OUTBOUND ARGUMENTS & USER FUNCTIONS

SAMPLE: SHOPPING CART

Last Update: Aug 2010

Table of Contents

1 Goal

In this tutorial we will continue walking through the development of a very simple application using Integranova. The tutorial covers the basic tools and steps involved in the development of applications from scratch as a way of introducing the reader to Integranova.

Following with the tutorial series, we will now expand the Tutorial 07 with this new tutorial.

In the Tutorial 07 we learned how to create global transactions to interact with several instances not related among them. In this new tutorial we will learn how to:

- ✓ Show a message after deleting a purchase order line and paying a purchase order.
- ✓ Show the shopping cart discount, net amount and gross amount after checking it out.
- ✓ Define a user function to send an email. This user function will be used in an outbound argument.

By the end of this tutorial, you will have learned how to define and use outbound arguments and user functions.

In this tutorial we will start from the model we created in the previous tutorial in the series: Tutorial_07_Final.oom.

2 Concepts

Let's see above the Integranova concepts that we will use in the current tutorial.

2.1 Outbound argument

Outbound arguments are used to manage or show the system response after the successful completion of a service execution.

When using events, the only way of assigning values to outbound arguments is through *value expressions*. Transactions (and operations) can use values expressions to assign value to the outbound arguments, but also, they can use another mechanism to set values to outbound arguments in the Transaction formula. This mechanism consists of using a service with outbound arguments and assign the returned value to its own outbound arguments. They assign the values to the outbound arguments:

- ✓ Keeping the value in an outbound argument of the transaction/operation. The value will be propagated from the first service to the transaction/operation which called it.
- ✓ Ruling out the returned value, selecting a NULL in the call.

2.2 User Function

A *User Function* is the interface to manage functionalities not enclosed in the model. This function must be directly implemented in the generated code. One typical example is to send an e-mail.

Note: A user function cannot be used in the same way as a service. The user functions always have a return value that is what must be used in the model.

In the generated code, the user functions are defined in the client and server side. These are the files where the user functions are located:

- ✓ **BLT_TRCS2**: Others\ONUserFunctions.cs
- ✓ **BLT_TREJB2**: src\java\<*AppServerName*>\global\UserFunctions.java
- ✓ **UIT_CS2**: Logics\UserFunctions\UserFunctionsLogic.cs
- ✓ **UIT_ASP2**: App_Code\Logic\UserFunctions\UserFunctionsLogic.cs
- ✓ **UIT_JSF**: JavaSource\<*AppClientName*>\logic\userFunctions\UserFunctionsLogic.java

Note: In this tutorial we are going only to use the **UIT_ASP2**, **UIT_CS2** and **BLT_TRCS2** Transformation Engines.

2.2.1 User function in server side

This is a generic example of a user function in **BLT_TRCS2** Transformation Engine:

```
public static <ReturnedType> <UserFunctionName>Function(ONContext onContext,
<Argument1Type> <Argument1Name>Arg,…)
{
      // TODO: Please implement the body of your user function
}
```

As you can see, only the header is defined. The body of the method is not automatically implemented. You should program it following your requirements.

2.2.2 User function in client side

This is a generic example of a user function in **UIT_ASP2** and **UIT_CS2** Transformation Engines:

```
public static <ReturnedType> Execute<UserFunctionName>(Oid agent,
<Argument1Type> <Argument1Name>Arg,…)
{
#error Implement user function code

      return null;

      // Uncomment in order to invoke remote user function code
      // return RemoteUserFunctions.Execute<UserFunctionName>(agent,
<Argument1Name>Arg,…);
}
```

Note: In **UIT_ASP2** and **UIT_CS2**, an error instruction is always generated to warn the user to implement the method.

In client side, the use of user functions has several options:

- ✓ **Directly used in the client side**. The user function is implemented in client side. Modify the body of method to implement it.
- ✓ **It is implemented in the server side**. The user function will call to the server side to execute the method. Delete the 'error' instruction and uncomment the 'RemoteUserFunctions' call:

```
public static <ReturnedType> Execute<UserFunctionName>(Oid agent,
<Argument1Type> <Argument1Name>Arg,…)
{
   return RemoteUserFunctions Execute<UserFunctionName>(agent,
<Argument1Name>Arg,…);
}
```

✓ **It isn't used by the client side**. If this user function will never be used by the client side you can delete the 'error' instruction and always return NULL value.

```
public static <ReturnedType> Execute<UserFunctionName>(Oid agent,
<Argument1Type> <Argument1Name>Arg,…)
{
   return null;
}
```

3 Showing an information message after deleting a purchase order line

After a customer deletes a purchase order line from the shopping cart, we want to show an information message. The message will be 'The item has been removed from the shopping cart'.

To do this, you need to add an outbound argument to the 'delete_instance' event of 'POLine' class:

1- Go to the *Servie* tab of the 'POLine' class and select the 'delete_instance' event. Click on *Outbound Arg.* tab:

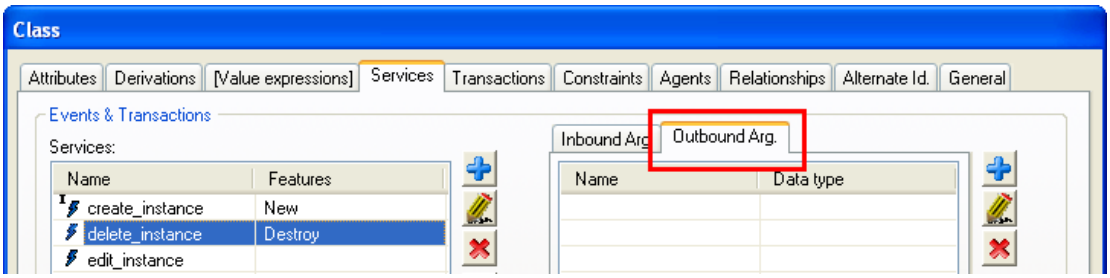

Figure 1 Outbound arguments tab

2- Add a new outbound argument (in the same way that you already added inbound arguments) with these data:

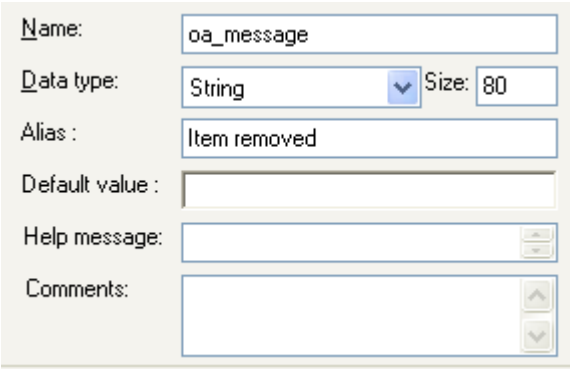

Figure 2 New outbound argument

3- Select the new argument and click on *Value expressions* button:

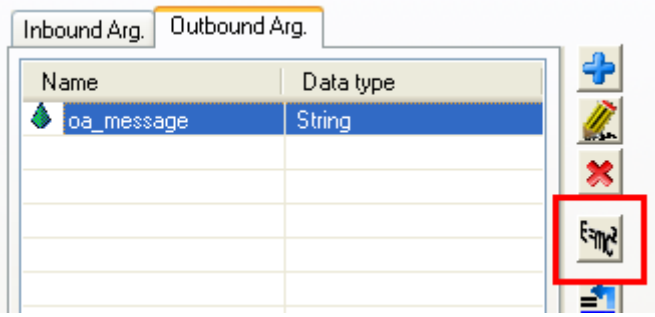

Figure 3 *Value expressions* button

4- Add the value expression of this argument. the following text should be assigned 'The item has been removed from the shopping cart'. When deleting a shopping cart line this message will be show.

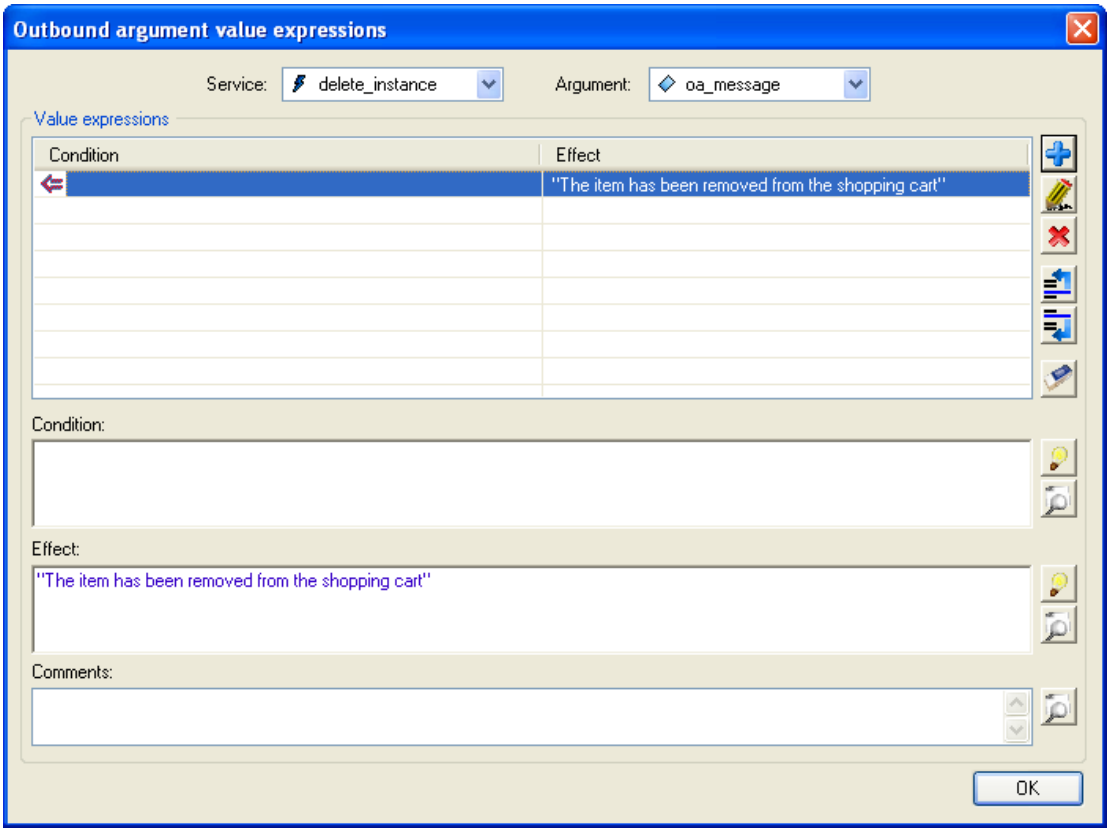

Figure 4 Outbound argument value expression

5- Now, the 'delete_instance' event of the 'POLine' class has another argument. The 'TDELETE' transaction of 'PurchaseOrder' class uses the 'delete_instance' event of the 'POLine' class, so you must modify its formula to take into account the new argument. Obviously, when a customer decides to delete a Shopping Cart, it is not necessary to inform him of each line deleted has been correctly deleted. That's why the value returned in the outbound argument is not necessary, so you can use a NULL to call it.

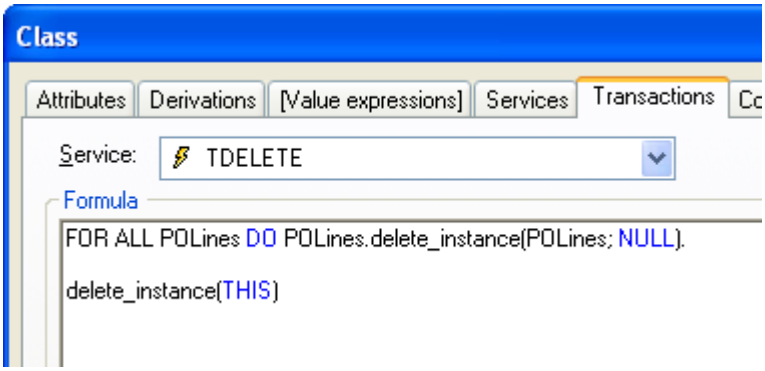

Figure 5 Modified 'TDELETE' transaction of 'PurchaseOrder' class to take into account the new argument

4 Showing an information message after paying the shopping cart

After a customer pays the purchase order, the application must show another information message. For that, add an outbound argument to the 'TPAY' transaction of 'Purchase Order' class:

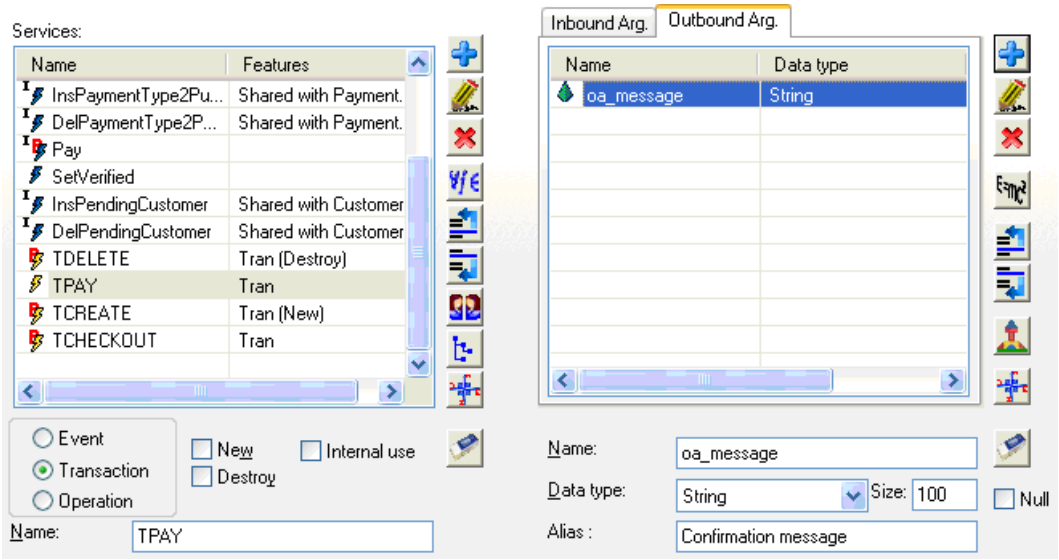

Figure 6 New outbound argument in 'TPAY' transaction of 'PurchaseOrder' class

Next, create a value expression for this outbound argument. No condition is needed, and its effect will be the desired message 'The payment has been successful. The purchase order is in process':

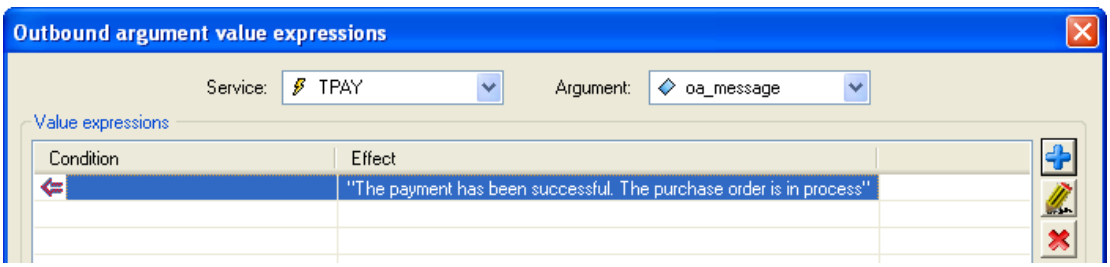

Figure 7 Outbound argument value expression

5 Showing shopping cart summary

Before paying the shopping cart, the customer should check it out. You must create a new service named 'TCHECKOUT' in charge of applying the discount (in case the discount exists), and allowing the customer to add comments in the shopping cart. After executing this service, a brief summary will be shown with the applied discount, the gross amount (without discount) and the net amount (with the applied discount).

1- In 'PurchaseOrder' class, there is the 'Amount' derived attribute which shows the final price. But, with this new requirement, instead of just one 'Amount' attribute, we will need two derived attributes: 'GrossAmount' and 'NetAmount'.

- The existing 'Amout' attribute can be renamed as 'NetAmount', since its derived formula calculates the net amount of the purchase order. For that, select the attribute, right click on it, click on *Rename* option, fulfill the new name and click on *OK* button:

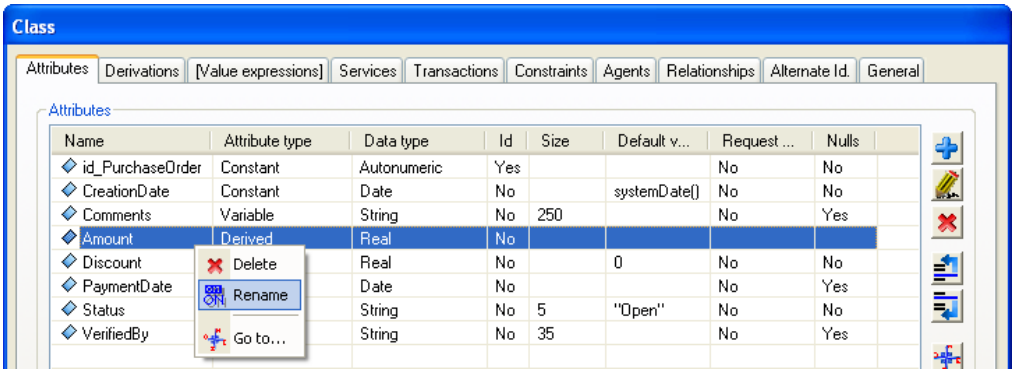

Figure 8 Attribute rename option

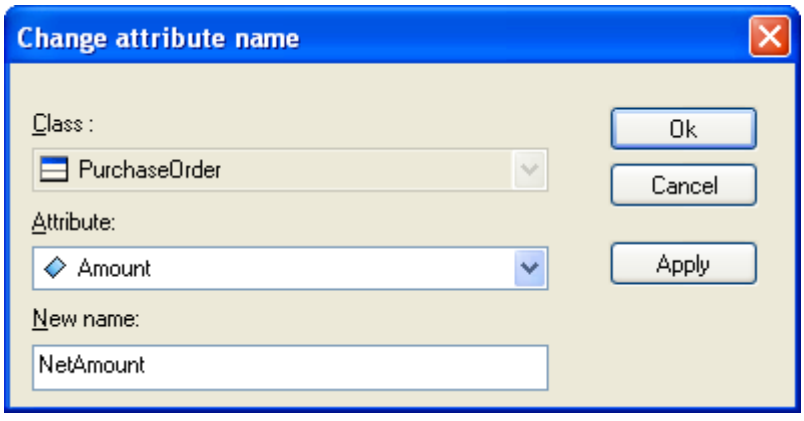

Figure 9 Filling out the new name

After that, modify its alias, the help message and comments.

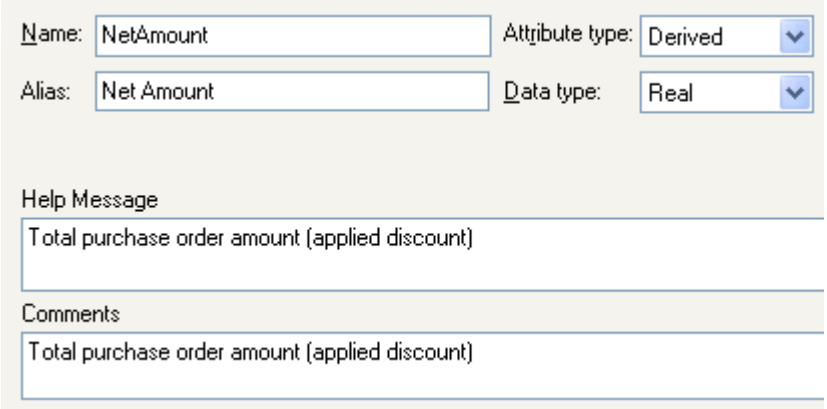

Figure 10 Changing the attribute properties

Finally, click on *Edit*  button to save the changes:

- Add a new attribute representing the amount **before** applying the discount (gross amount):

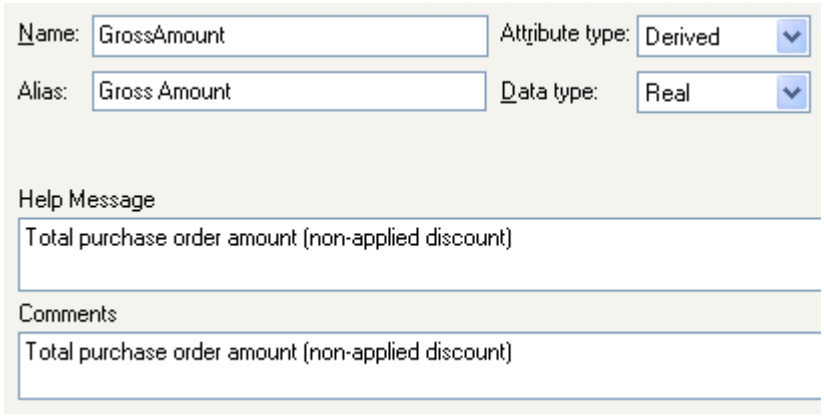

Figure 11 New "Gross Amount" derived attribute

The derived formula will be 'SUM(POLines.Amount)'. No condition is needed:

Figure 12 "Gross Amount" formula

Give the agent permissions to 'Customer' and 'Admin' classes in order to see this attribute.

2- Create a transaction called 'TCHECKOUT' with an argument to insert some comments:

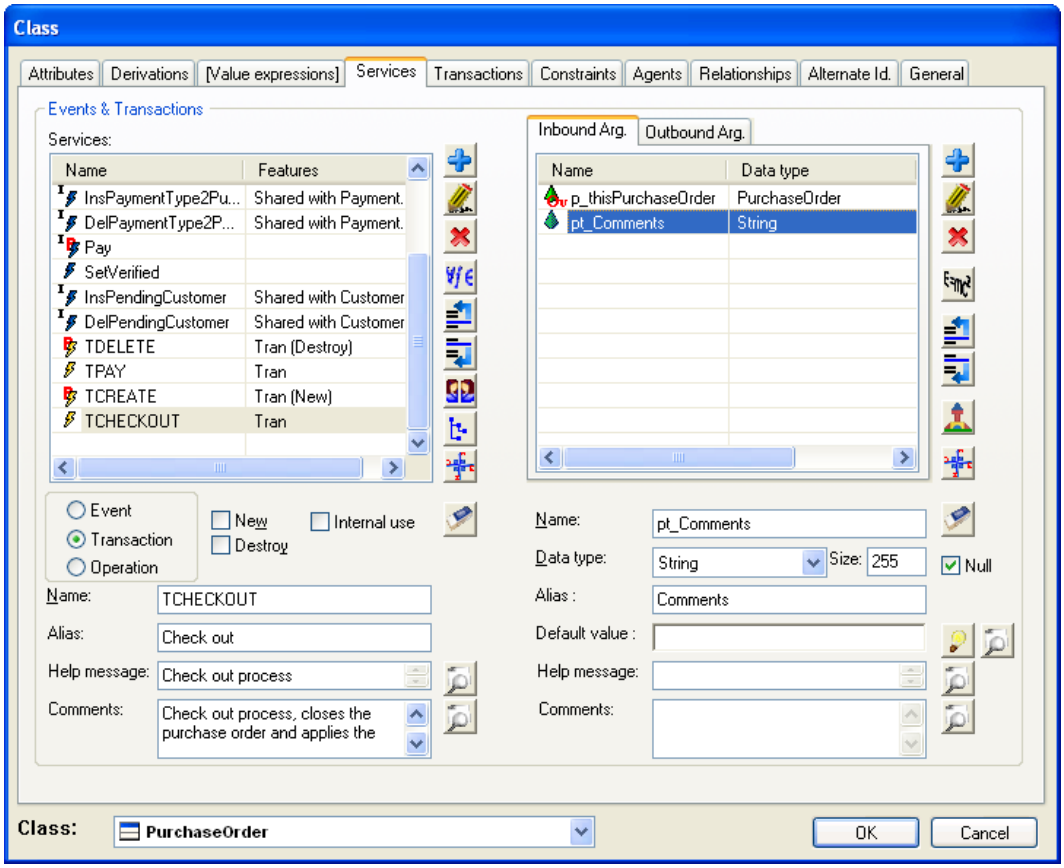

Figure 13 'TCHECKOUT' transaction

- Only *customer* will be able to execute the check out. That means you have to assign agent permissions for the 'Customer' class over the 'TCHECKOUT' transaction of 'PurchaseOrder' class.

- This transaction will apply the discount over the 'Discount' attribute and set the comments written by the customer in the 'Comments' attribute of 'PurchaseOrder' class. Its formula will be:

```
setDiscount(THIS).
edit_instance(THIS, pt_Comments)
```

Note: Mark the events used in this formula ('setDiscount' and 'edit_instance') as internal. They should not be directly executed.

- Only open purchase orders could be checked out. Then, for fulfilling this requirements, you should add a precondition to restrict other options:

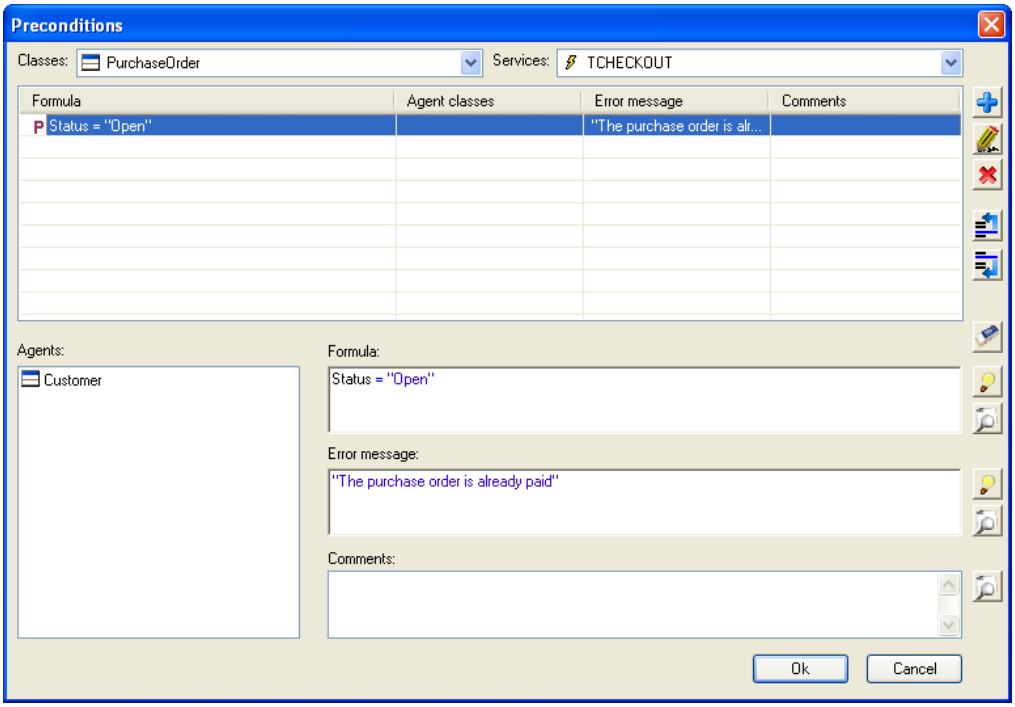

Figure 14 Precondition in 'TCHECKOUT' transaction

3- Finally, after the transaction functionality is defined, you need to add three real type outbound arguments (representing the discount, gross amount and net amount):

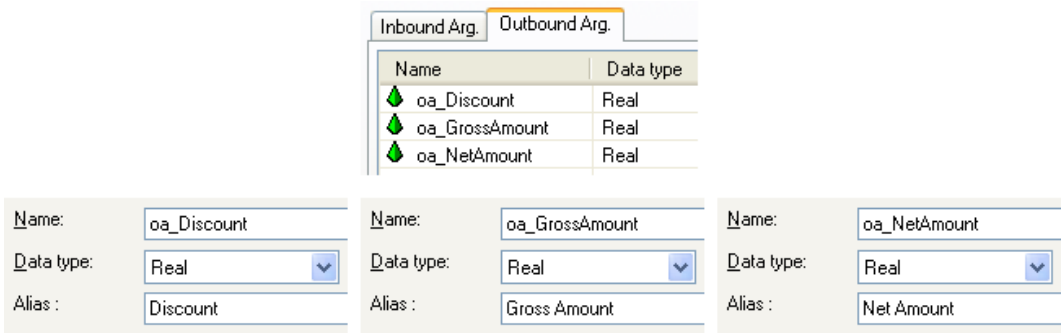

Figure 15 Outbound arguments

Add a value expression for each argument to retrieve the corresponding attribute value. No condition is needed:

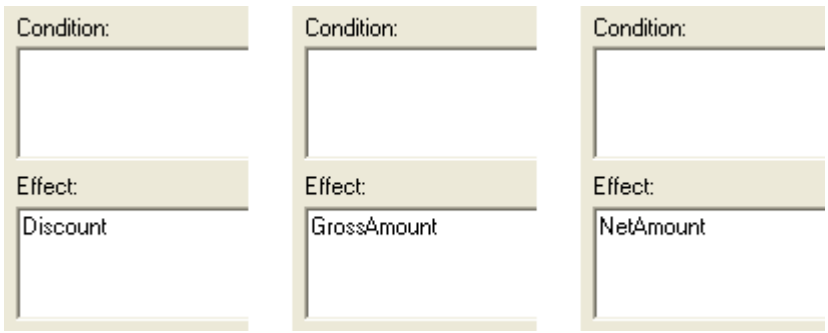

Figure 16 Value expressions of *discount*, *gross amount* and *net amount* respectively

6 Sending an e-mail

When a customer pays a purchase order, the application will automatically send an e-mail to the customer's e-mail address. This process cannot be directly defined in the model, for this reason you need a user function to manage it. After generating the code, this function must be implemented in the code.

6.1 Model Steps

1- The first step is to create a user function. Go to the *Edit* option from the *Main menu.* Select the *User Functions …* option:

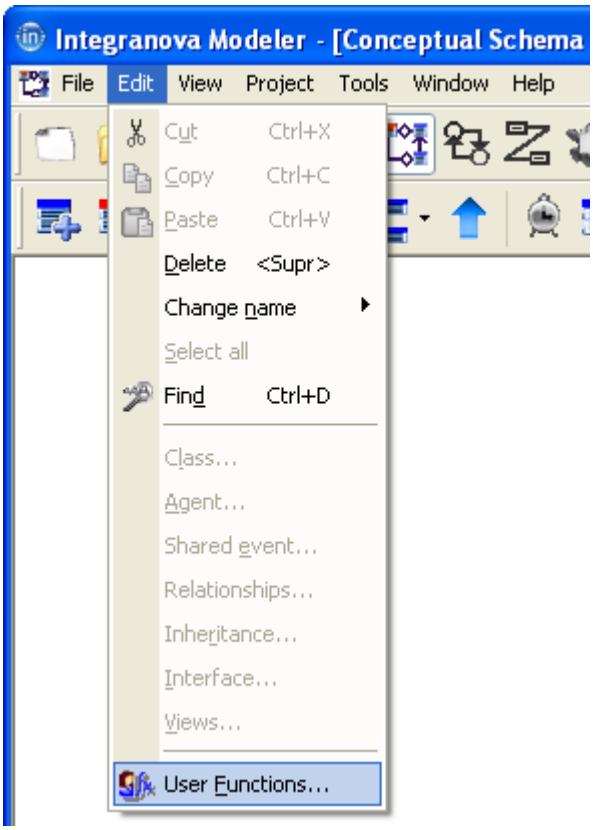

Figure 17 Access to User Functions

The User Function window will be shown (Figure 18). The list of user functions defined is displayed on the left list of the window. As you can see, there is a user function already defined. Its name will be 'uf_sendEmail' and its return type will be string. Also add an inbound argument to manage the email address (name 'pt_CustomerAddress', data type *String*):

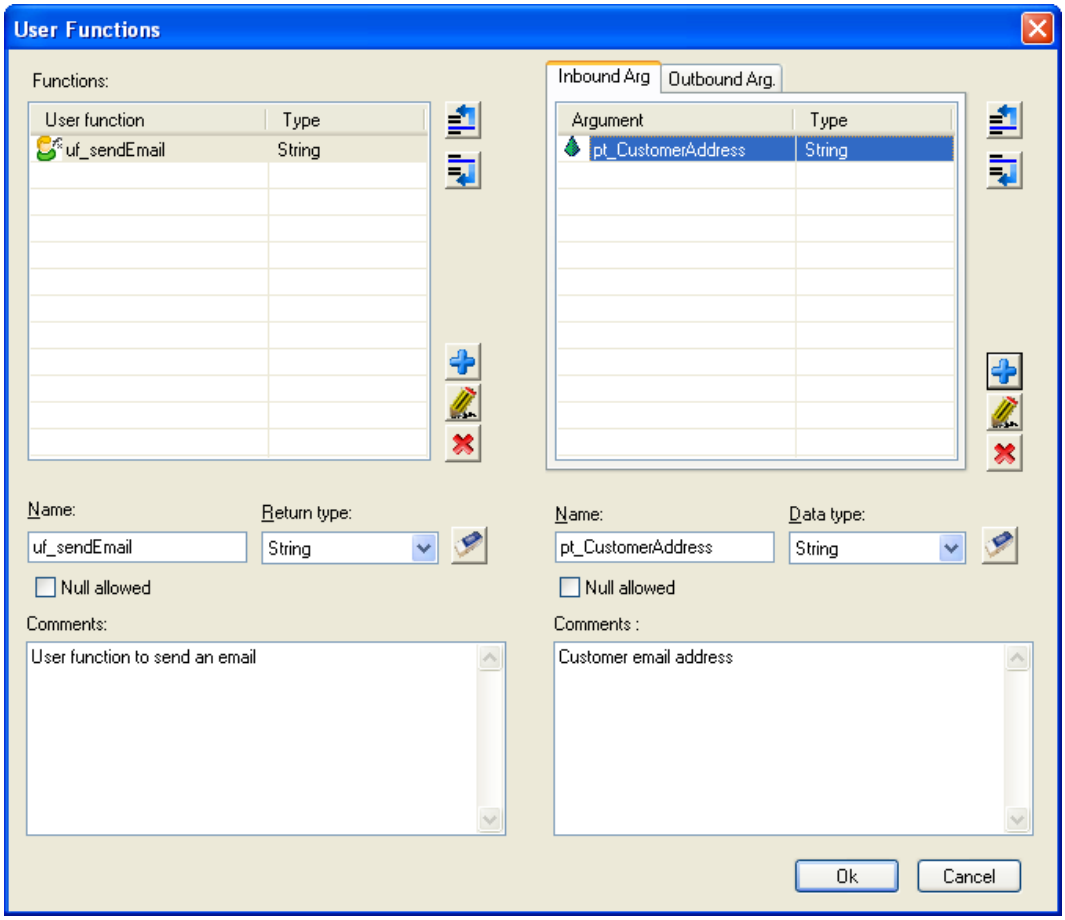

Figure 18 'uf_sendEmail' user function

2- Create an internal event in charge of calling this user function (i.e. 'sendEmail') in 'Customer' class. Add an outbound argument called 'oa_SendEmail'.

There are several ways in which user functions can be used. In this tutorial, it will be called from an outbound argument value expression, but it could be used in any formula of the model (preconditions, default values, valuations, arguments initializations…).

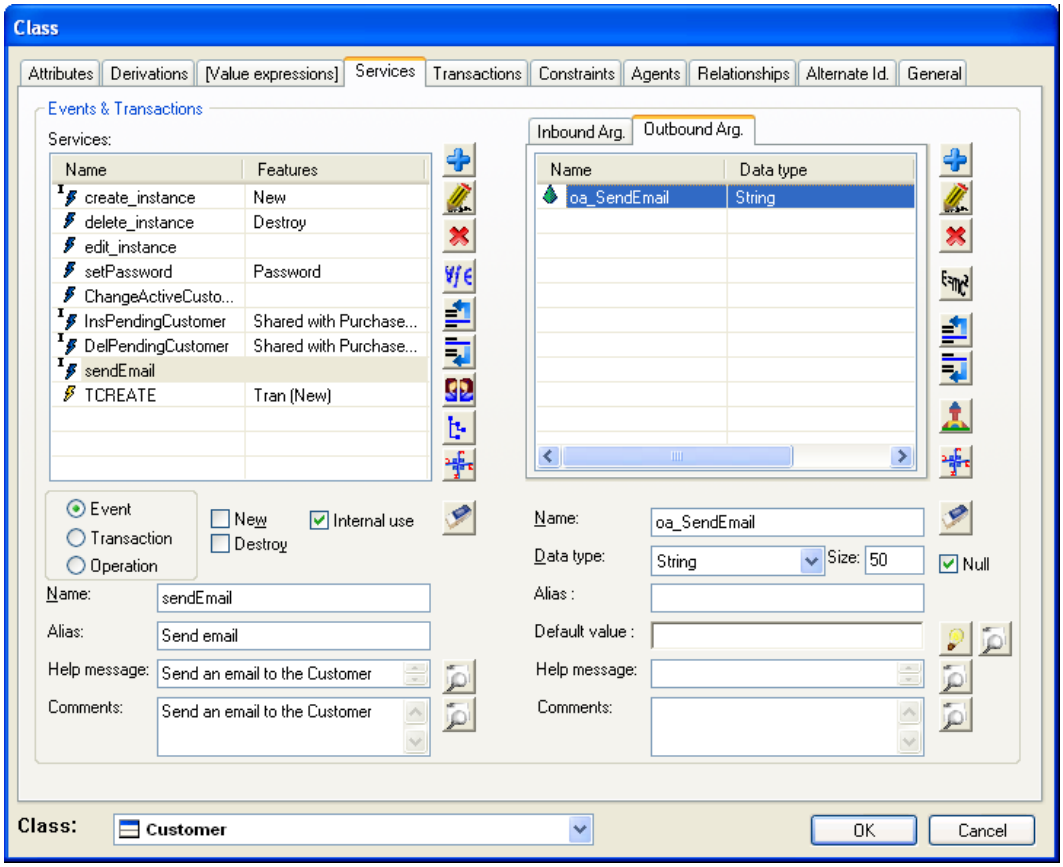

Figure 19 Event in charge of sending emails

3- The user function will be used in a value expression of this outbound argument:

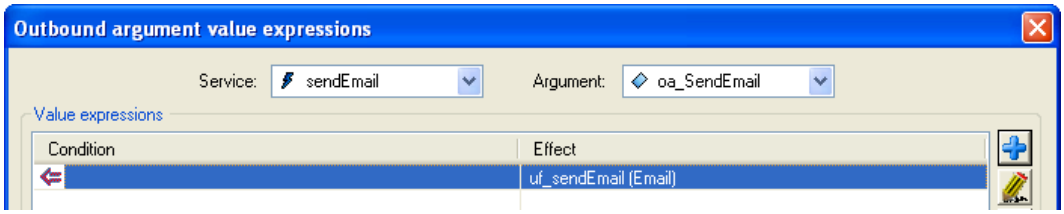

Figure 20 User function in value expression

4- Modify the 'TPAY' transaction formula to call this event:

```
IF( EXIST( PaymentType ) = TRUE) {
   DelPaymentType2PurchaseOrder(THIS, PaymentType)
}.

InsPaymentType2PurchaseOrder(THIS, pt_PaymentType).

DelPendingCustomer(THIS, PendingCustomer).

Pay(THIS).

Customer.sendEmail (Customer; NULL)
```

The 'sendEmail' event is been called with a NULL value as outbound argument. This means that the returned event outbound argument value will be ruled out.

6.2 Code Changes

After generating the code, the user function must be implemented. In this example, it will be implemented in the server side. Then the client and the server applications will be modified as following:

6.2.1 Client side (UIT_CS2 & UIT_ASP2)

This client side user function will not be used because the outbound argument value expressions are executed in the server side. But it has to be modified to avoid the default error instruction. You have two options:

- ✓ Return always a NULL value.
- ✓ Modify it to call the server side user function.

We are going to choose the second option. At this moment both options are correct, but in the future your requirements could change and you could need to send emails from the client side.

The generated method in the 'UserFunctionsLogic.cs' file is the following:

```
public static string Executeuf_sendEmail(Oid agent, string
pt_CustomerAddressArg)
{
#error Implement user function code

       return null;

       // Uncomment in order to invoke remote user function code
       // return RemoteUserFunctions.Executeuf_sendEmail(agent,
pt_CustomerAddressArg);
}
```

You should delete the 'error' line and the 'return null' and uncomment the 'RemoteUserFunctions' call:

```
public static string Executeuf_sendEmail(Oid agent, string
pt_CustomerAddressArg)
{
       return RemoteUserFunctions.Executeuf_sendEmail(agent,
pt_CustomerAddressArg);
}
```

6.2.2 Server side (BLT_TRCS2)

Here is where the user function will be implemented. It is located in the 'Others\ONUserFunctions.cs' file:

```
public static ONString Uf_sendEmailFunction(ONContext onContext, ONString
pt_CustomerAddressArg)
{
       // TODO: Please implement the body of your user function
}
```

You must adapt it to allow the email sending. This is a basic example to send an email:

```
public static ONString Uf_sendEmailFunction(ONContext onContext, ONString
pt_CustomerAddressArg)
{
       try
       {
              System.Net.Mail.MailMessage lmail = new
System.Net.Mail.MailMessage();
```

```
            lmail.IsBodyHtml = false;
            lmail.Priority = System.Net.Mail.MailPriority.Normal;

            lmail.From = new
System.Net.Mail.MailAddress("<SenderEmailAccount>"); // Email sent from...
            lmail.To.Add(pt_CustomerAddressArg.TypedValue); // Email sent
to...
            lmail.Subject = "Payment process finished";
            lmail.Body = "Payment process finished. Thank you for using our
online sale system. Your purchase order is in process.";

            System.Net.Mail.SmtpClient smtpMail = new
System.Net.Mail.SmtpClient("<SMTPServer>"); // SMTP Server
            smtpMail.Send(lmail);

            return new ONString("OK");
      }
      catch (Exception ex)
      {
            return new ONString("ERROR: " + ex.Message);
      }
}
```

Take into account that the "From" and "SmtpClient" depend on your system.

But if for any reason you cannot send emails, modify this user function to return a fixed value as for example:

```
public static ONString Uf_sendEmailFunction(ONContext onContext, ONString
pt_CustomerAddressArg)
{
      return new ONString("OK");
}
```

7 Default user interface

Integranova Modeler offers the possibility of creating a customized user interface using the features associated to the *Presentation Model*. All these features will be presented from Tutorial 9 in the series tutorials to the last one (Tutorial 15).

Meanwhile, **Integranova Modeler** provides a default user interface where all the functionality defined in the model can be accessed. It will be possible to access to the application using any of the agents defined in the model.

The main menu will have as many menu entries as classes defined in the model (the name will be the alias of the class). Each menu entry will have the following subentries: one scenario to show **one instance** of the class, another scenario to **list all the instances of the class** and as many scenarios as **non-internal services** we have defined in the class.

If there are global services defined in the model, a new menu entry labeled as "Global Services" is added to the main menu, where all the **global services** defined are available.

When an agent is connected to the application, he only will have access to the services that he can execute and the roles and attributes over which he has visibility. The menu entries which the agent has no permission (no execution, no visibility, no navigation) will be hidden.