Automatic Layout of Scilab/Xcos Diagrams

Chenfeng Zhu[1*], Umut Durak[2], Sven Hartmann[1], Clément David[3]

[1] Clausthal University of Technology, Department of Informatics, Julius-Albert-Straße 4, 38678 Clausthal-Zellerfeld, Germany; * *chenfeng.zhu@tu-clausthal.de*

[2]German Aerospace Center (DLR), Institute of Flight Systems

[3]Scilab Enterprises

SNE 27(3), 2017, 143 - 151, DOI: 10.11128/sne.27.sw.10384
Received: June 8, 2017 (Selected ASIM GMMS/STS 2017 Postconf. Publ.), Accepted: July 20, 2017
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna, ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. Scilab/Xcos is a graphical modeling and simulation environment for hybrid dynamic systems. It provides a graphical editor which allows representing models with block diagrams. While each block represents a computational function, links specify the data and event flow. However, as the number of the blocks and the links increases, the Xcos schema can quickly become messy and difficult to read. In this paper, we present an approach for automatically updating the layout of an Xcos schema by manipulating the links and the split blocks, so that the diagrams can be kept well-presented and readable. In this approach, we update the link styles with an optimal route and then, rearrange the positions of blocks. The proposed approach is exemplified with sample Xcos models. In addition to providing the automatic layout capability to the Scilab/Xcos user, an application programming interface is also specified for the Scilab/Xcos developer who want to further enhance the provided feature set.

Introduction

Scilab is a free and open source software about numerical computation for engineering and scientific applications [1]. Xcos is the graphical modeling environment of Scilab for modeling and simulation of hybrid dynamical systems [2].

When using Xcos to create models, we often create blocks to implement computational functions and use links to connect them for data and event flow. They are all well-organized at the beginning as we start building up the model.

However, as the model becomes more and more complex, and the number of blocks increases, we require layout rearrangements more frequently; we start moving the blocks or links every now and then. But manual layouting is hard, labor intensive and error prone. Such an effort usually ends up with a model in a messed up layout, which makes the diagram ugly and difficult for modification. Thus the readability and the maintainability of the model is decreased. Figure 1 is an example of a disordered Xcos diagram.

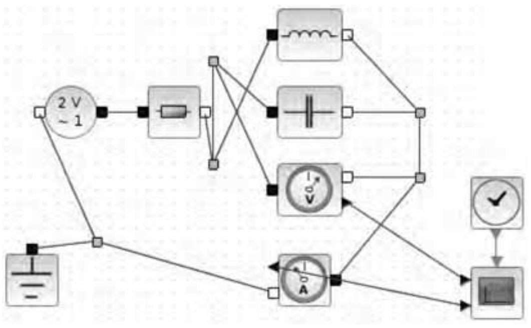

Figure 1: A Sample Xcos Model.

A constant manual work for relocating the blocks and rearranging the links between blocks is required in order to maintain the model readability. While block positioning is relatively straight forward and even may be more efficient manually, the link rearrangements are hard and cumbersome. This effort is about developing the capability for automatically improving the model layout by manipulating the links and split blocks that connect the links to each other, thereby keeping the model readable. The *Optimal Link Style (OLS)* that is introduced in the Section 3 proposes an optimal route for a link which could make the link clear in the diagram. In the Section 4, the *Split Block Automatic* Position (SBAP) that rearranges split blocks in better positions is presented. Lastly in Section 5, the conclusion is presented and we discuss future work for a better automatic Xcos layouting.

1 Related Work

1.1 Graph Theory

For finding the optimal link between two blocks, one can apply a graph search based approach. The approaches from the graph theory to find the shortest route with the minimum cost are already quite mature. The Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later [3, 4]. The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph [5]. The Floyd–Warshall algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles) [6, 7]. These theories are practical and useful for solving the shortest path problem which could also be extended to solve the minimum-cost problem. However, in modeling graphical model, there are always more than one alternative for the connection which look good. And sometimes, some links which look good and readable are not the shortest ones.

As to the problem about the positions of split blocks, we could use some basic graph drawing theories to reorder blocks. For instance, pseudo hierarchical tree and vertical or horizontal aligned layout would be helpful to make diagrams easy to read and clear to maintain. But for both cases, our approach to the problem was to develop heuristics that capture user insight for readability.

1.2 Layouting in MATLAB/Simulink

Other graphical modeling environments such as MATLAB/Simulink also provide capabilities for formatting the layout of their own diagrams. Simulink is also a graphical editor for Model-Based Design which provides customizable block libraries, and solvers for modeling and simulating dynamic systems [8]. Compared to the automatic layouting of other graphical modeling environments, Simulink achieves quite an outstanding work about this. It not only provides the beautiful layout, but also could implement the dynamic features. Simulink can automatically find the 'optimal path' so that the new signal line is as short as possible, has minimal 90 degree turns, and does not overlap other blocks and text.

Moreover, as you draw the signal line, Simulink lets you know exactly what the path is going to look like before you release the mouse button [9]. Additionally, Simulink provides one-click to beautify the model diagrams and autoarrangement the blocks and lines when building new functionalities [10].

1.3 Layouting in Scilab/Xcos

In fact, even in Scilab/Xcos, it is possible to improve the general look of a diagram in using the blocks alignment options and the links style [11]. Besides straight style and the free style with control points, there are only other 2 types of link styles provided for auto layout: vertical and horizontal.

When the diagram becomes too complicate, the results are obviously unsatisfying. So, the effort presented in this paper targets at rearranging the blocks by putting the blocks in some new reasonable positions and to find optimal routes for the connection. After this automated process, the layout of a diagram should be enhanced and beautified for readability and maintainability.

2 Technical Solution

2.1 Overview on Scilab/Xcos

Scilab/Xcos palette provides varieties of predefined blocks such as signal processing, mathematical operations and discrete and continuous system blocks while it is also possible to develop user-defined blocks. Despite of different types of blocks, when it comes to representation, they all belong to BasicBlock. We can abstract an Xcos diagram as shown in Figure 2.

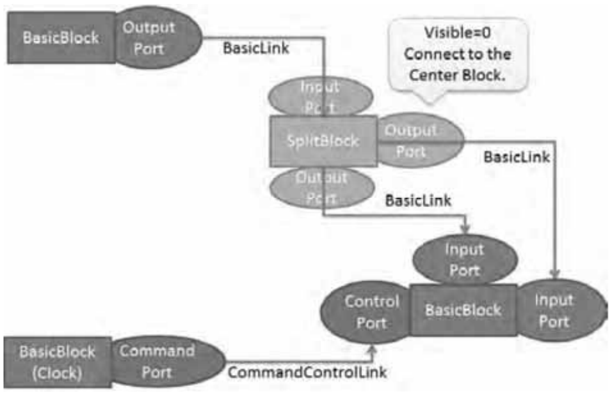

Figure 2: Basic Structure of Xcos Diagram.

Normally, every block from palette is a BasicBlock. Every block owns its port(s) of input or output as its children (BasicPort) which belongs to four types: Input-Port (subclasses: ExplicitInputPort or ImplicitInputPort) or OutputPort (subclass: ExplicitOutputPort or ImplicitOutputPort), ControlPort or CommandPort. Ports can be connected with links (BasicLink). ExplicitLink/ImplicitLink can be used to connect Input-Port and OutPutPort, and CommandControlLink can only be used to connect CommandPort and Control-Port. And there are also other classes implementing other functionality such as graph, palette and utilities.

All the Java codes of the Xcos program are in the 'org.scilab.modules.xcos' package. The structure is shown in Figure 3.

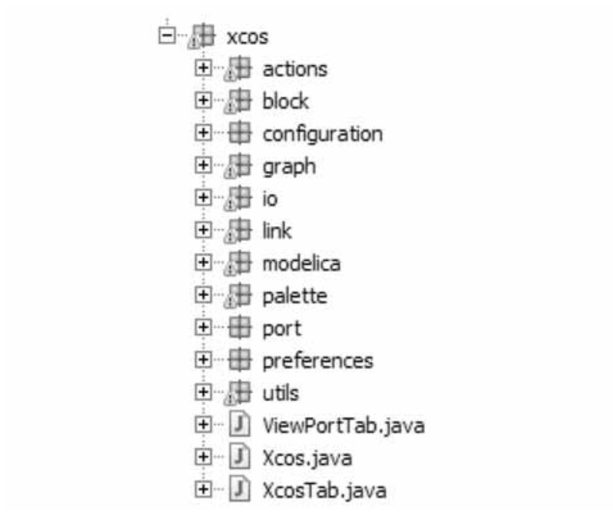
```
xcos
  actions
  block
  configuration
  graph
  io
  link
  modelica
  palette
  port
  preferences
  utils
  ViewPortTab.java
  Xcos.java
  XcosTab.java
```

Figure 3: Structure of Xcos Java program.

Scilab/Xcos code base consists of various types of files, such as C codes, Java codes, xml files for help documents, image resources and files for locales. The user interface is generated by using Java. So, in order to implement an automatic layouting for Xcos, we need to conduct the implementation in Java.

2.2 JGraphX

JGraphX [12] is the underlying graphics framework of Scilab/Xcos. Aligned with that, the implementation of the technical solution encompasses utilization of JGraphX for the development of autmatic layouting features. JGraphX is a Java Swing diagramming (graph visualization) library licensed under the BSD license. The library is strong and easy to extend and inherit.

The documentation and *Application Programming Interface* (API) is quite mature. JGraphX not only provides functionality for visualization and interaction with node-edge graphs, but also includes functionality like XML support which would help save the current layout of the diagram avoiding that the layout needs to be recalculated every time it is opened. Besides the features about graph interaction and graph layouts which are being used in Xcos, JGraphX provides an analysis package which includes a range of analysis functions which provided us with a number efficient building blocks for the automatic layouting.

The core architecture of JGraphX includes the JGraphX model, the transactional model and mxCell. The JGraphX model (mxGraph) is the core model that describes the structure of the graph. The class called mxGraphModel is the underlying object that stores the data structure of the graph [12]. The graph class ScilabGraph in Scilab extends mxGraph. The transactional model is a transaction of models update which contains a series of actions. Transaction starts with beginUpdate and ends with endUpdate. With the help of transactional model, a set of events for the compound changes could be fired together after transaction. The mxCell is the cell object for both vertices and edges [12]. The three key attributes for an mxCell is its value, its style and its geometry. The ScilabGraphUniqueObject extends mxCell and it is also the ancestor of BasicBlock and BasicLink.

We use the geometry to change the position of blocks and use the style to change the routes of links in Xcos. And we need to save all the states of Xcos diagram including the positions of blocks and the styles of links so that we do not need to re-calculate the layout every time re-opening an Xcos file.

3 Optimal Link Style

Optimal Link Style (OLS) is to find a route with more blank padding and with less turning and to use it as the style of a link.

3.1 The Functional Flow

The design of the functional flow can be briefed as below:

1. Change the style of the selected links one by one in a loop in one transaction.

2. Check whether the two points of the ports are aligned and make sure that there are no blocks between them. If so, make the link with straight style, e.g. connect them directly.
3. Otherwise, use two new points each of which is a distance away from its corresponding port (if it was SplitBlock, use its center directly instead of its port).
4. Then start with these two new points, try to find the new route with one single turning point or 2 turning points. Otherwise, get another new point away from the starting point and use this to find a route with the same method. This could be re-cursed in several times.
5. Remove the unnecessary points and get the final optimal route for the link.

We modify the source files and create the classes listed in Table 1 to implement the functionality:

Class	Description
StyleOptimalAction	Action events
XcosRoute	Compute route
XcosRouteUtils	Common utilities

Table 1: Classes Created.

3.2 The Methods

Here, we would like to introduce the methods and and underlying mathematical model for OLS. The method signatures are declared at the beginning of each section. Thus, the reader is informed about the application programming interface for that particular method.

Get the position of a cell

Method:
`mxPoint getCenterPoint(mxICell cell, XcosDiagram graph)`
This method is used to get the position of a cell where a link will connect to. There are three situations according to this cell:

- If it is a Port and its parent is a `SplitBlock`, use the center point of its parent.
- If it is a Port and its parent is not a SplitBlock, use the state of this cell `(graph.getView().getState(cell))` to get port's mxCellState to get the point.
- If it is a BasicBlock, use the center point according to its geometry attribute.

Check if a point is in a line segment

Method:
`boolean pointInLineSegment(double x1, double y1, double x2, double y2, double x3, double y3)`
This method is used to check P_1 in Segment (P_2, P_3) which is shown in Figure 4.

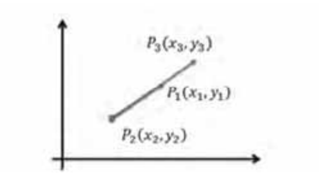

Figure 4: Point in Line Segment.

If it is, then P_2P_1 and P_2P_3 must have the same direction and P_1 is between P_2 and P_3.

For the same direction:

$$\angle\overrightarrow{P_2P_1} = \angle\overrightarrow{P_2P_3} \tag{1}$$

$$\frac{y_1 - y_2}{x_1 - x_2} = \frac{y_3 - y_2}{x_3 - x_2} \tag{2}$$

$$(y_1 - y_2)(x_3 - x_2) = (y_3 - y_2)(x_1 - x_2) \tag{3}$$

For P_1 between two points:

$$\min(x_2, x_3) \leq x_1 \leq \max(x_2, x_3) \tag{4}$$

$$\min(y_2, y_3) \leq y_1 \leq \max(y_2, y_3) \tag{5}$$

Check superimposition

Method:
`boolean linesCoincide(double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4)`
These two methods are used to check whether two lines coincide or not. The second one is to check strict superimposition of two line segments. In the first one, the lines would move parallel and then check all of them. This could avoid that two line segments are be too close.

Segment A (P_1, P_2) and Segment B (P_3, P_4) will coincide in these situations:

- Segment A is inside Segment B, e.g. both P1, and P2 are in Segment B;
- Segment B is inside Segment A, e.g. both P3, and P4 are in Segment B;
- Segment A and Segment B are parallel and one of the endpoints of one segment is in the other segment.

If lines are parallel,

$$(x_1 - x_2)(y_3 - y_4) = (x_3 - x_4)(y_1 - y_2) \tag{6}$$

Check obstacles

Method:
```
boolean checkObstacle(double x1, double y1,
double x2, double y2, Object[] allCells)
```
This method is used to detect whether there are obstacles between two points.

The definition of obstacles is: All top blocks and links and all the ports of blocks. EXCEPT: itself (link), its Source and Target (i.e. port) and SplitBlock.

If any of the below situations happens, it means that there is an obstacle between two points:

- If it is a Link,
  - Check lines superimposition.
  - Check whether points are in the link.
- If it is a Block,
  - Use mxRectangle.intersectLine to get an intersection if it exists according to its geometry.

Get orientation of ports

Method:
```
Orientation getPortRelativeOrientation
(BasicPort port, XcosDiagram graph) Orienta-
tion getNewOrien-tation(mxICell cell, double
cx, double cy, mxICell oth-erCell, double
ox, double oy, XcosDiagram graph)
```
These two methods are used to get the current orientation of a port according to its relative position to parent block.

We also consider the ports of different blocks, because the ports of a SplitBlock are not visible or the target point has no parent blocks.

- If its parent is a normal Block, calculate orientation according to the relative position of the port to its parent block as Figure 5 shows. For instance, the orientation of the port will be EAST if this port in the EAST zone of its parent block.
- If its parent is a SplitBlock, get the orientation of the InputPort of the SplitBlock according to the relative position of link's source (it is the same mathematical model in the first case); get the orientation of one OutputPort of the SplitBlock according to the orientation of the InputPort and the positions of both OutPorts. For instance, when one of the OUT target is on NORTHEAST to IN source, its orientation will be NORTH if the other out target is on its right side and the orientation of IN is not north; its orientation will be EAST if the other out target is on its left side.

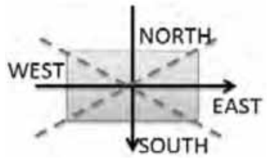

Figure 5: Orientation in Zones.

Get a point away from port

Method:
```
mxPoint getPointAwayPort(mxICell port, dou-
ble portX, double portY, Orientation orien,
Object[] allCells, XcosDiagram graph)
```
This method is used to get a new point away from a port according to the orientation of this Port as Figure 6 shows. If there are obstacles between the Port and the new point, reduce the distance and try another new point. Then use the new Point as the start/end point to compute the route.

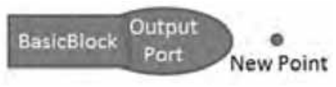

Figure 6: Get a New Point away from Port.

Choose an optimal line

Method:
```
double choosePoint(List<Double> list, double
p1, double p2)
```
This method is used to choose a better line (which is the average number in the widest range in a certain density) from the discrete numbers as Figure 7 shows. Consider the points between p1 and p2 as a priority as Figure 8 shows.

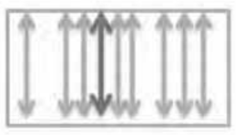

Figure 7: Choose an Optimal Line I.

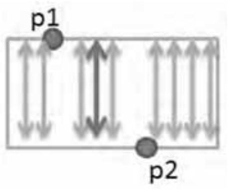

Figure 8: Choose an Optimal Line II.

3.3 Implementation in detail

Firstly, we get the position of the source and the target cell. If two points are aligned `(boolean isStrictlyAligned(double, double, double, double))` and there are no obstacles between them, then we connect two points directly and we do not need to do the steps further.

Then, we create a point away from the port according to the orientation of each port. Using these two new points as the new starting point and the ending point, find a simple route with 2 turning points. If the source is EAST/WEST orientation, we try the point(x2, y1) as the turning point and check the obstacles among the new source point, this point and the new target point. In this case, the away point for the source is unnecessary. Otherwise, we try point(x1, y2) and check the obstacles among the new source point, this point and the new target point. In this case, the away point for the source is necessary. The away point for the target is similar. This is shown in Figure 9. If the source is SOUTH/NORTH orientation, we try the point(x1, y2) and check the obstacles among them. In this case, the away point for the source is necessary. Try the point(x2, y1) and check the obstacles among them. In this case, the away point for the source is unnecessary. The away point for the target is also similar. This is shown in Figure 10. If we could get a route, we do not need to do the steps further.

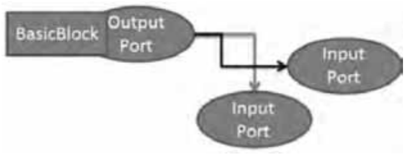

Figure 9: Single Turning I.

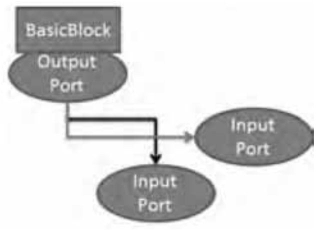

Figure 10: Single Turning II.

At the third step, we check all the possible horizontal or vertical connections of the points whose y or x is between the two points as Figure 11 shows ('possible' means no obstacles between points). In case that it is full of obstacles between two blocks, we extend the range of the detection.

If the orientation is horizontal, we check x firstly (the left one in the figure). Otherwise, check y firstly (the right one in the figure).

Finally, if there is no optimal solution in simple mode which is introduced above, we get new away points of the start point in 3 directions and use the new points to find a simple route. Otherwise, we try to find a complex route in a recursion.

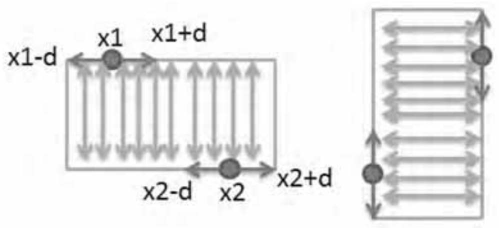

Figure 11: Simple Routes.

3.4 A sample application

In this example case, Figure 12 is the original diagram in a mess. Figure 13 is the diagram which we used OLS to format the links.

It looks better than previous layout and the links were clear for users to read. But it does not work well when there are split blocks. So we need to do some optimizations to make it more beautiful.

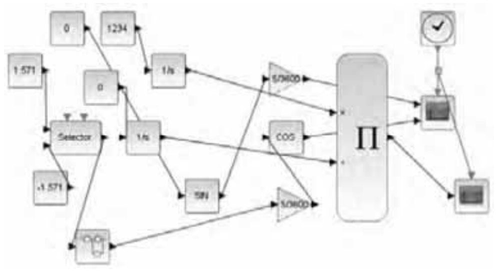

Figure 12: The Diagram in Original Version.

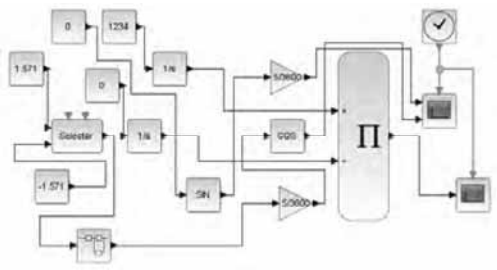

Figure 13: The Diagram after Using OLS.

4 Split Block Auto Position

Split Block Auto Position (SBAP) is to find a position for split block where the links which connect to it look clearer in the optimal routes.

4.1 The Functional Flow

The design of functional flow can be introduced as below:

1. In the whole connection where the split block is, get one of the normal blocks as the source and all other blocks as the targets.
2. Compute their optimal routes separately.
3. Choose the conjunct point of the routes to be the new position of every split block.
4. Update the orientations of ports in the split block according to the routes.
5. After getting new position(s) and new orientations, update the links.

We modified the source files and created the classes listed in Table 2 to implement this functionality:

Class	Description
AutoPositionSplitBlockAction	Action events
BlockAutoPositonUtils	Calculate position

Table 2: Classes Created.

4.2 The methods

We will explain the methodology which is used for *Split Block Auto Position*. In this section, the methods and and underlying mathematical model for SBAP will be presented. As it was for the OSL, the method signatures are declared at the beginning of each subsection in order to reveal the application programming interface for that particular method.

Get the root split block

Method:
```
SplitBlock getRootSplitBlock(SplitBlock
splitblock)
```
This method is used to get the root split block when there are multiple split blocks in the whole connection.

1. Check if the block which connects to the IN port of this split block is a normal block.
2. If it is a normal block, this split block is the root split block.
3. If it is a split block, then check as step 1 again and start this loop until find the normal block. Then the split block is the root one.

Adjust routes

Method:
```
void adjustRoutes(List<List<mxPoint»
listRoutes, Object[] allObstacles,
List<mxICell> listPorts)
```
This method is used to adjust routes after getting the optimal routes (using OLS). As shown in Figure 14, some segments in two links might be parallel. We move segments to make them superimposed if there are no obstacles. Then there will be more superimpositions between routes and the last conjunct point of routes will be more meaningful.

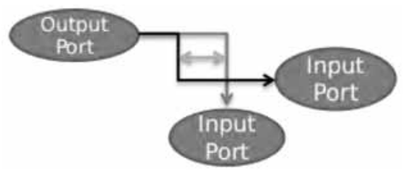

Figure 14: Adjusting Routes.

Get the last conjunct point

Method:
```
mxPoint getSplitPoint(List<List<mxPoint»
listRoutes)
```
This method is used to get the last conjunct point of all routes. After the routes are adjusted, there are different last conjunct points between every 2 routes. We choose the one which is in all routes.

Update orientation of port

Method:
```
void updatePortOrientation(SplitBlock split,
List<List<mxPoint» listRoutes, XcosDiagram
graph, BasicPort input)
Orientation getInputOrienttion(List<List
<mxPoint» list, mxPoint startPoint, mxPoint
splitPoint) Orientation getPortOrienta-
tion(List<mxPoint> list, mxPoint splitPoint)
```
These methods are used to get the orientation of ports in a split block according to the relative routes and update them. There are routes passing the split block. As shown in Figure 15,

1. For the IN port, from the split block point to the previous turning point is the orientation.
2. For the OUT port, there are 2 cases. If the split block is in the turning point, then from the turning point to next turning point is the orientation. Otherwise, from the previous turning point to the next turning point is the orientation.

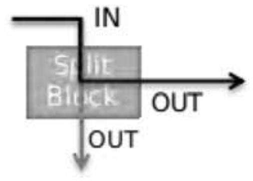

Figure 15: Get and Update Orientation.

4.3 Implementation in detail

When a link is split, there will be one split block generated. When the link is split several times, there will be several split blocks. Then one split block must have one IN port and two OUT ports.

1. Calculate the number of split blocks in this whole part of linking.
2. If there is only one split block,
   - Get the port which connects to the IN port of the split block as a source. And get the ports which connect to the two OUT ports of the split block as targets.
   - Find the optimal routes for the source to each target.
   - Use the last conjunct point in both routes as the new position of split block as shown in Figure 16
   - Find the orientations for each ports in the split block according to optimal routes.
3. If there are more than one split blocks,
   - Get the root split block. Get the port which connects to the IN port of this root split block as a source.
   - And get the ports of all basic blocks which connect to all other children split block as targets.
   - Find the optimal routes for the source to each target.
   - Use the last conjunct point in different routes as the new position of split blocks.
   - Find the orientations for each ports in the split blocks according to optimal routes.
4. After getting new position(s) and new orientations, update the links.

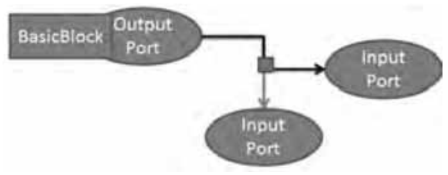

Figure 16: New Position of SplitBlock.

4.4 A Sample Application

In the case study, Figure 1 is the original diagram. Figure 17 is the diagram which we use SBAP and OLS to format.

Now, the layout is much better. The diagram is easy to read and maintain.

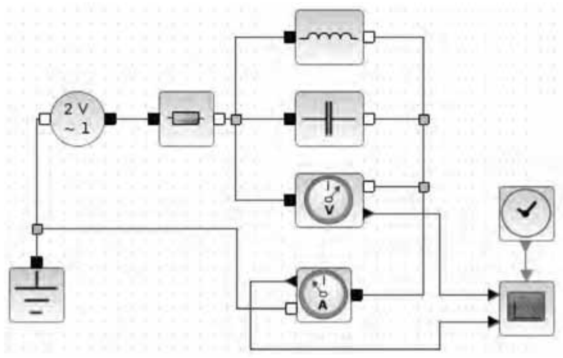

Figure 17: New Position of SplitBlock.

5 Conclusion and Future Work

The paper presents *Optimal Link Style* and *Split Block Auto Position* for automatic layouting Scilab/Xcos diagrams. *Optimal Link Style* focuses on the link styles which format the link, while *Split Block Auto Position* concentrates mainly on position changing of split blocks. According to his needs, user can decide in modeling time which one to apply.

Every project or every team has its own standards or criteria about the format of the diagrams. It is not easy to decide which layout is really perfect or fulfills the needs of users. What we have tried to achieve was not only providing an automatic layout, but also giving some options for users to make their own decisions about the final layout which they would like to maintain. On the other hand, user therefore needs to select the blocks or the links which he wants to change and click some buttons to make them in an optimal layout. That means that our automatic layout is still static instead of dynamic. And we could not get a preview of the automatic layout while we are drawing our diagrams or get a direct result about this automatic feature.

Sometimes, users would like to draw a link with the optimal route when creating the connection between blocks. They would also like to format the layout of the links after they move some blocks without clicking some buttons.

Based on the previous paragraph, the future work includes improving the user experience. While the application programming interface for the layouting methods provides a baseline, it is necessary to experiment various user interaction scenarios.

References

[1] Champbell SL, Chancelier JP, Nikoukhah R. Modeling and Simulation in Scilab/Scicos. Springer Science and Business Media, Inc., 2006.

[2] Scilab Enterprise, Xcos Features. Retrieved July 30, 2016 from www.scilab.org/scilab/features/xcos.

[3] Frana P. An Interview with Edsger W. Dijkstra. Communications of the ACM 53 (8): 41–47, 2010.

[4] Dijkstra EW. A note on two problems in con nexion with graphs. Numerische Mathematik 1: 269–271, 1959.

[5] Bang-Jensen J, Gutin G. The Bellman- Ford-Moore algorithm. Digraphs: Theory, Algorithms and Applications, Springer Science and Business Media, 2000.

[6] Cormen TH, Leiserson CE, Rivest RL. Introduction to Algorithms (1st ed.). MIT Press and McGraw-Hill, 1990.

[7] Rosen KH. Discrete Mathematics and Its Applications, 5th Edition. Addison Wesley.2003.

[8] MathWorks, Simulation and Model-Based Design. Retrieved July 30, 2016 from http://www.mathworks.com/products/simulink/.

[9] Roulaeu G. Smart Signal Routing. Retrieved July 30, 2016 from http://blogs.mathworks.com/simulink/2012/10/11/smart-signal-routing/.

[10] Klauske LK, Dziobek C. Improving Modeling Usability: Automatic Layouting for Simulink. Retrieved July 30, 2016 from http://www.mathworks.com/videos/improving-modeling-usability-automatic-layouting-for-simulink-93139.html.

[11] Scilab Enterprise. Xcos for Very Beginners. Retrieved July 30, 2016 from http://www.scilab.org/community/news/20130830.

[12] JJGraph Ltd. GraphX (JGraph 6) User Manual. Retrieved July 30, 2016 from https://jgraph.github.io/.