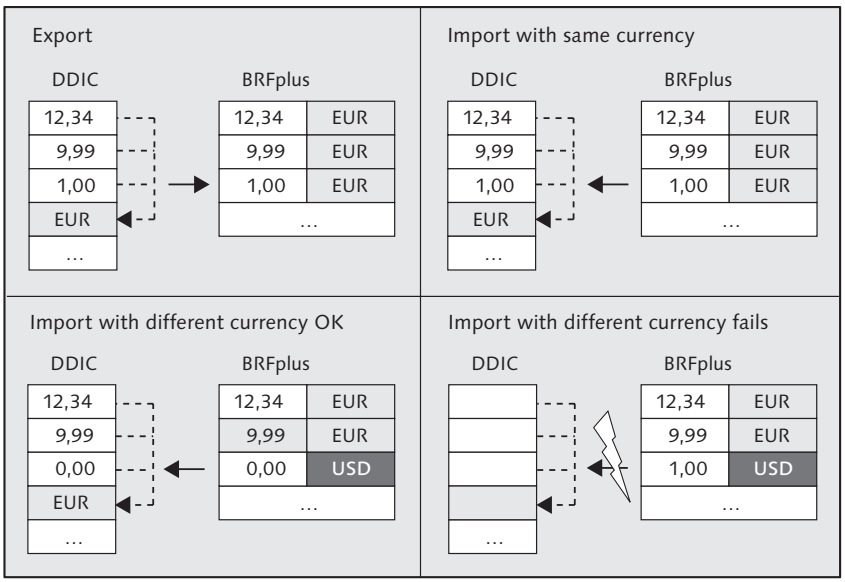


Figure 5.8 DDIC-Structure Data Exchange for Reference Types

3. The amount field references a currency field that is not contained in the bound DDIC structure but is part of an external (table) structure. In this case, an element of type amount that includes a currency field will be created for the BRFplus structure. At runtime it does not make sense to exchange data using the DDIC structure because the currency is not included. When the BRFplus structure is imported into the DDIC-structure, the currencies will be omitted. This scenario may be subject to change.

Structure API

Apart from the common data object API, interface `IF_FDT_STRUCTURE` allows you to set the components of the structure with method `SET_ELEMENTS`. The name of the method stems from SAP NetWeaver 7.0, enhancement package 1, when structures supported only elementary types.

The single table parameter `ITS_ELEMENT` has the line type `S_ELEMENT`. It consists of the position field the table is ordered by, and three component fields to separately store either an element-, structure- or table-ID. The extra fields to be used for structures or tables ensures backwards compatibility with the previous release.

5.2.4 Table

The table data object can be used to hold multiple lines of data at runtime. The line type can be defined either by an elementary or a structure data object type. By default, table data can be accessed only as a whole. To extract or manipulate single table lines and cells, you need to use expressions like the Table Operation expression, the Loop expression or also the Formula expression, described in Sections 5.5.11, 5.5.13, and 5.5.17.

When a table is added to the function signature, the data object defining the line type automatically becomes accessible as context, too. This feature makes the retrieval of table line data within rule definitions more convenient because no duplicate of the table line data object is required in addition.

Creation and Binding

The table data object has only a single property: the line type. In the BRFplus Workbench you can simply select or create an element or structure data object for that purpose. As for structures, the alternative approach is to use DDIC-binding, which will automatically create all required data objects for the line type.

In the case of a structure line type, the structure components are shown, but only editable within the structure data object itself.

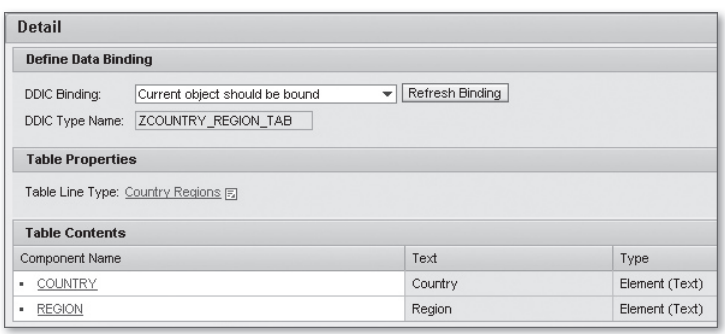


Figure 5.9 Table Data Object Definition

Table API

The API for table data objects is even more simple than that for structures. Interface `IF_FDT_TABLE` defines the single method `SET_STRUCTURE` to pass the ID of a structure or element, which shall be used as the table line type.

5.3 Rulesets

In BRFplus a ruleset object has three main tasks:

- It encapsulates a set of rules, but also defines under which conditions the rules shall be processed.
- It connects the contained rules to a specific function object, which becomes the main trigger for the rules processing.
- It enhances the function context with variables. Variables are additional, temporary data objects that can be accessed as context parameters within the nested rules and expressions.

At design time, the relationship between rulesets and functions is unidirectional. One or multiple rulesets can subscribe to a single function, but functions do not directly reference any ruleset. Functions need to be set up in event mode to allow rulesets to subscribe to them (see also Section 5.1.1). Changing a function object may have impact on many connected ruleset and rule definitions, whereas changing a ruleset object does not affect the referenced function object.

At runtime a function in event mode will trigger the processing of all subscribed rulesets. If multiple rulesets are involved, the order of processing can be influenced by a priority setting of the rulesets. With the default priority setting (zero), the subscribed rulesets may, however, be processed in arbitrary order. The event concept allows an easy and flexible way to enhance pre-delivered BRFplus functions. Even system functions can be extended with additional rules of storage type customizing or application data. More information on this topic is provided in Chapter 8.

5.3.1 Ruleset Header

The concepts and properties that are of importance before any rule is actually processed form the *header* of a ruleset.

A ruleset can be enabled or disabled as a whole. If the ruleset is not enabled, it will be completely skipped by the triggering function at runtime. But even if the ruleset is enabled, it can define an additional precondition besides the triggering function. If the precondition is not fulfilled, all contained rules will be skipped as well. Such a condition may, for example, be based on the passed context values or depend on the system status.

By default, rules of a ruleset can use the data objects that are defined in the assigned function signature. However, those data objects are mainly intended for external data exchange. Having multiple rules within a ruleset often requires additional

data objects to store and exchange intermediate results. Rulesets therefore allow the declaration of additional data objects as *variables*. Variables can be used just like context parameters within the rules.

Context Parameters

Within rule, expression, or action definitions, there is no differentiation between function context or result data objects and ruleset variables. They are all referred to as context parameters.

Variables are of a temporary nature at runtime, so values can neither be read nor set from outside. Before any rule is executed, the ruleset can set an initial value for each variable by using an adequate expression. In the simplest case this can be just some constant value.

If you expect or know that multiple rulesets are assigned to the same function, you can set a priority for each of them. With the priority setting a certain processing order can be enforced. Rulesets with a high priority then usually have more impact on the outcome of the function than rulesets with a lower priority.

By default no priority—or actually priority number 0—is set. It means that the ruleset may arbitrarily be processed before or after any other rulesets. If you set a priority, the according number ban ranges from 1 to 99. Priority number 1 is actually the highest priority! This is recommended for rulesets that shall intentionally be executed before any other ruleset. Rulesets with a lower priority number are evaluated before rulesets with a higher priority number. Rulesets with the same priority are executed in an arbitrary order.

In the BRFplus Workbench the ruleset header is shown in the area between the Detail toolbar and the Rules section. Figure 5.10 shows the initial display of the ruleset. With button Hide Ruleset Header you can hide the header part.

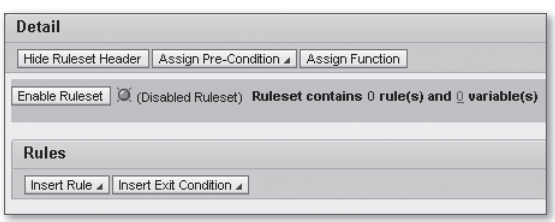


Figure 5.10 Initial Ruleset

Clicking on button Assign Function, you may first select the function that will trigger the execution of the ruleset. When the ruleset is created from within a

function, that function is automatically assigned to the ruleset by default. See also Section 5.1.1.

With button Assign Precondition you may add an additional condition that needs to be fulfilled to execute the ruleset. The condition can be defined in two ways:

- A simple condition that is based on a (context) value comparison can directly be entered.
- For other, more complex conditions, a separate expression with a Boolean result needs to be used.

Ruleset variables can be defined by clicking on the link displaying the number of variables. Or you select the menu entry Maintain Ruleset Variables, which is contained under the menu icon on the right-hand side of the Detail tray. There you also find entries for other actions related to the ruleset header.

The variables are maintained in a popup window. The first section contains a list of the used data objects. Here you may simply add or remove entries. The lower section contains a list of expressions to initialize the variables. Per variable, only one expression with a corresponding result data object is allowed. For variables of type table, you can add expressions that deliver only a line of table data, using the table's structure as result data object. The table line will then simply be inserted into the table.

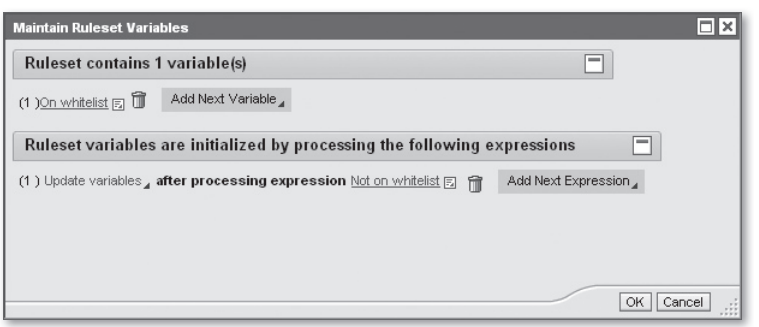


Figure 5.11 Ruleset Variables

If you want to set a specific ruleset priority that differs from the default, you need to choose menu entry Assign Ruleset Priority under the tray menu icon. An input field will then become available in the ruleset header area.

At the end of a ruleset definition, you must not forget to enable the ruleset as a whole by clicking on the button Enable Ruleset in the header section. The header may then look like Figure 5.12.

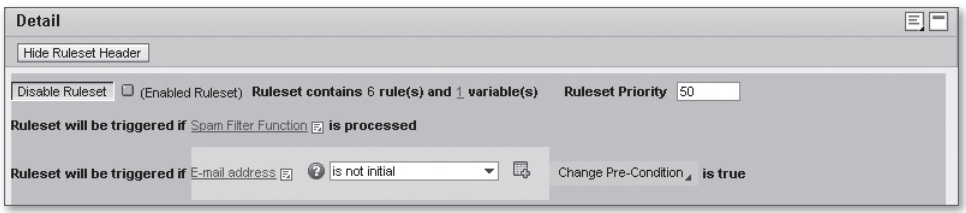


Figure 5.12 Ruleset Header

5.3.2 Rules in the Ruleset

Without any rules a ruleset is, of course, still incomplete. In the Rules section of the ruleset user interface, you define an ordered list of rules and exit conditions. They will be executed top-down, one after another. Exit conditions can be regarded as special rules that allow to skip any subsequent rules and thereby end the ruleset processing.

Figure 5.13 Rule Toolbar in Rulesets

For each rule, a toolbar as shown in Figure 5.13 will be displayed. On the left-hand side it contains a short summary of the header information. From left to right the toolbar has the following content:

1. A status icon indicating whether the rule is enabled or not.
2. An optional icon may indicate that a precondition is specified for the rule.
3. A link with the position and description of the rule. Clicking this link will show or hide rule details below the toolbar. Each such detail section can be closed again with an according icon on the right-hand side. Further options to show or hide the details of all rules are available in the menu of the Rules tray.
4. The validity of the rule with respect to time
5. For rules that are not reusable, an icon to edit the rule is shown next.
6. There are icons to copy, delete, or move the rule within the list.

7. The complete rule header can be made visible and editable with the glasses icon.
8. Eventually the Other Operations menu includes entries that allow you to directly insert or replace rules and exit conditions at a specific position.

With the button Insert Rule at the beginning of the Rules section you have two options to add rules to the top of the list:

1. You can select an existing, independently defined rule to be reused. For such rules you need to ensure that the data objects used by them are contained in the function context or as ruleset variables in respect. Referenced rules are displayed as links that allow navigating to the corresponding object.
2. You can create a new unnamed rule, which is only valid and changeable in the scope of the ruleset. Such rule definitions are edited inside a popup window, and a preview is directly shown in the ruleset list. To edit these rules once you have defined them, you have to click on the corresponding icon in the rule toolbar.

With the button Insert Exit Condition you can add an exit condition at the top of the list. You can either define a value range comparison for a context parameter or expression result. Or you reference any separate expression that provides a Boolean result.

Rule Header

For each rule, the ruleset stores additional settings that are relevant for the rules processing. Similar to the ruleset header, they are referred to as the *rule header*. The representation of the header in the Workbench is shown in Figure 5.14.

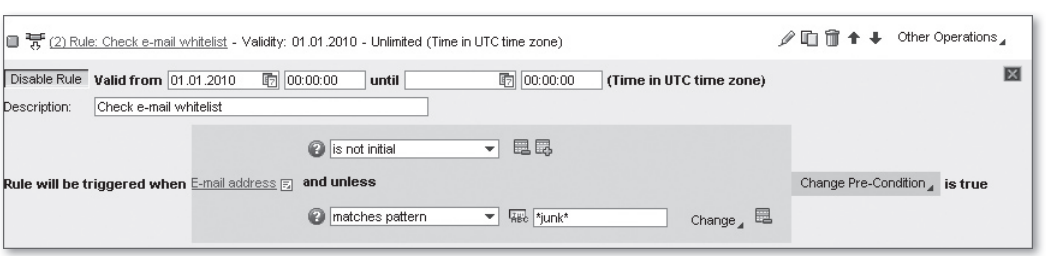


Figure 5.14 Rule Header in Rulesets

The rule header consists of the following settings:

- **Status**
  Each rule can individually be enabled and disabled within the ruleset.
- **Description**
  A textual description to easily identify the rule and its meaning.
- **Interval of validity**
  Within the ruleset a rule can be made explicitly time dependent through the definition of an interval of validity. If processing of the assigned function is triggered outside such a validity period, the rule will be skipped.
- **Precondition**
  It is even possible to define a precondition that must be fulfilled in order to actually process the rule. Such a precondition can be useful to define a more sophisticated period of validity or to prevent unnecessary execution of performance critical rules.

You can enable or disable the rule with a button at the beginning of the section. You can also enter the interval of validity and a description text. To set an optional precondition you need, however, to use the OTHER OPERATIONS menu in the rule toolbar and select entry ASSIGN PRECONDITION.

Example

Before going into details about rules in Section 5.4, let us have a look at an example. Figure 5.15 shows a ruleset that implements a very basic email filter.

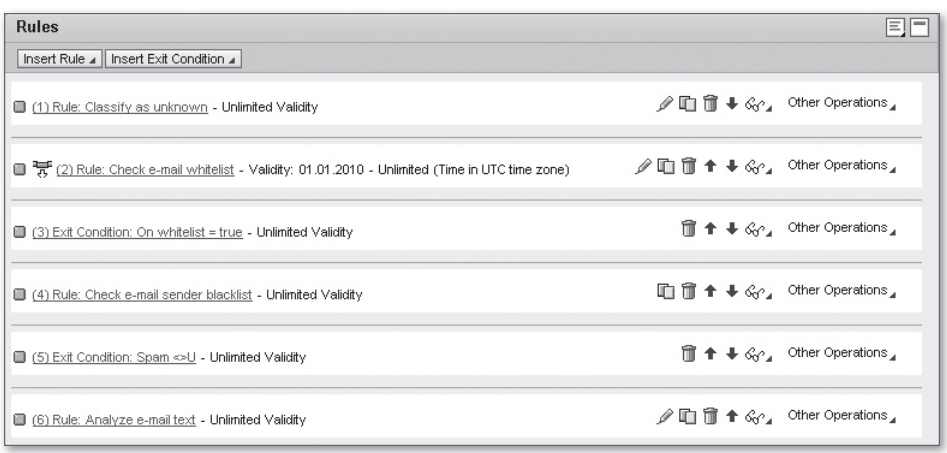


Figure 5.15 Ruleset Example

The function and the ruleset shall provide a classification, whether an email is spam, not spam, or of unknown/ambiguous type. The sender's email address and the email text are available for analysis as function context.

The first rule of the ruleset will unconditionally classify the mail to be of unknown type by default. The second rule checks whether the sender of the email is on a "white" list of trusted mail addresses. If this is the case, the classification is set to "no spam," and the processing is ended by the subsequent exit condition.

The rule in the fourth step will check the opposite: whether the sender is on a "black" list of untrustworthy addresses. If this is the case, the classification is set to "spam" and again an exit condition will end the processing in the next step.

Last but not least, the email text will be analyzed and the classification is either set to "spam" or left as "unknown."

Assuming that the email whitelist or blacklist in this example might become inconsistent and require maintenance, it would be easy to deactivate the according rules temporarily.

5.3.3 Deferred Ruleset Processing

Exit conditions are typically intended to stop the ruleset processing when a valid result could be determined and no further rules are required for processing. However, there may also be use cases where the evaluation of subsequent rules is not possible due to some temporary constraints, e.g., missing data. When the constraints do not exist any longer, the rules processing could continue. In another scenario, the evaluation of some rules might be not time critical but performance critical and should then rather be executed by a batch process. In both cases, it is desirable to keep the current state of the rule processing and to be able to continue with it at a later point in time. This is exactly what deferred ruleset processing makes possible.

If deferred ruleset processing is enabled, an exit condition of a ruleset can not only quit the processing, but it can also be set up to remember the current processing state, including the values of all context data objects. This information can even be stored to the database and be used later on to restart the ruleset again. Two distinct points of entry for a restarted ruleset are possible:

- The ruleset is restarted *before* the exit condition. The exit condition is evaluated again and the processing of the ruleset may possibly be deferred yet another time at the same position.
- The ruleset is restarted *after* the exit condition. Processing of the ruleset simply continues with the next following rule or exit condition in this case.

Because deferred ruleset processing requires special calls to the BRFplus API, it needs to be enabled on the application level. Consequently, this feature is available only for all or no rulesets within an application. In the BRFplus Workbench, the corresponding setting can be changed on the MISCELLANEOUS tab of the application UI. See also Section 3.4.2.

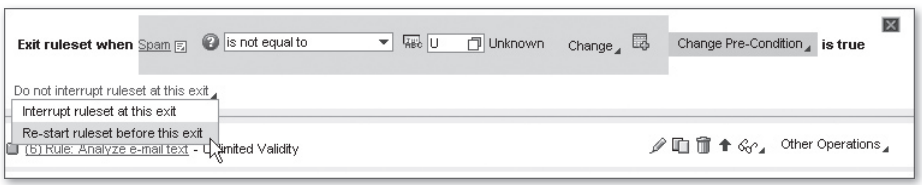


Figure 5.16 Deferred Ruleset Processing

When deferred processing has been enabled, the following additional options for exit conditions become available in the ruleset UI, as illustrated in Figure 5.16:

- DO NOT INTERRUPT RULESET AT THIS EXIT: This is the standard exit condition behavior.
- INTERRUPT RULESET AT THIS EXIT: The ruleset can be restarted after the exit condition.
- RE-START RULESET BEFORE THIS EXIT: The ruleset can be restarted before the exit condition.

5.3.4 Ruleset API

A ruleset is represented by interface `IF_FDT_RULESET`. Table 5.12 summarizes the methods that deal with properties of the ruleset header.

Method	Description
`SET_FUNCTION_RESTRICTION`	Assigns the triggering function to the ruleset. Simply a valid function ID needs to be passed.
`SET_RULESET_SWITCH`	The ruleset can be switched on and off (or rather be enabled or disabled) with the passed Boolean value.
`SET_RULESET_VARIABLES`	An ordered list of data object IDs can be set that extends the context with temporary variables.
`SET_RULESET_INITIALIZATIONS`	A list of expressions that will be evaluated prior to the rules. The result data objects of the expressions must be variables of the ruleset.

Table 5.12 Ruleset Header API

Method	Description
SET_RULESET_PRIORITY	Sets a specific priority for the ruleset. Allowed values are 0 (no priority) and 1 (highest) to 99 (lowest).
SET_RULESET_CONDITION	Assigns an optional precondition to the ruleset. One may either pass the ID of a Boolean element or of an expression with Boolean result in parameter IV_CONDITION_ID. Alternatively, a value range comparison can be set using parameter IS_CONDITION_RANGE. More details on such a range definition are provided in Section 5.5.3.

Table 5.12 Ruleset Header API (Cont.)

The list of rules and exit conditions are set with method SET_RULES. The desired rules must have been created as separate rule objects beforehand. The table parameter ITS_RULE is of the structure type S_RULE, which consists of the fields shown in Table 5.13:

Field	Description
POSITION	Defines the position of the entry in a rule list in ascending order.
SWITCH	Indicates whether the rule shall be enabled or disabled in general.
VALID_FROM	Represents an optional timestamp that limits the validity of the rule. The rule will be evaluated only if the processing timestamp is after this timestamp.
VALID_TO	Represents an optional timestamp that limits the validity of the rule. The rule will be evaluated only if the processing timestamp is before this timestamp.
CONDITION_ID	Contains the ID of an expression or elementary context parameter with Boolean (result) type, which serves either as a rule precondition or as an exit condition.
CONDITION_RANGE	Can be used instead of the CONDITION_ID and depicts a value range comparison, that serves then as precondition or exit condition. More details on such a range definition are given in Section 5.5.3.
EXIT_RULESET	Indicates whether this rule entry represents an exit condition. When set to "true," no rule will actually be processed but the ruleset is exited in case the condition is fulfilled.
RULE_ID	Contains the ID of the rule object to be processed. The field is (only) required, in case EXIT_RULESET is "false."

Table 5.13 Structure IF_FDT_RULESET=>S_RULE

Field	Description
RESTART_OPTION	Indicates for exit-condition entries whether the ruleset can be restarted. Three values available as constants are possible: GC_NO_RESTART: For standard exit conditions. GC_RESTART_BEFORE_EXIT: The ruleset can be restarted just before the exit condition. GC_RESTART_AFTER_EXIT: The ruleset can be restarted directly after the exit condition.
FUNCTION_ID	This field must not be used any longer and is only relevant for legacy usages.

Table 5.13 Structure IF_FDT_RULESET=>S_RULE (Cont.)

For convenience there are also the two methods ADD_RULE and DELETE_RULE available. They will not exchange the whole list of rules but rather modify it by adding or deleting some entries.

5.4 Rules

Even though rules are technically similar to expressions, they are treated as an object type in BRFplus. This is mainly because rules allow some special kind of operations and can thus be used only within rulesets and some specific expressions like the LOOP expression.

A rule basically implements an IF <condition> THEN <do operations> ELSE <do operations> logic. So a rule definition consists of three parts: the condition and possibly two lists of operations. The first list of operations is processed if the condition evaluates to true (or if no condition is specified at all). The other list of operations is optional and will be processed if the condition evaluates to false.

The operations comprise either the processing of nested rules and actions or the change of context values. In detail the following operations are possible:

- Reset a context data object to its type-specific, initial value.
- Set a value for a context data object.
- Exchange values among context data objects.
- Process an expression to determine a new value for a context data object.
- Process a nested rule.
- Execute an action to perform some other (external) changes.

Index

A

Action Plan, 413
Actions, 281, 341
Actions API, 283
Activation Exit, 363
Active, 129
Agents, 290
Aggregation, 236
Aggregation Mode, 239
Application Administration Tool, 321
Application architecture, 29
Application Data, 400
Application Exit, 370, 402
Application Exit Class, 382
Application Usage Tool, 317
Authorization Check Exit, 371

B

Binding, 183, 199, 201
Boolean Expression API, 227
Boolean Expression Example, 226
Boolean Expressions, 225
BOR objects, 293
BRFplus
  and SAP NetWeaver BRM, 35
  messages, 134
  origin, 33
  rules authoring, 33
  rules engine, 34
  rules repository, 34
  types, 133
BRFplus API
  constants, 133
  deep mode, 136
  messages, 134
  types, 133
BRMS, 23
Buffering, 354
Business Objects (BOR), 288
Business rule engine (BRE), 26
Business Rule Framework (BRF), 32
Business Rule Framework plus, 33
Business Rule Management systems, 23
Business Rule Service, 415, 420, 422
Business Rules, 412
  definition, 21
  SAP's approach, 32
Business Rules Group, 21
Business Rules Management
  application design, 30

C

Calling a Function, 338
Case Expression API, 236
Case Expression Example, 235
Case Expressions, 234
Catalog
  nodes, 164
Catalog View, 164, 165
Change and Transport System (CTS), 399
Change Notification Exit, 374
Changeability
  Exit, 377
Check Exit, 362
CL_FDT_WEB_SERVICE, 317
Cleanup Database, 327
Code Generation, 172, 301, 353
Comparison of Function Calls, 343
Comparisons, 191
Complexity of Rules, 342
Condition Columns, 243
Condition Technique, 32
Constant Expression API, 220
Constant Expression Creation, 219
Constant Expressions, 219
Context, 169, 171
Creation of Objects with the API, 405
Currency and Unit Conversion, 341

Custom Formula Functions, 378
Customizing Data, 400

D

Data Conversion, 184
Data Creation, 184
Data Dictionary (DDIC), 180
Data Objects, 169, 180
Data Objects API, 182
Data Retrieval, 236
Data Retrieval Mode, 237
DB Lookup Expression API, 239
DB Lookup Expressions, 236, 341
Decision table, 27
Decision Table API, 247
Decision Table Expressions, 241
Decision tree, 28
Decision Tree API, 250
Decision Tree Expressions, 248
Deletion of Marked Objects, 324
Deletion of Unused Objects, 323
Deployment, 399
Derivation, 184
Derivation Tool, 32
Dice Example, 269
Direct Value and Value Range API, 224
Direct Values, 223
Discarding of Object Versions, 325
Domain Values, 195
Downwards Compatibility, 263
Draft, 417
Dynamic Expression API, 253
Dynamic Expressions, 251

E

Education in BRFplus, 417
Effort Estimation, 423
Element API, 195
Element Creation Popup, 186
Element Types and Attributes, 187, 189
Element Values Exit, 374
Elements, 185
Embedding the UI, 389
Enforced Closing, 392
Events, 394
Excel, 50
Excel Support, 246
Execution Mode, 339
Existence Check, 236
Existence Check Mode, 239
Expert Mode, 255
Expression API, 217
Expression Types, 215
Expressions, 215
Extensions, 420

F

FDT, 131
Formula, 28
Formula Builder, 257
Formula Builder (Fobu), 32
Formula Expression API, 256
Formula Expression Example, 254
Formula Expressions, 253
Formula Function Class, 379
Formula Functions Exit, 376
Function, 169
  Event Mode, 170
  Functional Mode, 170
  modes of operation, 170
Function API, 173
Function Call Example, 179
Function Call Expression API, 259
Function Call Expressions, 258
Function Processing API, 174
Functional and Event Mode, 170

H

Hierarchical Derivation Service (HDS), 32

I

IF_FDT_ACTION, 283
IF_FDT_ACTN_EMAIL, 287

IF_FDT_ACTN_MESSAGE_LOG, 284
IF_FDT_ACTN_RAISE_EVENT, 294
IF_FDT_ACTN_STATIC_METHOD, 286
IF_FDT_ADMIN_DATA, 144
IF_FDT_BOOLEAN, 224, 227
IF_FDT_CONSTANT, 220
IF_FDT_CONSTANTS, 133
IF_FDT_CONTEXT, 175
IF_FDT_DATA_EXCHANGE, 311
IF_FDT_DB_LOOKUP, 239
IF_FDT_DECISION_TABLE, 247
IF_FDT_DECISION_TREE, 250
IF_FDT_DYNAMIC_EXPRESSION, 253
IF_FDT_EXPRESSION, 217
IF_FDT_FORMULA, 256
IF_FDT_FUNCTION, 174
IF_FDT_FUNCTION_CALL, 259
IF_FDT_LEAN_TRACE, 356
IF_FDT_LOOP, 261
IF_FDT_RANDOM, 270
IF_FDT_RANGE, 222
IF_FDT_RESULT, 177
IF_FDT_RULE, 213
IF_FDT_RULESET, 208
IF_FDT_SEARCH_TREE, 274
IF_FDT_START_WORKFLOW, 291
IF_FDT_STATIC_METHOD, 267
IF_FDT_STRUCTURE, 200
IF_FDT_TABLE_OPERATION, 278
IF_FDT_TRACE, 352
IF_FDT_TYPES, 195
IF_FDT_XSL_TRANSFORMATION, 281
Implicit Comparisons, 192
Import Queue, 332
Imported Object Versions, 307
Inactive, 129
Infix Operators, 254
Internet Communication Framework, 39,
Internet Pricing Configurator (IPC), 32

K

Key Structures, 289

L

Lean Trace, 353
Lean Trace API, 356
Local Scenarios, 401
Locking, 127
Log Message Action, 283
Log Message Action API, 284
Loop Expression API, 261
Loop Expressions, 260
Loss of Data, 322

M

Mass Testing, 302
Menu Bar, 43
Messages, 134
Minimized Data, 354
Multiple Match Mode, 243,

N

Names, 50
Namespace, 145
Navigation Panel, 43, 44
  context menu, 122
  Favorites View, 44
  Recently Used View, 44
  Repository View, 44
Normalization, 193

O

Object
  Creation, 122
Objects, 121
  activation, 129
  authorization, 127, 136
  checking, 128
  copying, 139
  deletion, 130
  lifecycle, 129
  locking, 138,

mark for deletion, 130
obsolete, 130
saving, 128
unlocking, 140
Object Changeability, 402
Object Changes, 394
Object Manager, 386, 391
Object Manager Configuration, 392
Ordinal Comparisons, 191

P

Parallel Processing, 342
Performance, 337
Performance Measurements, 343
Procedure Call Action API, 286
Procedure Call Actions, 285
Procedure Call Expression API, 267
Procedure Call Expressions, 263, 341

R

Random Expression API, 270
Random Number Expressions, 269
Recorded Data, 361
Remote Function Call, 315
Remote Scenarios, 405
Reorganization of Objects, 329
Repository View, 51, 164
Result Columns, 243
Result Data Object, 171
RFC, 315
Rule, 26
Rule API, 213
Rule authoring environment, 24,
Rule engine, 26
Rule flow, 27
Rule Header, 206
Rule Modeler, 32
Rule repository, 25
Rule representation, 26
Rule Toolbar, 205
Rules, 211
Rules engine, 34
Ruleset, 26
Ruleset API, 209
Ruleset Example, 207
Ruleset Header, 202, 205
Ruleset Variables, 204
Rulesets, 202

S

SAP NetWeaver BPM, 35
SAP NetWeaver BRM, 35
SAP NetWeaver Business Process Management (SAP NetWeaver BPM), 35
SAP NetWeaver Business Rule Management (SAP NetWeaver BRM), 35
SAP NetWeaver CE, 35
SAP NetWeaver Composition Environment (SAP NetWeaver CE), 35
SapScript, 153
Save Notification Exit, 374
Scorecard, 29
Search Tree Expression API, 274
Search Tree Expressions, 270
Send Email Action, 286
Send Email Action API, 287
Signature, 171
Simulation, 172
Simulation Tool, 299
Single Match Mode, 242
Sketch, 414
SOAMANAGER, 315
Source System Client, 143
Source System ID, 143
Start Workflow Action API, 291
Start Workflow Actions, 287
Static Function Processing API, 179
Status Bar, 43
Storage Type, 51, 404
String Comparisons, 191
Structure API, 200
Structure Definition, 199
Structures, 198
System Building Blocks, 411
System Data, 400

T

Table, 203
Table API, 201
Table Contents, 246
Table Data Object Definition, 201
Table Operation Expression API, 278
Table Operation Expressions, 274, 340
Table Settings, 245
TADIR, 331
Team Setup, 414
Technical Trace, 349
Testing, 383
Testing, 423
Text Exit class, 152
Text Symbols, 152
Token Types, 257
Tools, 51
Trace API, 352
Tracing, 172, 340, 347
Transport, 51, 128, 330
Transport Analysis Tool, 330

U

User Interface Integration, 385
Utility Reports, 299

V

Validation, Substitution, Rules (VSR), 33
Value Range Expression API, 222
Value Range Expression Example, 221
Value Range Expressions, 220
Value Ranges, 223
Variables, 203
Versioning, 353

W

Waterfall Model, 410
Web Service Generation API, 313
Web Service Generation Tool, 314,
Web Services, 172
Workbench, 121
Workflow, 33
Workflow Container, 288
Workflow Event Action API, 294
Workflow Event Actions, 293
WSDL, 315

X

XML Conversion, 354
XML Document, 349, 365
XML Export, 304
XML Export and Import API, 311
XML Export Tool, 305
XML Format, 308
XML Import, 304
XML Import Tool, 306
XSL Transformation Expression API, 281
XSL Transformation Expressions, 279

Y

Yasu, 35