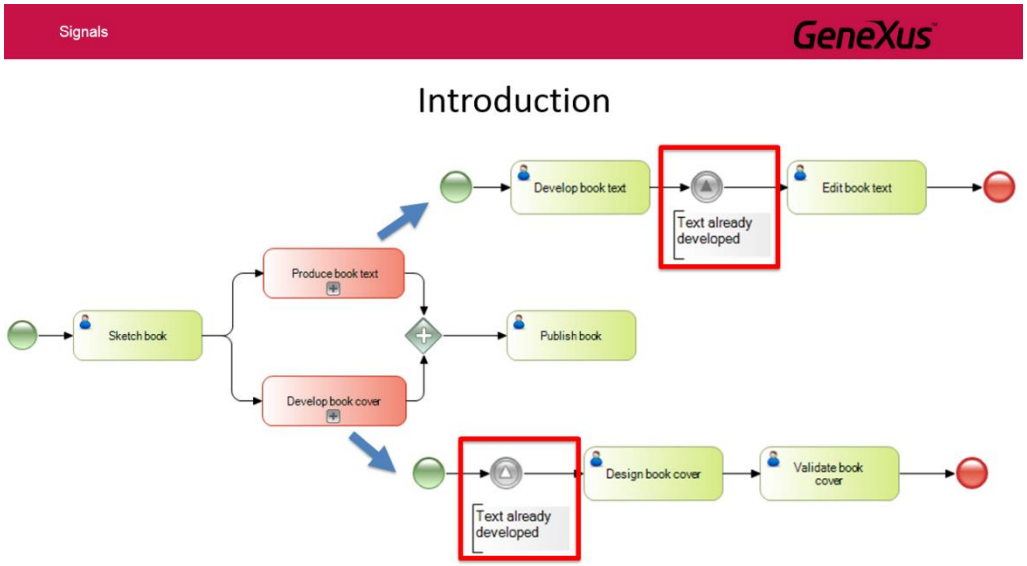
Signal Type Events

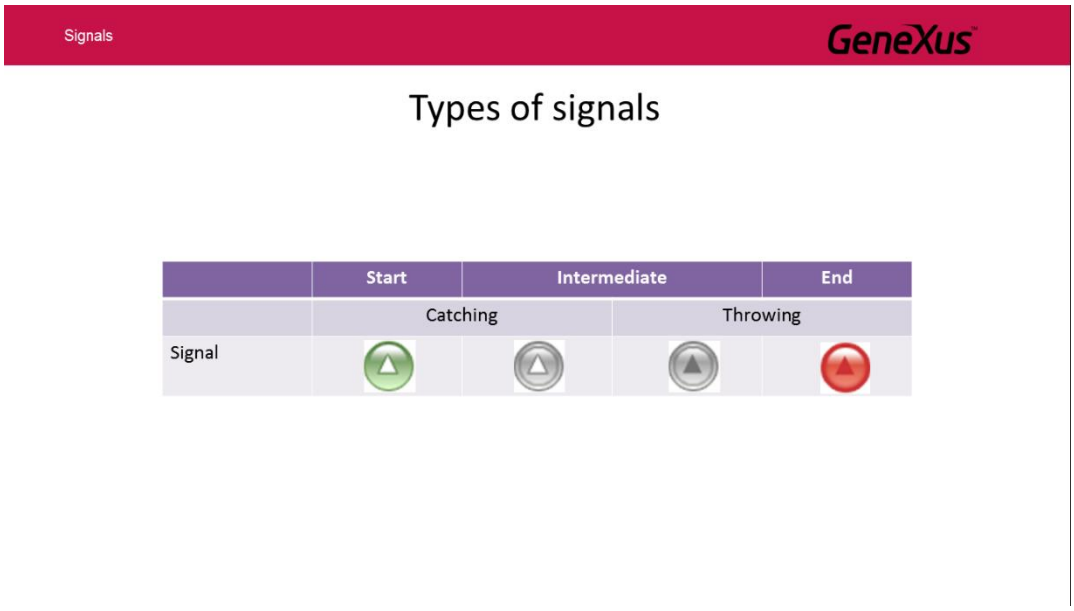
In this video we will see some ways to use the Signal type event.

Signal events are used to send or receive signals within or outside the process. For this reason, they are useful for both communication between parts of the same process and between processes linked by a hierarchical relationship.



This implies that it is possible to signal the occurrence of an event that can be detected in any part of a process, sub process or even parent process.

The signal behavior varies depending on whether they are start events, intermediate events or end events. In turn, these events can trigger a signal (throwing) or receive a signal (catching).



The notation used changes accordingly. For example, intermediate events have a double border, while start and end events have a simple border. When they are triggering events, the triangle is dark; when a signal is captured, the triangle is clear.

A start signal event allows a process to start when a signal is received from another part of the process or processes hierarchically related to the process where it is defined. For this reason, they are always of catch

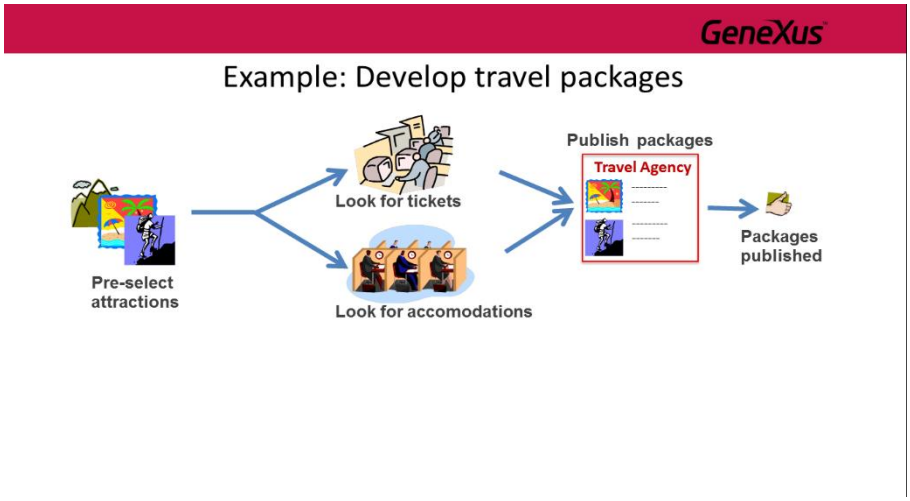
type.

When a process flow ends, a signal event of end type allows sending a signal to another part of the process or processes hierarchically related. They are always of throw type.

A signal event of intermediate type can be of throw or catch type, depending on the value of the “Is throw” property (True or False, respectively), and it can be placed in any part of the process, usually between activities.

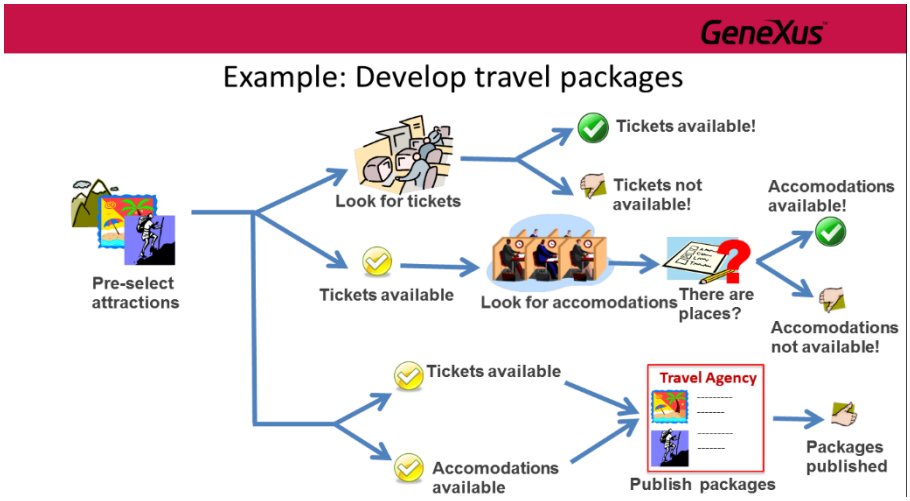
Let's see some of these symbols in action.

The travel agency has requested us to model the process used to publish travel packages for their clients. Each package includes several ticket and hotel reservations, and one package can be published only if there are tickets and hotels available.



This implies that the process must allow securing a certain number of tickets and rooms, so as to be able to promote packages of a certain tourist attraction.

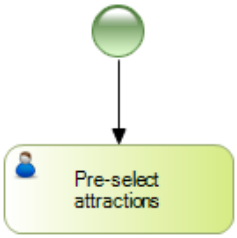
In addition, the process must be efficient. That is to say, the enquiries to airlines and hotels must start at the same time, and if the minimum number of tickets is not obtained, the search for hotel rooms must be canceled.



The process must guarantee that only packages are published, when both tickets and accommodation are booked.

To implement this process in GeneXus, we open the GeneXus Modeler, create an object of Business Process Diagram type and call it **DevelopTravelPackages**.

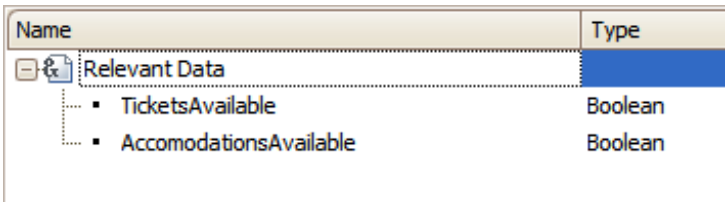
First, we drag a None Start Event symbol. Next, we insert an interactive task, call it "**Pre-select attractions**" and connect it from the None Start Event.



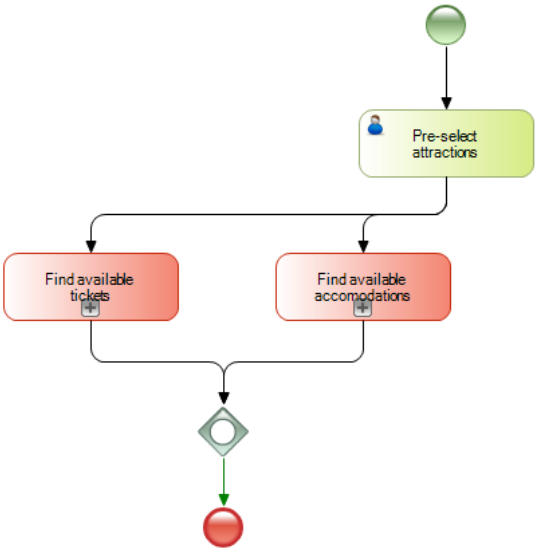
This task encompasses the activity of examining which attractions are more suitable to be included in a package.
Next, we must start one process to search for tickets and another process to search for hotels.

To do so, we drag the symbol of an embedded sub process, call it **Find available tickets** and join it from the task defined before. We add another embedded sub process called **Find available accommodation** and also join it from the task Pre-select attractions.

In order to complete the process of developing travel packages, we must make sure that both sub processes have been successfully completed. This may be done by defining a relevant data item of boolean type set in each process, depending on whether this process has been successfully completed.

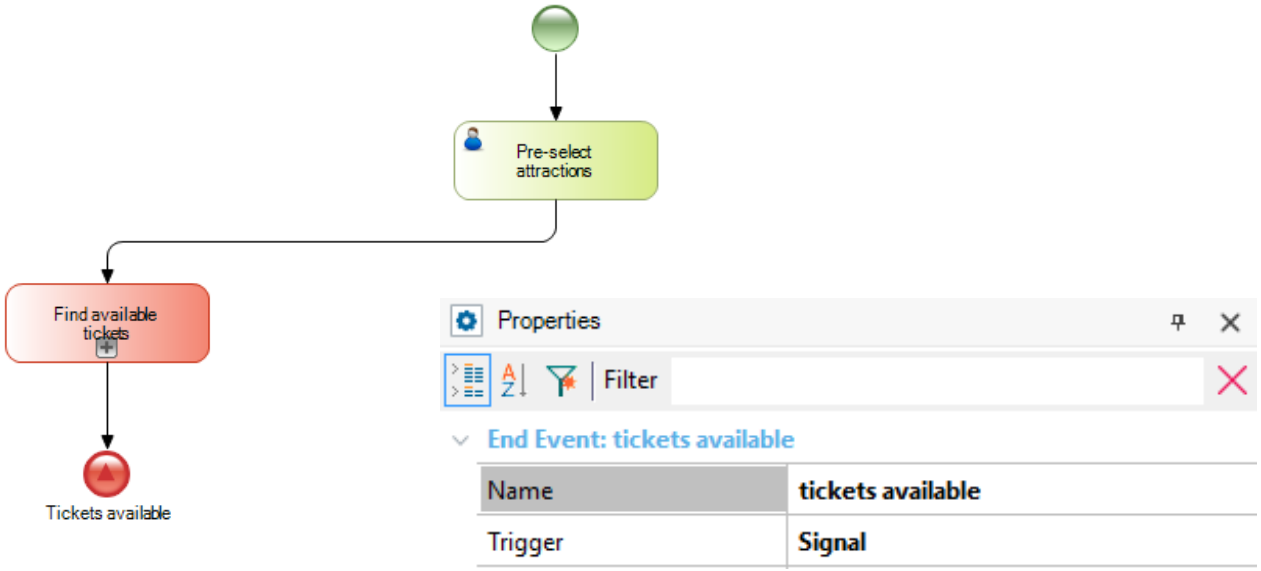


Next, we should use an Inclusive Gateway to synchronize the paths coming from each sub process, so that the package development process only ends if both sub processes have been successfully completed.

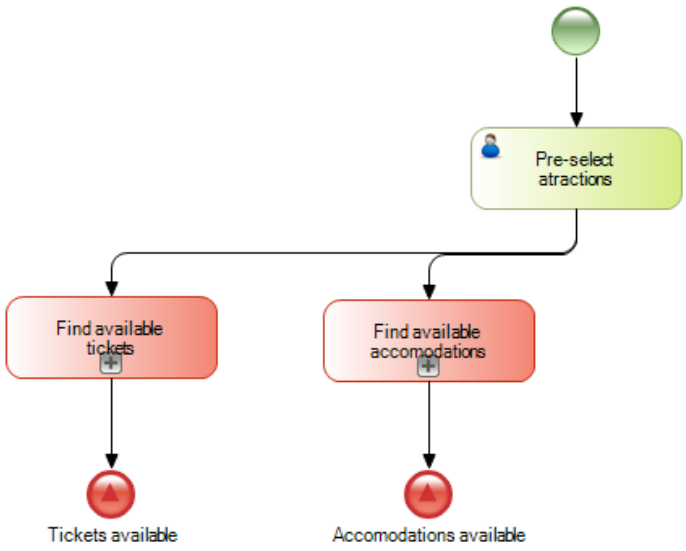


Even though this solution is possible, it implies defining and setting relevant data. We can obtain a similar result by using Signals, with no need for this relevant data.

Instead of the Inclusive Gateway, we insert an End Event of Signal type and connect it to the output of the sub process that looks for available tickets. We call the signal **Tickets available** and see that an **end event of signal type** is already of throw type because the triangle is dark. In a signal end event we can't change the value of the **Is Throw** property because it isn't available and its value is always True.

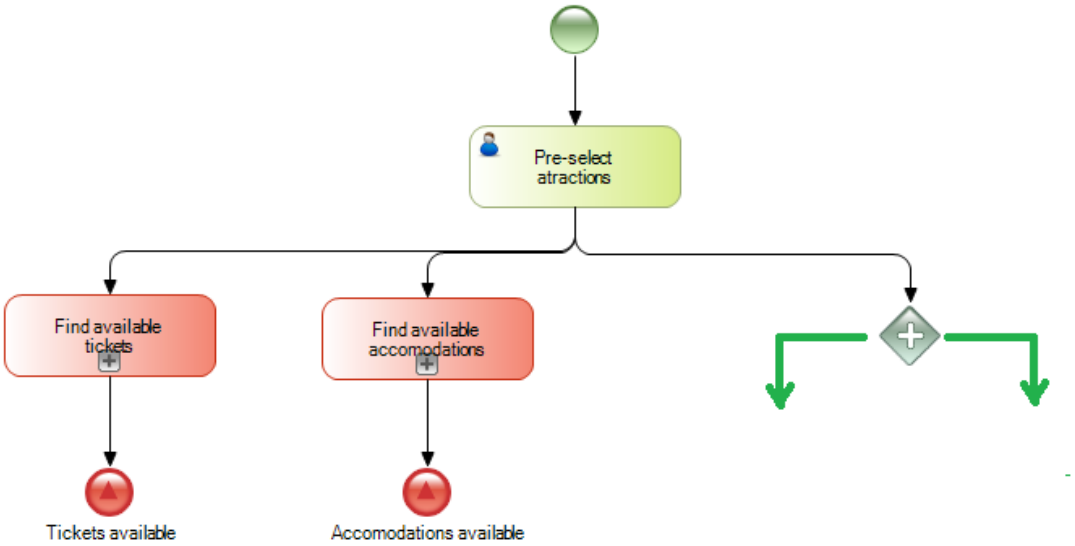


In the same way, we insert a Signal End Event to the output of the sub process that looks for hotels and call it **Accommodation available**.



Now we must catch these signals to continue with the process to develop travel packages, that is to say, to publish them and finally end the process.

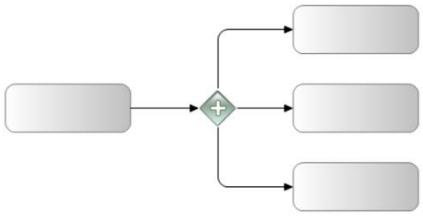
To do so, we insert a Parallel Gateway to which we connect from the Pre-select attractions task. This will allow us to fork the flow in two paths.



Unlike the Exclusive Gateway that is used when the path to be followed by the process flow depends on the evaluation of a true / false condition, a Parallel Gateway allows us to split the flow in two or more paths, without evaluating any conditions.

GeneXus

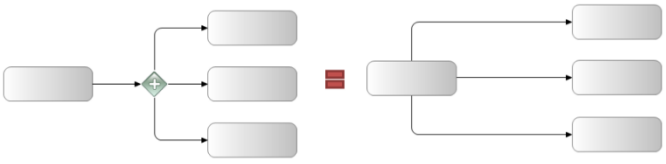
Paralell Gateway



For this reason, it is also possible to model this same behavior without using a **Parallel** gateway, simply by joining two connectors. However, the use of a **Parallel** gateway can help clarify the diagram in certain circumstances.

GeneXus

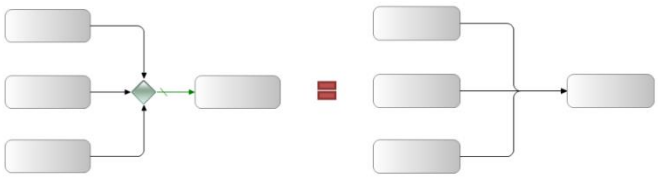
Paralell Gateway



Joining several paths to follow only one of them is called **synchronization**. In this context, **Exclusive** gateways can also be used for synchronization, even though they are rarely necessary for modeling because it can generally be modeled without them, obtaining the same behavior.

GeneXus

Exclusive Gateway

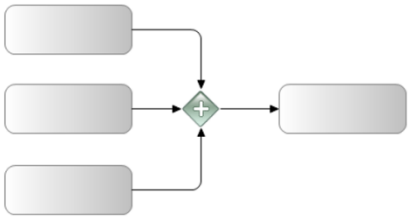


This modeling method is rather risky because since there is no synchronization, the task after joining the paths could be executed several times, once for each path that was joined, as the flow sequences of each path arrive.

To avoid this, we must use a **Parallel** Gateway to join paths. A Parallel Gateway waits for all the incoming flow sequences and doesn't continue until all of them have arrived.

GeneXus

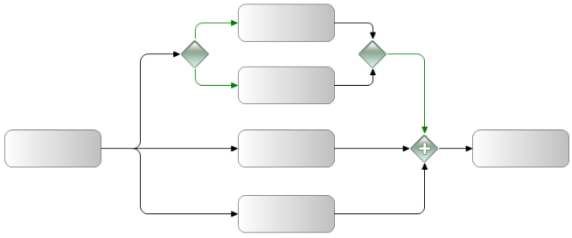
Paralell Gateway



However, in some cases an **Exclusive** gateway is required for synchronization.

GeneXus

Exclusive Gateway

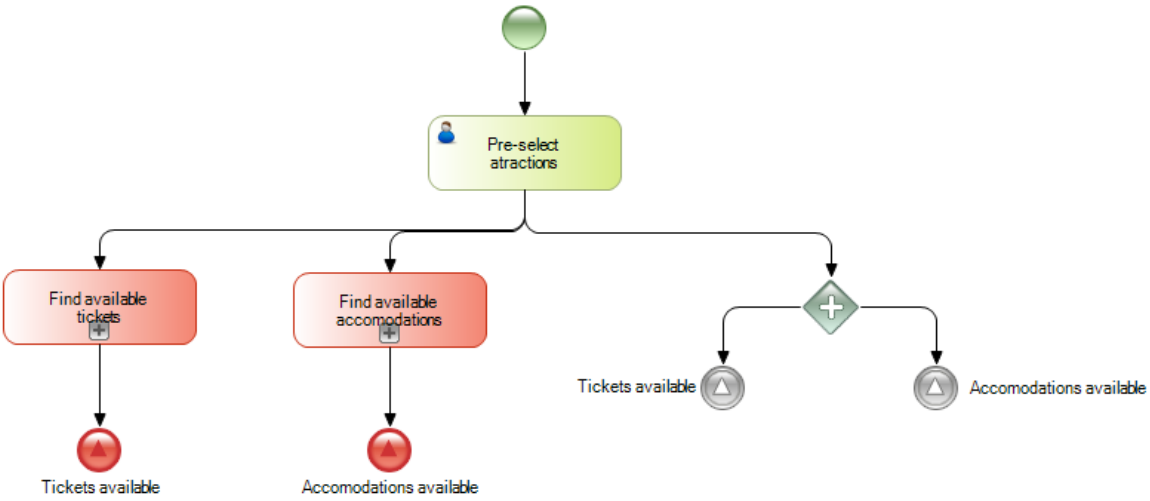


In this example, if the **Exclusive** gateway wasn't used to synchronize the result of a previous gateway, the **Parallel** gateway would have four input sequence connectors. However, only three of the four flow sequences could pass at a time (because of the Exclusive Gateway at the forking). Therefore, the process would get stuck at that **Parallel** gateway.

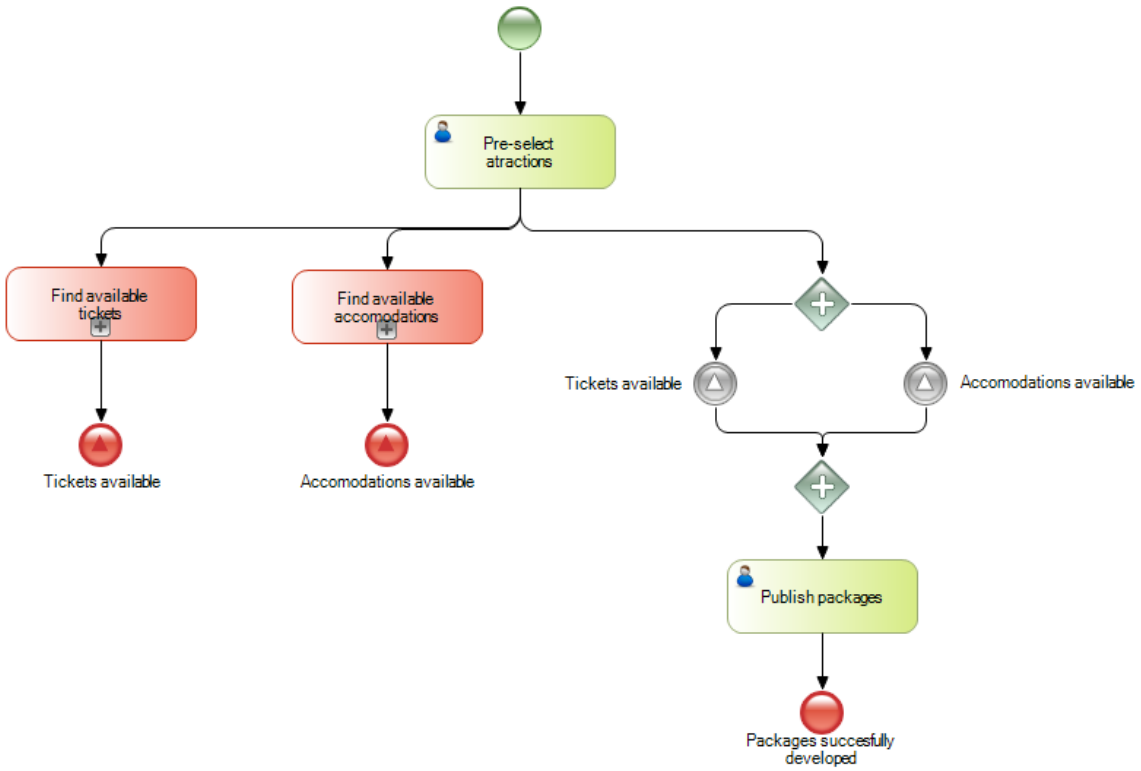
Going back to our model, we add an intermediate event of signal type for each path and connect them from

the parallel Gateway.

We call the signal on the left path **Tickets available** and leave the **Is throw** property set to False. Note that this signal is of catch type. It will catch the throw signal with the same name that is triggered when the ticket search process ends.
We do the same with the other signal. We call it **Accommodation available** and set its **Is throw** property to False. In turn, this one will catch the signal sent when the process to search for accommodation ends.



Next, we insert another parallel Gateway to join both paths. This will prevent the process from continuing unless both signals are received.
Lastly, we insert a task called **Publish packages**, connect it from the parallel Gateway and add a none end event to end the process, to which we add the description **Packages successfully developed**.



In this way, the implementation meets the requirement of not completing a process unless the necessary tickets and accommodation are obtained.

However, the process is not efficient. Remember that the Travel agency has requested that the process to search for accommodation must not start if the tickets haven't been previously obtained. To do this we also

use signals.

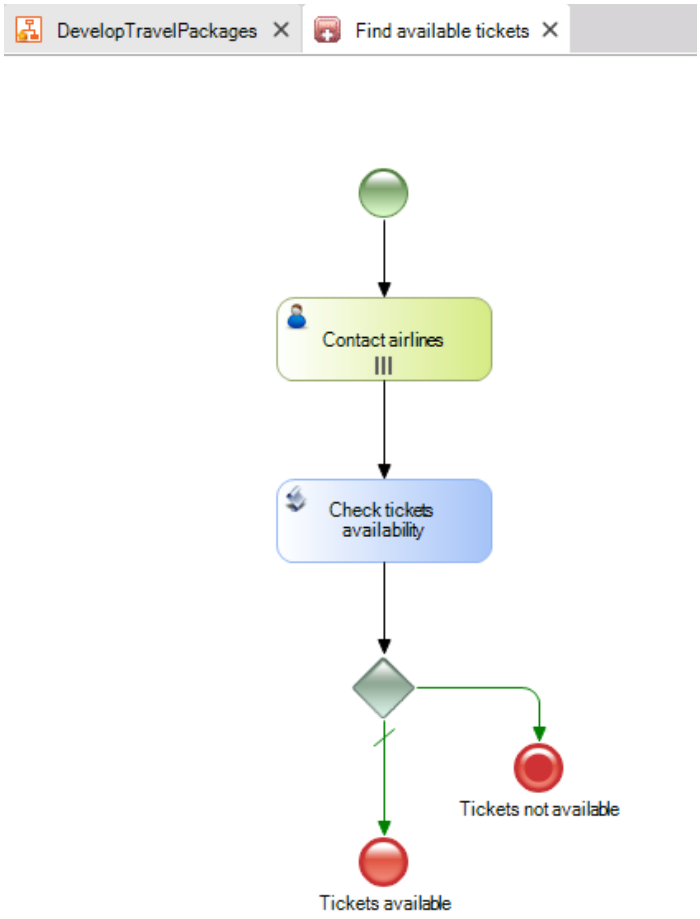
We double-click on the process **Find available tickets** to define it. A blank window is opened to add the symbols representing the process to obtain tickets.

First, we insert a None Start Event and a task called **Contact Airlines**. Since this task will be run several times, one for each airline contacted, we open its Loop type property and set it to Multi-Instance.

To examine the information obtained, once all the airlines have been contacted, we insert a script task called **Check availability** and connect it from the task **Contact Airlines**.

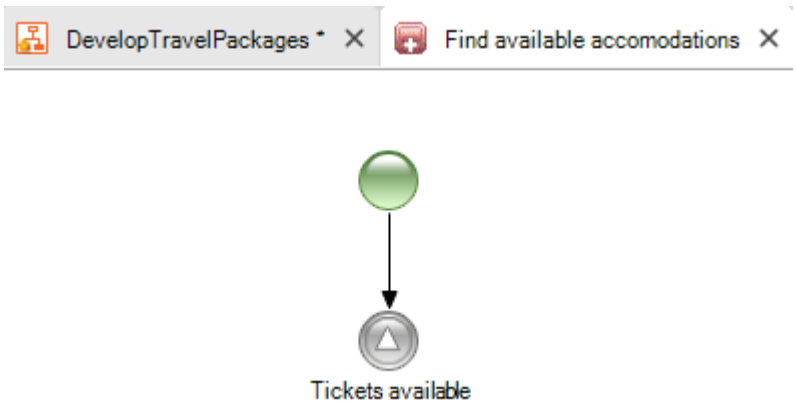
Now we drag an exclusive Gateway.

If there are tickets available, we end the process with an end event (the default path). If there aren't enough tickets available to create a travel package, we end the process with a terminate end event that not only ends the sub process, but also the main package design process.

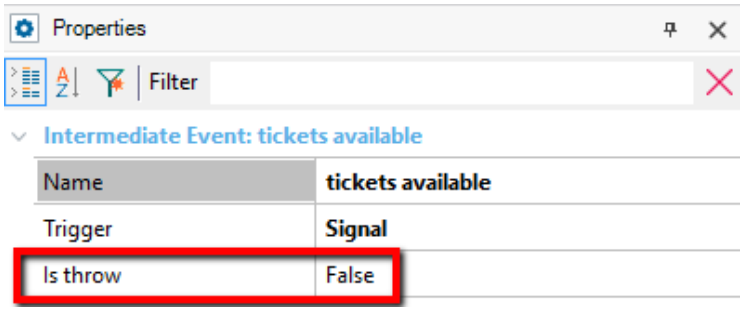


Next, we will define the accommodation search process to check if we can offer a package for the pre-selected attraction.

We double-click on the sub process **Find available Accommodation**. We drag a None Start Event; next, we insert the symbol of an intermediate event of signal type.



We call it **Tickets available**, like the Signal End Event at the end of the ticket search process for the package. We open its properties and check that the IsThrow property is set to False, because we want this event to catch the signal sent when the ticket search process is completed.

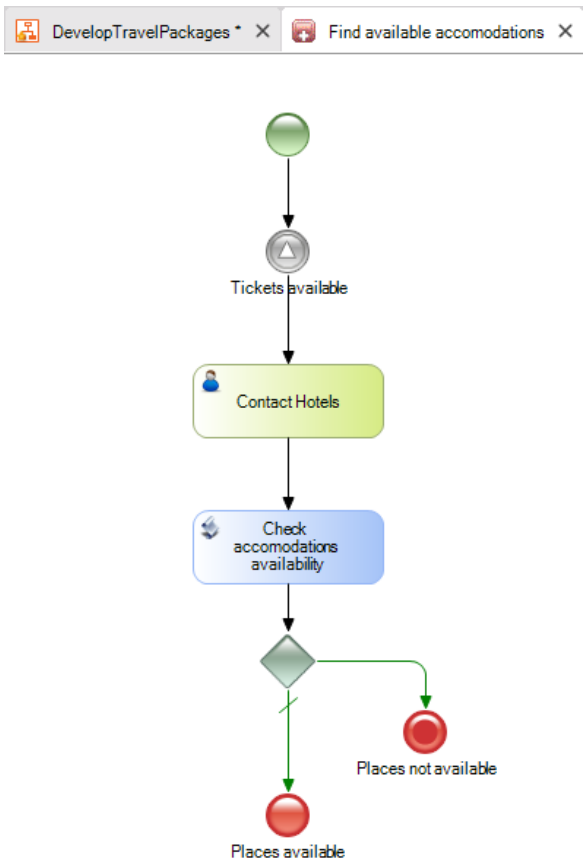


In this way, the process to search for accommodation will continue only if there are tickets available for the package.

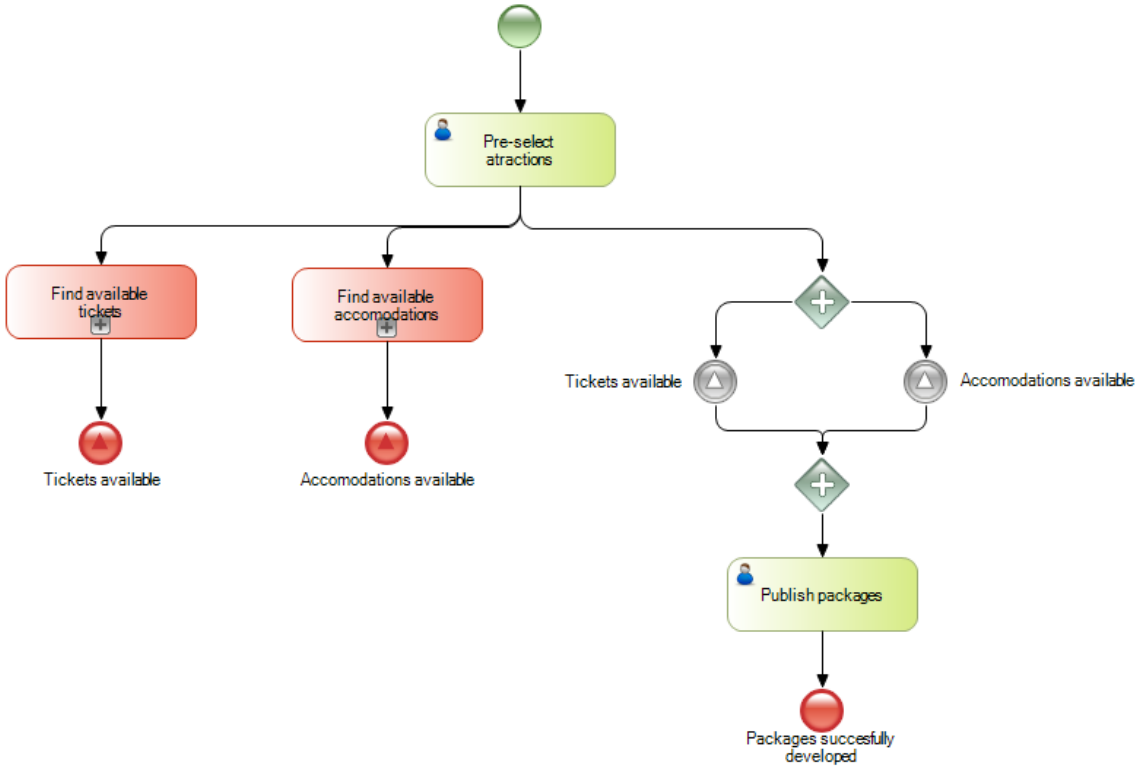
To continue defining the process, we insert an interactive task called Contact Hotels. We set its Loop type property to Multi-instance because this task should be run several times, once for each hotel contacted.

Next, just like in the ticket search process, we insert a batch task called **Check Accommodation availability** that will check if there are enough hotel rooms available to offer the package we're designing.

Lastly, we insert an exclusive Gateway and if there are hotel rooms available (the expected result), we end the process with a None End Event with the tag “Places available”. Otherwise, we insert a Terminate end event with the tag “No places available”.



We close the sub process window, return to the main process and save.



In this way we have defined a process that meets the objective of efficiently designing a travel package, just

like the Travel agency had requested.

The signals have provided us with a communication method between the various parts of the main process and between the main process and its sub processes, to achieve the objectives we had set out.

To learn more about this topic, click on the link displayed.

http://wiki.genexus.com/commwiki/servlet/hwikibypageid?24913