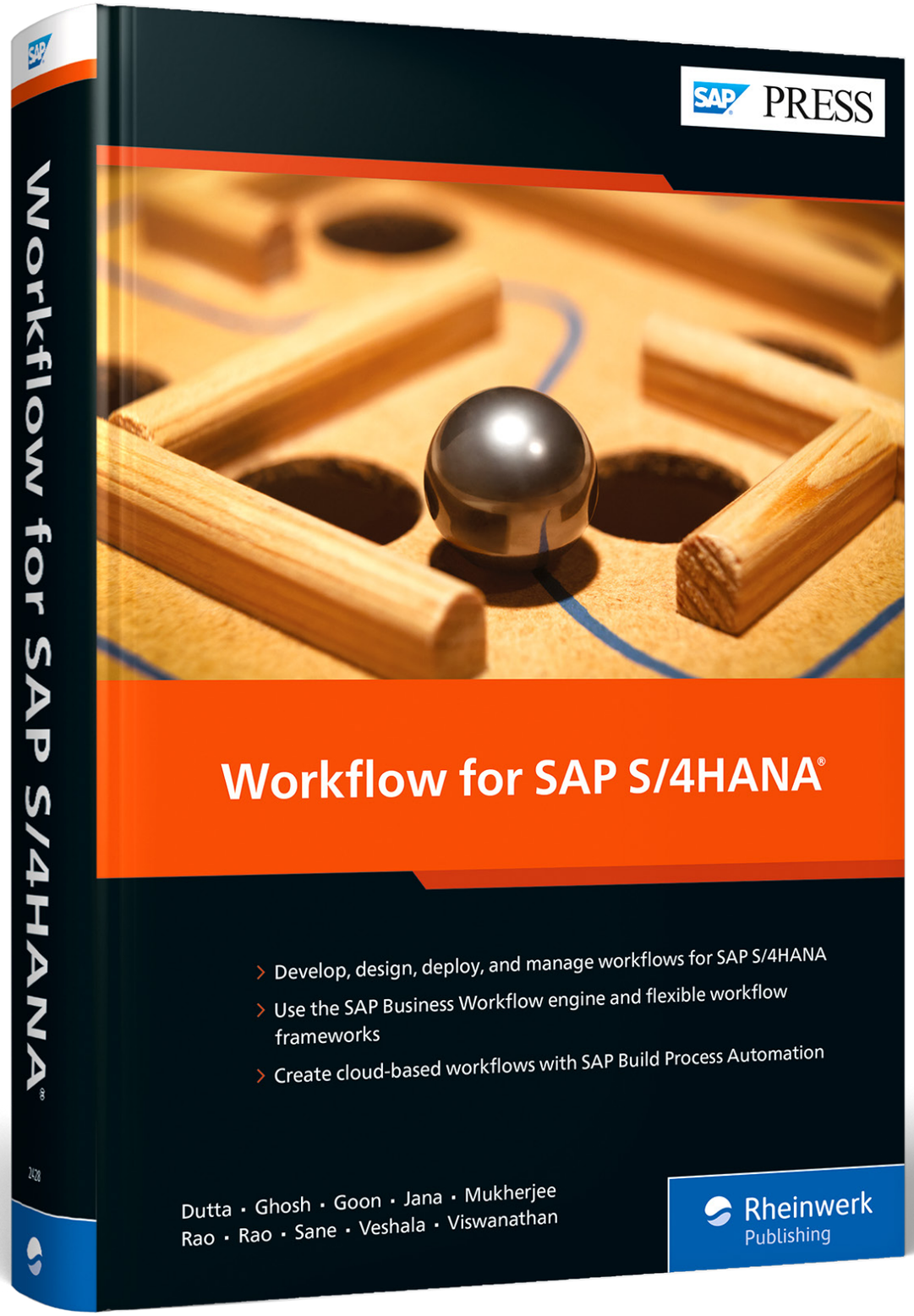


Reading Sample

This sample chapter discusses how to trigger events in SAP S/4HANA and gives details about event creators, receivers, and event linkage. It walks through how to implement a start condition in the workflow as well as how to use terminating events and instance linkage. The chapter concludes by providing information about check function modules and receiver function modules.

Dutta, Ghosh, Goon, Jana, Mukherjee, Rao, Rao, Sane, Veshala, Viswanathan

Workflow for SAP S/4HANA

686 pages | 11/2023 | $89.95 | ISBN 978-1-4932-2428-9

www.sap-press.com/5697

Chapter 5
Defining and Triggering Events

This chapter talks about different triggering mechanisms for workflow events. We look at each event-triggering technique in detail with the help of some practical examples. Then, we delve deeper into event creators, receivers, event linkage, and how to attach start conditions to event linkages. We also look at terminating events and instance linkages and finally close the chapter by explaining how to use check function modules and receiver function modules in event triggering.

An event describes the change in status of an object in the SAP system, for example, when a sales order is created or changed or when a purchase order is released. In the context of this book, we'll be referring to workflow events only.

To use an event in a workflow, it must first be defined in a business object type in the business object repository (BOR) or in an ABAP object-oriented (OO) class (which implements the `IF_WORKFLOW` interface). The event carries information for the business object (or ABAP class object) instance along with optional parameters (if relevant), which may be used by the receiver (workflow, single-step task, or function module) for further processing steps.

You need the following details to trigger an event in the SAP system:

- Business object type or ABAP OO class name in which the event is defined
- Event name
- Key field(s) of the BOR or class required to create the object instance
- Event creator name (by default, this is the current system user)
- Optional event parameters if defined

When you create an event in the SAP system, an instance of the BOR object type or ABAP OO class is automatically created via the key field information passed to the event. This object instance is passed to the receiver application. If the receiver is a workflow template or a single-step task, then the event object instance is passed via event container element `_EVT_OBJECT` to the workflow container or task container. Additional parameters (if any) on the event must be explicitly bound to the receiver container via the event container element with the same name as the parameter.

Events must be triggered explicitly by the SAP application. There are various techniques available in SAP to trigger events, some of which involve configuration steps

and others only pure ABAP code. Whenever an event is triggered in SAP, you can see it from the event trace Transaction SWEL (provided the event trace is switched **ON** via Transaction SWELS). We'll discuss the event trace further in Chapter 8.

In the next section, we'll discuss the various techniques available in SAP to trigger events, followed by a section on the concepts of event creators, receivers, and event linkage. You'll learn how to implement a start condition in the workflow so that you can filter and trigger your workflow or single-step task only under the exact conditions you prefer. We'll introduce you to the concept of terminating events and instance linkage in the next section. Finally, we'll talk about check function modules and receiver function modules that may be configured against an event linkage.

5.1 Event Triggering Techniques

Following are the common techniques used for triggering events in SAP (discussed in detail in this section):

- Event trigger via change documents: Pure configuration, no ABAP coding
- Event trigger via status management: Pure configuration, no ABAP coding
- Event trigger via message control: Pure configuration, no ABAP coding
- Event trigger via ABAP code in a user exit, business add-in (BAdI), or other ABAP programs

5.1.1 Event Trigger via Change Documents

Most standard SAP applications (master data or transaction data) use the concept of change documents to track the creation, updates, and deletion of the entire object or some attributes within the object. Change document objects can be viewed with Transaction SCDO. Figure 5.1 shows an example of the change document object VERKBELEG, which is used to track changes in sales document transactions.

The change document object lists the underlying tables that are tracked for create/change/delete actions on the corresponding application transactions or programmatically via Business Application Programming Interfaces (BAPIs).

The first step to configure an event trigger via change documents is to identify the correct change document object and the corresponding table (if you're interested in specific field updates).

Next, you go to Transaction SWED to verify whether the relevant change document object tracks the change action that is relevant for your requirement. For example, for change document object VERKBELEG, you can see in Figure 5.2 that all three actions—create, change, and delete—are tracked.

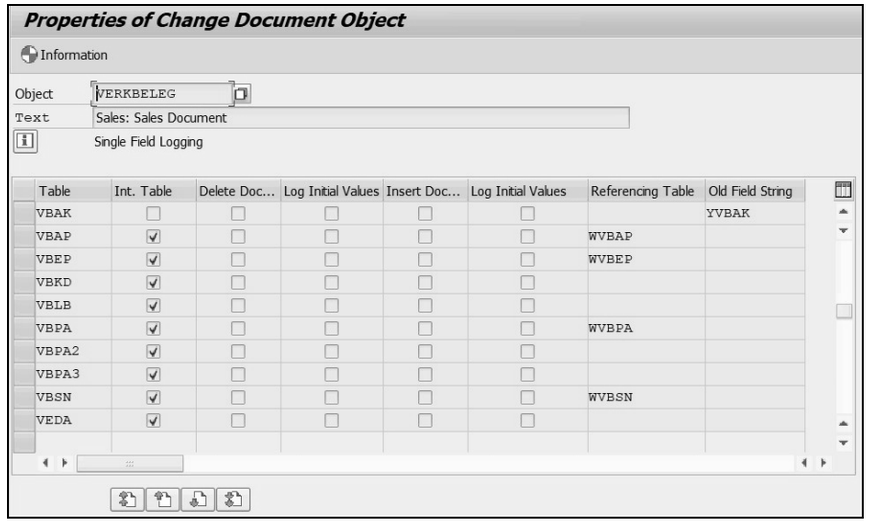


Figure 5.1 A Change Document Object in SAP

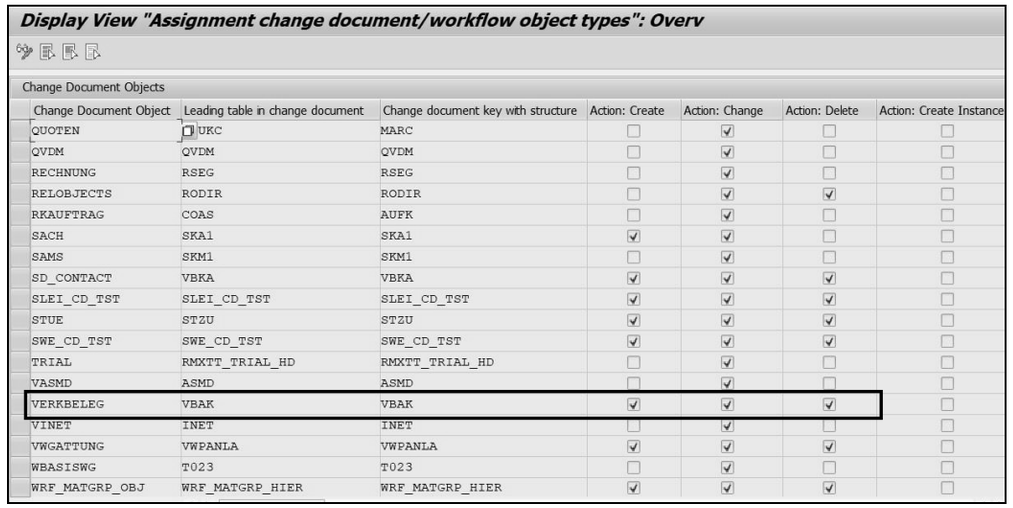


Figure 5.2 Viewing the Change Actions Maintained against a Change Document Object in Transaction SWED

Note

This is just a verification step as most common standard SAP change document objects will track all the actions. However, if you've created your own custom application with a custom change document object, then this step becomes a prerequisite before you can proceed to the next step to configure your workflow events against the change document object.

Expert Tip

In Transaction SWED, there is an option to configure a function module against a change document object. This function module is required when the key field of the primary table in the change document object differs from the key field of the business object used to trigger the event in Transaction SWEC. For example, for the change document object PROJ, primary table PROJ has a key field of PSPNR (internal project number). However, the business object for project BUS2001 has two key fields, external project number (PSPID) and internal project number (PSPNR). So, a conversion needs to happen from the change document key to the business object key via the Transaction SWED function module. Refer to function module CJPN_BUS_2001_2054_OBJECT_KEY for an example of the coding involved.

The next step is to configure your workflow event for the change document object and also configure the specific change action in Transaction SWEC. Once you click on the **New Entries** button, enter the change document object (which you selected from Transaction SCDO) and the business object type and event which you want to trigger. Select the **On Create**, **On Change**, or **On Delete** radio button per your requirement. Save your entries.

You can also maintain field restrictions to track changes in specific fields of the tables listed under the change document object. This additional field restriction is relevant for change actions only. Here you simply need to select the field you want to track and enter a change condition (optional) using the old value and new value variables of this field. Figure 5.3 shows the change document configuration of a sample business object/event against the object VERKBELEG.

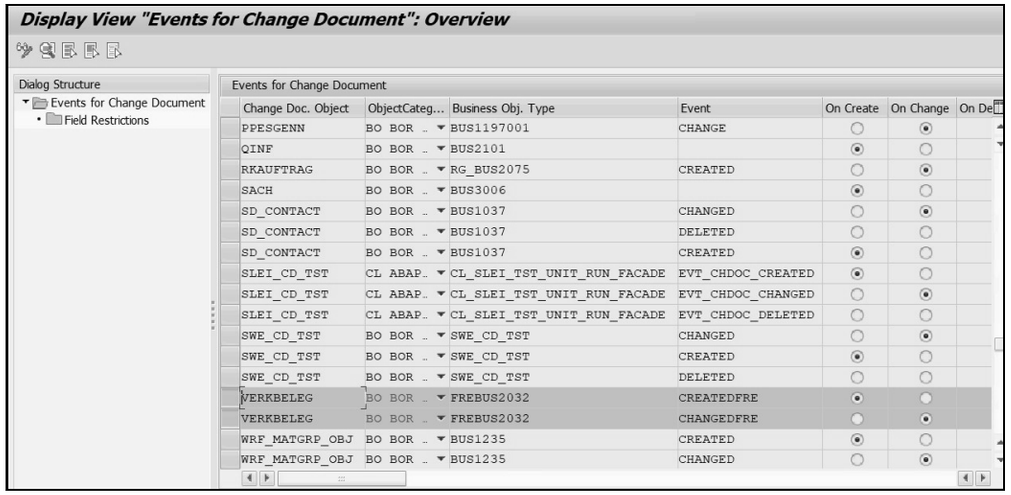


Figure 5.3 Configuring BOR/Class Events against a Change Document Object in Transaction SWEC

Once you've linked your business object (or ABAP class) and event to the change document object/action, then your event triggering configuration is complete. Now you may proceed to test your application by creating/updating/deleting an object and checking if the event was triggered from the event trace (Transaction SWEL).

Let's now look at an example in which an event is triggered when a credit memo request is blocked for billing in SAP. We've identified the standard business object for the credit memo request application as BUS2094 and verified that no standard event is available that is triggered under the same conditions. Here, we've created a subtype of BOR BUS2094, delegated the supertype to the subtype, and defined our custom event ZWEBillingBlockCreated in the subtype (refer to Chapter 3 for details on subtype creation and delegation). Figure 5.4 shows the view of the subtype after the addition of the custom event.

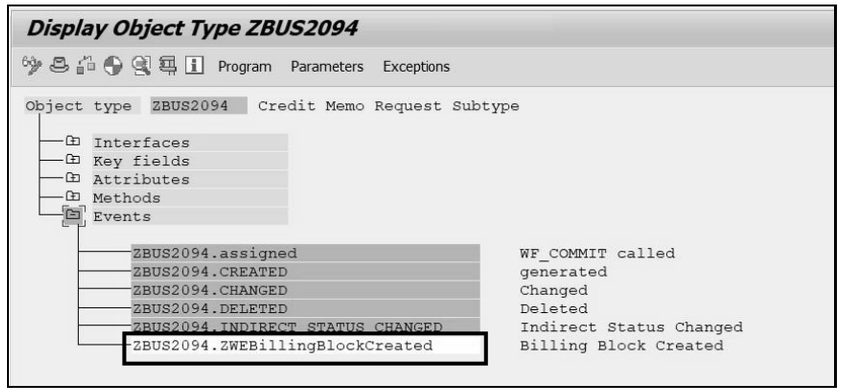


Figure 5.4 Subtype with Custom Event Added from Transaction SWO1

Now follow these steps to proceed with the event configuration:

1. Identify the table and field name for which the new custom event should be raised, and identify the specific value that needs to be checked (if relevant). In this case, the field for the billing block on credit memo request transaction is VBAK-FAKSK.
2. Identify the change document object, which includes table VBAK from Transaction SCDO. In this example, the object name is VERKBELEG.
3. Proceed to Transaction SWEC to enter the configuration steps as mentioned in the following steps. (Note: Because you're dealing with a standard transaction and a standard change document object, you don't need to configure the change actions registered in Transaction SWED.)
4. Under the **Events for Change Document** node, click on the **New Entries** button, and enter the details as shown in Figure 5.5: enter **Change Doc. Object** (change document object) as "VERKBELEG", choose **BO BOR Object Type** from the **Object Category** drop-down, enter **Object Type** as "BUS2094", and enter **Event** as "ZWEBILLINGBLOCK-CREATED".

5. Choose the **On Change** radio button under **Trigger Event**.

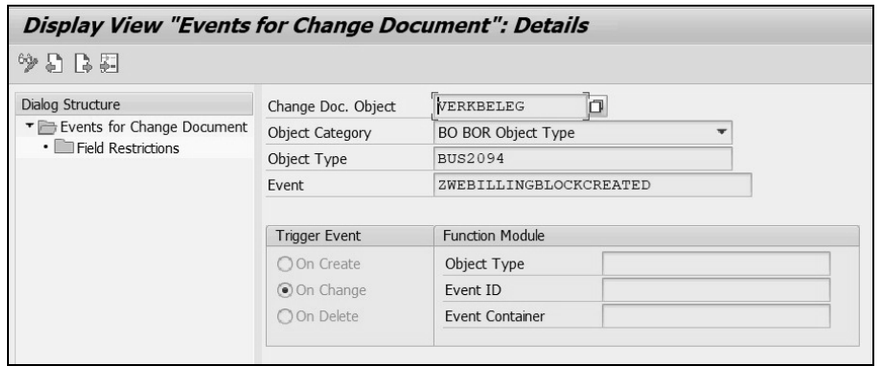


Figure 5.5 Custom Event Configured for the Change Document Object in Transaction SWEC

6. Navigate to the **Field Restrictions** child node, and enter the field change details for triggering the event. Here you have two options to enter the field restriction:
   - If your condition is a simple comparison of one or more values, then enter it directly as shown in Figure 5.6 with the **Table** name, **Field Name**, **Old Value**, and **New Value** fields.

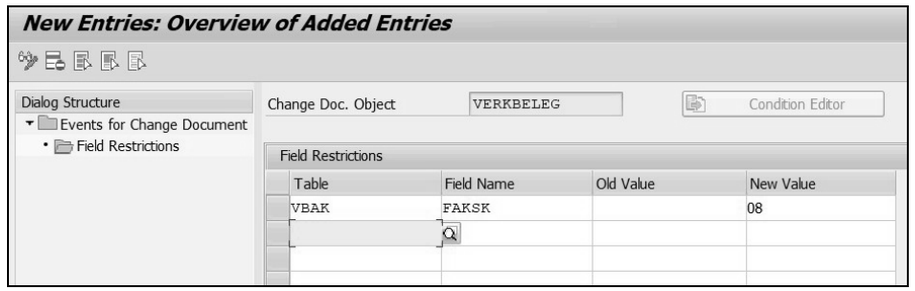


Figure 5.6 Maintaining Field Restrictions for the Event Trigger via Change Documents in Transaction SWEC

   - If your condition involves a more complex expression, then use the **Condition Editor** button to navigate to the screen shown in Figure 5.7, and enter the condition as required. You can make use of a wide range of expression operators, such as **>**, **>=**, **<**, **<=**, **EX**, **NX**, **CE**, **NE**, and so on, along with logical operators, such as **And**, **Or**, and **Not**, to frame your condition. You can also use parentheses in your expression as appropriate and use multiple table fields together in a single expression.
7. Go back and save your changes. You will be prompted for a transport request.

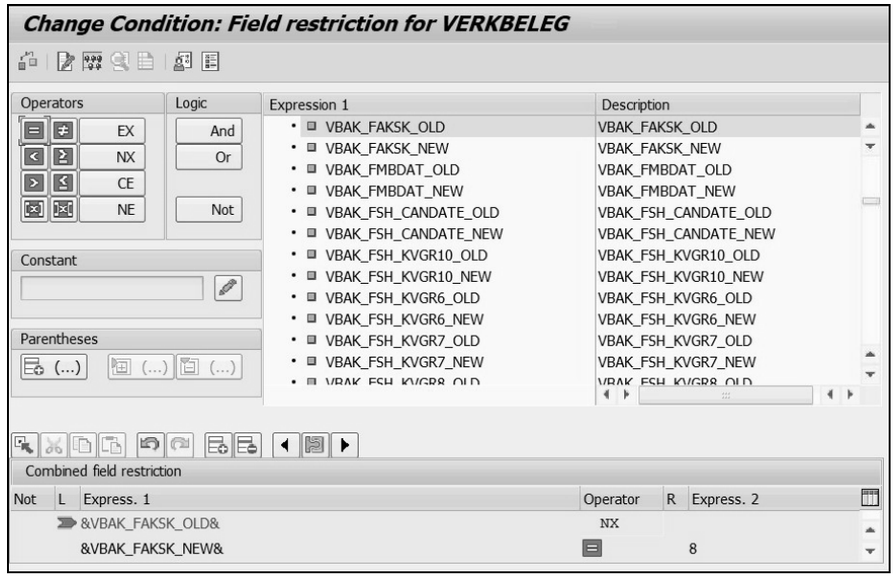


Figure 5.7 Maintaining Field Restrictions with the Condition Editor in Transaction SWEC

Next, it's time to test your changes. As shown in Figure 5.8, create/change a credit memo request in SAP S/4HANA from Transaction VA01/VA02, and enter the **Billing Block** as "08" before saving the document. Write down the document number.

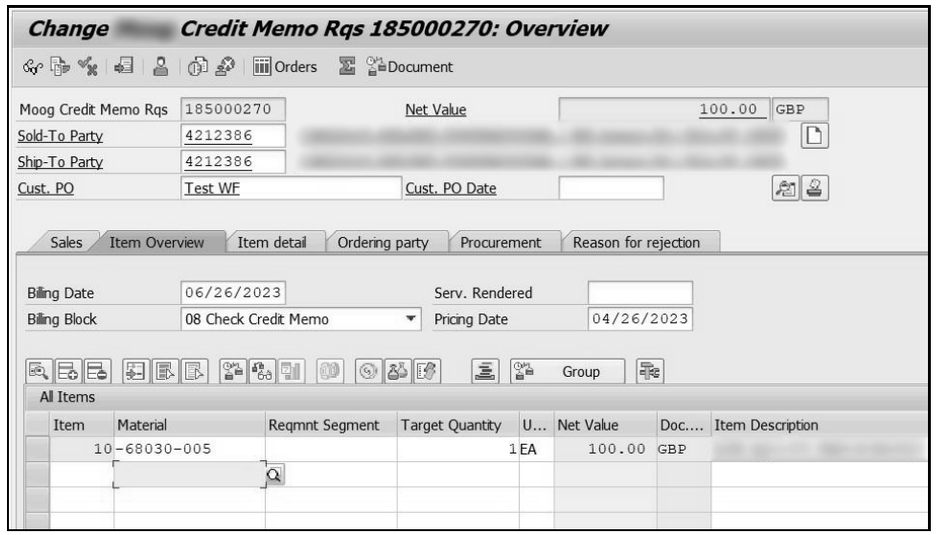


Figure 5.8 Changing a Credit Memo Request in SAP S/4HANA with the Billing Block Set

Now, check the event trace from Transaction SWEL (make sure the event trace is switched **ON** via Transaction SWELS first), as shown in Figure 5.9. Enter your business

object type ("BUS2094") in the **'Creator' object type** field to filter out the trace entries, and make sure the date and time are selected appropriately in the **Created From Date/Time** field.

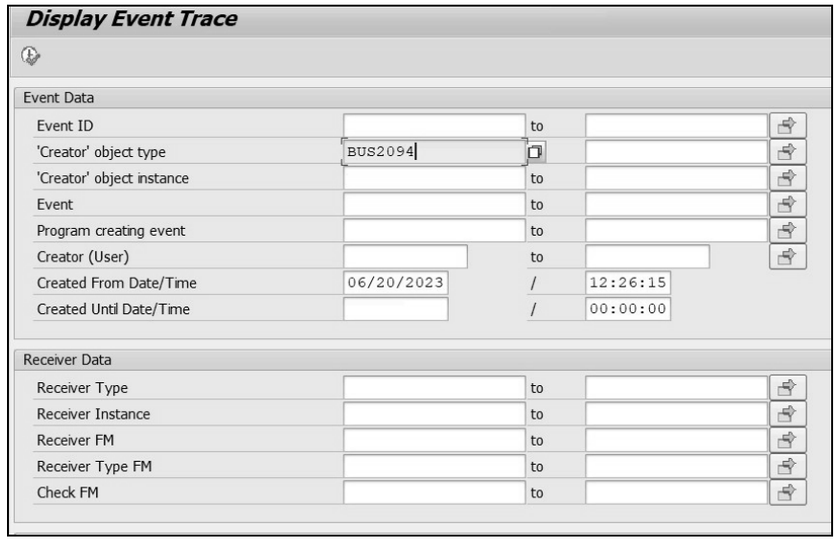


Figure 5.9 Checking the Event Trace from Transaction SWEL

Then execute and check the trace as shown in Figure 5.10. If the entry exists with your business object type and event, the object key matches the document number that you saved (in this case, the credit memo request in Transaction VA01/VA02) on the details screen (see Figure 5.11), then your test result is successful.

Event trace reveals that your custom event has been successfully triggered (see Figure 5.10), and the credit memo request number in the object key (see Figure 5.11) matches the document you've posted.

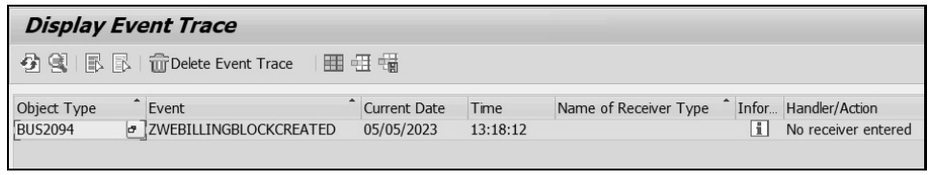


Figure 5.10 Event Trace Showing Event Has Triggered

On clicking the **Details** button (or double-clicking on the event trace line), you can see the instance information shown in Figure 5.11 with object type, object key, event, and event creator.

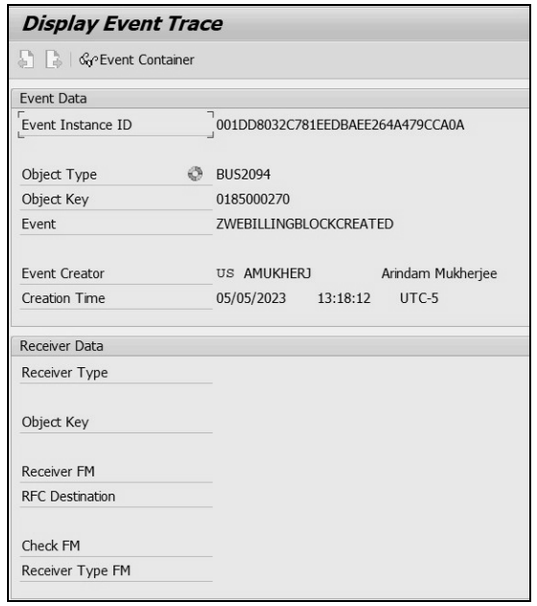


Figure 5.11 Event Trace Details Showing Event Creator Data

Note

The **Receiver Data** section is empty here as we haven't yet configured any receiver (workflow template or single-step task) for this event. This section will be dealt with in detail in Section 5.2 when we talk about receivers and event linkage.

5.1.2 Event Trigger via Status Management

Sometimes, you may want to trigger your workflow event via status management. Status management with respect to a workflow includes both system statuses and user statuses. System statuses are specific standard processing statuses of a document. For example, when you release a production order in SAP, the system status `I0002` is set for that order. On the other hand, user statuses are usually customer defined and are included under a status profile, which is either assigned to the order type (header level) or to an item category (item level). Detailed configuration steps required to create a status profile and assignment to an SAP transaction is beyond the scope of this book. However, note the following points when dealing with status management related to workflow events:

- Both system statuses (`IXXXX`) and user statuses (`EXXXX`) may be used to trigger workflow events.
- An event may be triggered when a specific system/user status becomes active or when an **Active** status is set to **Inactive**.

- System or user status entries for a particular document are logged in table JEST. The key field of this table is object number (OBJNR), which can be retrieved from the application table for which the status is logged, for example, table AUFK for production order header, table VBAK for sales order header, table VBAP for sales order item, and so on. Other important tables related to status include table JSTO (status object information) and table JCDS (change documents for status).
- Status profile may be maintained from Transaction BSO2.
- Workflow events may be configured for system statuses in Transaction BSVX. Figure 5.12 shows an example. These entries are usually provided by standard SAP. Here, you maintain the business object type and the event name. Under the **Status restrictions** node (see Figure 5.13), you can maintain one or more system statuses for which the event will be triggered.

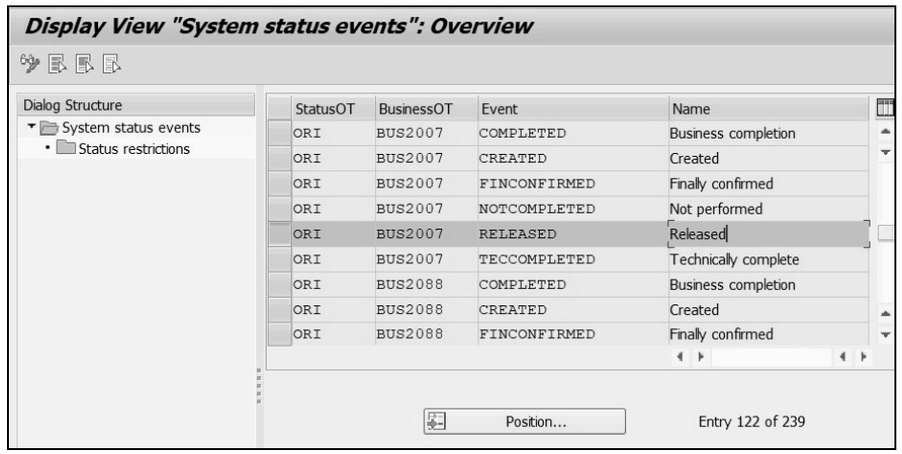


Figure 5.12 Event Configuration with System Status in Transaction BSVX

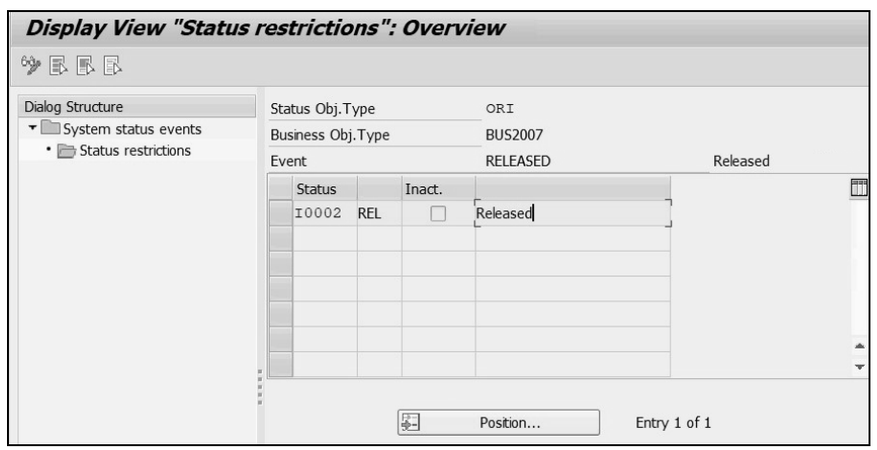


Figure 5.13 System Status Restrictions for Event Configuration

- Workflow events may be configured for system/user statuses in Transaction BSVZ (see Figure 5.14 for an example). These entries are always maintained by the customer and not provided by SAP. Here, we've configured the event **ZWECHANGED** of business object **BUS2007** and a custom status profile for status object type **ORI** (maintenance order).

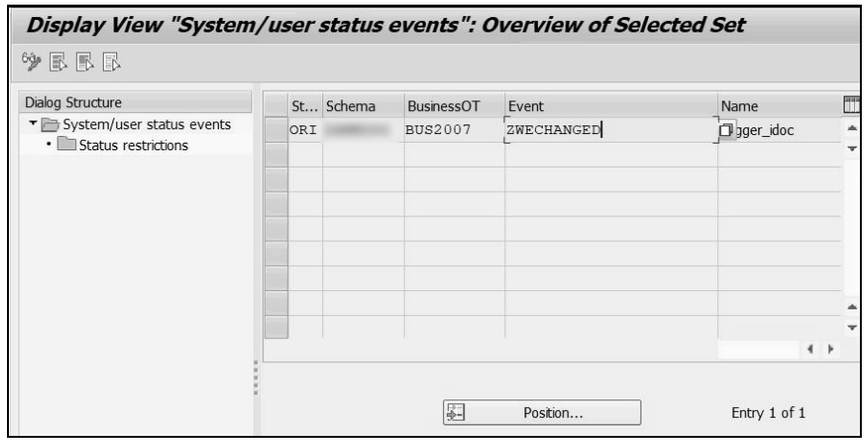


Figure 5.14 Event Configuration with User Status Profile in Transaction BSVZ

- Under **Status restrictions**, shown in Figure 5.15, we've maintained the user status **QTCR** (**E0003**) with the **Inact.** (inactive) checkbox unselected. This means that the event ZWECHANGED of business object type BUS2007 will be triggered when the user status QTCR is set for a particular service order at the header level.

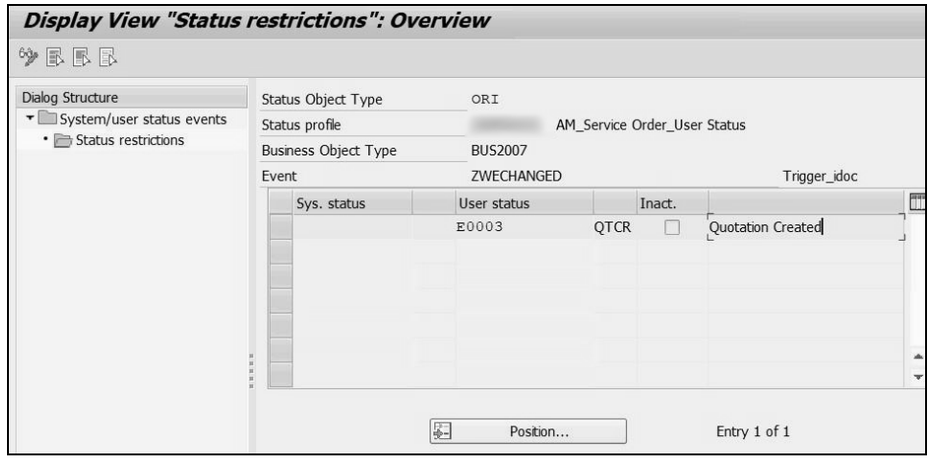


Figure 5.15 User Status Restrictions for Event Configuration

5.1.3 Event Trigger via Message Control

This is one of the less commonly used techniques for triggering events via configuration. In this technique, you use an output type based on traditional table NAST to trigger the event. Steps to configure the output type in Transaction NACE are very much like print, email or Application Link Enabling (ALE) outputs. The only difference is the transmission medium, which in this case, should be **9 Events (SAP Business Workflow)**. Detailed configuration steps for output type configuration in Transaction NACE is beyond the scope of this book, but Figure 5.16 lists the important steps involved in this configuration.

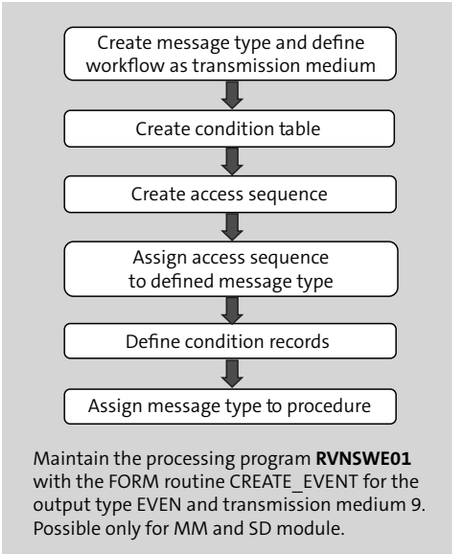


Figure 5.16 Steps for Event Configuration via Table NAST Message Control Configuration

Following are some important points about the restrictions applicable for this event triggering technique:

- This technique only applies to those standard applications that can use output types based on table NAST. With the introduction of new Business Rules Framework plus (BRFplus) output types in SAP S/4HANA, the list of standard applications may be further reduced because the output determination technique doesn't support workflow channels in SAP S/4HANA yet.
- Message control can only raise business object type events. Class-based events aren't supported with this technique.
- The business object type and event name are maintained in condition records, so business users have the flexibility of activating and deactivating the event trigger separately in each system. They also have the flexibility to raise different events according to different condition criteria.

Let's now look at an example of triggering a custom event in an inbound delivery transaction via message control. For this example, you'll trigger an event called *quality inspection request* after posting the goods receipt against an inbound delivery. This event may further be used to trigger a workflow that would manage the quality inspection approval of the goods received from a vendor. The standard business object for inbound delivery transaction has been identified as BUS2015. We've created subtype ZBUS2015 of the standard BOR type, delegated supertype BUS2015 to the subtype, and defined custom event ZWEQualityInspection in the subtype. (Refer to Chapter 3 for details on subtype creation and delegation.) Figure 5.17 shows the view of subtype ZBUS2015 after adding the custom event from Transaction SWO1.

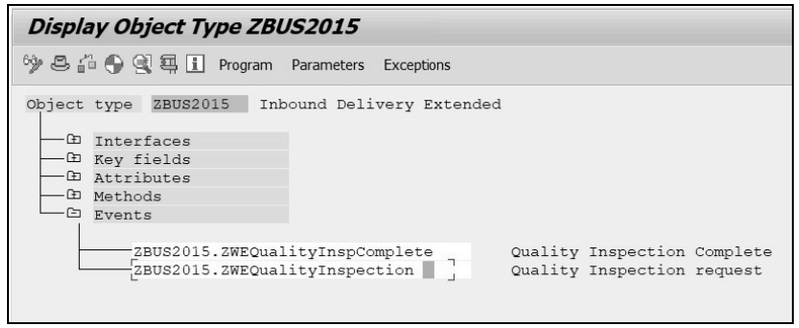


Figure 5.17 BOR Subtype Definition with Custom Events

Next, you'll see the steps to configure your custom output type with transmission medium **9 Events (SAP Business Workflow)** and use it for triggering your quality inspection event after goods receipt posting for the inbound delivery. Note that although Transaction NACE output type configuration details aren't in the scope of this book, we've listed some high-level steps, mostly related to workflow event configuration for the sake of understanding the concepts, as follows:

1. Define your custom output type in the **Application E1** (inbound delivery) in Transaction NACE. Once you select the application, click on the **New Entries** button, and enter the output type name and short description. Figure 5.18 shows the custom output type definition in Transaction NACE.
2. Click on **Processing routines** in the **Dialog Structure**, and then maintain Transmission medium as **9 Events (SAP Business Workflow)**. Maintain the **Program** as "RVNSWE01" and **Form Routine** as "CREATE_EVENT", as shown in Figure 5.19.
3. Maintain the access sequence for the output type in Transaction NACE in the **Access Sequences** screen per your business requirement. In this case, we've assigned an access sequence based on shipping point/delivery type. This step depends on your business requirement. You can use a standard access sequence if it meets your requirement or create a custom one according to your needs. We won't go into

details about access sequence and access creation here as this is an elaborate topic by itself and not related to workflows.

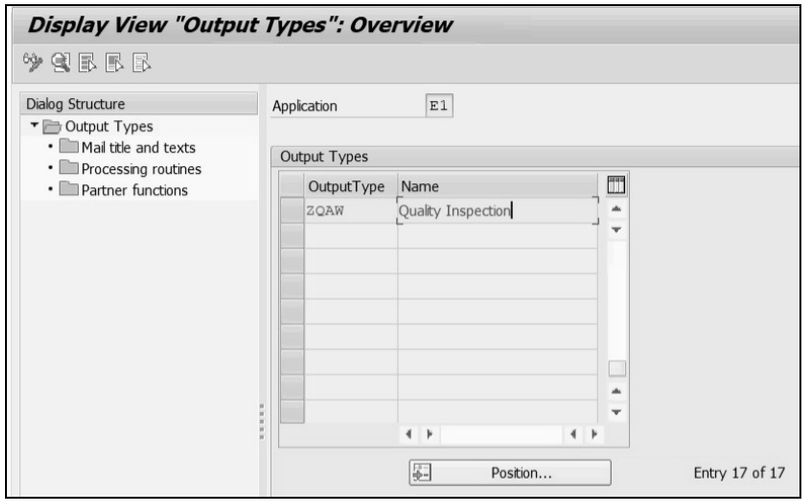


Figure 5.18 Custom Output Type Definition in Application E1 in Transaction NACE

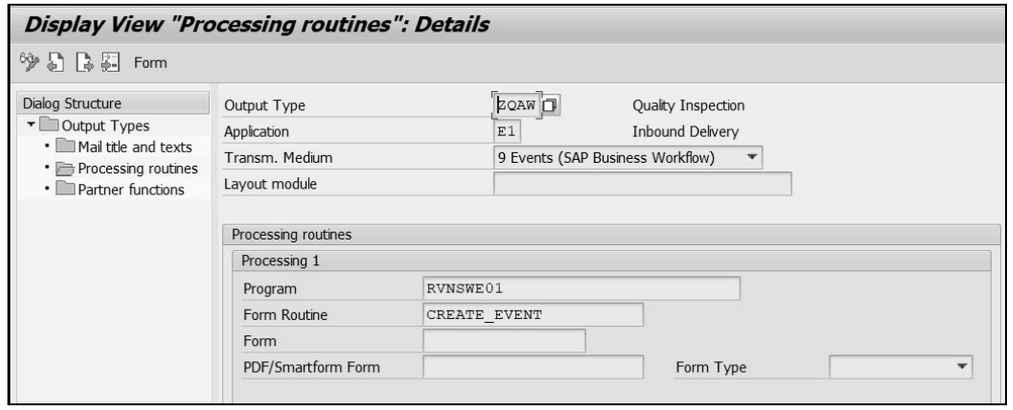


Figure 5.19 Maintenance of Processing Routine for Transmission Medium 9 in Transaction NACE

4. Assign the output type to an output determination procedure by clicking on the **Procedures** button in Transaction NACE and then adding the output type under a suitable procedure (see Figure 5.20) Make sure this procedure is assigned to the relevant delivery type(s) in Transaction SPRO Customizing under **Logistics Execution • Shipping • Deliveries • Define Delivery Types**. Select your delivery type, and click the **Details** button. Then, enter the procedure name in the **OutputDet.Proc.** field. In

Figure 5.20, we've also maintained a requirement routine in the output procedure against our custom output type to restrict the output trigger after post goods receipt only.

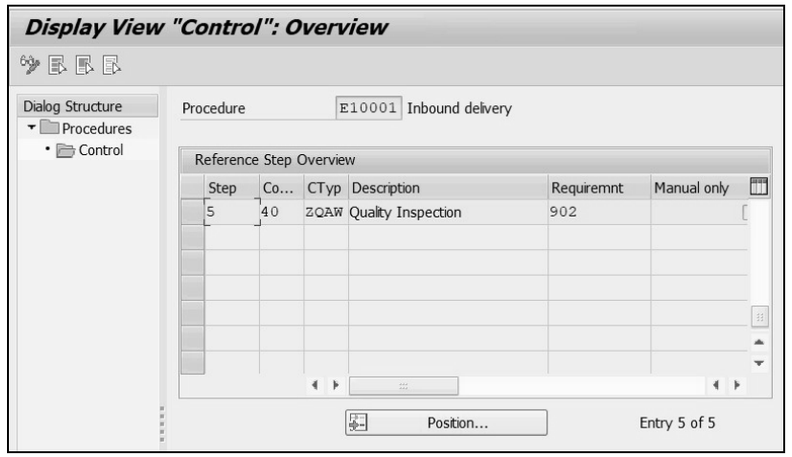


Figure 5.20 Addition of Custom Output Type to the Output Procedure along with Requirement Routine

5. Maintain the condition record for the output type in your test client, as shown in Figure 5.21. You can navigate to this screen by clicking on the condition records button in Transaction NACE; then choose your output type and enter the appropriate criteria per your access sequence definition. In the condition record, you must enter the transmission medium as "9".

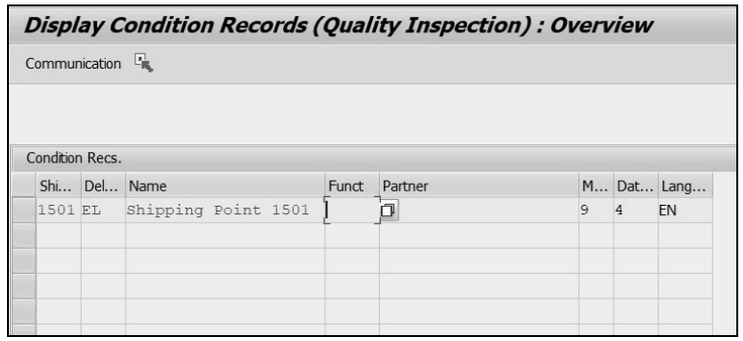


Figure 5.21 Condition Record Maintenance for Custom Output Type

6. Maintain the object type and event under the **Communication** tab, as shown in Figure 5.22. To navigate here, click on the **Communication** button on the application toolbar of the condition record screen shown previously in Figure 5.21.

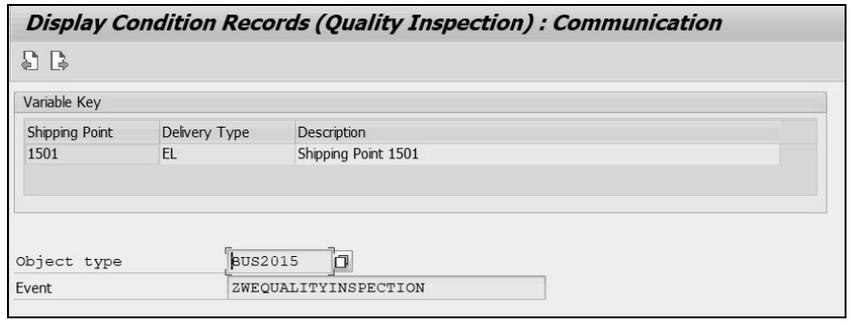


Figure 5.22 Enter the Communication Data for the Condition Record with BOR Type and Event

Now it's time to test your changes. Create an inbound delivery based on a purchase order item in your test system. Perform post goods receipt for the delivery (via inventory management or embedded extended warehouse management (EWM), depending on your storage location settings). After post goods receipt, the output type `ZQAW` will be automatically triggered, which will fire our custom BOR event. Figure 5.23 shows the custom output type **ZQAW** triggered from inbound delivery after post goods receipt. The transmission medium shows **9 Events (SAP Business Workflow)**, as required by our scenario.

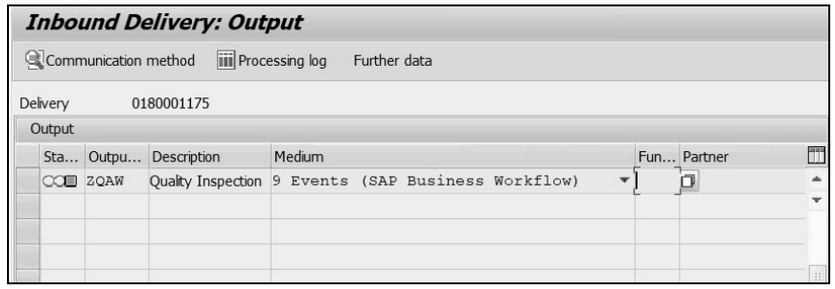


Figure 5.23 Workflow Event Output Generated after Post Goods Receipt of Inbound Delivery

You can check the event trace (Transaction SWEL) for tracking the event (same steps as detailed in the test scenario for the event trigger with change documents). In this case, you filter the trace with your object type `BUS2015`. Figure 5.24 shows the event trace with our custom event **ZWEQUALITYINSPECTION** triggered for object type **BUS2015**.

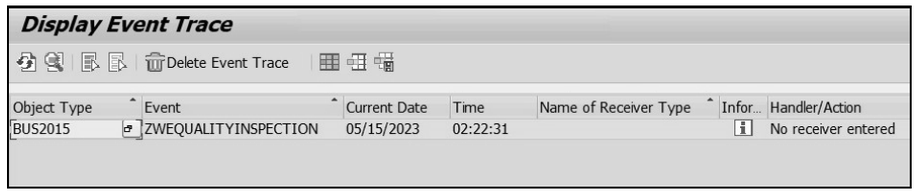


Figure 5.24 Event Trace Showing Event Triggered for the Output Type

On clicking the **Details** button (or double-clicking on the event trace line), you can see the instance information, as shown in Figure 5.25, with **Object Type**, **Object Key**, **Event** name, and **Event Creator** details. The **Object Key** number is the key field of the business object type `BUS2015`, that is, the inbound delivery number.

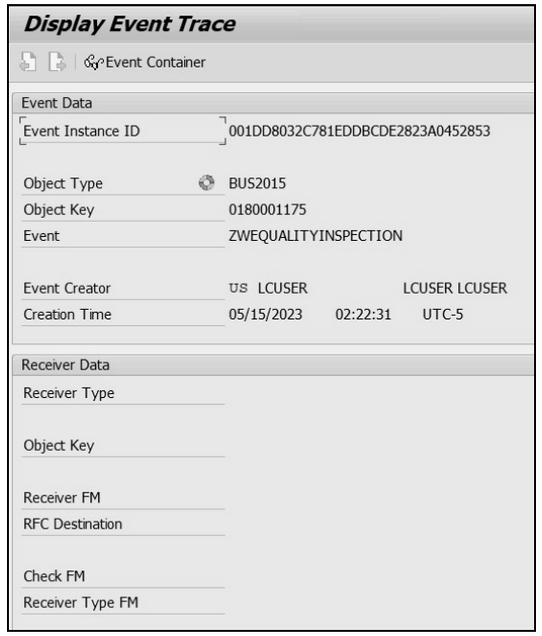


Figure 5.25 Event Trace Details Showing Event Creator Instance Information

> **Note**
>
> The **Receiver Data** section of the screen is empty here because we haven't yet configured any receiver (workflow template or single-step task) for this event. This section will be discussed in more detail in Section 5.2 when we talk about receivers and event linkage.

5.1.4 Event Trigger via ABAP Code in User Exits, Business Add-Ins, and Custom Programs

Finally, you always have the option to trigger a workflow event via ABAP code. For this, you need a BAdI, user exit, function exit, or some other form of enhancement in a standard SAP application program. If you've developed your own custom program, then you can call the workflow event API from any appropriate point in that program (preferably a save or update module). An explicit `COMMIT WORK` is required to register the event in the system, which may be part of your custom program, or for an enhancement, it should be part of the standard Transaction LUW processing.

Some of the common function modules or APIs that can be used to raise a workflow event are as follows:

- Function module SWE_EVENT_CREATE (used for BOR events)
- Workflow API SAP_WAPI_CREATE_EVENT (used for BOR events)
- Update function module SWE_EVENT_CREATE_IN_UPD_TASK (used for BOR events)
- Method RAISE of class CL_SWF_EVT_EVENT (used for both BOR events and ABAP class events)
- Workflow API SAP_WAPI_CREATE_EVENT_EXTENDED (used for both BOR events and ABAP class events).

All these APIs perform the same thing in the SAP system, that is, raise a workflow event with required instance data and additional parameters. The event API interface parameters are as follows:

- **Object type**
  This is the name of the BOR type, for example, BUS2032.
- **Object key**
  This is the concatenated key of the business object instance. For example, for BUS2032, it's the sales order number, whereas for BUS2009, it's the purchase requisition number and the item number.
- **Event**
  This is the name of the BOR event that you want to trigger. This event must be defined as part of a BOR definition.
- **User**
  The user ID should create the event (available as _EVT_CREATOR in the event container). By default, this is the system user calling the workflow API.
- **Event container**
  This container may be used to pass any event parameters. For example, if you have sales orders being created via multiple interfaces and directly within SAP as well, then you may want to add a parameter to identify the system or the application that triggered the sales order. This will be an event parameter and can be passed via the event container table parameter of the workflow API.
- **Event ID and return code**
  The output of the workflow API is the event ID number (internal) and a return code indicating success or error. In case of error, additional parameter(s) for messages provide the details.

For class-based events, the method parameter requires the business class name and event in place of the BOR type name and event. The rest of the parameters are like the event function module.

Now let's look at a couple of source code examples to learn how to trigger workflow events via APIs. In Listing 5.1, we'll raise event ZWEQualityInspComplete for BOR type BUS2015, which will be triggered at the end of the quality inspection review process discussed in our case study in Section 5.1.3. Event ZWEQualityInspComplete has event parameter ZWCDecisionCode, which will pass either APPROVED or REJECTED values through the event. In Listing 5.2, we'll raise an event for a custom workflow business class to release a sales and distribution invoice to accounting.

> **Note**
> For both of these examples, the enhancement or the custom program from where the event should be triggered depends on your application design and business requirements.

In Listing 5.1, we'll raise an event for the business object type, along with an additional event parameter. The inbound delivery number, which is the key field of object type BUS2015, is passed as an input to the program as P_INBDEL, along with the decision code that is passed via parameter P_ACTION. Workflow API SAP_WAPI_CREATE_EVENT triggers event ZWEQualityInspComplete for the entered delivery number and maps the action code to the event container.

```
INCLUDE: <cntn01>. " Include for Container Macros

***Selection screen
PARAMETERS: p_inbdel TYPE likp-vbeln OBLIGATORY, " Delivery
            p_action TYPE char10 OBLIGATORY.     "Approved or Rejected

***Data declarations
DATA: v_object_key    TYPE swo_typeid, " Object key
      v_subrc         TYPE sysubrc,    " Return Code
      s_evt_cont      TYPE swr_cont,   " Container (name-value pairs)
      t_evt_container TYPE swrtcont.

***Constant declarations
CONSTANTS: c_object_type TYPE swo_objtyp VALUE 'BUS2015', " Type
           c_event       TYPE swo_
event VALUE 'ZWEQualityInspComplete', " Event ,
           c_evt_param   TYPE swc_
elem VALUE 'ZWCDecisionCode'.         " Element


*Fill object key for event
DATA(v_delivery) = CONV vbeln_vl( |{ p_inbdel ALPHA = IN }| ).
v_object_key = v_delivery.
```

```
*Fill event container with decision code
s_evt_cont-element = c_evt_param.
s_evt_cont-value = p_action.
APPEND s_evt_cont TO t_evt_container.
CLEAR s_evt_cont.


CALL FUNCTION 'SAP_WAPI_CREATE_EVENT'
  EXPORTING
    object_type     = c_object_type "BUS2015
    object_key      = v_object_key
    event           = c_event
    commit_work     = abap_true
    event_language  = sy-langu
    language        = sy-langu
    user            = sy-uname
  IMPORTING
    return_code     = v_subrc
  TABLES
    input_container = t_evt_container.
```

Listing 5.1 Example Source Code to Trigger a BOR Event with Parameters via the Workflow API

In Listing 5.2, we'll raise an event for an ABAP class, along with an additional event parameter. The billing document number, which is the key attribute for ABAP class `ZCLOTC_INV_RELEASE_ACCOUNTING` is passed as input to the program. The additional event parameter for auto release is set to true (X). The method `RAISE` of class `CL_SWF_EVT_EVENT` raises the event `APPROVE_INV_WORKFLOW` for the entered billing document number and maps the parameter for auto release to the event container.

```
PARAMETERS: p_vbeln TYPE vbrk-vbeln. "SD Billing document number

***Data declarations
DATA : lr_event_parameters TYPE REF TO if_swf_ifs_parameter_container, "
Container for Transfer of Parameters
       lr_catch            TYPE REF TO cx_root,   " Abstract Superclass for All
Global Exceptions
       lv_msg              TYPE string,
       lv_objkey           TYPE char32,           " Objkey of type CHAR32
       lv_id               TYPE char01,           " Id of type CHAR01
       lv_param_name       TYPE swfdname.         " Element ID (32 Characters,
Unique, Not Language-Dependent)
```



```
***Constant declarations
CONSTANTS : lc_objtype TYPE sibftypeid
   VALUE 'ZCLOTC_INV_RELEASE_ACCOUNTING', " Type
            lc_event   TYPE sibfevent
   VALUE 'APPROVE_INV_WORKFLOW', " Event
            lc_catid   TYPE sibfcatid
   VALUE 'CL'. " Category of Objects in Persistent Object References

cl_swf_evt_event=>get_event_container(
   EXPORTING
     im_objcateg  = cl_swf_evt_event=>mc_objcateg_cl
     im_objtype   = lc_objtype
     im_event     = lc_event
   RECEIVING
     re_reference = lr_event_parameters ).

* set up the name/value pair to be added to the container
lv_param_name  = 'AUTO_RELEASE'. "parameter name of the event
lv_id          = abap_true.

* Add the name/value pair to the event conainer
TRY.
    lr_event_parameters->set(
      EXPORTING
        name  = lv_param_name
        value = lv_id ).

  CATCH cx_swf_cnt_cont_access_denied INTO lr_catch.
    lv_msg = lr_catch->get_text( ).
  CATCH cx_swf_cnt_elem_access_denied INTO lr_catch.
    lv_msg = lr_catch->get_text( ).
  CATCH cx_swf_cnt_elem_not_found INTO lr_catch.
    lv_msg = lr_catch->get_text( ).
  CATCH cx_swf_cnt_elem_type_conflict INTO lr_catch.
    lv_msg = lr_catch->get_text( ).
  CATCH cx_swf_cnt_unit_type_conflict INTO lr_catch.
    lv_msg = lr_catch->get_text( ).
  CATCH cx_swf_cnt_elem_def_invalid INTO lr_catch.
    lv_msg = lr_catch->get_text( ).
  CATCH cx_swf_cnt_container INTO lr_catch.
    lv_msg = lr_catch->get_text( ).
ENDTRY.

DATA(lv_vbeln) = CONV vbeln_vf( |{ p_vbeln ALPHA = IN }| ).
```

```
*       assigning the billing no to object key
lv_objkey = lv_vbeln.
*       Raise event to trigger the workflow
TRY.
    cl_swf_evt_event=>raise(
      EXPORTING
        im_objcateg        = cl_swf_evt_event=>mc_objcateg_cl
        im_objtype         = lc_objtype
        im_event           = lc_event
        im_objkey          = lv_objkey
        im_event_container = lr_event_parameters ).

  CATCH cx_swf_evt_invalid_objtype INTO lr_catch.
    lv_msg = lr_catch->get_text( ).
  CATCH cx_swf_evt_invalid_event INTO lr_catch.
    lv_msg = lr_catch->get_text( ).
ENDTRY.
```

Listing 5.2 Example Source Code to Trigger a Class-Based Event with Parameters via a Method Call

5.2 Event Creators, Receivers, and Event Linkage

An event creator is the object that raises an event. In the previous section, we discussed the various options for raising an event from an application. Technically, the event creator is the user who raised the event, with any of the previously mentioned event publishing mechanisms. The name of the event creator is visible in the event trace (Transaction SWEL), and normally when the event is used to trigger a workflow, the event creator is bound to the workflow container element `_Wf_Initiator` (workflow initiator). Figure 5.26 shows the event creator details in the event trace Transaction SWEL.

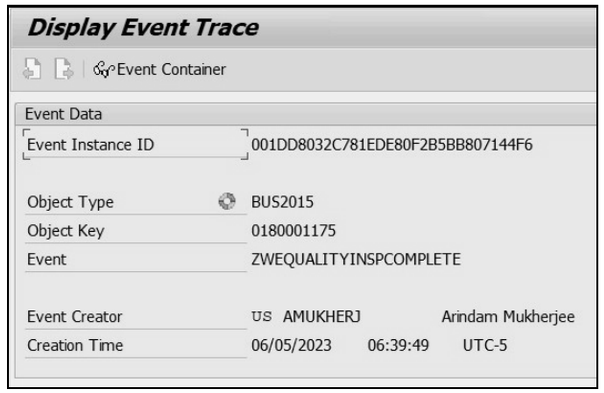


Figure 5.26 Event Trace Showing Event Creator Information

An event receiver is the object that receives the event and performs subsequent processing. Technically, the event receiver could be a workflow template (multistep task), a single-step task, a function module, or a handler class method. In the event trace Transaction SWEL, you can see the event receiver details (if any) under the receiver data. In Figure 5.27, you can see the event receiver details in the event trace in Transaction SWEL. Here, the receiver is a workflow template (multistep task) and the receiver **Object Key** shows the work item ID of the workflow triggered from the event.

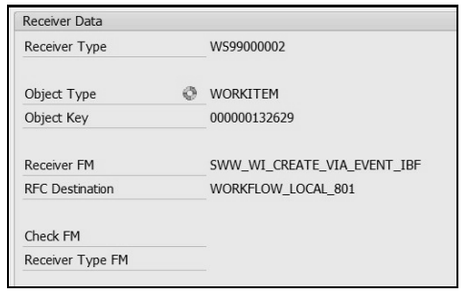


Figure 5.27 Event Trace Showing Event Receiver Information

Event linkage is the link between an event creator (semantically, the BOR object type/class and event) and a receiver (workflow template/task/function module/method call). Event linkages can be created in Transaction SWE2 or Transaction SWETYPV. This linkage is also implicitly created when you enter a triggering event on a workflow template or single-step task from Transaction PFTC. Figure 5.28 shows a sample event linkage entry from Transaction SWE2 with various details. We'll look at the components on the screen to understand the significance of that entry in the event linkage.

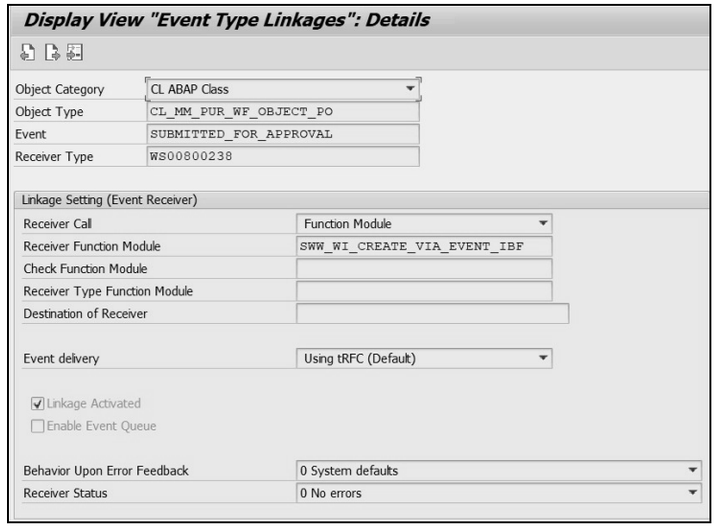


Figure 5.28 Event Linkage from Transaction SWE2 or Transaction SWETYPV

The following details are visible in an event linkage entry under the **Linkage Setting (Event Receiver)** section of the screen, as shown in Figure 5.28:

- **Receiver Call**
  This dropdown field lets you choose between a function module or method call as the event handler and acts as a trigger for the receiver type. If you choose **Function Module** in this option, then you must enter a receiver function module in the **Receiver Function Module** field that appears. This function module must use the interface mentioned in the **Receiver Function Module** bullet point later in this list. For single-step or multistep tasks as receiver, the function module is the default receiver call, and `SWW_WI_CREATE_VIA_EVENT_IBF` is the default function module.

  If you choose a **Method** call in this field, then you must enter a class that implements interface `BI_EVENT_HANDLER_STATIC`. The **Method Name** field is defaulted as **ON_EVENT**, which must implement the logic to handle the event. Figure 5.29 shows an example of an event linkage with a method call as the receiver call.

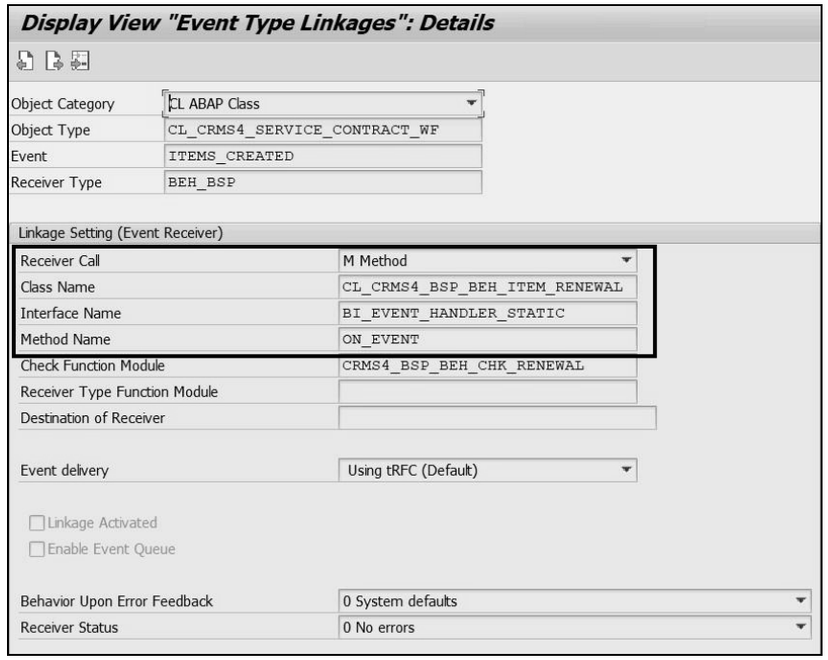


Figure 5.29 Sample Event Linkage Showing the Receiver Method Call

- **Receiver Type**
  For a multistep task or single-step task, you'll see the workflow or task ID in this field starting with a prefix of WS or TS, respectively. This is the workflow template or the task that has been triggered by the event. The event must be maintained in the **Triggering Events** tab of the workflow template or task, and the event linkage must be

active (green). If the receiver is a function module or a class method, then this field acts as a dummy entity. No object exists in the system with this ID.

- **Object Type/Object Key**
  These fields appear in the event trace in the **Receiver Data** area of the screen (refer to Figure 5.28). If the receiver type is a workflow or task, then **Object Type** is **WORKITEM**, which is the runtime object type for a workflow or task. The **Object Key** then represents the work item ID of the triggered workflow instance (or single-step task). If the receiver is a function module or class method, the **Object Type** and **Object Key** fields are both empty.

- **Receiver Function Module**
  This field appears when you choose **Function Module** from the **Receiver Call** type dropdown. When the receiver type is a workflow or a single-step task, this function module is defaulted as **SWW_WI_CREATE_VIA_EVENT_IBF**. If you choose some other receiver type, then you can choose your own function module or class method based on the **Receiver Call** dropdown. You can choose the receiver as function module or method when you simply want to update some transaction data via a BAPI or other API once the event is raised from an application. For example, you may want to create an outbound delivery automatically when a sales order is saved. The receiver function module must implement the same interface as template function module `SWE_TEMPLATE_REC_FB` (BOR-based objects only) or template function module `SWE_TEMPLATE_REC_FB_2` (both BOR- and class-based objects).

- **Check Function Module**
  A check function module is a source code triggered synchronously by the event creator if an active event linkage is found and before triggering the event receiver. The check function module may be used to determine if the event receiver should be triggered or not based on the event instance data. For example, you've developed a workflow for approval of purchase orders, but you only want to trigger the workflow for specific purchase order types. The check function module may be attached to the event linkage to check the purchase order type (`EKKO-BSART`) before you decide whether to trigger the workflow (receiver) for a given purchase order. The check function module must implement the same interface as template function module `SWE_TEMPLATE_CHECK_FB` (BOR-based objects only) or template function module `SWE_TEMPLATE_CHECK_FB_2` (both BOR- and class-based objects).

- **Receiver Type Function Module**
  This is used when you have multiple receivers linked to the same event (event linkages), and you only want to trigger one receiver at runtime based on the object instance data. For example, you've developed three different workflow templates for the approval of three types of purchase orders, such as, direct purchases, indirect purchases, and stock transfer orders. All three workflows have the same triggering event. Now when the purchase order `Created` event is raised, you need to check the

purchase order document type and decide which workflow to trigger. This requirement may be achieved via receiver type function module. The output of the function module is the receiver type (workflow or single-step task or function module/method). The receiver type function module must implement the same interface as template function module `SWE_TEMPLATE_RECTYPE_FB` (BOR-based objects only) or template function module `SWE_TEMPLATE_RECTYPE_FB_2` (both BOR- and class-based objects).

- **Destination of Receiver**
  In this field, you can enter the logical Remote Function Call (RFC) destination of the receiver, if the receiver is in a different system from the event creator.
- **Event delivery**
  Event delivery from creator to receiver can happen via transactional RFC (tRFC; default) or via queued RFC (qRFC). This setting goes together with the **Enable Event Queue** checkbox in the same screen. We'll study more about event delivery and event queue administration in Chapter 8, Section 8.7.
- **Linkage Activated**
  This indicates that the event linkage is active. Receiver determination and triggering only happens when the event linkage is active. This flag can also be updated from Transaction PFTC when you activate the triggering event on a workflow template or a task.
- **Behavior Upon Error Feedback**
  This setting determines how the system should react if an error occurs while delivering an event to the receiver. Default setting is **0 System defaults**, which means that the global setting from event administration (Transaction SWEAD) is used. Other options for this field include the following:
  - **1 Deactivation of Linkage**: If an error occurs, then the event linkage will be automatically deactivated.
  - **2 Mark linkage as having errors**: If an error occurs, then the event linkage is marked as `Errors`. This setting influences the next field on the event linkage, that is, **Receiver Status**.
  - **3 Do not change linkage**: There is no impact on the event linkage with this setting even if an error occurs.

  Normally, you'll maintain the setting **0 System defaults** in each individual event linkage and control the error feedback behavior via the global setting in Transaction SWEAD. More details on event administration will be covered in Chapter 8.
- **Receiver Status**
  Based on the error feedback setting in the previous field (or global setting), the **Receiver Status** field may show as **0 No errors** or **1 Errors** after an error occurs while trying to trigger an event receiver.

5.3 Start Conditions in Workflows

In the previous section, you learned about check function modules in event linkage, which can be used to evaluate any condition before deciding to trigger the receiver for a particular object instance. For example, you might want to trigger a workflow for the release of purchase orders, but you want to restrict the approval for specific purchase order types only. This condition may be evaluated in a check function module. If you raise the exception `NO_RECTYPE` in this check function module, then the workflow is not triggered by the event.

An alternative and more recommended option compared to the check function module is `Start Conditions` in workflows. This is more of a configuration approach, provided that the fields or the variables you want to use for the condition are already available as an attribute in the BOR type or the ABAP class, which defines the event. If not, then you first need to create a custom attribute in the business object or ABAP class that can be used for configuring the start condition in Transaction SWB_COND (or via the **Start Events** tab under the header details of a workflow definition in Transaction SWDD). The event linkage should exist before you can create a start condition for the same. The start condition itself is a Customizing object and can be transported to other systems or clients in a Customizing transport request.

In Section 5.1.3, we looked at an example of how to trigger custom event `ZWEQualityInspection` from the BOR type `BUS2015` (delegated to subtype `ZBUS2015`). Now, let's suppose that we've created an event linkage of this event with a custom workflow, but we only want to trigger the workflow for certain delivery types. There are many ways to apply this condition filter, but, in this example, we'll explore the approach of start conditions using Transaction SWB_COND:

1. Ensure that an event linkage exists in Transaction SWE2 or Transaction SWETYPV for the concerned event and workflow/task because that is a prerequisite to creating a start condition. Figure 5.30 shows an event linkage entry from Transaction SWE2 without a start condition. Note that the **Check Function Module** field is empty in this case.
2. Ensure that the field(s) to be used in the start condition exist as attribute(s) in the BOR type of the event. In this example, you need the `Delivery` type as an attribute in business object type `BUS2015`. Because this attribute doesn't exist in the standard, you must create a custom attribute in the delegated subtype `ZBUS2015`, as shown in Figure 5.31. (Refer to Chapter 3, Section 3.1.2, for details on how to create a database attribute in a business object type.) We've created custom database attribute `ZWADeliveryType` for this purpose.

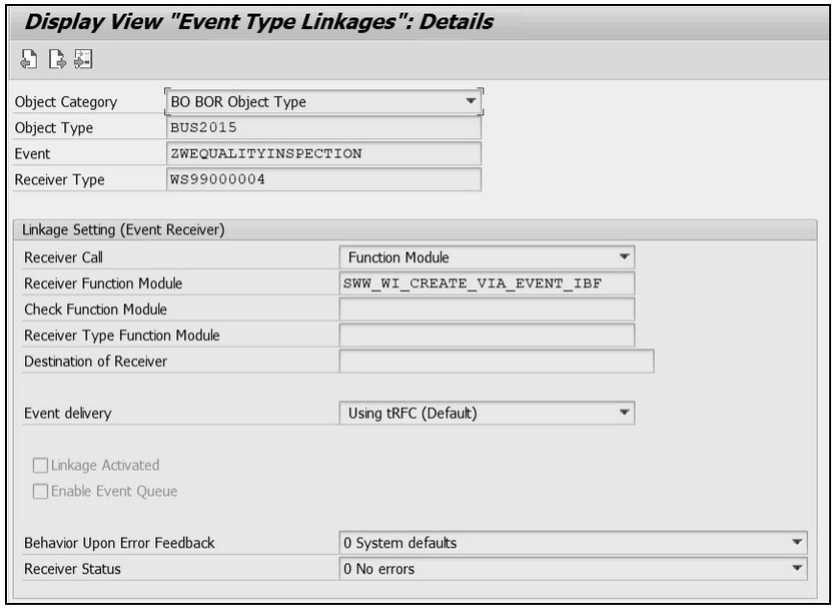


Figure 5.30 Event Linkage before Assigning a Start Condition

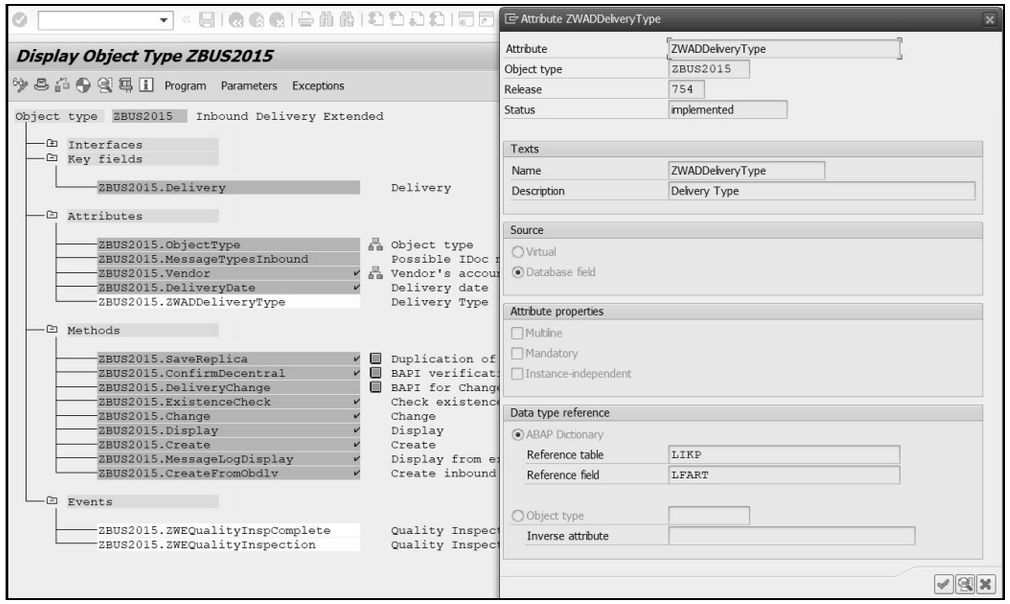


Figure 5.31 Creation of a Custom Attribute in the BOR Subtype Definition

3. Now you can create the start condition using `Delivery` type attribute for the event linkage in Transaction SWB_COND. Figure 5.32 shows a view of the event linkage entry in Transaction SWB_COND before creating a start condition for the same. (You

can search for the event linkage based on the business object type/ABAP class, event name, receiver workflow, or task ID.)

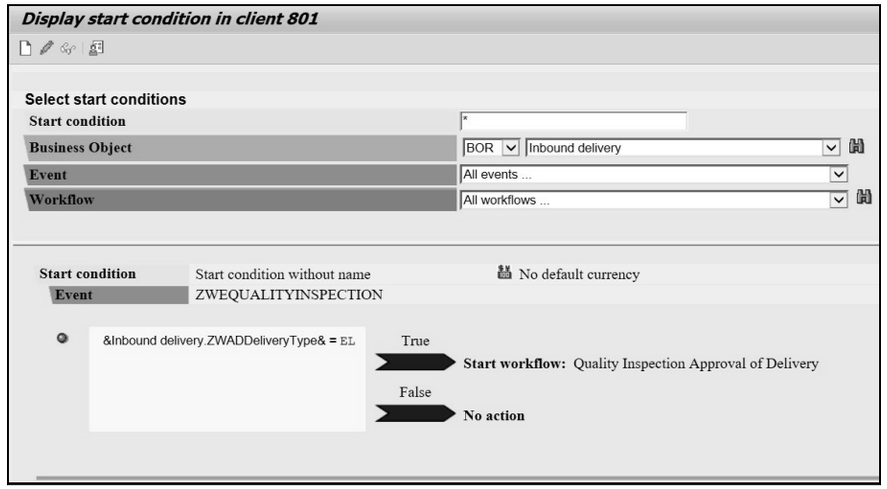


Figure 5.32 Creation of Start Condition for Event Linkage from Transaction SWB_COND

4. Click on the **Create** button on the toolbar, and then click on the selected event linkage. A popup screen will appear with the condition editor that allows you to create flexible start conditions. In the screen shown in Figure 5.33, enter the start condition per your requirement. (In this case, we're checking for delivery type = **EL**).

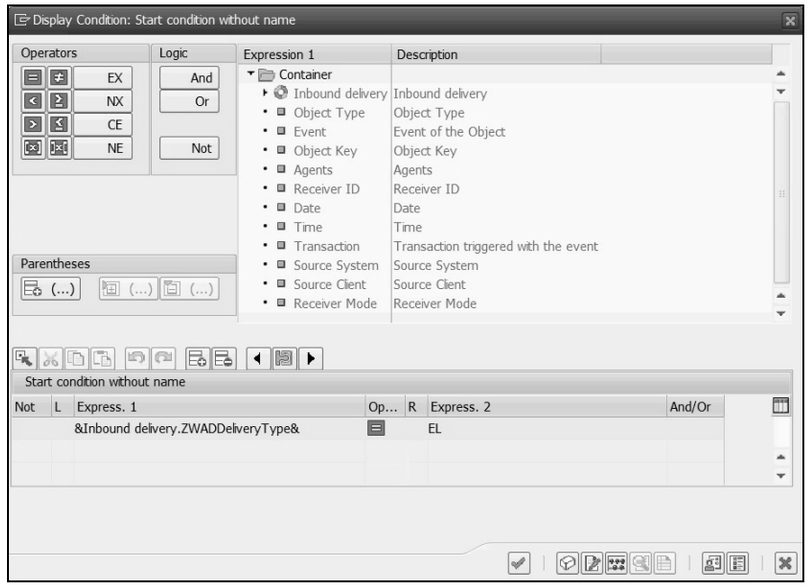


Figure 5.33 Maintaining the Start Condition Using the Condition Editor in Transaction SWB_COND

5. Confirm the condition popup screen once editing is complete.
6. In Figure 5.34, activate the condition by clicking on the red-light icon until it turns green. Save the condition and capture it in a Customizing transport for moving to other systems/clients.

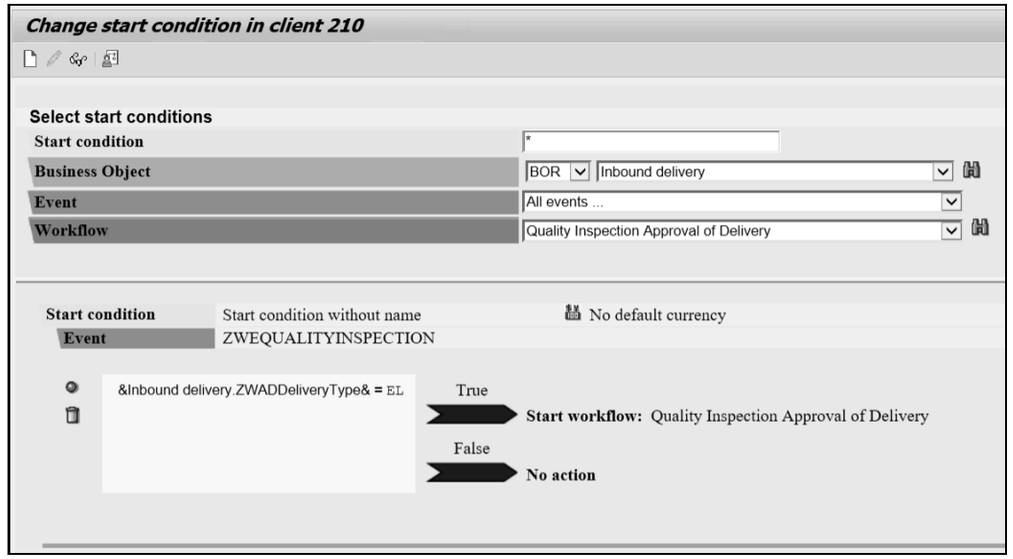


Figure 5.34 Activate the Start Condition (Indicated by the Green Icon to the Left)

Now it's time to test the start condition. When you trigger event `ZWEQualityInspection` for a delivery with type `EL`, the start condition will evaluate to `True`, which may be viewed from the event trace in Transaction SWEL. Figure 5.35 shows the event trace with a successful receiver trigger after the start condition evaluation.

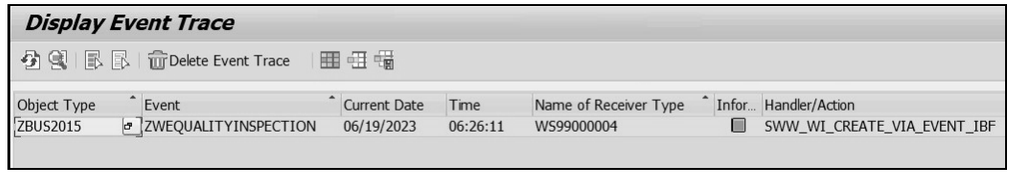


Figure 5.35 Event Trace Showing Event Triggered after Evaluation of Start Condition

Now if you trigger the event for a delivery with any other type, for example, `DIG`, then the start condition will evaluate to `False`, and the receiver workflow won't be triggered from the event trace, as shown in Figure 5.36.

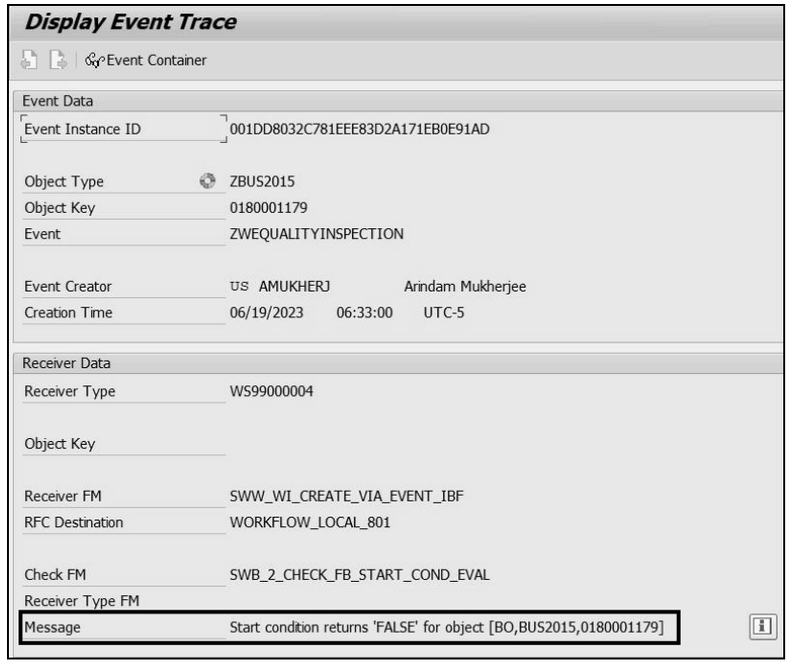


Figure 5.36 Event Trace Showing Exception Raised due to Start Condition Being Evaluated to False

5.4 Terminating Events and Instance Linkage

Up to this point, we've discussed triggering events and their linkage to workflows (multistep tasks) or single-step tasks. These events are maintained in the **Triggering events** tab of a workflow or single-step task. Tasks can also have terminating events that are used to signal the end of processing for a dialog work item. Usually, terminating events are attached to asynchronous dialog tasks. This means that the underlying BOR method is declared as asynchronous (**Synchronous object method** checkbox is unchecked in the method definition in Transaction SWO1). For ABAP classes, however, this synchronous/asynchronous attribute doesn't exist at the method level. It's flagged at the task level only from Transaction PFTC.

In Figure 5.37, the task for releasing a purchase order is marked as asynchronous.

On the **Terminating events** tab, events **RELEASED**, **RESET**, and **SIGNIFICANTLYCHANGED** are maintained as terminating events for this task, as shown in Figure 5.38. Terminating events are entered by selecting the object type container element, from where the BOR object type or the ABAP class name is derived. Then, you must select the event name from the BOR object type or ABAP class. Binding can be maintained between the event container and task container, similar to triggering events.

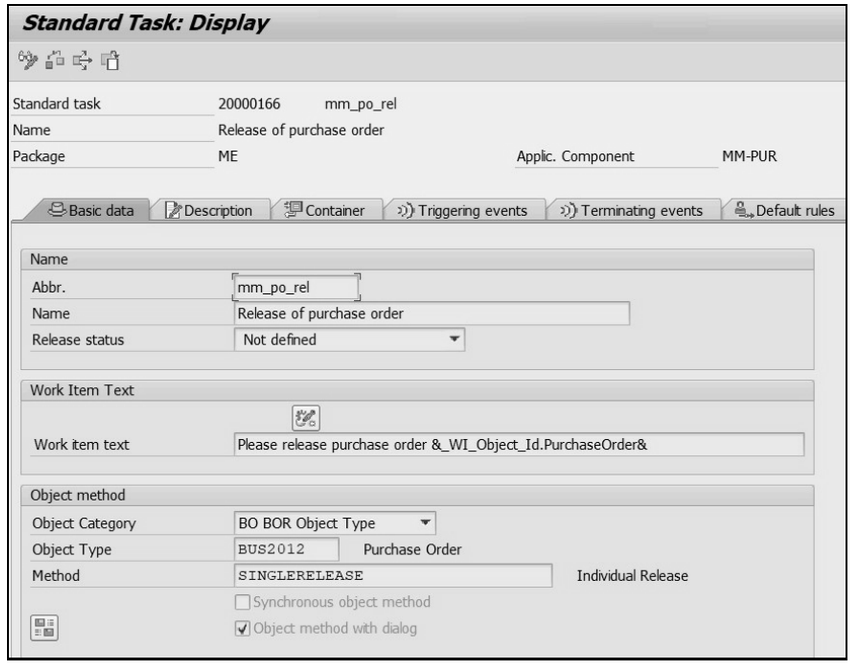


Figure 5.37 Example of an Asynchronous Standard Task

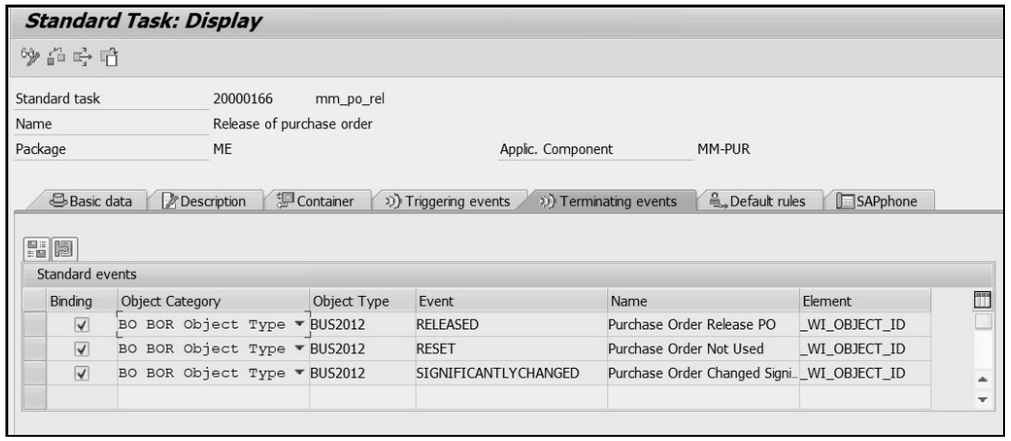


Figure 5.38 Terminating Events in an Asynchronous Dialog Task

Whenever the system triggers any one of the preceding events for a purchase order; any **Ready** or **In Process** work items for task TS20000166 will be set to **Completed** status. Terminating events may be raised using the same kind of mechanisms as a triggering event, namely change documents, status management, message control, or ABAP code in enhancements. Binding may be done from event to task container to receive the updated object instance into the task/workflow.

Moving on to our next topic, instance linkages are automatically created by the workflow runtime system, whenever a work item is created for an asynchronous task that has terminating events attached to it. Unlike event linkage for triggering events, instance linkages aren't created manually. They may be viewed or updated manually (as an administrator for exception situations) via Transaction SWE3 or Transaction SWEINST. Each instance linkage consists of a header record with the BOR/class, event, and receiver type, along with the type-linkage active and event queue active indicators. Under the header entry, you'll see the object details data, with the object key fields and the receiver key (work item ID if task is the receiver type). Figure 5.39 shows the view of instance linkages created through workflow runtime data in Transaction SWE3 or Transaction SWEINST.

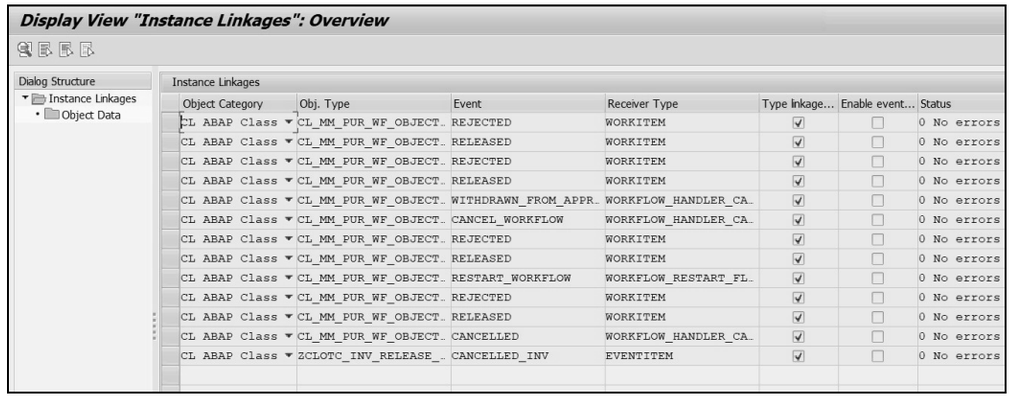


Figure 5.39 Instance Linkages from Transaction SWE3 or Transaction SWEINST

On selecting any row from the table and clicking on the **Object Data** node, you can view the object instances (list of documents) for which the instance linkage header entry was created (see Figure 5.40).

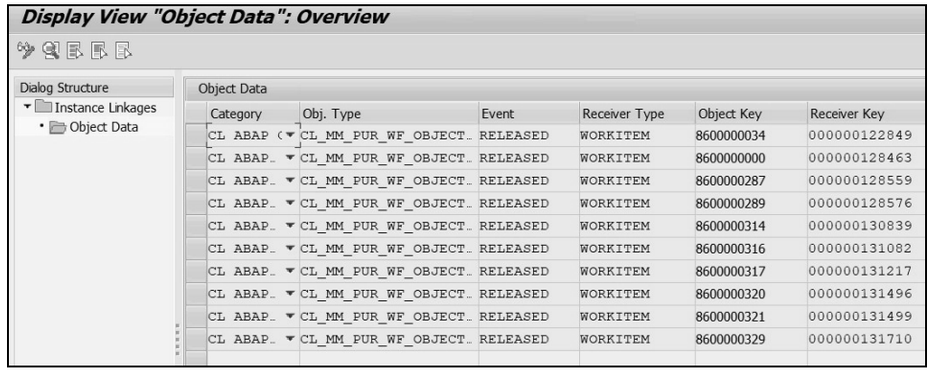


Figure 5.40 Object Instances for an Instance Linkage from Transaction SWE3 or Transaction SWEINST

Instance linkages work automatically in the background based on design-time definitions entered in the task. Usually, no manual intervention is required. You can use the instance linkage transaction for analysis to check which objects are currently awaiting a terminating event. Once the terminating event is raised for an object instance, the corresponding entry is deleted from this table.

5.5 Check Function Module and Receiver Function Module for Events

In Section 5.2, you learned the definition and purpose of check function modules and receiver function modules, which are maintained in the event linkage entry. Both function modules may be entered automatically by the system or manually by the developer.

When you configure a start condition for an event linkage via Transaction SWB_COND (or directly via the **Start Events** tab under the header details of a workflow definition in Transaction SWDD), then default check function module `SWB_2_CHECK_FB_START_COND_EVAL` is inserted in the event linkage entry. If no start condition is maintained in Transaction SWB_COND, then adding a check function module is a manual task.

Similarly, when you maintain a workflow or a single-step task as the receiver for an event, then default receiver function module `SWW_WI_CREATE_VIA_EVENT_IBF` is automatically added to the event linkage entry. For nonworkflow receivers, you must manually enter a receiver function module or a class and method name as the event handler object.

In Section 5.3, we used the start condition approach to add a filter based on the delivery type on the quality inspection approval workflow event linkage. In this case, we explored an alternate approach using the check function module. The advantage of this approach is that you don't need the condition field(s) to be created as attributes in the leading BOR type or ABAP class. The disadvantage is that you need to write some code for formulating the start condition. In a real business scenario, you must carefully consider both approaches and decide which one is the cleaner and more efficient option for you. Listing 5.3 contains sample source code of a custom check function module. Note the use of exception `NO_RECTYPE` in determining the result of the check.

```
FUNCTION z_delivery_qinsp_wf_check.
*"----------------------------------------------------------------------
*"*"Local Interface:
*"  IMPORTING
*"     VALUE(OBJTYPE) LIKE  SWETYPECOU-OBJTYPE
*"     VALUE(OBJKEY) LIKE  SWEINSTCOU-OBJKEY
*"     VALUE(EVENT) LIKE  SWETYPECOU-EVENT
*"     VALUE(RECTYPE) LIKE  SWETYPECOU-RECTYPE
*"  TABLES
```



```
*"     EVENT_CONTAINER STRUCTURE  SWCONT
*"  EXCEPTIONS
*"     NO_RECTYPE
*"----------------------------------------------------------------------

*--Local constant declarations
  CONSTANTS: lc_inb_delivery TYPE lfart VALUE 'EL'. " Delivery Type

  IF objtype = 'BUS2015' AND
     event = 'ZWEQUALITYINSPECTION'.

    DATA(lv_vbeln) = CONV vbeln_vl( objkey ).

***Fetch delivery type from delivery header table
    SELECT SINGLE FROM likp " SD Document: Delivery Header Data
      FIELDS lfart
      WHERE vbeln = @lv_vbeln
      INTO @DATA(lv_lfart).
    IF sy-subrc = 0.
      IF lv_lfart <> lc_inb_delivery. "EL
        MESSAGE 'Invalid delivery type for Quality Inspection WF' TYPE 'E'
RAISING no_rectype.
      ENDIF. " IF lv_lfart <> lc_inb_delivery
    ENDIF. " IF sy-subrc = 0

  ENDIF. " IF objtype = 'BUS2015' AND


ENDFUNCTION.
```

Listing 5.3 Sample Code for a Check Function Module in Event Linkage

Receiver function modules (or method calls) for nonworkflow receiver types may be used for a variety of purposes in real business scenarios, for example, if you want to trigger an interface after a sales order is saved. You would ideally want to send the sales order number or the updated data from the order in the interface, so you first look for an exit or BAdI in the update task, or you may decide to go for a custom receiver function module configured in an event linkage. Because receivers are triggered via an event that is raised after the standard save of the transaction, you can query all database tables related to the transaction in the receiver function module. It also allows you the flexibility of configuration to activate or deactivate the interface or maintain start conditions per requirement. Another common scenario for using custom receiver function modules is to create a follow-on document from a transaction on save, for example, if you want to create the delivery for a sales order automatically after saving.

Receiver function modules must implement the same interface as template function module `SWE_TEMPLATE_REC_FB` (BOR-based objects only) or function module `SWE_TEMPLATE_REC_FB_2` (both BOR- and class-based objects). For receiver method calls, you must create a custom class using interface `BI_EVENT_HANDLER_STATIC`. Method `ON_EVENT` of this interface must be implemented for the handler logic.

5.6 Summary

In this chapter, we started by explaining the concept of events with respect to workflows. Then we discussed the different event triggering techniques in detail, some involving configuration and others involving ABAP code. Then, we talked about event creators and event receivers, followed by describing each element of an event linkage entry. We also talked about event instance linkages. Finally, we looked at some examples to understand the concept of start conditions, check function modules, and receiver function modules with respect to SAP Business Workflow.

Contents

11 Integrating Workflows with User Interface Applications and External Applications 357

12 Migrating SAP ERP Workflows to SAP S/4HANA 375

13 Workflow in SAP Master Data Governance 391

Part II Flexible Workflow in SAP S/4HANA

14 Introduction to Flexible Workflow 441

Index

C

D

E

F

G

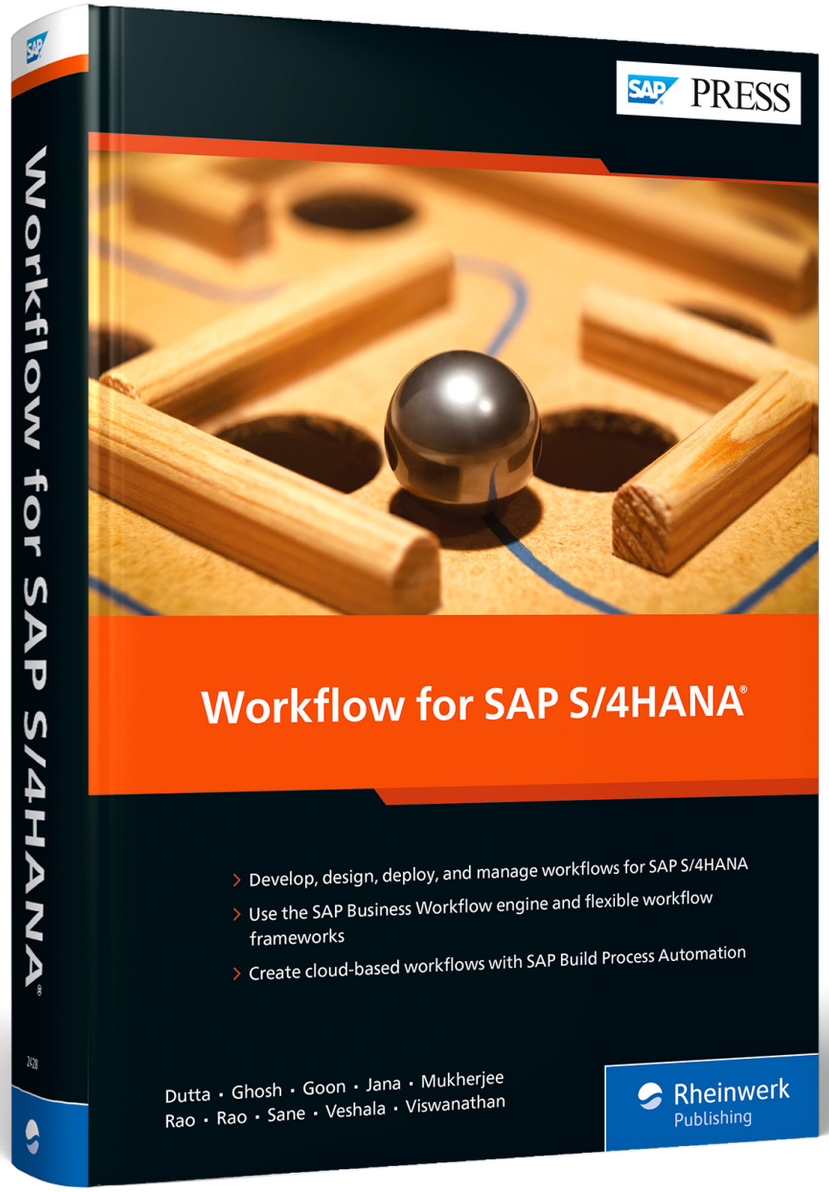
I

T

U

V

W



The Authors

Sabyasachi Dutta, Nilay Ghosh, Kousik Goon, Sandip Jana, Arindam Mukherjee, Srinivas Rao, Yogeendar Rao, Yogesh Sane, Naveen Veshala, and Kiran Viswanathan are a team of expert consultants, architects, and technical leaders at IBM Consulting.

Dutta, Ghosh, Goon, Jana, Mukherjee, Rao, Rao, Sane, Veshala, Viswanathan

Workflow for SAP S/4HANA

686 pages | 11/2023 | $89.95 | ISBN 978-1-4932-2428-9

www.sap-press.com/5697