

Thoughts about a new UI for the Eclipse BPEL Designer

Author: Vincent Zurczak – EBM WebSourcing

Version: 1.0

Status: draft

Date: 10/02/2011

Table of Content

| | | |
|-----|---|----|
| 1 | Context..... | 3 |
| 1.1 | BPEL modeling?..... | 3 |
| 1.2 | Few words about BPEL..... | 3 |
| 1.3 | Who writes BPEL processes?..... | 3 |
| 1.4 | How about the BPEL Designer?..... | 4 |
| 2 | Alternative UI..... | 5 |
| 2.1 | Partner links..... | 5 |
| 2.2 | Message reception and invocations..... | 8 |
| 2.3 | Conditions, loops and parallel flows..... | 12 |
| 2.4 | Variables, assignations and validation..... | 16 |
| 2.5 | Notations and icons..... | 17 |
| 2.6 | Conclusion..... | 20 |
| 3 | Others extensions..... | 21 |
| 3.1 | Analysis and verification..... | 21 |
| 3.2 | Graphical mapping..... | 21 |
| 3.3 | An extended XML source..... | 23 |
| 3.4 | Graphical debug..... | 23 |

1 Context

1.1 BPEL modeling?

There are two ways to consider BPEL modeling.

An indirect approach consists in going through an intermediate model that will be transformed in BPEL. Such intermediate models could be BPMN (see Intalio's work), UML or even SoaML. We could also think about a new process modeling or a scripting language that would ease the generation of BPEL. So, basically, it is based on code generation.

The second approach is a direct one: the model behind the editor is the BPEL model. There is no required transformation or export. This approach is the one used in the BPEL Designer.

In this document, we are only interested in the direct approach.

The objective is to describe another way of representing BPEL processes graphically, with a focus on ergonomics and ease of use. Any approach should aim at hiding the complexity of BPEL and only show what is important.

If indirect approaches are not discussed here, they should however not be seen as opposites to direct approaches. It depends on the target user and its background experience.

1.2 Few words about BPEL

Even if BPEL stands for Business Process Modeling Language, the truth is that BPEL is a programming language, written with a XML syntax.

Some people think it is more than a programming language because of the compensation mechanism. But in fact, compensation aims at replacing transaction managers in SOA. Indeed, in SOA, low-coupling is a fundamental key and transactions induce tight-coupling between services. Compensation is somehow a return in history, just like if transactions did not exist. It is not a progress, it is a necessity that comes from the low-coupling requirement.

So BPEL can be seen as any other programming language, like Java or C.

What makes it particular is the XML syntax and the fact that all the conditions are written with Xpath and that variables are defined by XML schemas.

Representing graphically a BPEL process is quite the same thing than representing a Java program. This is true for all the direct approaches in BPEL modeling.

1.3 Who writes BPEL processes?

BPEL being a programming language, it would seem obvious that only developers should write BPEL processes.

The BPM methodology may have let think that people with less technical skills could write, directly or indirectly, BPEL processes. Articles that sold BPMN to BPEL transformations as natural and simple may be responsible for that.

In the rest of the document, we will only focus on developers.

What is interesting to know, though, is whether this developer needs to have a specific knowledge about BPEL or not. Said differently, did he read the BPEL specification or not?

Thus, we will consider advanced BPEL developers (that know the BPEL syntax and can hover the source directly) and other developers (with only a programming background).

1.4 How about the BPEL Designer?

Until there, we could think the current BPEL Designer fills in all the criteria to satisfy our needs:

- ✓ The BPEL model is edited directly and graphically.
- ✓ This editor is intended for developers (mostly with a BPEL experience or training).
- ✓ It offers some nice features, through wizards and properties.

However, there are some little things that make this editor not very intuitive.

- ✗ You have to read a lot of labels (mark-up names) to understand what a BPEL process does. Besides, all the BPEL mark-ups are shown. Even sequences. It is hard to determine whether a process diagram shows the execution flows or a syntactic tree of the language.
- ✗ The dashboard palette, that shows variables and partner links, can be mistaken with the real palette. It is confusing for new users.
- ✗ Imbricated boxes represent blocks of code. Depending on the process complexity (e.g. 5 imbrications), it can quickly become hard to read or navigate in the diagram.
- ✗ Invocations to external partner links are not explicit at all (unlike in Netbeans, where exchanges with partner links appear on the diagram).

There are some people that are used to this representation.

But from the feedback I could get from clients and from training session participants, it is not easy to master it, and even when it is done, opening existing processes that were created by someone else is not simple. Watching the diagram or reading the source code would be almost the same.

What I suggest in the rest of the document is not removing the current representation of BPEL diagrams. It is one page of the multi-page editor.

What I do suggest is to add new pages in this editor. It could be achieved nicely with an extension-point. Each contributor would provide a composite or an editor ID to add in the multi-page editor. Each contributor would receive the editor input and the MVC pattern would guarantee the synchronization between the pages.

Users could then define in their preferences which pages would be visible in their BPEL designer instance. Ideally, the current representation should also be provided as a contribution to this extension-point.

2 Alternative UI

This section describes new pages that could be added to the multi-page editor. There are at least two for the moment, but there could be more.

One of the most important weakness of the BPEL Designer is its lack of accessibility for new beginners. It mixes several things together, which make users confused. These new pages target developers with no or few BPEL experience. Programming notions should be enough to design graphically BPEL processes. Some other pages, advanced properties and an extended XML editor could be available for experienced BPEL developers.

The two main pages to add would be:

- ◆ One for partner links, since every user will have to deal with them directly.
- ◆ One to represent the diagram. The diagram will show the executions flows as an oriented graph. All the BPEL activities will not be visible on the diagram (sequences as example). Graph vertices will be conditional bounds (conditions, loops, invocations). This will result in smaller diagrams, abstracted from the BPEL specification or syntax.

Another difference between the new diagram page and the current one, is the absence of the dashboard palette. The dashboard palette is the area of the BPEL Designer that lists partner links, variables, correlation sets, etc. This diagram page would only show the execution flows.

Let's talk a deeper look at specific BPEL activities and concepts, and how they could be shown and edited.

2.1 Partner links

Partner links are a key concept in BPEL. Every user will have to deal with them. So the more they are simple to define and update, the better it is.

To not confuse users, they should be defined and updated in their own page. We could see this page as a catalog of partner links.

In addition to a name, a partner link type and roles, a partner link may be associated with an image. This image would be reused on the diagram view: exchanges between the process and a partner (**invoke**, **receive** or **pick** activities) will link a graph node and this image.

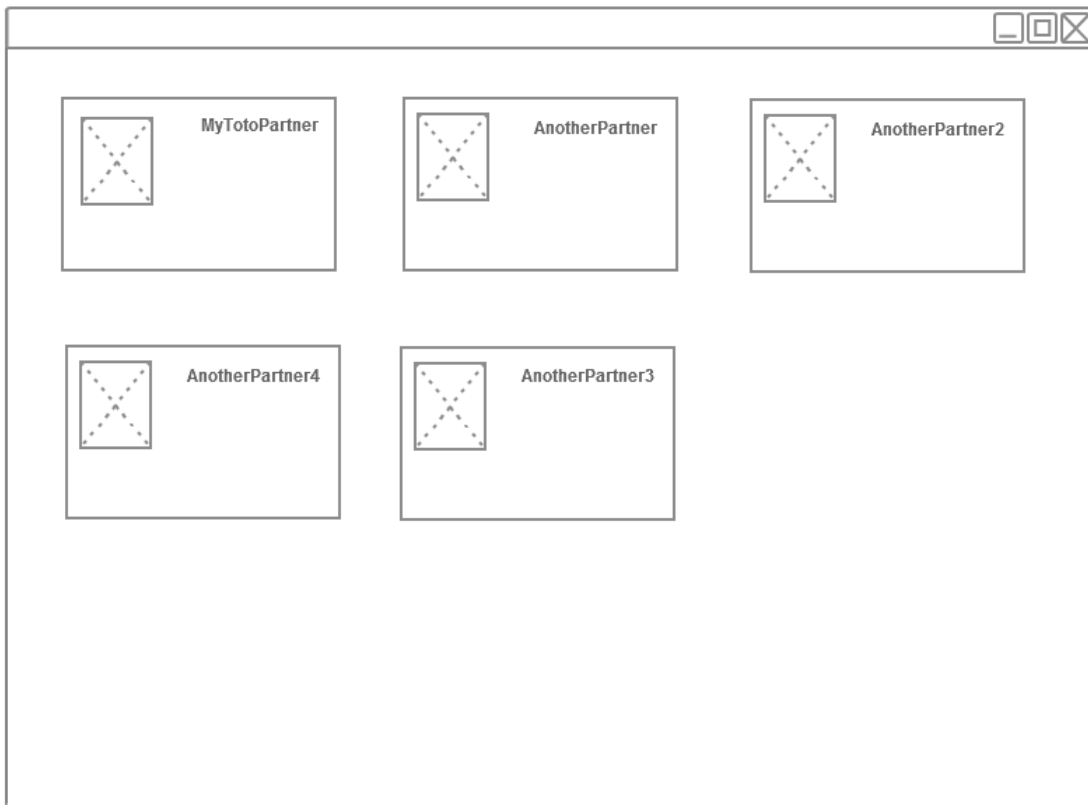
This page will list all the partner links and allow to edit their properties.

Three modes could be proposed:

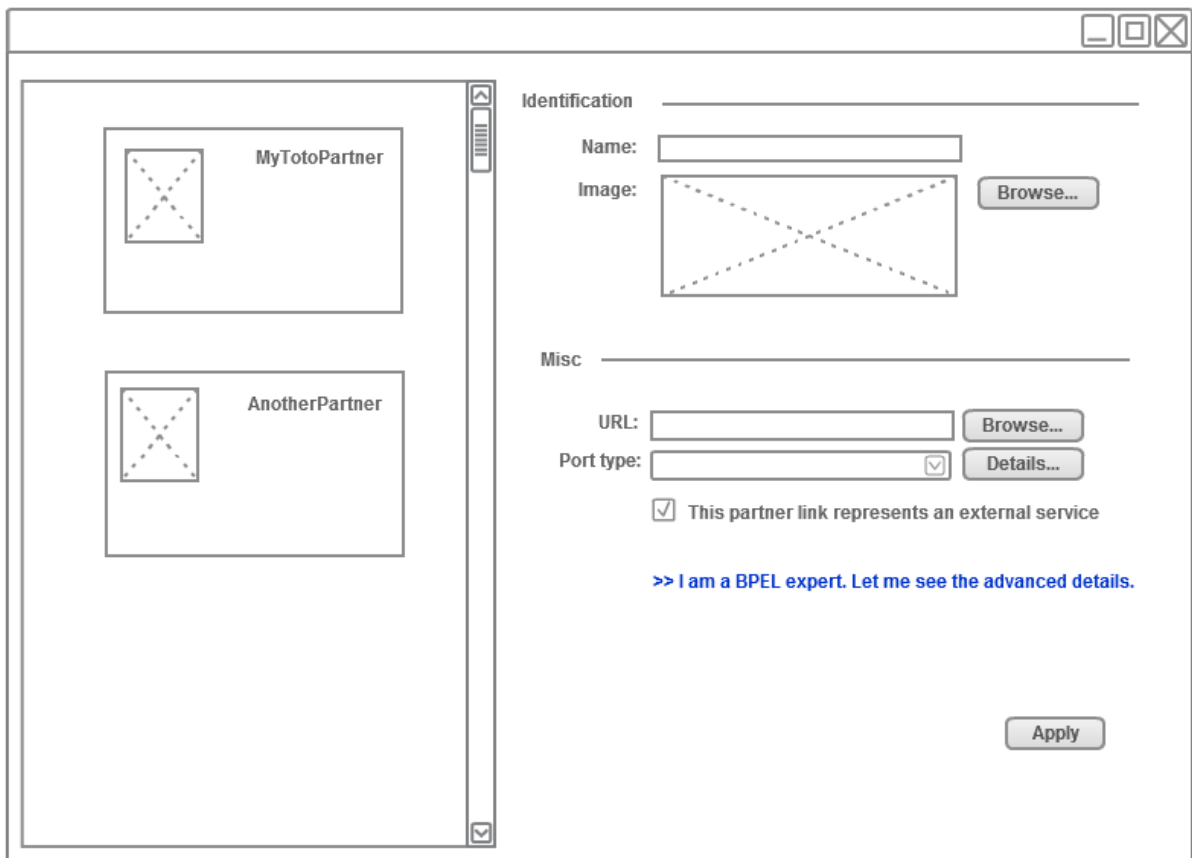
- ◆ Overview (display all the partner link's card ID).
- ◆ Horizontal browser (the top part shows the partner link's card ID) and the bottom part shows the properties to edit.
- ◆ Vertical browser (a derived version of the previous mode: list on the left, properties on the right).

Here are some rough sketches.

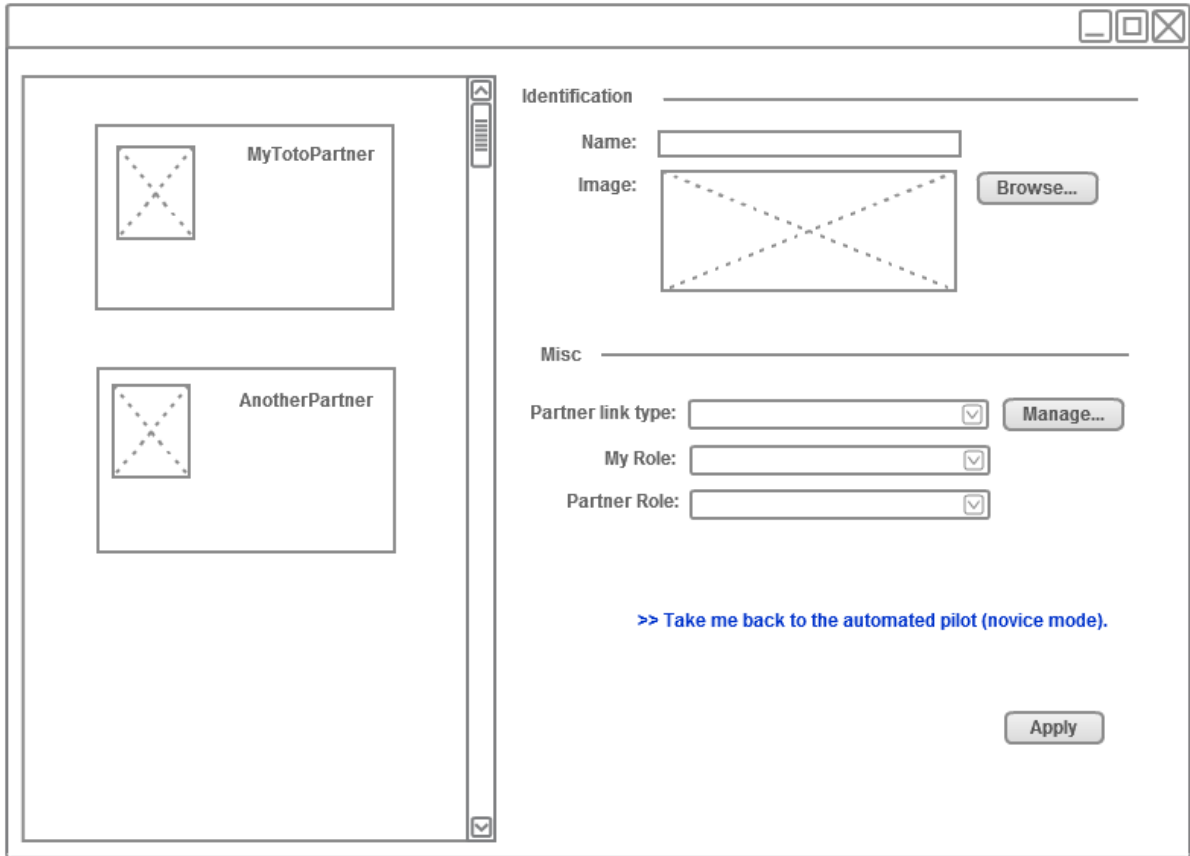
The first one lists all the partner links. When we select one, we reach the vertical / horizontal browser mode.



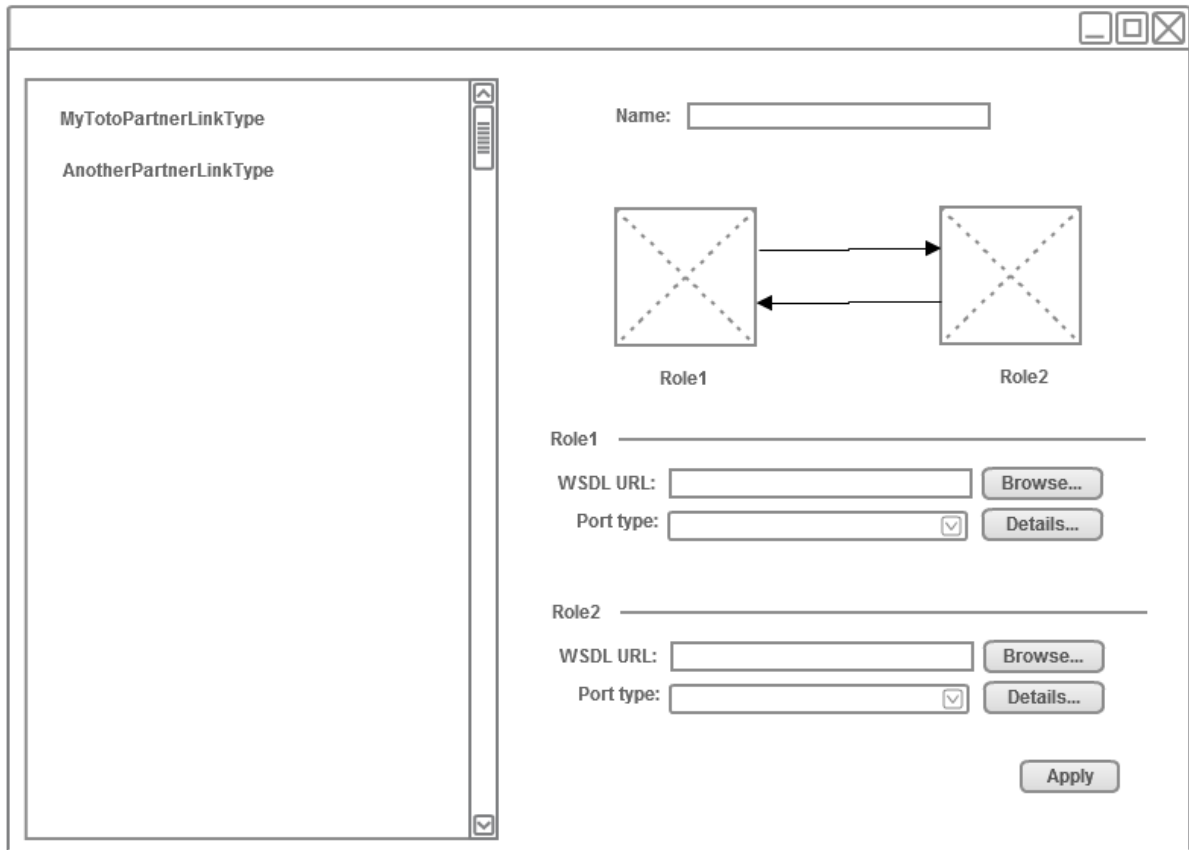
List all the partner links (useful only with a lot of partners)



Novice view of the partner links (vertical browser)



Expert view of the partner links (to manage partner link types)



Management dialog for partner link types

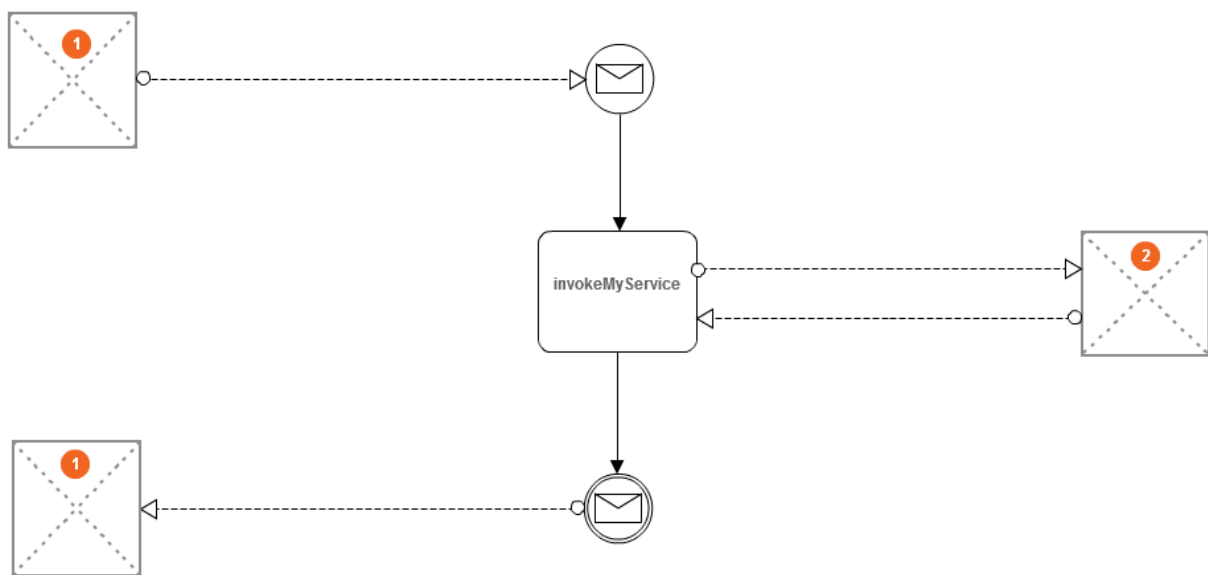
3 sketches (among the 4) show the content of the “Partners” page in the editor.
The last one is a dialog, which is tightly coupled with the management of partner links.

It is also possible that we have to define a convention (background color?) to directly identify partner links that are associated with myRoles or partnerRoles.

In the novice mode, partner link types are managed automatically. User do not need to edit them.
A partner link type will be created and associated with the selected port type, and roles will be created and set automatically.

2.2 Message reception and invocations

Receive and **invoke** activities will be displayed exactly like Netbeans does (but with less boxes).



There is no starting and ending points.

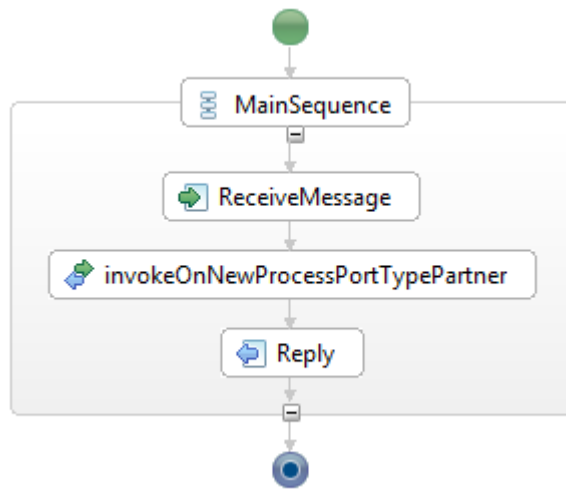
Here, we simply receive a message and forward it to an external service.

And we return its response as the process response. Message transformation could be added with minor changes (see the section about assignments and variables). Roughly, the diagram would be almost the same.

The orange circles (1) and (2) help to identify a partner link.

A partner link may be identified by an image, or by a default image with its name otherwise.

The same process would look like this in the BPEL Designer.

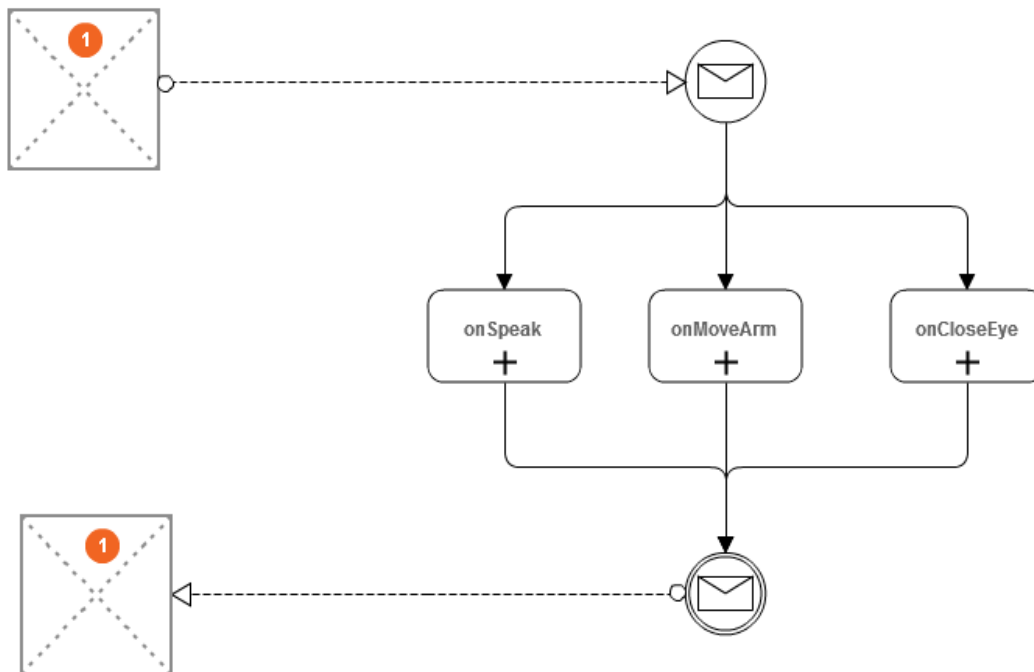


Pick activities will look similar to **receive** activities, except that each potential message will give distinct branches in the graph (just like in the BPEL Designer). **OnMessage** mark-ups will be represented by sub-processes.

A sub-process hides a subset of activities in the process. It is a front-end, that does not make sense in the BPEL source (although we could associate one with a scope).

We can explore a sub-process either by expanding it (not my preference, but reasonable for small sub-processes), or by showing a new area in the editor that details the sub-process content and that is linked with the sub-process.

Here is a pick activity...

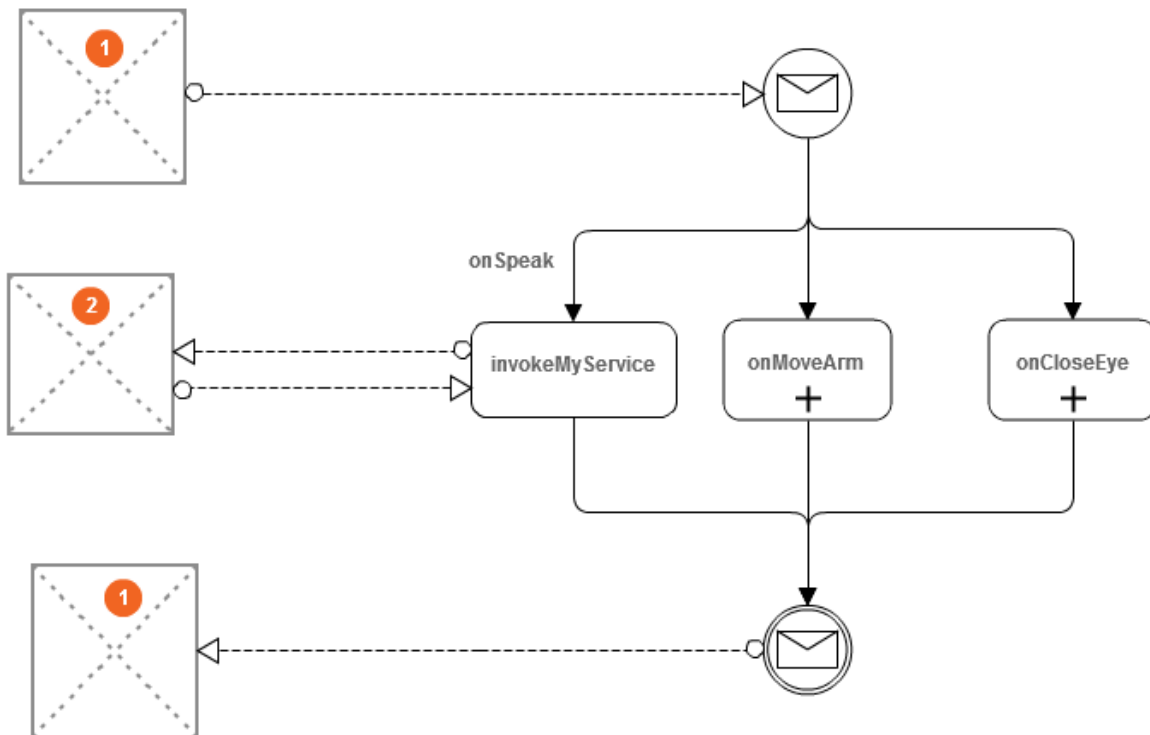


"Code blocks" (that would be placed between curly brackets) are placed in virtual sub-processes. These sub-activities could be extended (like in the BPEL Designer) or could result in a new editor area that would detail the content of the sub-process.

For the moment, I do not know whether both solutions should be kept. The second one has my preference. It makes the diagrams easier to read. Then, you have to zoom in on sub-activities to see what is under.

As big graphs may be hard to read, it is better to explore it by branch.

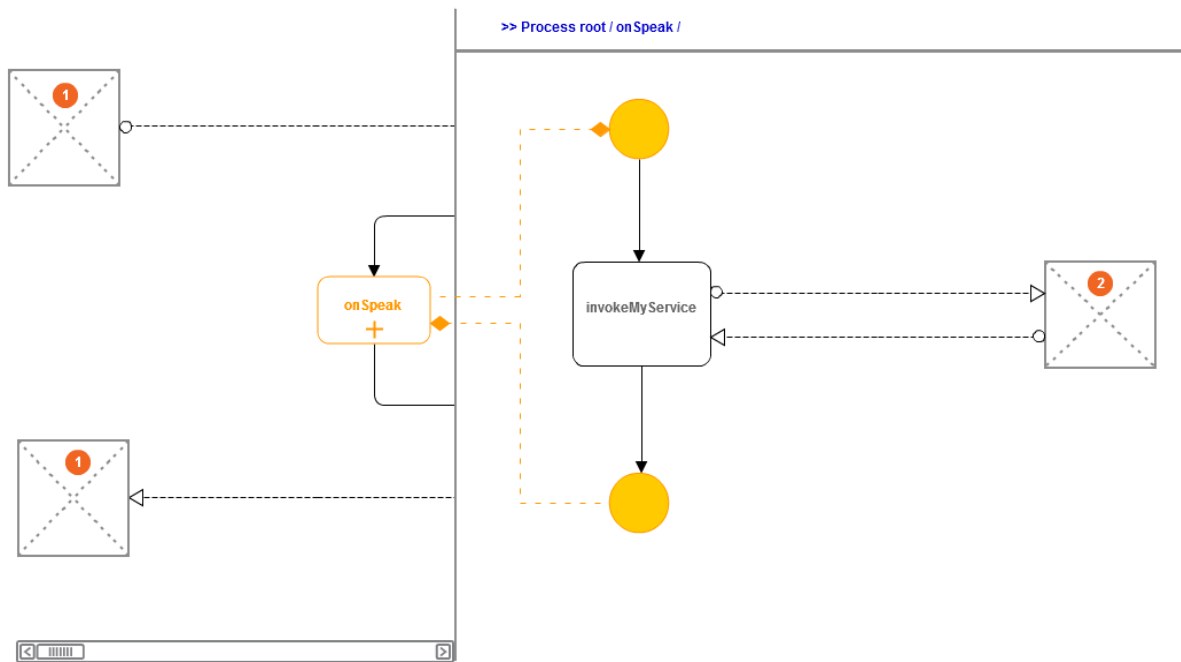
If we expand a sub-process...



It is only suitable for small sub-processes.

So, I don't know if we should keep this option or not. In terms of layout, it would be hard, for few advantages.

The best approach would be to have a new editor area (in the same editor, we simply divide it).



Process | Sub-process 1 | Sub-Process 2...

This gives us a hierarchy, which is in fact a path in the graph of the execution flow.

“In the process, I decided to go into the sub-process 1. And in the sub-process 1, I had a choice and I went into the sub-process 2”.

We graphically and progressively build the execution branches.

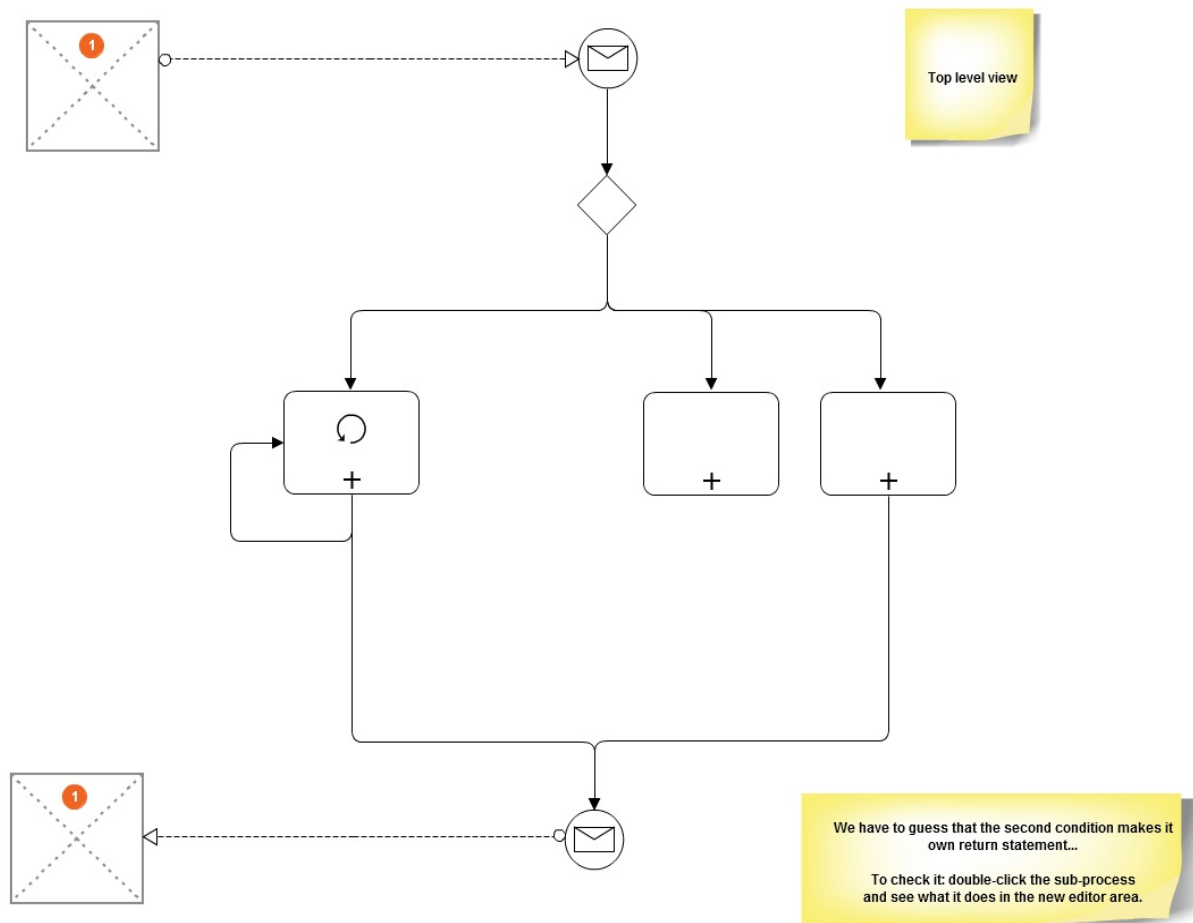
To ease the navigation and help the user to know where he is, we would have bread scrums in the top part.

We could even imagine display all the execution branches as hyperlinks, with a table of content.

The idea is to build all the small pieces and paths, and then to have analysis tools to make sure it is correct and coherent (see the chapter 3).

2.3 Conditions, loops and parallel flows

Conditions, loops and flows will make an important use of *sub-processes*. Here are some sketches.



Here is a sample process.

- ◆ On request reception (**receive** activity), we make a test on the request's content.
- ◆ Condition 1: we have a looping sub-process.
- ◆ Condition 2: the processing is opaque at the top level.
- ◆ Condition 3 (default processing): the processing is opaque, but this branch joins the one of the first condition to reply to the client. We can only assume that the second conditional branch makes its own return statement.

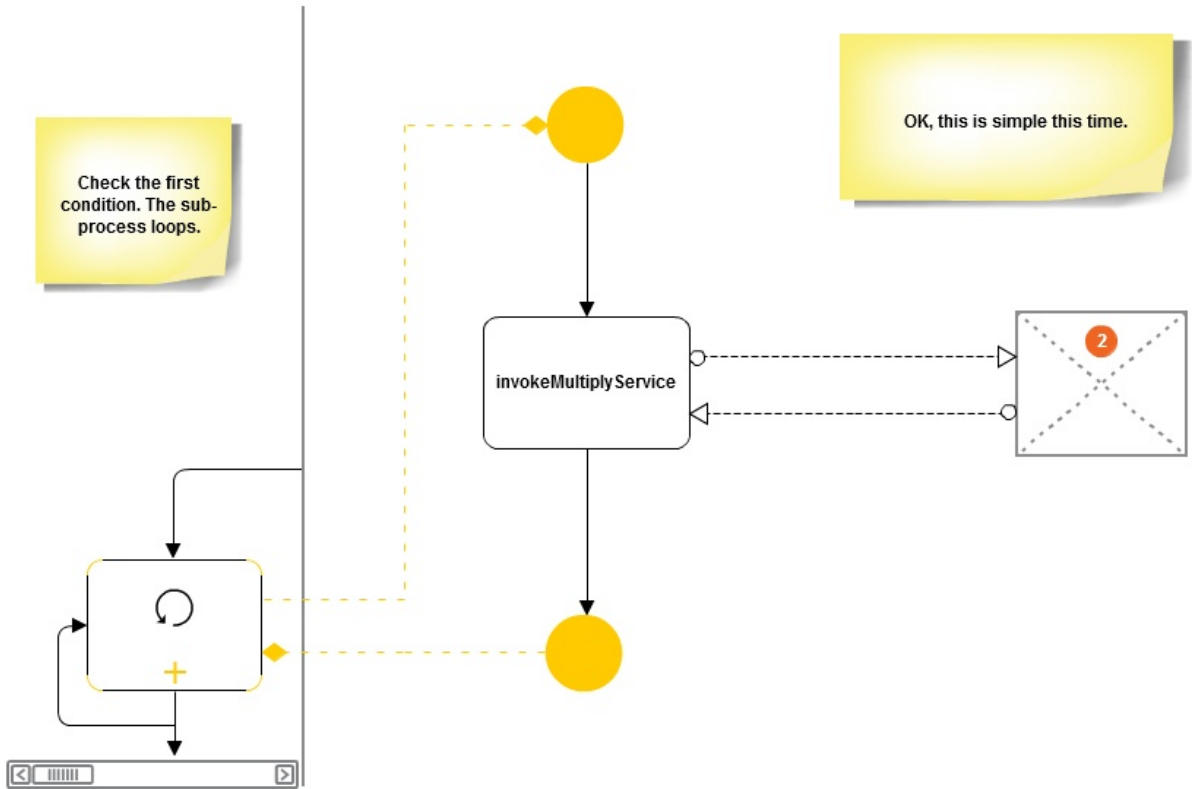
Let's take a look at the loop's sub-process.

The sub-processing is very simple, since we only invoke a partner link.

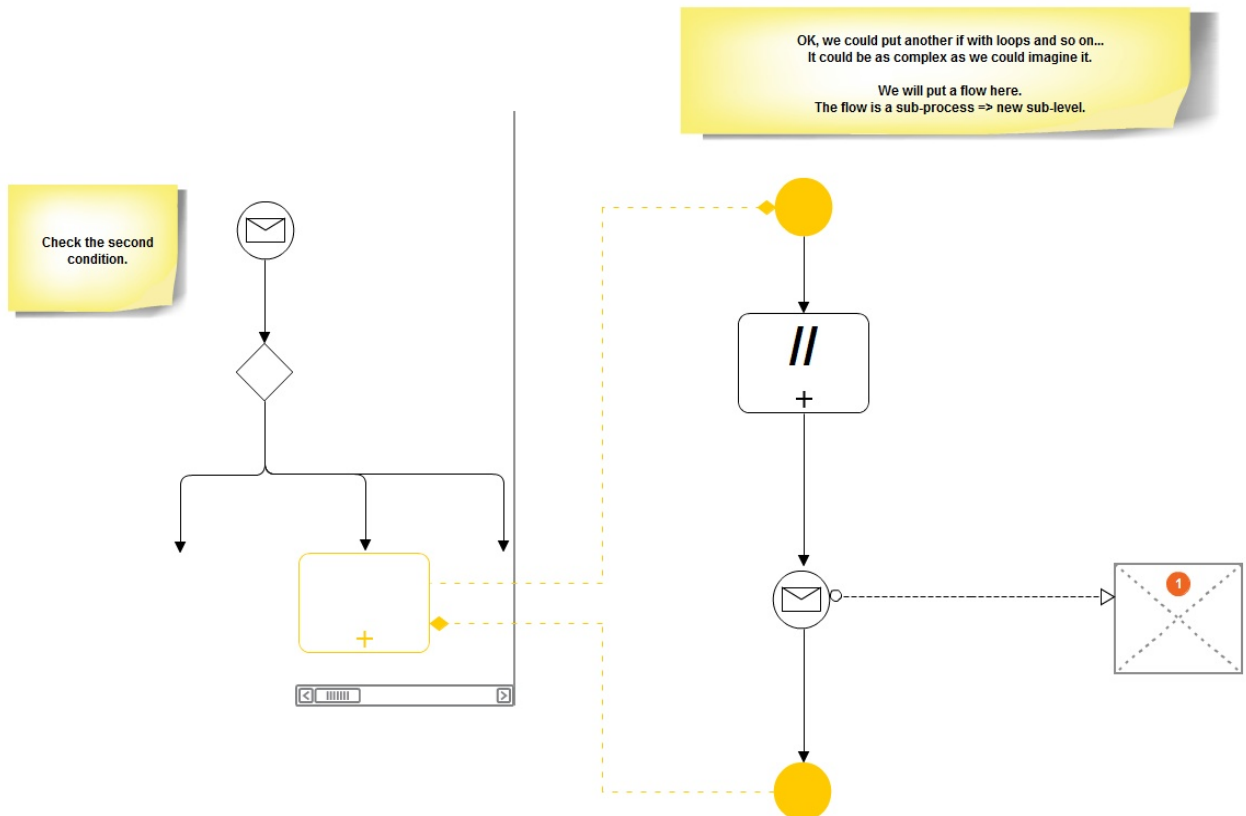
We will invoke it until the loop condition is satisfied.

Note that the notation used for the looping sub-process is a shortcut.

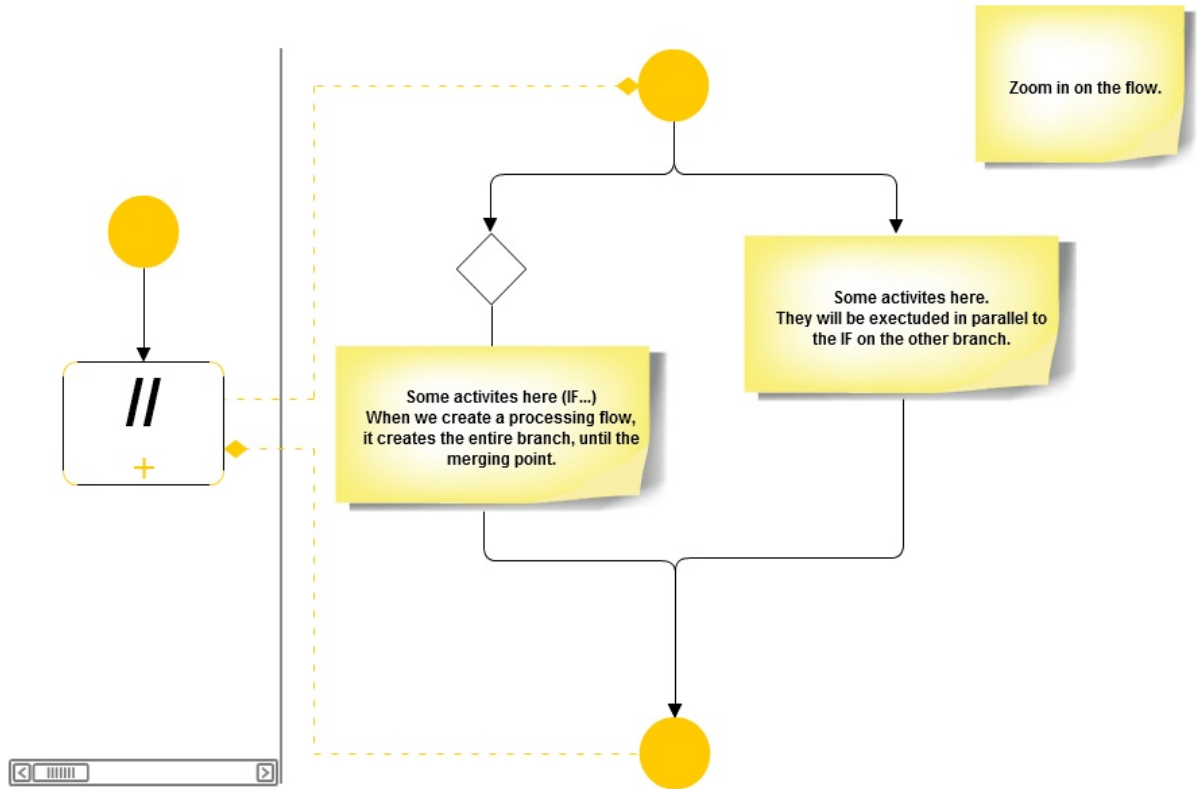
By saying the sub-process is looping, we avoid the situation where we have a sub-process and this sub-process starts with a loop activity that forwards to another sub-process.



Let's see the second conditional branch.
 This one involves a **flow** as the starting activity, followed by a **reply** activity.
 A **flow** activity is necessarily represented by a sub-process.



This second sub-process is show below.
In parallel, we have conditional branches and some other activities.
Everything that is external to the flow will be represented at the upper level, near the flow sub-process.



Note that I did not put the breadscumbs on these images.
But they should be on them. Making the navigation in the execution paths is very important.
As the logic is not build the process in one block, we must ease the creation and the assembly of all the sub-parts.

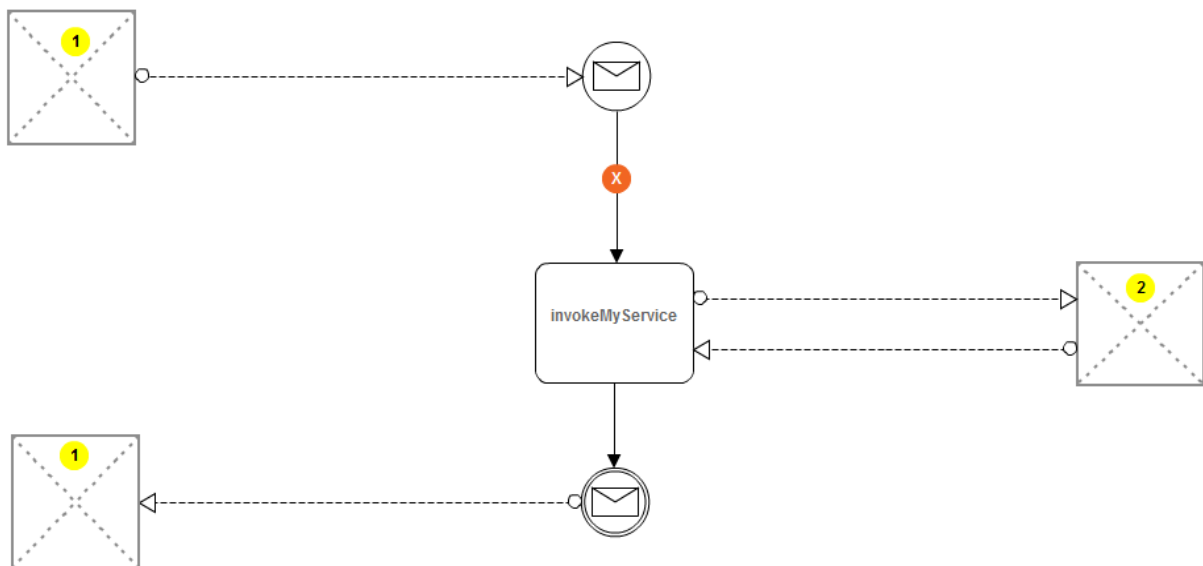
2.4 Variables, assignments and validation

Variables are tricky objects, that can be responsible for many errors at runtime. They are created, updated or destroyed, depending on their definition and the location in the execution flow.

Variables do not have to be visible on the diagram. It is already the case with the BPEL Designer (if we exclude the dashboard palette). They are used in some activities, like **assign**, **validate** and **invoke** activities.

Validate and **assign** mark-ups can be represented as simple *interceptors* on the execution flow. It means they are decorators of the execution flow. The goal is to make them discreet. We do not have to consider them as other activities. In practice, they are used to make the glue between other activities.

As an example, here is our first process we used in this document. We simply added it an assignation between the **receive** and the **invoke** activities. Note that I changed the color of the partner IDs (yellow instead of orange, as it was conflicting with the color of the assign).



Validate can be represented the same way, but with a different symbol (see the chapter 2.5). Variables could either have their own page, as a new page in the multi-page editor, or be defined during the execution flow.

As an example, before a loop, one could create an **assign** in which he defines a new variable and its initial value. In this case, the (x) symbol is more than assignments. It becomes an access to variable management. Wherever we want to deal with variable definition or update, we will use this symbol.

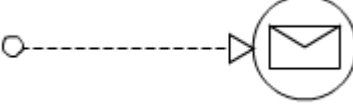
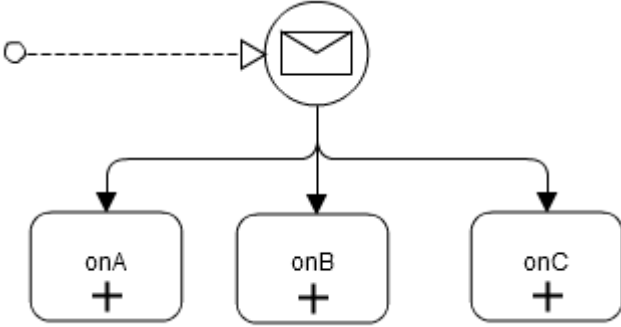
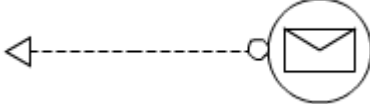
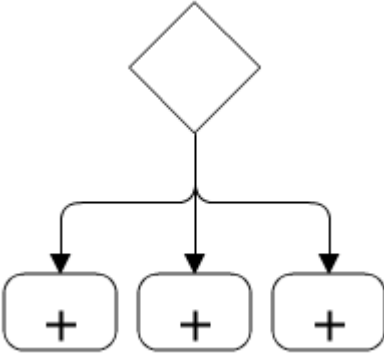
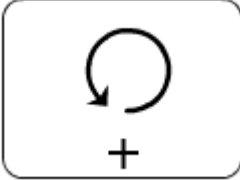
Where variables are stored in the BPEL source, or their scope, is not important for most of the users. Later on, if the user needs to access variables, only those that are on the same execution path will be proposed. Once again, we hide the complexity. And we provide validation and analysis tools.

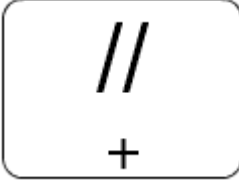


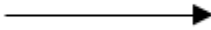

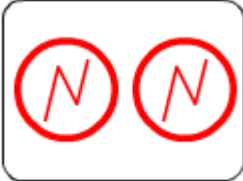

Clearly, nothing is determined for variables, as there are many possibilities.

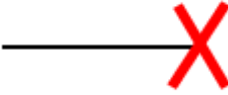

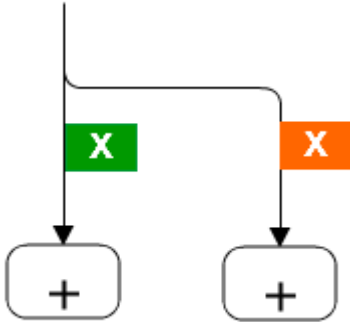
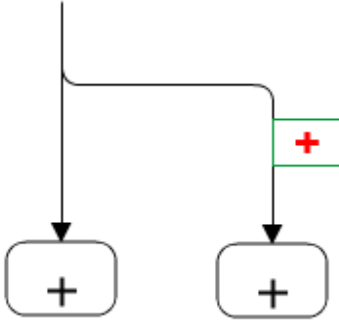


2.5 Notations and icons

Here is a list of the images used in this document.

Most of them come from UML activity diagrams or BPMN. There is no need in reinventing images. It is more about the way we use and assemble them together. Some of them also look like the icons used by the BPEL Designer.

| Graphical element | Image |
|---|--|
| Receive |  |
| Pick / On Message |  |
| Reply |  |
| If |  |
| While / For Each / Repeat Until / Loop sub-process |  <p data-bbox="555 1980 1345 2049">A loop sub-process is a sub-process on which we iterate. Said differently, the sub-process only contains a loop activity.</p> |

| | |
|-------------------------------------|--|
| Flow / Flow sub-process | <div style="text-align: center;">  </div> <p>A flow sub-process is a sub-process which only contains a flow activity. A flow is always a flow sub-process (UI constraint).</p> |
| Partner link | <p>An image, as specified by the user in the definition of the partner link. If no image was specified, a default one is used, with the partner name.</p> |
| Empty (activity) | <div style="text-align: center;">  </div> |
| Assign / Variable management | <div style="text-align: center;">  </div> |
| Execution flow | <div style="text-align: center;">  </div> |
| Throw | <p>Not sure about this one.</p> <div style="text-align: center;">  </div> |
| Rethrow | <p>Not sure about this one.</p> <div style="text-align: center;">  </div> |
| Wait | <div style="text-align: center;">  </div> |

| | |
|------------------------------|---|
| Exit |  An execution flow that is interrupted. |
| Sub-process |  A sub-process. When clicked, its content is shown, either in the same diagram, or in a sub-area of the editor. By default, a sub-process represents a sequence but can be marked as a loop or a flow. |
| Fault handlers |  |
| Compensation handlers |  |
| Compensate |  |
| Validate |  |

2.6 Conclusion

First, there are things we did not mention in this document.

For the moment, message exchanges, correlation sets and extended activities have not been studied. About scopes, there may be a relation with sub-processes (as an option in the properties).

Anyway, this document is a basis of reflexion and not a complete specification. It aims at starting a discussion.

Besides, readers may have noticed that what is described in the alternative UI is not revolutionary:

- ◆ Show less elements on diagram. Show less boxes.
- ◆ Stop the edition of huge diagrams. We edit processes part by part and progressively.
- ◆ Reduce the displayed texts. Use images, tooltips and conventions to represent elements.
- ◆ Use the execution flow as the visualization paradigm of the process. No more syntactic representation of the process.
- ◆ Make shortcuts for new beginners, by using default values, conventions and generated elements. Experts may have additional properties, but may also go in the code to edit manually some parts that are not accessible in the UI because it is too advanced / tricky.

Eventually, I would like to insist on the fact that BPEL is a programming language, and that somehow, direct graphical edition for BPEL is a non-sense (as it would be for any usual programming language – for which we would rather use a code generation approach). A scripting language that would be converted or transformed in BPEL would make much more sense for developers (people write C or Java very easily – if we remove the XML mark-up constraint, it would go very fast to write a BPEL, with curly brackets and so on).

BUT, whatever I think of the idea of a graphical editor for BPEL, a lot of people in the SOA ecosystem and in IT organizations have been convinced that it was feasible, and that BPEL was not too difficult, in particular with graphical editors. There is no perfect solution or tool. This alternative UI is one solution among others, but I think it can really help most of the developers, reduce the pain of using BPEL and be more simple to use than the current one.

Once again, this document is a first draft and the discussion is, hopefully, only starting... :)

3 Others extensions

3.1 Analysis and verification

One of the things the alternative UI pushes is the cut of big diagrams into sub-diagrams, sub-processes. However, people may want to see the overall picture.

If it is something to not recommend for edition, it is on the contrary a good feature for documentation. Thus, there should be an image export of the process, that could rebuild the entire process without having sub-areas. Sure, people will end with huge pictures (just like with BPMN), but they will ask for this feature anyway.

We could also have a documentation export, based on the documentation properties. One other notion that is highly used in the alternative UI is the branch or path notion. A path of execution flow. We could easily generate a document with all the possible execution paths, as a set of sequences. That could help for code review.

Code review also implies analysis tools.

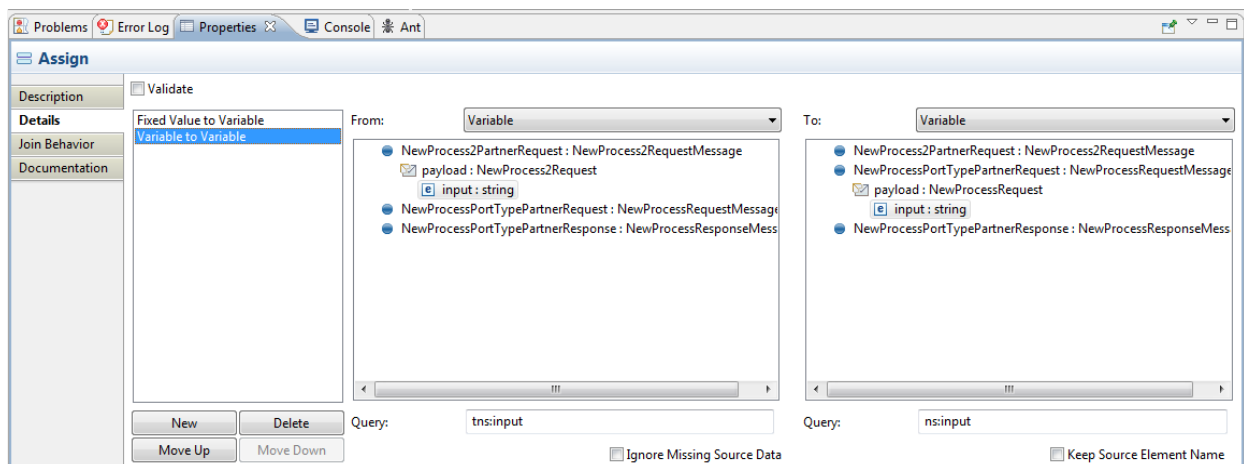
Because not everything can be represented at the same level, because complexity can change over execution branches, analysis tools could be provided. As an example, we could easily find if in the process, one or several branches do not return something to the client, or if they return something while they should not (fault, variable).

In the same way, useless variables could be found and removed.

It is not about syntactic validation, but about optimization and semantic validation.

3.2 Graphical mapping

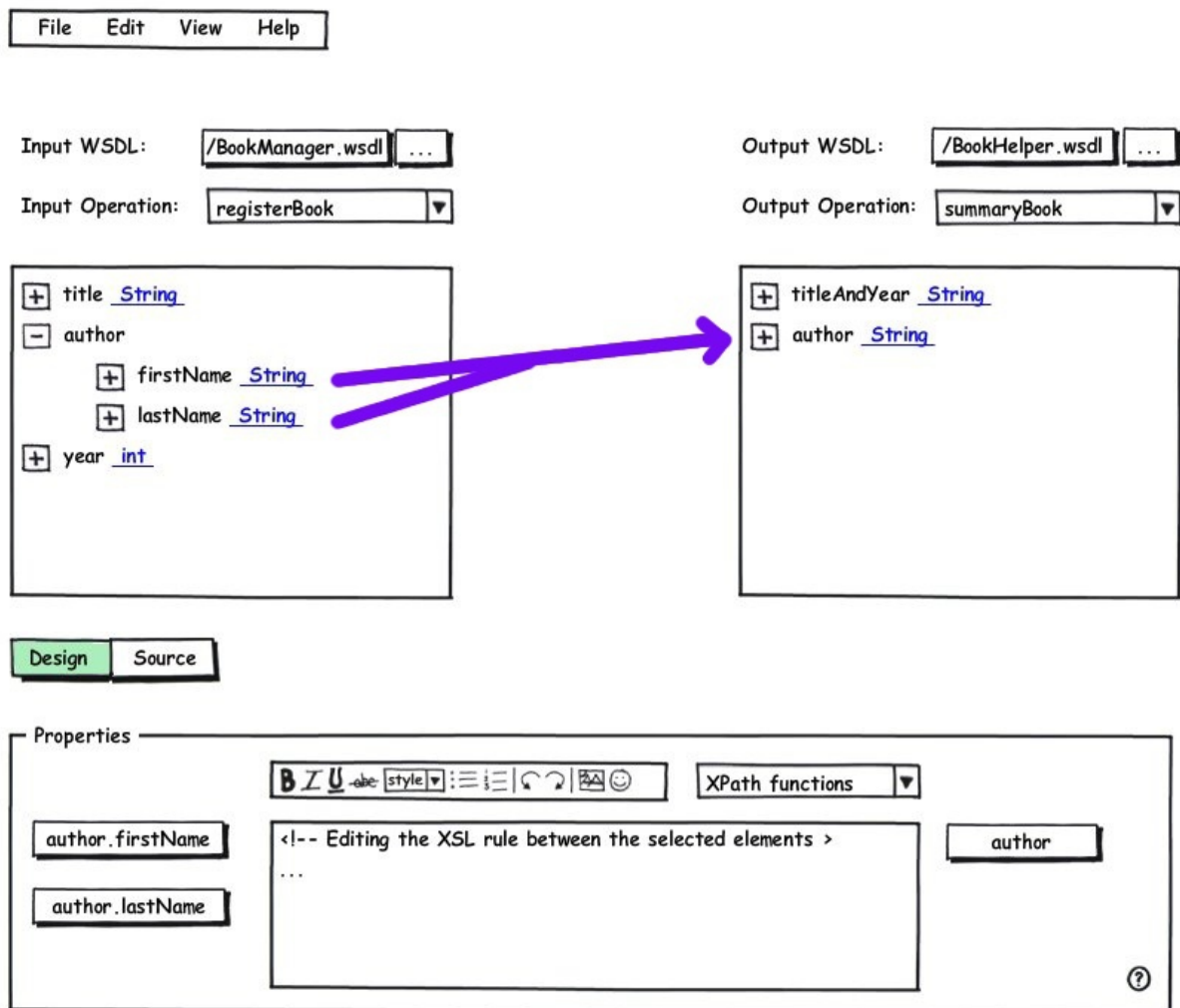
In a first time, assignments can be edited with the current tabbed properties of the BPEL Designer.



However, my company is *thinking* about a graphical editor for XSLT.

Graphical in the sense “ETL”. The idea is to have an editor that can take XML, XSD and WSDL (to access its XML schemas) documents as source, and from them, be able to generate the right assertions for XSLT. For the moment, it is still at the stage of idea. It should be seen as a long-term ambition (end of the year, sooner if we can have new resources to work on tools).

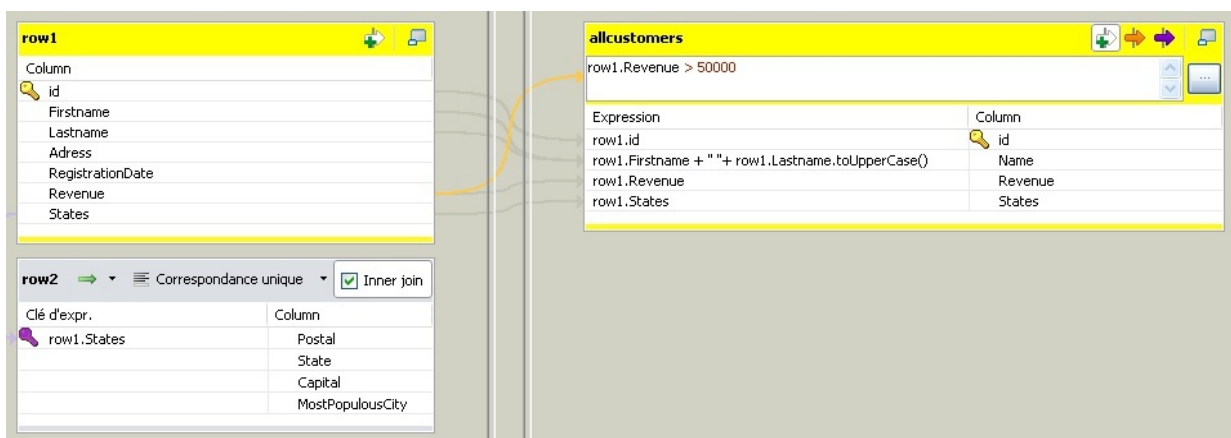
Here is a sketch of this editor, that I made several months ago.



created with Balsamiq Mockups - www.balsamiq.com

But with nicer arrows.

Something like what Talend offers, but for hierarchical data structures.



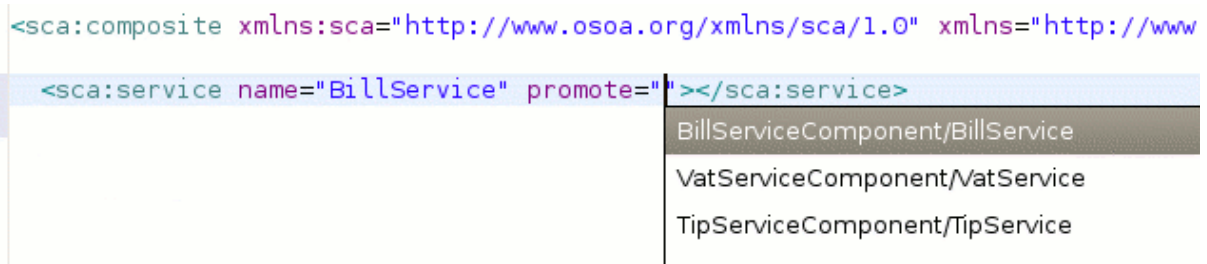
In my opinion, this project would have its place in Eclipse WTP.

The idea would be to make its core reusable, so that we could embed it in other tools, like the BPEL Designer. It would then be usable for assignments, as an alternative (but not exclusive) solution.

3.3 An extended XML source

The graphical view is always a great help, but when experience comes, it is sometimes better to directly go in the code. And from this point of view, the XML editor could be enhanced to simplify the use of the source page. In addition to the content assistance for BPEL mark-ups, we could have content assistance for mark-up and attribute values, as well as resource navigation (CTRL + mouse to open imported files).

It consists in customizing and adding new proposal processors to the WTP XML editor. This assistance could be used for variable names, port types, partner link type names, etc. This is something we already did in the SCA editor. It is a tiny feature, but generally, it is appreciated from developers.



```
<sca:composite xmlns:sca="http://www.osoa.org/xmlns/sca/1.0" xmlns="http://www
<sca:service name="BillService" promote="*"></sca:service>
```

The screenshot shows an XML editor with a dropdown menu for the 'promote' attribute. The dropdown lists three options: 'BillServiceComponent/BillService', 'VatServiceComponent/VatService', and 'TipServiceComponent/TipService'. The first option is highlighted.

3.4 Graphical debug

One last thing is about debug.

Indeed, one feature many BPEL developers would have is the debug of BPEL processes on their BPEL engine. Independently of the required API to make Eclipse interact with these engines, here are few ideas about how debug could be used in the alternative UI (and maybe even in the current process view). The source view is obvious, it will be like Java.

There is nothing magic, break points should be set as interceptors (decorators) on the execution flows. Except that the alternative UI already makes an important use of them. It could be confusing for users and lower readability. This why the debug mode should add a grayed layer on the process diagram. When we want to add break points, a gray veil comes on the diagram and colors are no more visible. Only break points will be colored. This is just about break point definitions, not process definition (we can think about 2 modes, one for edition, one for setting / removing break points).

About following the execution and the break points, I think we should have a specific view. In this view, the execution flow is built from the BPEL process. In the process, we may have conditional branches. It means several execution paths. But at runtime, only one will be *chosen*. This is this one we will rebuild graphically. When we reach a break point, the execution flow display will be interrupted, stopped at the break point.

We won't navigate in the process diagram.

We simply build the sequence of all the process activities that have been executed by the engine. Such a sequence is one of the execution paths that may have been exported for code review.

And of course, when we reach a break point, we should be able to see variable contents. But this more about debug than UI.