ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221542866

# A Simple Algorithm for Automatic Layout of BPMN Processes

**Conference Paper** · July 2009

DOI: 10.1109/CEC.2009.28 · Source: DBLP

CITATIONS
37

READS
12,355

**4 authors**, including:

Daniel Lübke
Leibniz Universität Hannover
**71** PUBLICATIONS **899** CITATIONS
SEE PROFILE

Leif Singer
University of Victoria
**40** PUBLICATIONS **2,149** CITATIONS
SEE PROFILE

**Some of the authors of this publication are also working on these related projects:**

Project Executable Business Process Testing View project

Project Microservice API Patterns (MAP) View project

All content following this page was uploaded by Leif Singer on 20 May 2014.

The user has requested enhancement of the downloaded file.

2009 IEEE Conference on Commerce and Enterprise Computing

© 2009 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

# A Simple Algorithm for Automatic Layout of BPMN Processes

Ingo Kitzmann, Christoph König
Leibniz Universität Hannover
Software Engineering Group
Welfengarten 1, D-30167 Hannover, Germany
{ingo.kitzmann, christoph.koenig}@se.uni-hannover.de

Daniel Lübke, Leif Singer
Leibniz Universität Hannover
Software Engineering Group
Welfengarten 1, D-30167 Hannover, Germany
{daniel.luebke, leif.singer}@inf.uni-hannover.de

## Abstract

*Badly and inconsistently layouted business processes are hard to read for humans and therefore lack comprehensibility. Furthermore, processes generated by software have no layout at all.*

*If stakeholders cannot comprehend the process descriptions, they are unable to validate them and find mistakes.*

*By offering a fully automatic layout algorithm for BPMN, it is possible to layout business processes in a consistent and clear way so that stakeholders can better and quicker comprehend the contents.*

*This leads to better comprehensiblity and thus to better communication in BPM / SOA projects and allows for consistent process layouts throughout projects and enterprises – independent from the model source. In addition, this permits using generated models without manual layouting.*

## 1 Introduction

Nowadays, the *Business Process Modeling Notation* (BPMN, [11]) gets more and more established as the standard graphical notation for business processes. This success is partly based on offering simplicity to the user, while still providing rich expressiveness. Additionally, BPMN is strongly linked with BPEL, the *Business Process Execution Language*. Different conversions between BPMN, BPEL and other formats exist (e.g., [8] and [7]), serving several needs for interoperability.

Even though the conversions to and the generation of BPMN models required in this context produce semantically sound process models, a clear visual layout would be beneficial. This is all the more important if the produced BPMN models are to be used as tools in discussions about the actual processes, e.g., in requirements engineering or BPM activities. Automating this layouting process to produce BPMN diagrams that can be easily understood should ideally be just another step of the conversion / generation of the BPMN models.

Similarly, manually created models benefit from automatic layout, which reduces the required effort and guarantees a consistent layout for different models created by different people.

This paper presents a layout algorithm which takes a BPMN model in eRDF-format as input and layouts it. eRDF is used by the *Oryx Editor*[1], a web-based modeling tool supporting business process modeling in BPMN, created by researchers of the Hasso-Plattner-Institut in Potsdam, Germany. The algorithm was developed as a contribution for the InformatiCup 2008[2], a student competition by the Gesellschaft für Informatik e.V. The *Oryx Editor* and its format were chosen since it is freely available and needs no installation on the client.

The main focus is the positioning of the *elements*, having the greatest impact on the final layout. Reduced effort was put into positioning the edges.

## 2 Related Work

There are many different graph layout algorithms in use today. The characteristics of the generated layouts of the al-

[1]http://www.oryx-editor.org
[2]http://informaticup.de/

gorithms differ strongly, as most algorithms target a special application of the created diagrams – e.g., organizational charts or circuit design. Most of the general graph layout algorithms are not suitable for BPMN since they do not take into account the special constraints that can improve the readability of BPMN diagrams.

Force-based layouts (e.g., [4]) and spectral layouts (see [6]) produce rather organic diagrams and target dynamic, often changing graphs. These are not suitable for visualizing business processes. Also, their results may differ significantly between different runs, lacking the consistency the approach presented in this provides.

Orthogonal layouts – e.g., [1] – produce results less organic, but may not really be focused on clarity. Instead, common goals of these algorithms are, e.g., minimizing the area taken up by a graph. This may ignore or even contradict the clarity required from BPMN diagrams to make them effective tools for communication. Specialized orthogonal layout algorithms may nonetheless be able to produce satisfying results for BPMN.

Hierarchical layouts, as in [2], are even better suited for BPMN. As the hierarchies given by splits and joins, pools, lanes, and subprocesses may be taken into account, the process can be understood much better when layouted by such an algorithm. Although general hierarchical layout algorithms will still lack in some aspects concering BPMN layouts, they can be improved with constraints tailored for the requirements BPMN. As the algorithm presented in this paper respects and utilizes all these hierarchies, it is considered to fall into this class of algorithms.

[3] introduces the concept of *divisions* to distribute a diagram across multiple pages. While this approach addresses the lack of overview inherent in large diagrams, it does not aim to improve existing layout algorithms significantly – any such improvements seem to be considered side effects of the divisions. The actual layouting is done using existing algorithms, namely an orthogonal one for an initial layout and a sketch-driven orthogonal one in a second run after the division. The report does not go into much detail concerning the runtime properties of the approach. Considering the algorithm layouts the whole process once, and, then again each division created in the intermediate step, a more detailed description of its runtime behaviour would be desirable.

Graphviz [5] is a framework for graph layout, supporting several different layout algorithms. Due to its open license, it is used in many applications dealing with process visualization – e.g., ProM [10]. Preliminary trials using the example process from section 4 as input resulted in the graph shown in figure 1. While the produced layout is certainly good enough to provide some understanding of the process, the algorithm presented in this paper further improved comprehensibility. Especially the vertical association of opening and their corresponding closing gateways is a huge advantage in the subjective opinions of the authors of this paper. Still, a more thorough examination of Graphviz might lead to more satisfactory results.

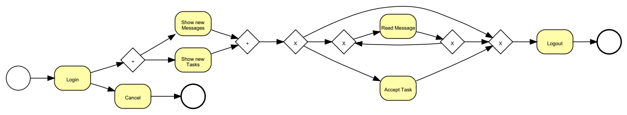

**Figure 1. The layout produced by the `dot` tool from Graphviz.**

## 3 Algorithm Description

In conceiving the algorithm presented in this paper, the authors started by deciding that it would be favourable for a graph layout algorithm to have no need for an understanding of the whole process described by a BPMN model.

The approach therefore operates mainly locally and avoids having to analyse too much of the process. This section presents the algorithm. It begins with the prerequisites, explains the basic idea and then goes into the details of some parts.

### 3.1 Prerequisites

BPMN models must conform to some rules in order to being processed by the algorithm. Most rules are related to how processes are syntactically modeled. The only technical rule is that all elements (except for pools, lanes, edges and uncollapsed subprocesses) must have set a size.

This was delegated to the generation of the model because this way the algorithm does not have to distinguish between generated and manually created models. In the latter, elements are typically sized according to their content, e.g., the length of the text. To preserve the effort that may have gone into this, the sizes are not being overridden with standard values. For generated models, the generation step can set standard sizes for all elements.

Syntactical modeling rules are:

- The model must have at least one start event or at least one element without any incoming links.
- Each edge must be connected (no open ends).
- A subprocess must not be empty and must be isolated, i.e., there are no edges between the subprocess and the parent process.
- Each element is connected with another one by only one edge and must have no link to itself.

- Groups are not supported by the algorithm because the eRDF does not save a logical connection between the group and the elements which are grouped together.

Most of these rules ensure the syntactic correctness of the input models. Because of their simplicity, they should be easily satisfiable.

## 3.2 Layout Goals

Some requirements from the competition are concerned with the layout and are supposed to ensure consistency. The following paragraphs will explain why these requirements are useful.

First, the process flow should be from left to right. This matches with the horizontal progression of text in western handwriting.

Exception handling of tasks should be below the task, to not desturb the process flow.

Elements after a split-gateway (which has more than one outgoing edge) should be right of it (not directly under or above). The inverse applies for elements before a join-gateway (which has more than one incoming edge), i.e., the preceding elements should be left of it. Additionally, joins should be at the same height as the correspondent split. Both requirements help in identifying corresponding gateways.

## 3.3 Basic Idea

Only a few simple ideas lead to the algorithm. The first one gives a more general look at the BPMN process: elements were first classified by those attributes important to the layout. The result of this classification are the types shown in table 1. An element can have different types, e.g., a gateway can be a split and a join at the same time.

**Table 1. Types for a more general look at the process**

| Type | Description |
|---|---|
| Edge | Sequence flow, message flow, data flow |
| Element | Every element of the process which is not an edge |
| Start Event | All types of start events |
| End Event | All types of end events |
| Join | An element with more than one incoming edge |
| Split | An element with more than one outgoing edge |
| Docked Event | An intermediate catching event, which is docked at a task |

The first problem to address is the global effect of local changes. A displacement of one element has effects on the elements connected with it, and so on. A solution was needed that allowed doing local changes while automatically maintaining the relative alignment of other elements – i.e., whether an element is above, below, to the left or to the right of another. That solution, called the *Grid*, is presented in section 3.4.

The processing order was chosen to be along the progression of the process. Therefore, a topological sort seemed helpful. Unfortunately, BPMN allows cycles, making it impossible to use an ordinary topological algorithm. In section 3.5, a slight modification is shown which gives the desired result.

With these preparations, the process can be layouted from left to right, and each element is positioned relative to its predecessors. This is described in section 3.6. After that, section 3.8, describes some heuristics enhancing this general approach. Once each element has its position in the *Grid* assigned, the coordinates for the elements are calculated.

Finally, the edges are set. Based on the type of the edge, the types of the source and target element, and their relative position, one type of edge is chosen. Amongst others, there are a 90 degree corner, a step, and a direct connection. Each edge is layouted independent of all others, i.e., other edges have no effect on the layout of one edge.

## 3.4 The Grid

One part of the solution presented in this paper, the *Grid*, is inspired by spreadsheets. Spreadsheets provide functions to insert rows or columns (i.e., local changes) and move the other cells away (a global effect). But their relative positions are maintained.

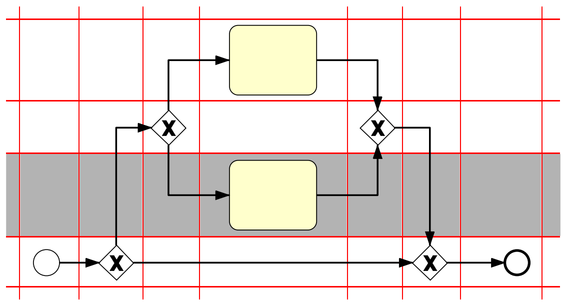

**Figure 2. Sketch of the Grid with newly added row.**

This idea was adapted to the *Grid*. It starts off with one cell, while rows and columns are added as needed. With the *Grid* it is possible to add branches without influencing the relative positions of elements which were already set. Figure 2 shows a simple example, in which the lower task added a row and pushed the upper gateways and task outwards. But they are still above and between the lower gateways. As the algorithm layouts from left to right, each split leads to more rows.

For calculating the real coordinates of the elements from the *Grid*, for each row and column the maximal height and width is determined. These are set for the according row and column. Then each element is placed in the center of its cell. This way, direct connections to succeeding elements with different dimensions will still be straight.

## 3.5 Modified Topological Sort

To be able to layout the BPMN from left to right, an ordering of its elements is required. The ordering must ensure that all predecessors of an element are layouted before that element (i.e., those left of it). To accomplish this, a modified topological sort was developed. The modification is based on the variant of the topological sort which uses an incoming link counter in the nodes.

**Listing 1. Pseudocode for the modified topological sort**

```
G ← Set of nodes to sort
L ← Empty list for the sorted elem.
S ← Empty Set for nodes with no incoming edges
B ← Set of backwards edges
while G is non-empty do
  // search free nodes
  foreach node n
    if incoming link counter of n = 0 then
      insert m into S
  if S is non-empty then
    // ordinary top-sort
    remove a node n from S and G
    insert n into L
    foreach node m with an edge e
            from n to m do
      remove e
      decrement incoming link counter from m
  else // cycle found
    // find loop entry
    foreach join j in G do
      if incoming link counter of j
        < initial incoming link counter of j then
        J ← j and break
    // process loop entry
    foreach remaining incoming edge e of J do
      replace l with backwards edge b
      add b into B

output: L,B
```

As opposed to the standard topological sort, the version presented here (listing 1) is able to handle cycles like those appearing in BPMN diagrams. The standard algorithm is used until it has detected a cycle and cannot go on. In this case, the algorithm seeks an entry point into the cycle, which is an element that has an already processed incoming edge. This is based on the idea of *dominant nodes* (see [9]). Now all remaining unprocessed incoming edges are flipped virtually (they become outgoing edges) to break the cycle as shown in figure 3. The standard topological sorting can go on to the next cycle, and so on.

After this process, the graph is acyclic and a topological ordering of the graph has been achieved, ensuring the criteria from above.

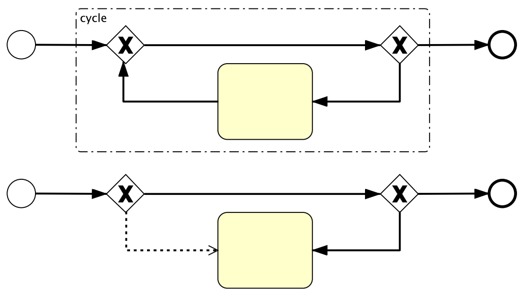

**Figure 3. The process before and after the first topological sort. The cycle was removed. The virtual reversed edge is shown dashed.**

Since elements like the task element in figure 3 now only have incoming edges, they will be ranged and layouted after the second gateway. As this is undesirable, to achieve a layout as in figure 3, the reversed edges are being traced back to the next split or join gateway and flip the edges on the way as well. In figure 3, the lower right edge would also be reversed. This works best with cycles with a single exit. Cycles with multiple exits (in general all cycles with additional splits or joins on a backwards path) tend to look a bit odd because backtracking stops at the splitting node, which may lead to a long backwards edge.

In some cases, this may lead to new cycles. If the gateways in figure 3 were merged into one (see figure 4) this would be such a case. But these types of cycles (splits without joins) should be avoided anyway, as they are not as clear as the one in figure 3.

Nevertheless such situations have to be dealt with, which is why the modified topological sort will be run again on the resulting graph – only this time without backtracing the reversed edges.

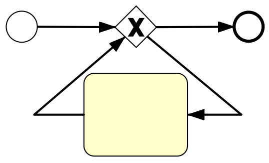

**Figure 4. One type of cycle, where backpatching of reversed edges leads to a new cycle.**

## 3.6 Positioning of elements

With the *Grid* and the topologically ordered elements, all preparations for positioning the elements are in place. The ordering ensures that at each step all preceding elements of the current element are already in the grid. For the current element, one of the following three options is chosen:

- In the simplest case, the current element has only one incoming edge and the preceding element is a basic element (i.e. no split or join etc.). Then the current element is put into the cell right of the preceding element.

- If the current element is a join, it is put into the column right to the rightmost one of the preceding elements. When the corresponding split is found, the current element is put into its row. Else, the middle row of the preceding elements is used (comp. fig. 5).

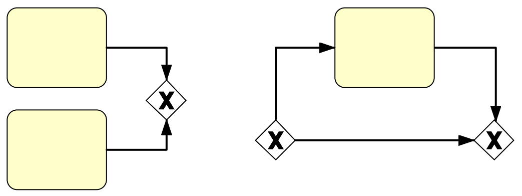

**Figure 5. On the left, the join is centered. On the right, the row of the split is preferred.**

- The succeeding elements of a split are distributed equally right of the split, centering them vertically relative to the split. For this, new rows are inserted into the *Grid*. Because the current element – one of the succeeding elements of the split – does not know its siblings (and they have possibly not been positioned yet) the positions of the elements after a split have to be prepared by the split. It marks the cells it deems reasonable for its successors (see section 3.8.2). Then the current element either uses this mark or rejects it if there is a rule with higher priority – e.g., joins trying to align with a different split.

## 3.7 Pools, Lanes and Subprocesses

Because subprocesses are isolated, they can be processed easily. For each subprocess, the layout algorithm runs once. The complete process is being processed from the most inner to the most outer subprocesses, so the algorithm can calculate the size of the subprocess for each parent process. After that, the parent process is layouted without any attention given to the content of the subprocess. This works since in eRDF, elements of a subprocess have coordinates relative to the parent element.

Collapsed pools also do not need much special handling, because they do not contain other elements. They are just placed at the top of the diagram, get a common height and are spanned across the whole width of the diagram. Technically, they get their own row in the grid and are removed from the regular processing order.

But more special handling is needed for uncollapsed pools and lanes, because they are partly isolated, as elements of one lane must not be placed in the area of another lane, and partly unisolated, as elements of lanes and the global process can be connected with message and sequence flows nearly without restrictions.

The chosen solution for this problem is to sort the elements of the global process and the elements of lanes and pools together (as described in section 3.5). Then the elements are placed in seperate grids: one grid for each pool or lane and the global process. In this way, the relations between process parts are preserved while elements cannot float into areas they do not belong into.

But for this solution the seperate grids need a sort of synchronisation, e.g., the grids must have the same width at all times. This is achieved by a supervising datastructure named the *SuperGrid*.

Also, positioning of elements (see section 3.6) needs some modifications in cases where the predecessor of the current element belongs to another lane. E.g., for the simple case, the current element is placed into the row that is nearest to the lane of its predecessor.

If the lane of the predecessor is above the element's lane, it is placed in the topmost row of the target lane. If it is beneath, the bottommost row is chosen.

To conclude, the order from top to bottom is: collapsed pools, pools and lanes, and the global process.

## 3.8 Heuristics

Some simple heuristics were added to the algorithm which improve the layouts for some cases.

### 3.8.1 Interleaving

As discussed in section 3.6, a split may insert new rows into the grid, pushing back older rows. While this works great for nested split / join pairs, it leads to an inconvenient and space-consuming layout for *sequential* split / join pairs. As shown in figure 6, the rows of the later split / join pair push apart the rows of the first split / join pair.

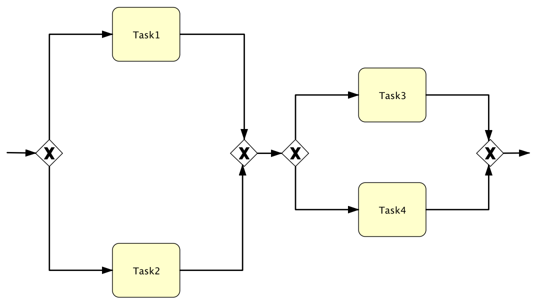

**Figure 6. The process before the Interleaving. Task 1 and 2 have been moved outwards by the rows of Task 3 and 4.**

To conquer this, the Interleaving heuristic was developed. It simply checks whether there are any rows in the grid that can be interleaved after all elements have been placed. Two rows can be interleaved if and only if they are adjacent and for each used cell (a cell where an element is placed) in one row, the adjacent cell in the other row is free.

If two rows can be interleaved, all used cells of one row are moved to the other row. The row that is now empty is being deleted. This process continues as long as there are any rows which can be interleaved. This leads to a more compact grid and mitigates the problem mentioned above in many cases (see figure 7).

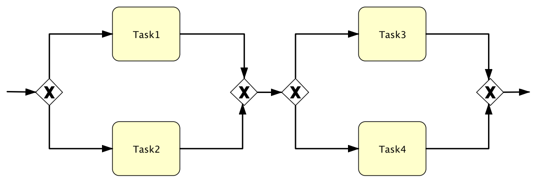

**Figure 7. The process after the Interleaving**

### 3.8.2 Prelayout

As written in section 3.6, a split prepares the layout for its following elements. This helps to arrange them equally.

A simple heuristic is used to improve the ordering of the following elements: text annotations at the top, data objects below, and finally sequence flows at the bottom. If one of the sequence flows is a direct connection to a join, this connection is centered. This should avoid text annotations between data and sequence flows. Also, directly connected joins have a straight egde between them.

### 3.8.3 Using the Grid for Edge Layout

It was tried to use the information in the *Grid* to enhance the layout of the edges. To show the *Grid* may be used for edge layout as well, a simple proof of concept was implemented. The algorithm looks for a direct horizontal edge if there are elements in the row between the source and the target of the edge. If there are any, the algorithm chooses a different layout for the edge. This could be adapted for other layout types of the edges. But as the focus of the algorithm primarily lies on the positions of the elements, this was not pursued further.

## 4 Example

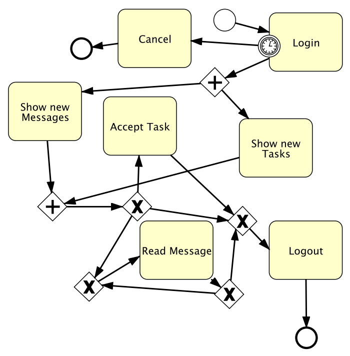

**Figure 8. The BPMN-model before the layouting**

To establish further understanding of the algorithm, an example will now be given. It begins with a manually modeled process as shown in figure 8. It describes an online service into which the user may log in, get shown their new messages and tasks and then may accept a task or read any number of messages and finally log out. The login has a timeout, which leads to a cancellation of the process.

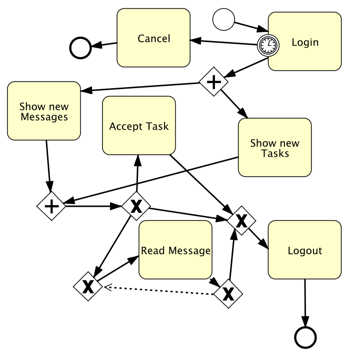

**Figure 9. The BPMN-model after the topological sort**

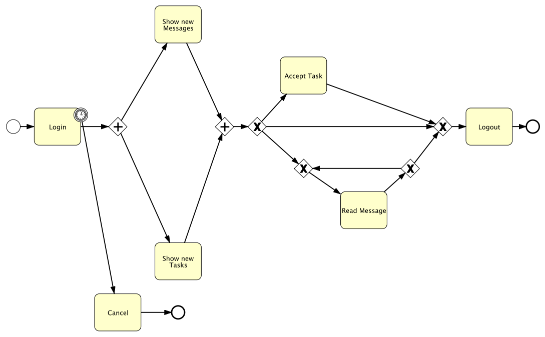

**Figure 11. The BPMN-model after the insertion of all elements into the Grid**

At first, the elements were topologically sorted as shown in section 3.5. In this step, backwards edges are detected and virtually reversed. In the example, the dashed edge in figure 9 is the only virtually reversed egde. As a possible topological sort, the following order is the result: *Start Event, Login, And, Cancel, End Event, Show new Messages, Show new Tasks, And, Xor, Accept Task, Xor, Read Message, Xor, Xor, Logout, End Event* (the docked event is fixed to the login).

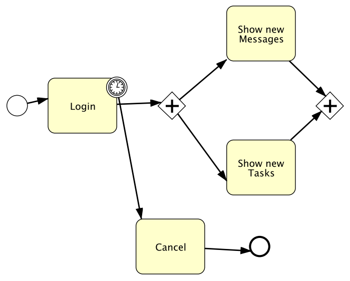

**Figure 10. The BPMN-model during the insertion of elements into the Grid**

Now, the elements are put into the grid. Figure 10 shows the situation after insertion of the second And-Gateway, figure 11 shows the situation after all elements have been put into the grid. It must be pointed out that the edges are not set yet, they are only shown for better overview. It is plain to see how the tasks *Show new Messages* and *Show new Tasks* are pushed outwards by the later branches.

The *Interleaving* (see fig. 12) works against this effect: both tasks and the cancel-branch are pulled more to the middle. Nearly all elements have their final positions in the grid now; only the docked event is missing. The real coordinates are now being calculated from the grid. Finally, the docked event is being aligned with its task.

All preconditions for calculating the edges are now fulfilled. The result is the layout shown in figure 13.

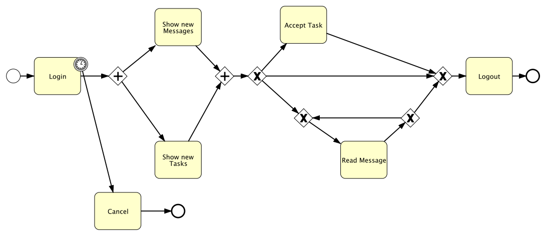

**Figure 12. The BPMN-model after the Interleaving**

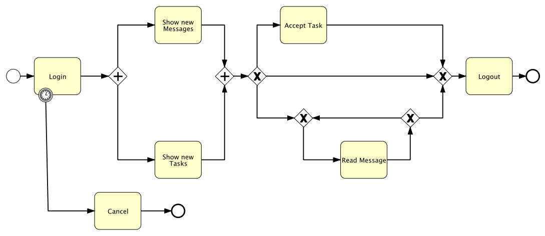

**Figure 13. The final layout of the BPMN-model**

## 5 Implementation

The algorithm was implemented as a small Java tool. It has a simple GUI, which allows to specify the input and output files. But it can also be used via the command line. This permits using it on a server or as part of a tool chain.

To further assist the user, the GUI has special support for the *Oryx Online Repository*[3]. It allows the user to drop public models from the repository into the tool without needing to export, download and open them. This can save a lot of time.

## 6 Runtime Properties

As the runtime of the algorithm depends much on the type of the input model (e.g., whether it contains subprocesses) it is nearly impossible to give formal runtime properties. To get a rough estimation of the runtime, models which fulfilled certain criteria (e.g. a constant ratio of connections to elements) were randomly generated. Then, the algorithm layouted these models, running in the Java SE Runtime Environment version 6 on a computer with an AMD Athlon 64 X2 Dual Core 3800+ CPU, 2 GB of RAM, and Microsoft Windows XP as its operating system.

Models with 100 elements and a total of 150 connections between them were layouted in an average time of 370ms. Models with 500 elements and a total of 750 connections between them needed an average time of 5680ms. Each average time was calculated based on 25 test runs in which 5 different random models were layouted. Thus, the algorithm should be fast enough for reasonably complex models.

## 7 Conclusions & Outlook

Quite good layouts were achieved for BPMN processes with a very simple approach. It was shown that it is possible to layout a BPMN model with a focus on the syntax and not on the semantics of the model elements. Also, it is possible to operate mainly locally on the graph, avoiding much global analysis and complex algorithms. Satisfying solutions to the main challenges of BPMN layouting were found, namely cycles, pools, and lanes.

There are some points where the algorithm could be improved. First, there should be a much more intelligent layout algorithm for the edges. It could avoid calculating edges which run through elements. For this, the *Grid* could be used as in section 3.8.3 or combined with exisiting algorithms for edge layout.

Also, there could be more differentiation between the different types of edges (sequence flows, message flows, data flows). The non-differentiating approach works fine if there is mainly one type. Sometimes, it gives strange results if the types are intensively mixed. One possible variation could be to privilege the most frequent type to dominate the layout.

Lastly, it could be feasible to calculate different variants of a model and have the algorithm choose the best one. Possible variants might differ in the arrangement of pools and lanes and the positions of branches after splits. The biggest challenge here is finding a metric to rate each variant. The number of crossed edges could be a starting point for such a metric. While the algorithm doesn't even consider edges yet and thus treats planar graphs the same as non-planar graphs, the application of such metrics might improve the comprehensibility of the latter significantly.

The next step is integrating the algorithm with the aforementioned *Oryx Editor* in cooperation with the Hasso-Plattner-Institut. This will allow everyone to evaluate the algorithm or download its source code as part of the *Oryx* source.

[3]`http://oryx-editor.org/backend/poem/repository`

## References

[1] T. Biedl and G. Kant. *Algorithms ESA '94*, chapter A better heuristic for orthogonal graph drawings. Springer Berlin / Heidelberg, 1994.

[2] P. Eades, X. Lin, and R. Tamassia. An algorithm for drawing a hierarchical graph. *International Journal of Computational Geometry and Applications*, 1996.

[3] P. Effinger, M. Siebenhaller, and M. Kaufmann. Improving business process visualizations. Technical report, Eberhard-Karls-Universität, Tübingen, 2009.

[4] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software - Practice & Experience*, 21(11):1129–1164, 1991.

[5] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. In *Software-Practice and Experience*, 2000.

[6] Y. Koren. *Computing and Combinatorics*. Springer Berlin / Heidelberg, 2003.

[7] D. Lübke, K. Schneider, and M. Weidlich. Visualizing Use Case Sets as BPMN Processes. In *Proceedings of REV Workshop, co-located at RE 2008, Barcelona, Spain*, 2008.

[8] C. Ouyang, M. Dumas, S. Breutel, and A. H. ter Hofstede. Translating standard process models to bpel. In *CAiSE 2006*, 2006.

[9] R. T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *IRE-AIEE-ACM '59 (Eastern): Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138, New York, NY, USA, 1959. ACM.

[10] M. Song and W. van der Aalst. Towards comprehensive support for organizational mining. In *Decision Support Systems*, 2008.

[11] S. A. White. *Business Process Modeling Notation (BPMN) Version 1.0*. Business Process Management Initiative, BPMI.org, 2004.

View publication stats