AUTOMATED LAYOUT OF PROCESS DESCRIPTION MAPS DRAWN IN SYSTEMS BIOLOGY GRAPHICAL NOTATION

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Begüm Genç

July, 2014

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Uğur Doğrusöz(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Öznur Taştan

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Özgür Şahin

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

AUTOMATED LAYOUT OF PROCESS DESCRIPTION MAPS DRAWN IN SYSTEMS BIOLOGY GRAPHICAL NOTATION

Begüm Genç
M.S. in Computer Engineering
Supervisor: Assoc. Prof. Dr. Uğur Doğrusöz
July, 2014

Evolving technology has increased the focus on genomics. The combination of today's advanced studies with decades of molecular biology research yield in huge amount of pathway data. These models can be used to improve high-throughput data analysis by linking correlation to the causation, shedding light on many complex diseases. In order to prevent ambiguity and ensure regularity of the research, a need for using a standard notation has emerged.

Systems Biology Graphical Notation (SBGN) is a visual language developed by a community of biochemists, modellers and computer scientists with the intention of enabling scientists to represent networks, including models of cellular processes, in a standard, unambiguous way. SBGN is formed of three languages: process, entity relationship and activity flow. This research is focused on its process diagram branch.

Automated layout is commonly used to clearly visualize the information represented by graphs. Considering the fact that, biological pathways includes nested structures (e.g., nucleoplasms), we have made use of a force-directed automatic layout algorithm called *Compound Spring Embedder (CoSE)*, which supports the compound graph structures. On top of this layout structure, we have developed a specialized layout algorithm called SBGN-PD layout.

SBGN-PD layout enhancements mainly include properly tiling of complex members and disconnected molecules, placement of product and substrate edges on the opposite sides of a process node without disturbing the force-directed structure of the algorithm.

Keywords: information visualization, graph layout, automatic layout, biological networks, biological process diagrams, graph algorithms, graph visualization, systems biology graphical notation, compound spring embedder.

ÖZET

SYSTEMS BIOLOGY GRAPHICAL NOTATION KULLANILARAK ÇİZİLEN PROSES DİYAGRAMLARININ OTOMATİK YERLEŞTİRMESİ

Begüm Genç
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Assoc. Prof. Dr. Uğur Doğrusöz
Temmuz, 2014

Gelişen teknolojiyle birlikte genom bilimi üzerindeki ilgi arttı. Günümüzdeki ileri düzey çalışmalar ve yüzyıllardır süregelen moleküler biyoloji araştırmaları birleşince yüksek miktarda yolak verisi oluştu. Bu modeller, bağıntıları nedenlere bağlayıp, yüksek hacimli veri analizinde kullanılarak, bir çok karmaşık hastalığı aydınlatabilir. Bu çalışmalardaki anlam belirsizliğini önlemek ve düzenliliği sağlamak amacıyla standart bir notasyona ihtiyaç duyuldu.

Systems Biology Graphical Notation (SBGN), bir grup biyokimyacı, modelleme uzmanı ve bilgisayar bilimcisi tarafından, bilim adamlarının hücresel süreçler de dahil olmak üzere pek çok ağ yapısını stadart ve belirsizliğe yer vermeyen bir şekilde yansıtması için geliştirilmiştir. SBGN üç dilden oluşur: proses, varlık ilişki ve etkinlik akış. Bu araştırma, bu dilin proses diyagramları kolu üzerine yoğunlaşmaktadır.

Otomatik yerleştirme, çizgelerle temsil edilen bilginin açıkça görselleştirilmesini sağlamakta yaygın şekilde kullanılır. Biyolojik yolakların içiçe yapılar (nükleoplazma vb.) içerdiğini göz önünde bulundurarak, bu yapıları destekleyen kuvvet-yönlendirilmiş bir otomatik yerleştirme metodu olan *Compound Spring Embedder (CoSE)* algoritmasını kullandık. Bu algoritma yapısını kullanarak, özelleştirilmiş bir SBGN-PD yerleştirme algoritması geliştirdik.

SBGN-PD yerleştirme algoritması genel olarak; kompleks üyelerinin ve bağlantısı olmayan moleküllerin düzgünce döşenmesi, proses düğümlerinin substrat ve ürün bağlantılarının, ilgili proses düğümlerinin zıt taraflarına yerleştirilmesi ve bu işlemlerin algoritmanın kuvvet-yönlendirilmiş yapısını bozmadan yapılmasını içerir.

Anahtar sözcükler: bilgi görselleştirme, çizge yerleştirme, otomatik yerleştirme, biyolojik ağ diyagramları, biyolojik proses diyagramları, çizge algoritmaları, çizge görselleştirme.

Acknowledgement

I would like to express my gratitude and thanks to my supervisor Uğur Doğrusöz for the understanding, patience and expertise through the learning process of this master thesis. His supports and advices are greatly appreciated.

Besides my supervisor, I would like to thank to the rest of my thesis committee: Asst. Prof. Dr. Öznur Taştan and Asst. Prof. Dr. Özgür Şahin for reviewing my work and the questions.

Furthermore, I would like to thank to my friends Mecit Sarı, Doğukan Çağatay, Merve Çakır, İstemi Rahman Bahçeci, Can Çağdaş Cengiz and Muhsin Can Orhan for the valuable friendship and coffee breaks.

Of course I would like to thank my mom and dad for unconditional support throughout my degree, Selen for being such an understanding sister and Arya for being a charming niece. A very special thanks goes to my brother Burkay Genç, without whose encouragement and assistance, I would have not finished this thesis.

Finally, I would like to thank TÜBİTAK for making this research possible by providing financial assistance.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 5

2 Background and Related Work **7**

2.1 Graphs 7

2.2 Systems Biology Graphical Notation 9

2.3 Automated Layout 11

2.4 Compound Spring Embedder 14

2.4.1 CoSE Layout Structure 18

2.5 Rectangle Packing 21

2.5.1 Tiling 23

2.5.2 Polyomino Packing 24

2.6 Compaction 26

3 SBGN-PD Layout **28**

3.1 Nested Rectangle Packing 31

3.1.1 Packing Complex Members 33

3.1.2 Packing Non-Complex Members 37

3.1.3 Performing Compaction 40

3.2 Handling Port Nodes 42

3.3 Orienting Products and Substrates 45

4 Results **50**

4.1 Evaluation 62

5 Conclusion **72**

List of Figures

1.1 "ATM mediated phosphorylation of repair proteins" biological network shown in Pathway Commons [1] 2

1.2 Visualized version of ATM mediated phosphorylation of repair proteins biological network shown in Fig. 1.1, using VISIBIOweb [2] . 2

1.3 Random node positioning (left) and automatically laid out (right) versions of a biological network that respects cell compartments. . 3

1.4 Systems Biology Graphical Notation (SBGN) reference card [3] . . 4

1.5 Result of an automated layout algorithm applied on a biological process diagram 6

1.6 Proper SBGN layout of the biological diagram shown in Fig. 1.5 . 6

2.1 Sample compound graph created randomly using Chisio graph editor [4]. 8

2.2 IFN regulation, a gene regulatory network. 9

2.3 From left to right: complex between phosphorylated cyclin and CDK2 forming the maturation promoting factor in yeast, a tetramer of complexes between globin and heme, and a complex between calcium-calmodulin kinase II and another complex, itself formed of calmodulin and calcium [5]. 10

2.4 Reaction between ATP and fructose-6-phosphate to produce fructose-1,6-biphosphate, ADP and a proton [5]. 11

2.5 Illustration of a generic force-directed method: the graph (top-left) represented as springs and particles (top-right), reached stable state (bottom-right), corresponding graph (bottom-left) [6]. 13

2.6 A sample compound graph (left) and its corresponding model in CoSE (right) [7] 15

2.7 Total area is divided into 3: A (rectangular objects), B (wasted area) and C (additional wasted area if aspect ratio is specified as 2). 21

2.8 A sample disconnected graph layout given the aspect ratio, with and without packing, respectively. 22

2.9 A sample result of tiling algorithm, where the numbering indicates the order of processing. 24

2.10 Polyomino constructed from a simple graph (left) and its bounding box (right). 24

2.11 Packing results of a arbitrarily shaped object set using two methods: tiling (left) and polyomino packing (right) [8]. 25

2.12 A simple polygon (left) with no obstacles and the visibility graph edges (right) 26

2.13 A set of objects (top), its horizontal visibility graph (bottom-left) and vertical visibility graph (bottom-right), with no directions specified. 27

3.1 CoSE fails to separate substrate, production and effector nodes (left), desired result (right). 29

3.2 A disconnected graph laid out using CoSE (top) and SBGN with tiling (bottom). Disconnected molecules are shown using red, where complexes are represented by octagonal boundaries. 32

3.3 Polyomino representation and the polyomino cells of two objects. 34

3.4 Polyomino packing result on a sample graph. 36

3.5 A dummy complex is created to group each disconnected molecules at each depth (large margins for dummy complexes are used to get a better understanding). 38

3.6 Two sample dummy complexes that have non-zero degree child nodes. 39

3.7 Vertical visibilities of a node set (top-left), vertical compaction result (top-right), horizontal visibilities (bottom-right) and result of horizontal compaction (bottom-left). 42

3.8 Process node handling in CoSE (left), how SBGN defines it to be (right) 43

3.9 Illustration of a process node (orange) using dummy compound (yellow), port nodes (green) and rigid edges (red) in SBGN-PD layout. 43

3.10 Rotational force acting on a sample process node. 46

3.11 A sample layout, where net rotational force fails to detect 90-degree rotation. 47

3.12 Reference card for rotational force signs for four possible orientations: left-to-right, right-to-left, top-to-bottom, and bottom-to-top. 48

4.1 Comparison of tiling, polyomino packing without compaction and polyomino packing with compaction 51

4.2 Percentage of properly oriented edge count to total number of edges to be oriented with different value sets. 53

4.3 Effect of approximation distance on results. 54

4.4 Effect of rotation period on results. 55

4.5 Effect of angle tolerance on results. 55

4.6 Effect of 90-degree rotation constant on results. 56

4.7 Effect of 180-degree rotation constant on results. 57

4.8 Effect of approximation period on results. 57

4.9 Effect of phase 1 maximum iteration count on results. 58

4.10 Comparison of CoSE and SBGN-PD average results. 58

4.11 Comparison of CoSE and SBGN-PD average execution times. . . 59

4.12 Effect of (N+E) on the average result for non-compound graphs. . 60

4.13 Effect of (N+E) on average execution time for non-compound graphs. 60

4.14 Effect of (N+E) on the average result for compound graphs. . . . 61

4.15 Effect of total number of nodes and edges on the average execution time for compound graphs. 62

4.16 Comparison of polyomino packing (left) and tiling (right). 63

4.17 Btg family proteins and cell cycle regulation network laid out using SBGN-PD. 64

4.18 Vitamins B6 activation to pyridoxal phosphate network laid out using SBGN-PD. 65

4.19 Process between ATRIP and CHEK1 laid out using SBGN-PD. . 66

4.20 5HT1 type receptor mediated signalling pathway laid out using SBGN-PD. 67

4.21 Aspirin blocks signalling pathway involved in platelet activation laid out using SBGN-PD. 68

4.22 Process between ATM and RBBP8 laid out using SBGN-PD. . . . 69

4.23 Carm1 and regulation of the estrogen receptor pathway laid out using SBGN-PD. 70

4.24 Tie2 signaling pathway laid out using SBGN-PD. 71

Chapter 1

Introduction

1.1 Motivation

One of the most famous idioms states that: "A picture is worth a thousand words". That phrase lays the foundation of information visualization. Fig. 1.1 and Fig. 1.2 illustrate the fact that it is easier to interpret data in visual format. Since the beginning of mankind, visualization has been effectively used. Even the ancient people were drawing hieroglyphs and diagrams on the cave walls to express themselves. It is stated that, the diagrams directly address people's innate cognitive abilities [9]. Due to the fact that symbols, diagrams and other graphical representations spread all over the world, it is necessary to have a common interpretation. Therefore, standard notations play an important role in communication and rapid development of many research areas. For instance, in the field of electronics, people use a common circuit design notation, which fastens the development and research quality.

Going back a few years ago, biologists did not have a common language for representation of their research results [10]. Studies included intensive usage of graphical elements but there were poor similarities between them when compared. There were no standard notation to visualize biochemical reaction pathways, signal transduction on the cell surface or inside the cell and genome regulation

Figure 1.1: "ATM mediated phosphorylation of repair proteins" biological network shown in Pathway Commons [1]

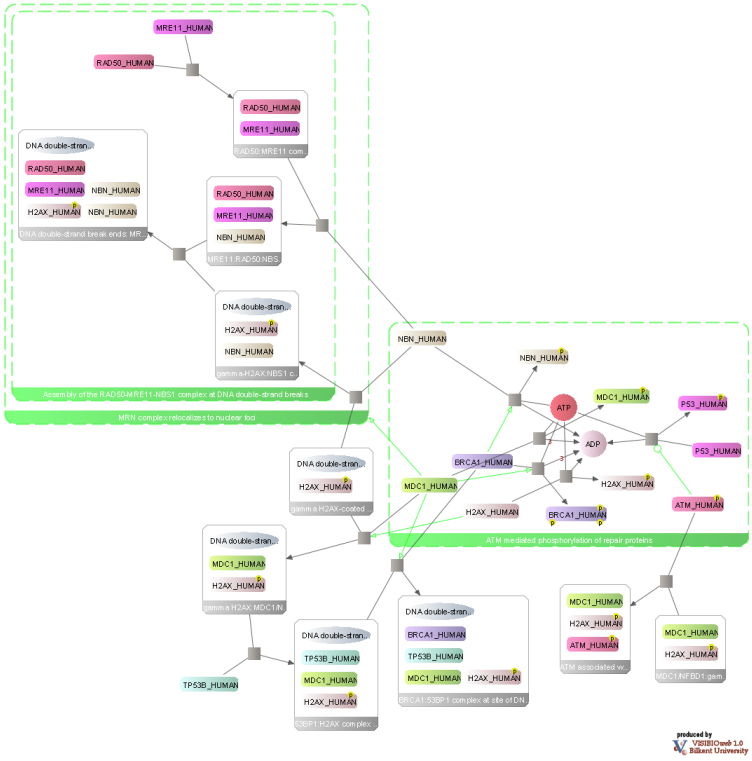

Figure 1.2: Visualized version of ATM mediated phosphorylation of repair proteins biological network shown in Fig. 1.1, using VISIBIOweb [2]

diagrams. Already present notations were fulfilling the needs of some groups of researcher but when used by others, resulting in ambiguity [11].

In 2005, a group of modellers, biochemists and software engineers have started working on a standard notation and in 2009, they published the Systems Biology Graphical Notation (SBGN) [**?**]. This notation allows scientists to represent biological pathways and networks in an easy-to-understand and efficient way. It consists of three complementary languages: process descriptions (PD), activity flows (AF) and entity relationships (ER).

In the last years, scientist have developed some other non-graphical notations for representing biological network information [12] and in the meantime over 300 biological network and protein interaction databases have been formed [13].

As intended, SBGN attracted considerable attention of the scientists and many software supporting this notation have been developed [14]. A person can create a new pathway or load a previously created one from listed databases using those tools. The tools also allow automated layout of those networks. This feature is crucial because, a nicely laid out graph helps the user to understand the underlying data faster, whereas a bad drawing may be misleading and confusing as can be seen in Fig. 1.3.

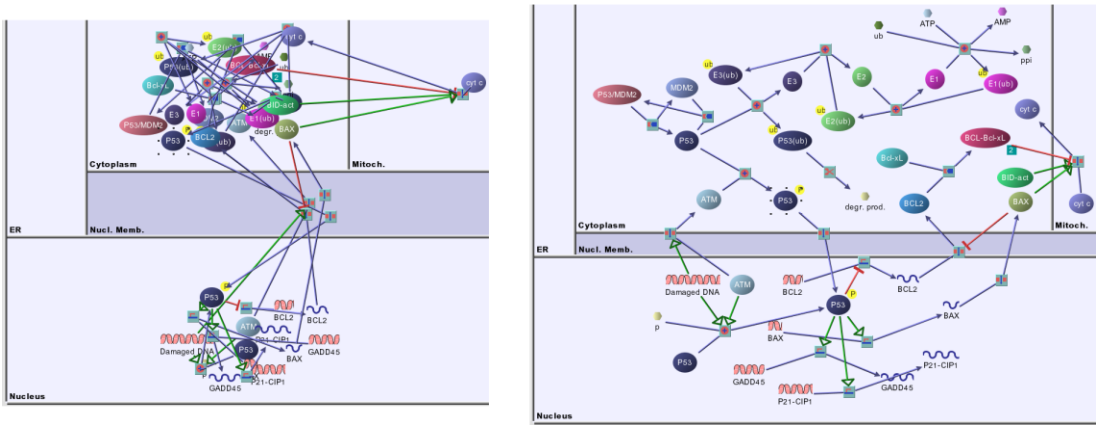

Figure 1.3: Random node positioning (left) and automatically laid out (right) versions of a biological network that respects cell compartments.

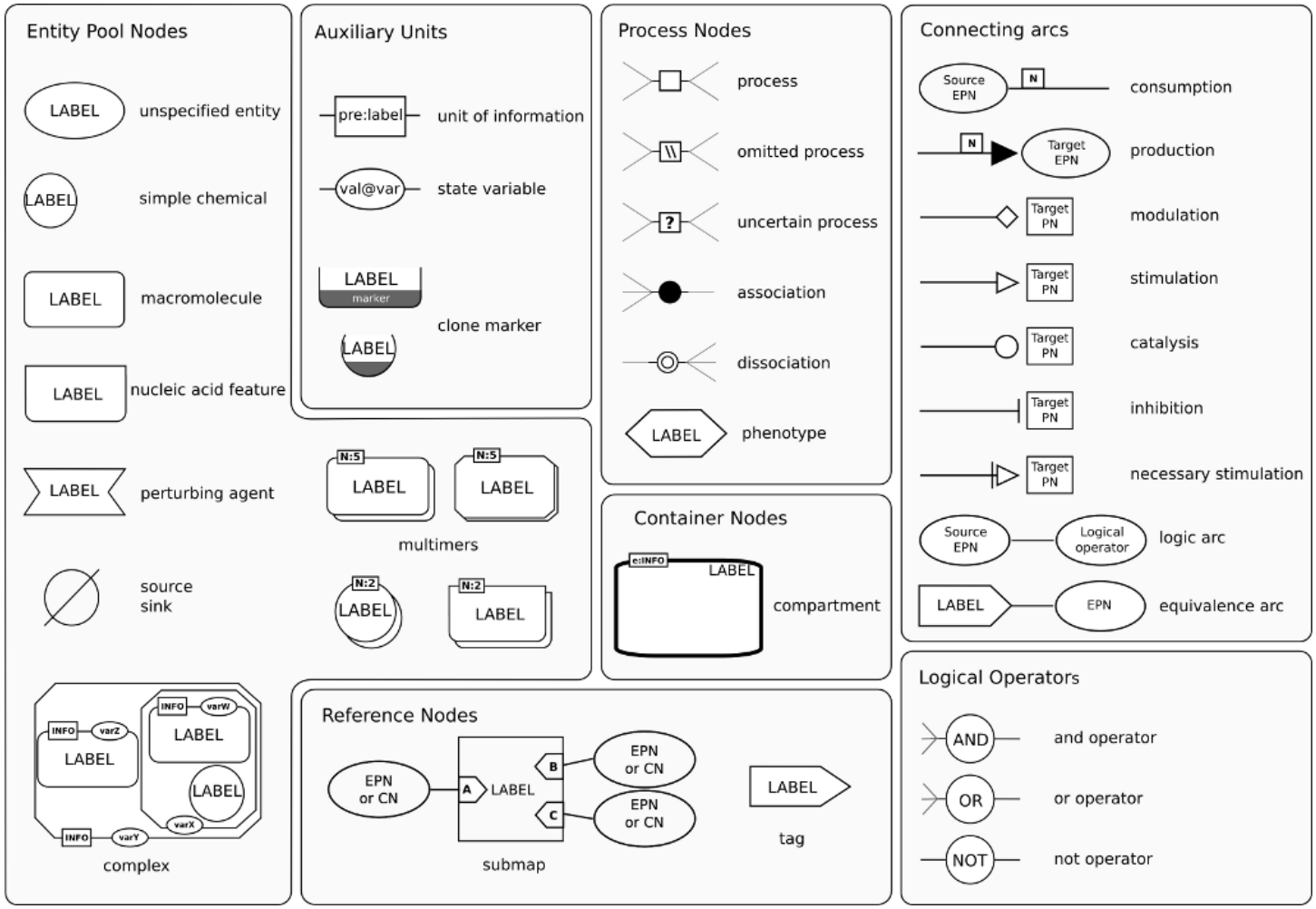

Figure 1.4: Systems Biology Graphical Notation (SBGN) reference card [3]

Until recently, most of the studies about automated layout have been focused on visualization of biochemical pathways. There are many software tools supporting SBGN and many studies on layout of biological networks [15] [16] [17] [18] [19]. However, none of them are specializing in automatic layout of SBGN process diagrams.

Process descriptions are intended to show *change*. A process description map represents processes that transform physical entities into other entities as a result of different influences [11].

The main purpose of this research is to represent process description maps in SBGN as clear and understandable as possible. We aim to speed up the comprehension and analysis steps of biological pathways for scientists by supporting the SBGN standard notation.

1.2 Contribution

We propose a new automated layout algorithm that enforces SBGN-specific rules. Those rules include separating different reactions by using associated compartments, tiling of zero degree members and proper orientation of processes and their neighbors.

The sample biological diagram in Fig. 1.5 is laid out using a general purpose layout algorithm. The tool [20] supports SBGN by drawing nodes and edges properly. However, that layout exhibits three problems that we address to solve in our proposed layout algorithm.

First problem is that, SBGN states that product and substrate edges of a process node (represented as small gray squares) should be placed on opposite sides of associated process nodes (A). Moreover, each process node should have two port nodes: an input and an output port. The products of a process should be connected to the output port and substrates should be connected to the input port. Secondly, the classical layout does not tile degree zero members inside the

complexes (represented as rectangles with information bulbs) (B). Lastly, the reactions that occurs inside the nucleus is not separated from the other reactions occuring in other places (C).

In this thesis, we propose a new layout algorithm to we obtain a more proper and compliant SBGN process description diagram (Fig. 1.6).

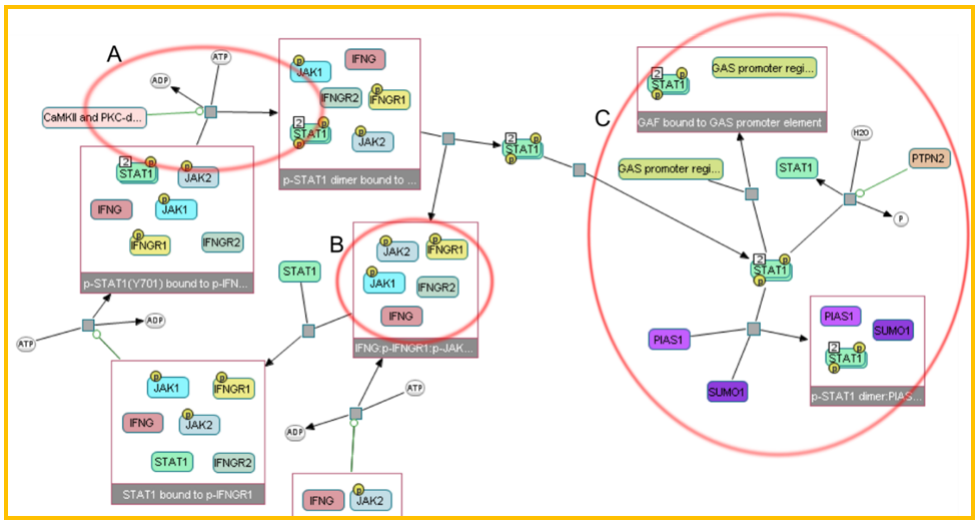

Figure 1.5: Result of an automated layout algorithm applied on a biological process diagram

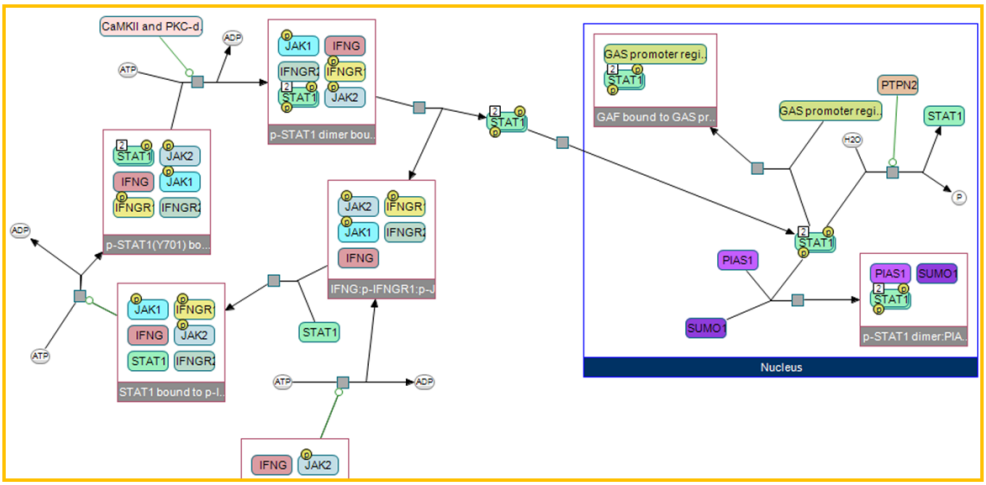

Figure 1.6: Proper SBGN layout of the biological diagram shown in Fig. 1.5

Chapter 2

Background and Related Work

2.1 Graphs

A graph is a representation of a set of discrete objects and their relations. The objects are called *nodes or vertices* and the links are called as *edges*. Mathematical notation of a graph is $G = (V, E)$, where vertex $v \in V$ and edge $e \in E$. An edge e represents a connection between two nodes and is represented by $e = (u, v)$, where $u, v \in V$. If two vertices u and v are connected to each other by an edge e, those two vertices u and v are said to be *adjacent vertices* and they are *incident* to e.

A graph may have a *directed* or *undirected* structure. If pairs (u, v) are ordered, the graph is said to be a directed graph. In that case, *in-degree* of a node v is calculated as the total number of incident edges that have their target nodes as v and *out-degree* is the total number of incident edges where their source nodes are v, respectively. Undirected graph vertices do not have a specific in/out-degree. In both cases, *degree* of a node is the total number of incident edges of that node.

A graph may have child graphs. In that case, if $G = (E, V)$, $G_i = (E_i, V_i)$ and $G_i \in V$ and similarly, parent of G_i is G. A node that have children nodes but, does not have a parent node is referred as *root* of the graph and child graph

of the root is referred as *root graph*.

A *compound graph* can be identified as having a number of edges that connect two nodes, whose parents are different from each other. It is shown as $C = (V, E, F)$ where V stands for vertex set, E for edge set and F is the inclusion edge set. Inclusion graph $T = (V, F)$ is an inclusion tree. T does not include any edges connecting a node to its children or parent nodes. Fig. 2.1 illustrates a compound graph, where blue nodes represent simple nodes, which do not have any children and conversely red labelled nodes are compound nodes, which have child graphs. For instance, 4 is a compound node, which has 1 as its parent node, which, in turn, is a compound node with children 8, 9, 31, 33, 38, 39.

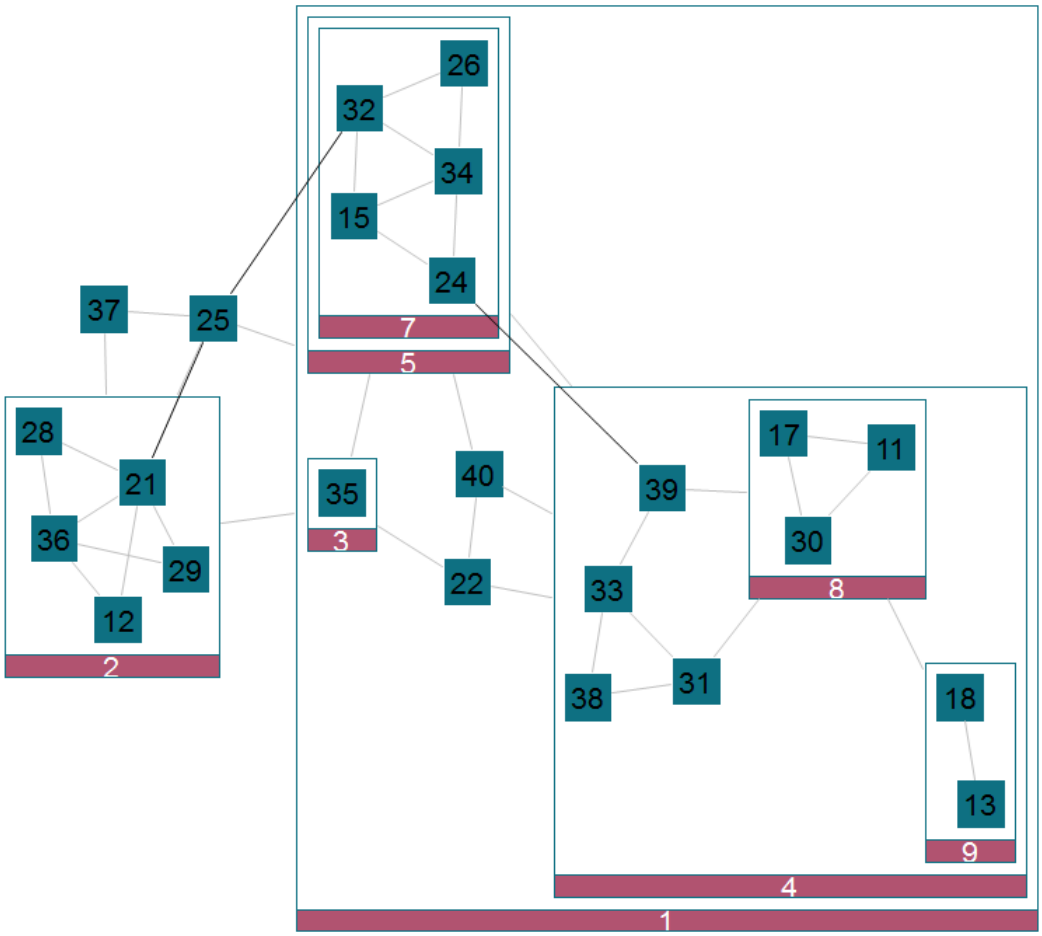

Figure 2.1: Sample compound graph created randomly using Chisio graph editor [4].

One common data structure to implement a compound graph is a *graph manager*, which stores the nested structure of compound graphs. A graph manager $M = (S, I, F)$ is defined by a set of inter-graph edges I, a set of compound graphs $S = G_1, G_2, ..., G_n$ and a rooted nesting tree $F = (V^F, E^F)$ [7]. Each node may have different sizes. If a node is a compound node (i.e., it has a child graph), its bounding box is calculated with respect to the bounds of its child graph in a nested manner.

2.2 Systems Biology Graphical Notation

The Systems Biology Graphical Notation (SBGN) is a standardized visual representation of biochemical cellular processes. It is developed by a group of software engineers, modellers and biochemists. It defines some sets of symbols and rules by defining their usage. Fig. 2.2 ilustrates a sample SBGN process description map.

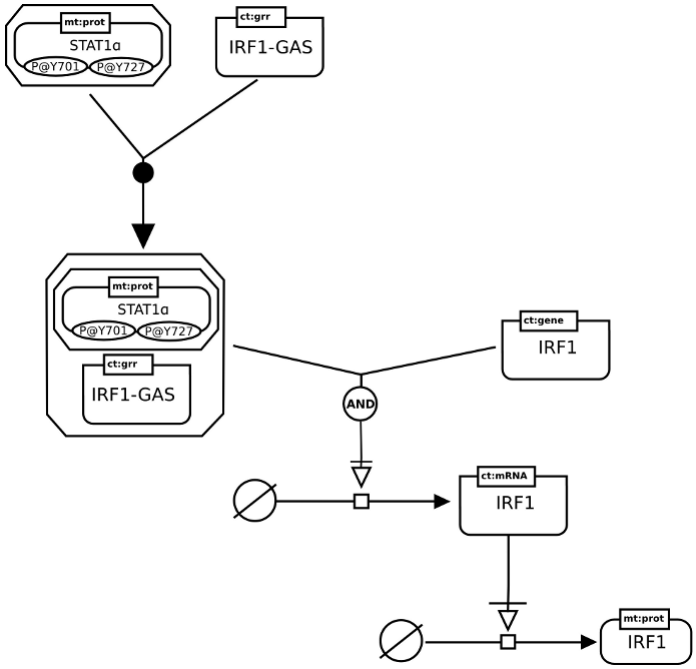

Figure 2.2: IFN regulation, a gene regulatory network.

Biochemical and cellular process has been visualized for nearly sixty years [11].

In the recent years, many software tools have been developed for pathway and network design such as PATIKA, CellDesigner, NetBuilder [21] [22] [23]. However all those tools have used their own notations. Therefore, representation of a network resulted in different visuals, causing ambiguity and slowing down the research process. This problem raised the need to have a standard notation and triggered SBGN to be formed.

An *SBGN Process Description* map is fundamentally a bipartite graph, i.e., its vertices can be divided into two disjoint sets such that each vertex in one set connects to a vertex in the other set, with no edges allowed between two nodes in the same set. One of those sets corresponds to *process nodes* and the other one to *entity pool nodes*. Another structure is *compartments*, which may contain some entity pool nodes. The entities are referred to as *glyphs* and the edges as *arcs*.

There are six glyph classes in entity pool nodes: unspecified entity, simple chemical, macromolecule, nucleic acid feature, and complex. Those materials may contain extra information such as *state variables*, *units of information* and *clone markers* (Fig. 1.4).

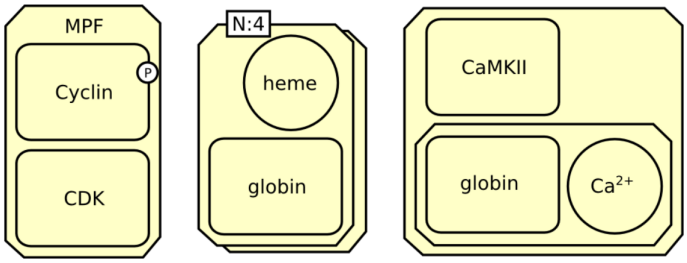

Figure 2.3: From left to right: complex between phosphorylated cyclin and CDK2 forming the maturation promoting factor in yeast, a tetramer of complexes between globin and heme, and a complex between calcium-calmodulin kinase II and another complex, itself formed of calmodulin and calcium [5].

For the layout process, one of our main concern are complexes. A complex node represents a biochemical entity, which is composed of some other biochemical entities. A complex is represented by a node that have octagonal boundaries. Children of complexes do not have edges to other entities. Hence, they

are disconnected nodes. The rule for complex member placement is to tile those disconnected members. Fig. 2.3 illustrates a number of complexes in SBGN.

Another important concern for us is the process nodes. A process node represents processes that transform a number of entity pools into a number of another entity pools. The generic rule for the process nodes is that, they have two port nodes. One of them is for *substrates* (input) and the other one is for *products* (output). SBGN specifies that the port nodes should be placed on the opposite sides of its corresponding process node. Fig. 2.4 illustrates a process node with two inputs (left) and three outputs (right).

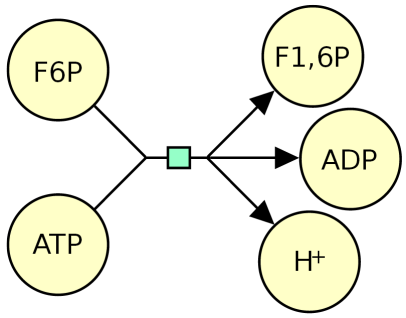

Figure 2.4: Reaction between ATP and fructose-6-phosphate to produce fructose-1,6-biphosphate, ADP and a proton [5].

2.3 Automated Layout

Information visualization plays a crucial role to improve understanding of people on a given data set. After data has been queried from databases and its irrelevant parts have been filtered out, it is ready to be visualized for analysis.

There has been lots of research on graph layout [24] [25]. The purpose of performing layout on a graph is to make visualization as clear and pleasant to the eye as possible. Because a poor layout of a graph may confuse the user's mind, whereas a good, aesthetically pleasing layout helps with understanding the data easily (Fig. 1.5 and Fig. 1.6). Criteria of a good layout may differ from person to person. However, the following are some generally accepted ones [24]:

- Minimize total drawing area
- Minimize number of edge-edge crossings
- Minimize total edge length
- Support uniform edge length
- Reflect the symmetry of the graph

In addition to those items, for our case, considering the fact that biological pathways include nested diagrams, it is important to use a layout algorithm that respects the nested structure. In other words, one of our criteria is to find an automated layout algorithm that places a child node of a compound node such that the boundaries of the compound node completely contains the boundaries of the child node plus user-defined margins.

Force-directed layout algorithms (also known as spring embedders) is probably the most popular approach for automatic layout of graphs. In 1984, Eades used a novel modelling for a graph: he considered nodes as steel rings and edges as stretchable springs [26]. In this algorithm, each node applies repulsion force to all other nodes in the graph and attractive force to its adjacent nodes.

The following equation is used for calculating the repulsion force between two non-adjacent nodes u, v [6]:

$$f_{rep}(p_u, p_v) = \frac{c_r}{\|p_v - p_u\|^2} \cdot \overrightarrow{p_u p_v} \tag{2.1}$$

He uses his own formulations to calculate spring forces, instead of using the exact Hooke's law formulations. If two nodes u and v are adjacent, which means that they have an edge connecting them to each other, the force acting between those two nodes is calculated using the edge, as follows:

$$f_{spring}(p_u, p_v) = c_s \cdot \log \frac{\|p_v - p_u\|^2}{l} \cdot \overrightarrow{p_u p_v} \tag{2.2}$$

Here, p_u denotes the position vector of u, c_r is the repulsion constant, c_s is the spring constant to control spring strength and l is the length of the edge. The algorithm performs those calculations, until the total displacement of nodes falls under a predefined threshold, indicating that the system became stable (Fig. 2.5).

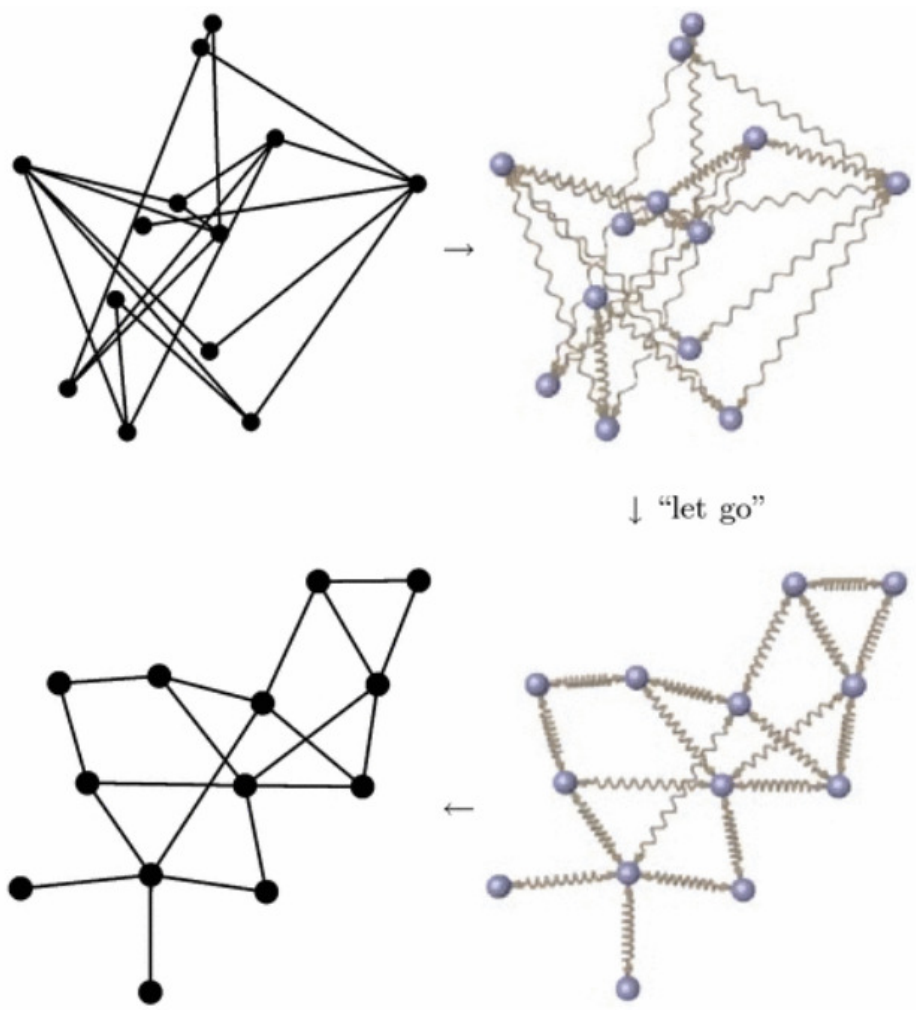

Figure 2.5: Illustration of a generic force-directed method: the graph (top-left) represented as springs and particles (top-right), reached stable state (bottom-right), corresponding graph (bottom-left) [6].

Kamada and Kawai enhanced Eades' algorithm by introducing the desirable (Euclidean) distance concept, using graph theoretic distance [27]. By doing so, they calculate attractive forces between not only adjacent nodes, but also nodes that are not adjacent but within an ideal distance. They also worked on minimizing the total spring energy of the system. As a result, their algorithm produced

successful results on drawing symmetric graphs and resulted in small number of edge crossings.

Later, Fruchterman and Reingold improved Eades' algorithm and proposed an advanced version [28]. They have used the uniform edge length concept. Their algorithm has two main principles: if two nodes are adjacent, draw them close to each other, but do not draw any two nodes too close to each other. They use a formula similar to Hooke's law formulations for spring force calculations. In Hooke's law, force F is defined as $F = -kx$, where k is defined as spring constant and x is defined as the distance from spring's current position to desired position.

Although those algorithms perform well especially on small scaled graphs, they do not produce good results for the graphs that have a few hundred or more nodes. The main reason of this problem is that, layout process has many local minima. In order to skip local minima, simulated annealing's cooling process has been imitated but Kobourov states that even more complex approaches do not guarantee to skip local minima [29].

2.4 Compound Spring Embedder

Compound Spring Embedder (CoSE) is a force-directed graph layout algorithm that aims to satisfy the generally accepted graph drawing criteria for compound graphs [7]. CoSE is based on Fruchterman and Reingold's algorithm discussed in the previous section. Additionally, CoSE has support for inter graph edges, non-uniform node sizes and clustered graphs.

This automated layout algorithm treats the nodes as charged particles, edges as springs and it symbolizes each compound node as a "cart". Therefore, a nested compound structure can be represented with multiple carts on top of each other, where a cart may contain a number of objects and other carts.

During the layout process, two nodes can repel or pull each other depending on the distance between them. If those nodes are adjacent, force calculation

depends on the tension of their edge. If they are not adjacent and located too close, they apply repulsion forces on each other, which is a similar approach to the one that Fruchterman and Reingold have proposed. Repulsion forces ensure avoiding overlaps between nodes as much as possible.

CoSE also uses *gravitational forces*. The aim of this force is to keep children of a node closer to the center of their parent, thereby by keeping the children nodes together. Similarly, in the case of the root graph, the gravitational forces keep disconnected graphs together. The results show that this force is useful for reducing the total drawing area.

To illustrate, in Fig 2.6, the nodes represented with filled rectangles in the sample graph are simple nodes, whereas the nodes with no fill are compounds. In the corresponding model, it is seen that all nodes are positively charged. There are gravitational force locations at the center of each compound node and at the center of the root graph. The distance between the gravitational force locations and the nodes affected by that point are shown using red lines.

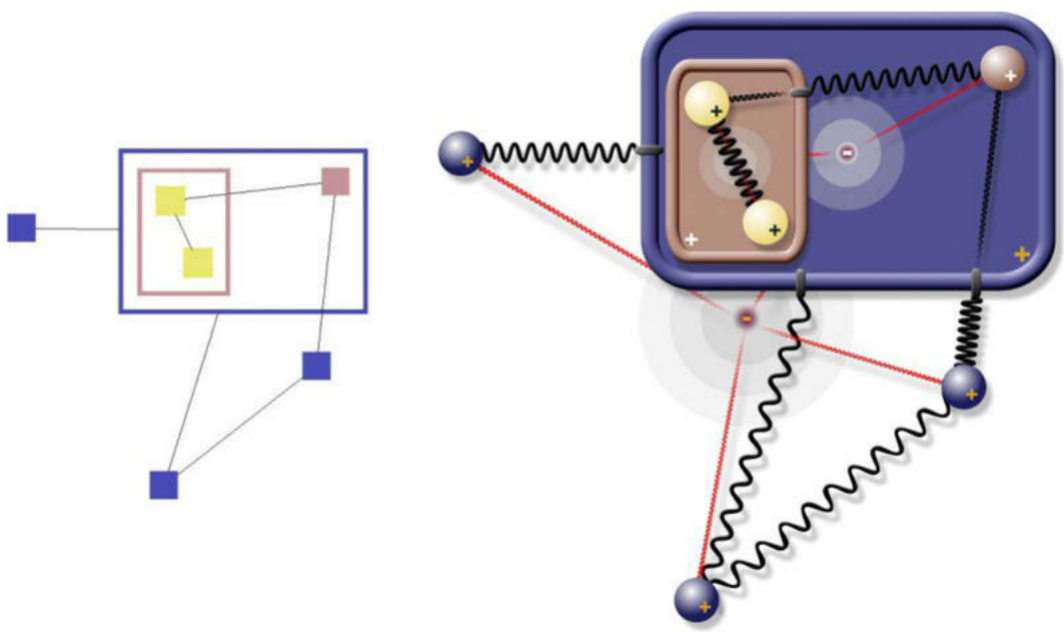

Figure 2.6: A sample compound graph (left) and its corresponding model in CoSE (right) [7]

CoSE supports non-uniformly sized nodes. The algorithms that do not support varying node sizes, can make use of uniform edge lengths. However, having different node sizes results in some difficulties such as adjusting the edge length. For instance, a predefined edge length may be shorter than the distance between the centres of two nodes, if at least one of them has a big bounding box. Therefore, if non-uniform node sizes is supported, a specialized ideal edge length calculation should be provided. CoSE overcomes this problem by using *clipping points*. Clipping point of a node is defined as the intersection point between the edge, which is defined from center point of the source node to the center point of the target node, and the bounding box of the node. The ideal edge length is then defined as the length between two clipping points.

Because of the fact that there are varying node sizes, finding an ideal edge length, that is valid for all nodes is not possible. Thus, CoSE defines an ideal edge length for each edge, depending on the number of nesting structures it spans. In order to do that, lowest common ancestors of two end nodes are found first. Then, using the ancestors, clipping points of the nodes are calculated and the distance from nodes to the corresponding clipping points are added to the ideal edge length. After ideal edge lengths for all edges are found, initial positioning of the nodes are performed. If the user does not run incremental layout, initial positioning is performed randomly or radially, depending on the graph type, which in this case being flat and forest or not. The pseudo code of the initialization step can be found in Algorithm 1.

After the initialization step is completed, the forces (repulsion, spring and gravitational) are applied on nodes until the system reaches a steady state or a maximum iteration count. If the total displacement between two iterations is below some predefined threshold, the algorithm is said to be stable. The cooling scheme used by CoSE is a linearly decreasing function (Algorithm 2).

Algorithm 1 CoSE Initialization

```
function classicLayout()
    calcLowestCommonAncestors()
    calcInclusionTreeDepths()
    calcIdealEdgeLengths()
    if layoutType ≠ incremental then
        if graph is forest and flat then
            positionNodesRadially()
        else
            positionNodesRandomly()
        end if
    end if
    initSpringEmbedder()
    runSpringEmbedder()
end function
```

Algorithm 2 Perform Layout

```
function runSpringEmbedder()
    while totalIterations < maxIterations do
        totalIterations ← totalIterations + 1
        if isConverged() then
            break
        end if
        updateBounds(graphManager)
        calcSpringForces()
        calcRepulsionForces()
        calcGravitationalForces()
        moveNodes()
    end while
end function
```

2.4.1 CoSE Layout Structure

CoSE makes use of a graph manager that keeps track of all subgraphs, all edges (including inter-graph edges) and all nodes.

As discussed previously, CoSE calculates spring, repulsion and gravitational forces at each iteration and moves each node with respect to total force acting on it. SBGN-PD layout uses the existing forces of CoSE with some modifications. Therefore, in order to specify the modifications, we explain the methods in detail.

Spring forces are calculated for each edge in CoSE. This method first updates the edge length using the end nodes centres, as opposed to clipping points to simplify calculations involved or using the rectangles representing the geometry of the nodes. Then, the spring force is calculated and applied on the end nodes.

Algorithm 3 Calculate Spring Forces

1: **function** CALCSPRINGFORCES()
2: **for all** $e = (u, v) \in E$ **do**
3: $edge \leftarrow$UPDATELENGTH(edge)
4: $springForce \leftarrow c_s * (edge.length - idealLength)$
5: $u.\vec{F_{spring}} \leftarrow u.\vec{F_{spring}} + springForce$
6: $v.\vec{F_{spring}} \leftarrow v.\vec{F_{spring}} - springForce$
7: **end for**
8: **end function**

Repulsion forces are calculated for each pair of nodes that are either adjacent nodes or within some distance. In order to find if two nodes are within some distance, CoSE uses either grid variant structure proposed by Fruchterman and Reingold or simply checks for the distance between each node. It is important to note that the repulsion force calculations for each node is different for those two cases:

- If FR grid variant is active, the surrounding nodes of the given node are detected first and then the repulsion force is applied if both of them are in the same graph.
- Otherwise, overlap amount between two nodes are detected first. If those

Algorithm 4 Calculate Repulsion Forces

```
function calcRepulsionForces()
    if type is useFRGridVariant then
        grid ←calculateGrid()
        for all v ∈ V do
            addNodeToGrid(v)
        end for
        for all v ∈ V do
            calculateRepulsionForceOfANode(v, grid)
        end for
    else
        S := {}
        for all v ∈ V do
            S := S ∪ {v}
            for all u ∈ V − S do
                calcRepulsionForce(u, v)
            end for
        end for
    end if
end function
```

two nodes overlap, repulsion force is calculated using the overlap amount. If they do not overlap, the distance between them is updated according to the clipping points and force is calculated using the updated distance.

Gravitational forces keep the components of a graph as close as possible. If the graph is a disconnected graph and gravitational forces are not used, disconnected components may be placed far from each other, yielding in arbitrarily huge drawing areas. In order to minimize the total drawing area and keep the components close to each other as much as possible, children nodes of graphs are attracted towards the center point of the bounding rectangles of them.

During the calculation of gravitational forces, this force is applied on a node only if the node is "roughly" outside the bounds of the initial estimate for the bounding rectangle of the owner graph. Gravity range factor g_f, gravity constant c_g, compound gravity range factor g_{cf} and compound gravity constant c_{gc} are constants to relax the estimated sizes of the owner graphs because their bounding boxes are tight. Estimated bounding box of the root graph is relaxed more, when

compared to relaxation of a compound node.

Algorithm 5 Calculate Gravitation Forces

```
function calcGravitationForces()
    for all v ∈ V do
        o ← v.ownerGraph
        distance ← v.center − o.center
        if (o is rootGraph) ∧ (|distance| ≥ o.estSize ∗ g_f) then
            v.F⃗_grav = v.F⃗_grav − distance ∗ c_g
        else if (o is compound) ∧ (|distance| ≥ o.estSize ∗ g_cf) then
            v.F⃗_grav = v.F⃗_grav − distance ∗ c_g ∗ c_gc
        end if
    end for
end function
```

After all the forces are calculated, each node is moved with respect to the net force acting on it, where $\vec{F_{net}} = \vec{F_{rep}} + \vec{F_{spring}} + \vec{F_{grav}}$. In this location translating process, if a node is a compound node, the forces acting on the compound is propagated to its children. As a result, not only the compound node, but also the locations of its children are updated.

Algorithm 6 Propagate Displacement To Children

```
function propagateDisplacement(v, F⃗_disp)
    for all u ∈ v.childGraph do
        u.center ← u.center + F⃗_disp
        if u is compound then
            propagateDisplacement(u, F⃗_disp)
        end if
    end for
    updateBounds(v.childGraph)
end function
```

Considering the fact that, attractive forces are calculated for each edge ($|E|$), repulsive forces are calculated for each vertex pair ($|V|^2$) and gravitational forces are calculated for each vertex ($|V|$), CoSE is resulting in a per iteration complexity of $O(|E| + |V|^2)$.

2.5 Rectangle Packing

Most of the graph drawing algorithms try to minimize drawing area by assuming that the graph is connected [30]. This minimization is achieved by finding the bounding box of a graph, or in other words, finding the smallest rectangle that the graph can fit in. However, if the graphs have disconnected members and the layout algorithm is poor, resulting layout may contain large wasted area. Additionally, if user specifies an aspect ratio, which is defined as the ratio between resulting container's width and height, then the packing result should respect the desired ratio and the result may contain even more wasted area (Fig. 2.7).

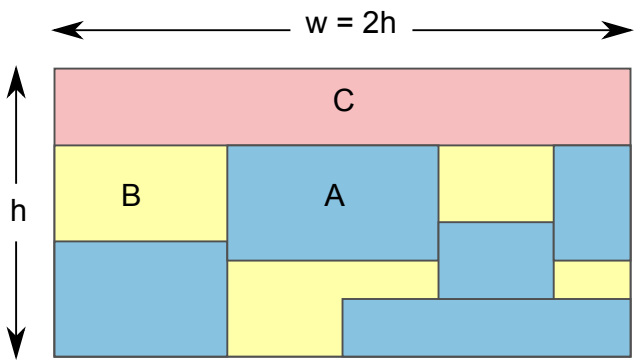

Figure 2.7: Total area is divided into 3: A (rectangular objects), B (wasted area) and C (additional wasted area if aspect ratio is specified as 2).

Rectangle packing problem can be defined as packing a number of non-uniformly sized, rectangle shaped objects into a container such that there will be no overlaps between the objects and the container will be as compact as possible. More formally, given a set of n rectangular items $i \in S_{1,\ldots,n}$, each defined by a width w_i and h_i, find the container R that covers each item i and the wasted area is minimum. This problem, defined in two-dimensions, is an NP-hard problem [31].

Fig. 2.8 illustrates the affect of packing on wasted area. The sample graph has two disconnected graphs and each of those graphs have 10 nodes. The layout algorithm is given an aspect ratio. It is seen that properly tiling disconnected components gives more aesthetically pleasing results with less wasted area.

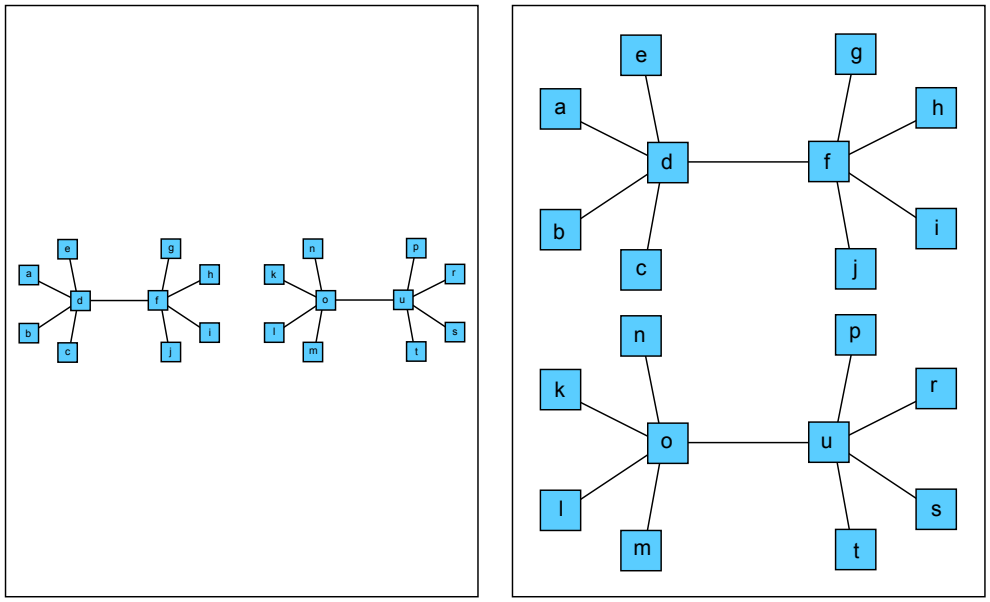

Figure 2.8: A sample disconnected graph layout given the aspect ratio, with and without packing, respectively.

Fullness of a drawing is calculated as the total area covered by the objects divided by the total drawing area. It is denoted by $F(L)$. *Adjusted fullness*, on the other hand, expresses the fullness of a drawing, when the aspect ratio is being used. It is denoted by $AF(L)$. For Fig. 2.7:

$$F(L) = \frac{A}{A+B} \tag{2.3}$$

$$AF(L) = \frac{A}{A+B+C} \tag{2.4}$$

Aspect ratio performance(ARP) of a packing algorithm can be calculated as follows:

$$ARP(L) = \frac{min(AR(L), DAR(L))}{max(AR(L), DAR(L))}, \tag{2.5}$$

where L denotes the list of objects, AR(L) is the aspect ratio produced by the algorithm and DAR(L) is the desired aspect ratio.

There are a number of different methodologies for rectangle packing such as *bin packing*, *strip packing*, *alternate-bisection method*, *tiling* and *polyomino packing* [30] [32] [8]. Below we examine two of those algorithms, which are tiling and polyomino packing. Tiling method gives good results in adjusted fullness and respects the aspect ratio [30]. On the other hand, polyomino packing supports not only rectangle shapes but also arbitrarily shaped objects.

2.5.1 Tiling

The general view to the tiling algorithm is that: the resulting container is composed of a number of strips, that are placed on top of each other. A strip is a horizontal container, that stores a number of objects. Therefore, tiling is defined as a type of strip packing problem. In strip packing, the strip is given with a fixed width value, which is calculated using the desired aspect ratio. However, tiling uses a strip, whose width changes dynamically.

The algorithm first, sorts the objects in descending order of their width sizes. Then, the first level strip is created and the first object is placed in it. Its width becomes equal to the width of that object. One by one, all objects are placed into the container by deciding whether to place the next object into the shortest strip or create a new strip. The condition for an object to be placed into the shortest strip is having enough space.

Fig. 2.9 shows a sample result of the algorithm. The experiments show that if the objects are not sorted in increasing order of their heights, the results become more compact [30]. The algorithm takes $O(n \lg n)$ time and has serious advantages in terms of adjusted fullness and aspect ratio performance over its competitive types.

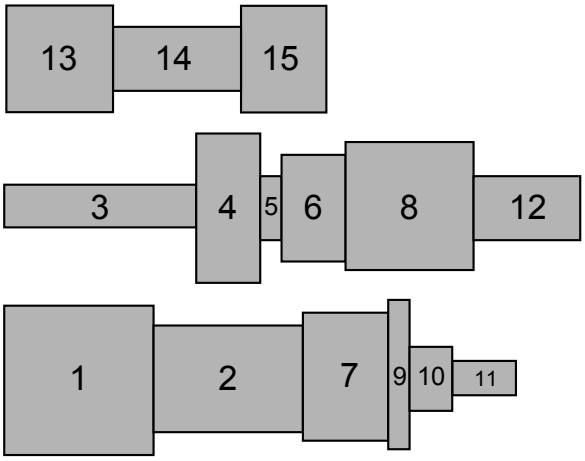

Figure 2.9: A sample result of tiling algorithm, where the numbering indicates the order of processing.

2.5.2 Polyomino Packing

A *polyomino* is a geometric figure formed by joining unit squares at the edges [8]. A polyomino represents a graph object. Thus, if the graph is a disconnected graph, each disconnected component is represented using a polyomino. Fig. 2.10 illustrates a sample polyomino of a graph.

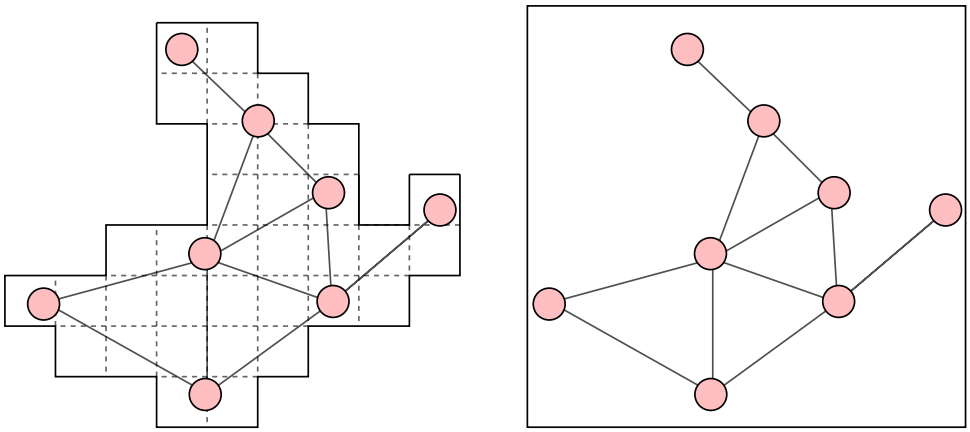

Figure 2.10: Polyomino constructed from a simple graph (left) and its bounding box (right).

The polyomino packing algorithm uses a greedy heuristic. Initially, the polyominoes of the objects are created. Then, a finite grid, that can cover all polyominoes fully or partially, is constructed. After the initialization is completed, each

polyomino is inserted into the grid one by one. The optimal place of a polyomino is where cost function $max(|x|, |y|)$ is minimum. In order to find the optimal place, each empty cell is sequentially scanned in the increasing order of the cost function. This means that the cells near to the already placed polyominoes are checked first. If the polyomino fits in a position without causing any overlaps, the corresponding polyomino cells are marked as full.

It is very crucial to set a good grid cell size s. Because, increasing the cell size means increasing the wasted area, whereas using a small grid size increases the running time of the algorithm by almost checking each point in the grid. The quadratic equation below can be used to calculate l, where c is some constant, s is the average polyomino size and $s \leq c$.

$$(cn - 1)l^2 - \sum_{i=1}^{n}(W_i + H_i)l - \sum_{i=1}^{n} W_i H_i = 0 \tag{2.6}$$

For the analysis, the algorithm scans each grid cell to find a suitable grid for each polyomino. When an empty grid cell is found, each neighboring cells corresponding to polyomino cells are checked to see if the polyomino fits into that empty grid cell. Therefore, the algorithm takes $O(n^2 s^2)$ but since s is a constant, the algorithm is of $O(n^2)$, where n is the number of objects to be packed. Fig. 2.11 shows a comparison of polyomino packing and tiling layouts.

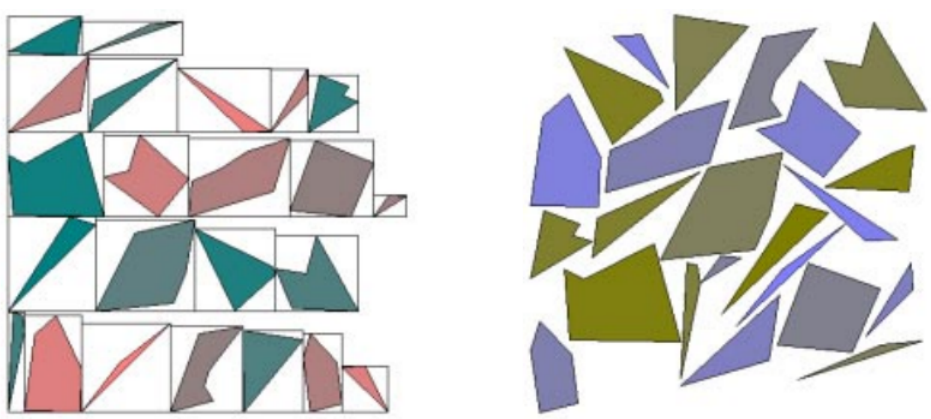

Figure 2.11: Packing results of a arbitrarily shaped object set using two methods: tiling (left) and polyomino packing (right) [8].

Considering that, a grid cell size may not have been properly set. As indicated previously, large grid cells result in large wasted area. In order to reduce that wasted area, a compaction may be performed on the resulting layout as post-processing.

2.6 Compaction

Compaction process is used to obtain more compact results from the tiled graphs. If the objects are drawn more compact, it may decrease the wasted area and result in smaller bounding box, which is a requirement for obtaining more aesthetically pleasing drawings.

Visibility graphs may be used to perform compaction. Visibility graphs are commonly used for planning robot paths. For instance, Nilsson described a system for planning the motion of robotic systems among obstacles in 1969 [33] [34]. He introduced the visibility graph and combined it with A* heuristic to find the shortest, collision-free path amidst obstacles between the robot's location and the destination point.

A visibility graph is described as $G_v = (E_v, V_v)$ and it indicates the visibility relation between a set of nodes. The *visibility* refers to the feasibility of drawing a collision-free straight line between two nodes (Fig. 2.12).

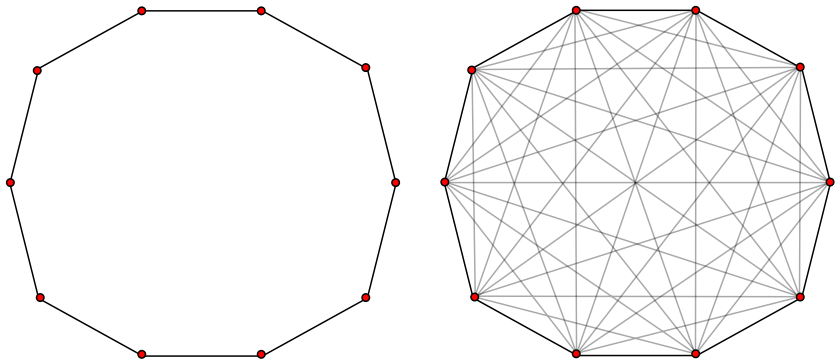

Figure 2.12: A simple polygon (left) with no obstacles and the visibility graph edges (right)

In this context, nodes are defined as point locations and edges are visible connections. However, for the simplicity of the problem, we use the set of objects as the set of nodes instead of using point locations.

There has been intensive research on visibility graphs in 1980's [35] [36] [37] [38]. The naive approach runs in $O(n^3)$ and the studies show that it can be improved to $O(n^2)$ by using sweep lines.

The visibility graphs can be applied to a number of shapes such as convex polygons, line segments and simple polygons. An illustration of visibility graph on rectangular objects can be found in Fig. 2.13. We are interested in only the vertical and horizontal visibilities.

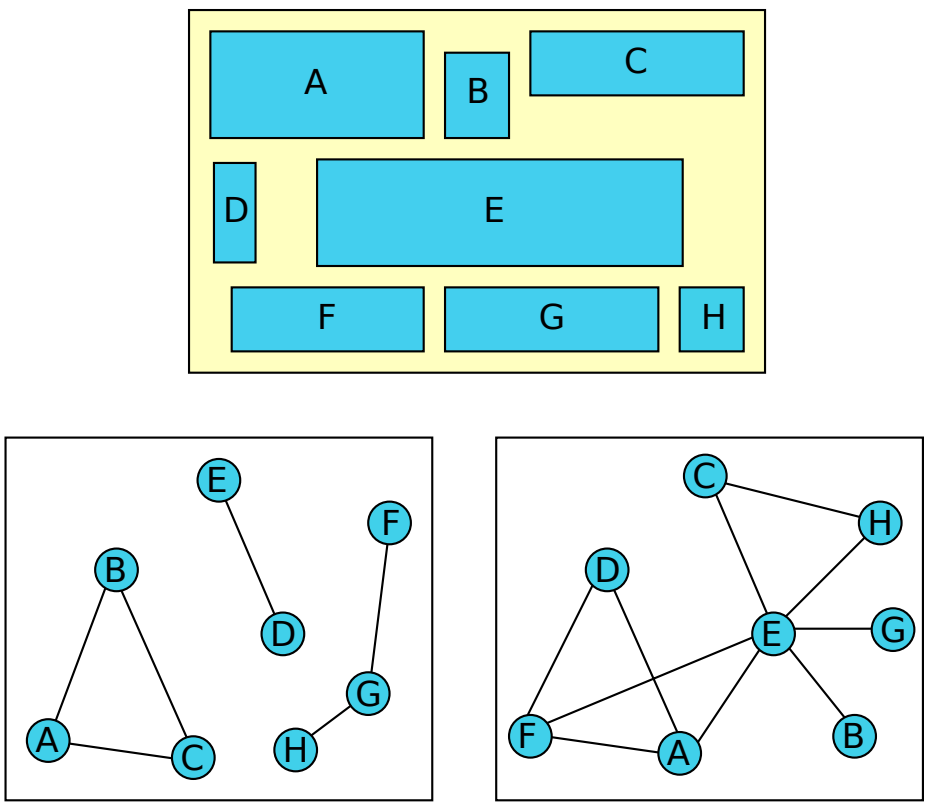

Figure 2.13: A set of objects (top), its horizontal visibility graph (bottom-left) and vertical visibility graph (bottom-right), with no directions specified.

Chapter 3

SBGN-PD Layout

SBGN-PD layout is a specialized automated layout algorithm for SGBN process description maps. It uses the force-directed CoSE layout algorithm as basis. Satisfying SBGN rules brings additional constraints to this layout problem. On top of the CoSE structure, we add a new type of force and associated procedures. In the following sections, enhancements to the original CoSE algorithm and adaptation of previously discussed algorithms are going to be explained in detail.

Spring force and repulsion force calculations are modified by SBGN-PD layout to prevent wrong relocation of process and port nodes (see section 3.2 for details). Additionally, a new force called as *rotational force* is introduced.

CoSE has only one phase for performing layout (see Algorithm 2), whereas SBGN-PD layout uses a two-phase algorithm. The first phase is almost identical to CoSE procedure, except it has a relatively small, fixed number of maximum iterations and it uses the modified force calculation methods.

In the second phase, the maximum number of iterations is initially calculated by taking the logarithm of total number of node and edges. Logarithm function ensures a slow growth. The reason behind using this calculation is to have less maximum number of iterations for small graphs in the second phase, whereas more iterations may be needed for a more complex graph.

Algorithm 7 Second Phase of SBGN-PD Layout

```
function doPhase2(G)
    maxIterations ← log(G^V.size + G^E.size) * c_0
    initialCoolingFac ← c_cool
    while totalIterations < maxIterations do
        totalIterations ← totalIterations + 1
        if approximationPeriod is reached then
            approximateLocations()
        end if
        if isConverged() and areEdgesProperlyOriented() then
            break
        end if
        updateBounds(graphManager)
        calcSpringForces()
        calcRepulsionForces()
        calcGravitationalForces()
        moveNodes()
    end while
end function
```

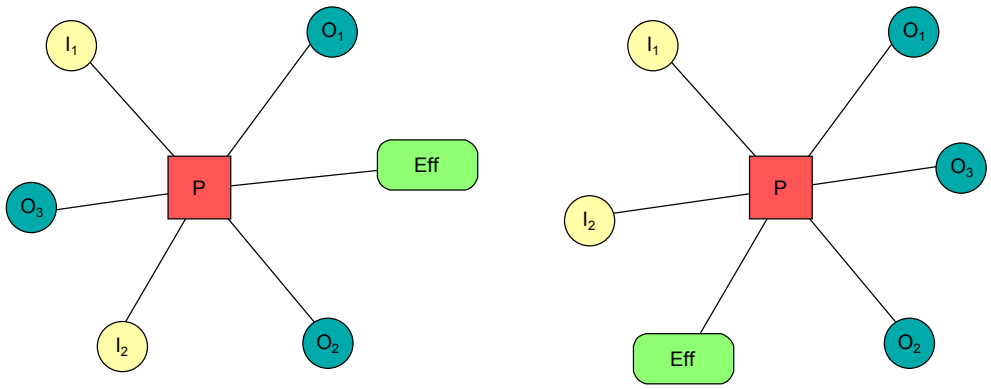

Figure 3.1: CoSE fails to separate substrate, production and effector nodes (left), desired result (right).

Additionally, in the second phase, the cooling factor starts from a smaller value to not to alter the result of first phase. However, this brings along some problems such as not being able to place substrate and production nodes close to each other or put the effector node in between those (Fig. 3.1). Because, CoSE does not respect the node types, it only considers the shapes and their overlap amounts. Therefore, in addition to those modifications, this phase applies a location approximation algorithm for product and substrate nodes and another one for effector nodes at certain iterations.

The approximations are only applied a few times. Product and substrate node approximation is performed to prevent two nodes that connect to different port nodes getting stuck on the reverse sides. Because of the fact that placement of nodes at the end of the first phase does not respect port node locations, the approximation is needed. In addition to this, cooling factor is decreased to a small value to prevent alteration. However, decreasing the cooling factor also prevents, for instance a product node to jump over a substrate node and get located near its own port node.

Although we want all nodes to be close to their own port nodes, the algorithm should not interfere much with a highly connected node. Because, for these types of nodes, forcing their locations to another point may cause long edges and poor layout. Therefore, we only apply approximation for degree-one nodes.

The calculation is done as follows: find all port nodes and their production or substrate nodes. Group multi-edge nodes and degree-one nodes separately. After that, for each group, find an approximation point using the locations of multi-edge nodes. On the other hand, if a port node has only degree-one nodes, then one of those nodes is selected randomly for the approximation point. At the end, all degree-one nodes are moved within a circle, whose center point is the approximation point and radius is a constant.

Location approximation for effector nodes is performed similarly. An effector node may get stuck between some production or substrate nodes. In order to prevent this, all effector nodes of a process node are found and grouped separately. Then, location of each effector node is changed with respect to the orientation of

the associated process node. Location of the effector node is always approximated within an ideal edge length distance from process nodes' center. There are four cases to consider: process node is oriented vertically and effector should be placed on its left or right, or the process is oriented horizontally and effectors should be placed on its top or bottom.

For instance, illustration of a vertical-left scenario works as follows: if port nodes of a process node is placed vertically, its effector nodes should be placed either on its left or right. If an effector is already placed on the left of the process node, calculate a target point from process node to ideal edge length distance to the left and place the effector node within an circle close to the approximation point. The other cases are handled similar to this one.

As mentioned previously, SBGN-PD layout introduces rotational forces. A rotational force is calculated at each step and it is applied while moving the nodes. This force is discussed in detail in section 3.3.

3.1 Nested Rectangle Packing

In order to achieve compact and proper drawings of SBGN process maps, disconnected nodes must be packed. A disconnected node may either be a member of a complex node or a degree-zero molecule that is not owned by a complex. A graph may need rectangle packing for two reasons:

- *Tiling of complex members*: Tiling of complex members is required by SBGN. It is important to note that, a complex node may have another complex node as its child. Therefore, the problem becomes a nested rectangle packing problem.
- *Tiling of disconnected molecules*: Disconnected graphs may result in large wasted area. In order to reduce it, we perform packing for all disconnected molecules at each level, that are not owned by any complex nodes. In the rest of the document, a *disconnected molecule* refers to a degree zero node,

whose owner is not a complex, for convenience.

Fig. 3.2 illustrates layout of a sample disconnected graph. CoSE does not perform packing and results in a huge wasted area when compared to SBGN-PD (Fig. 3.2).

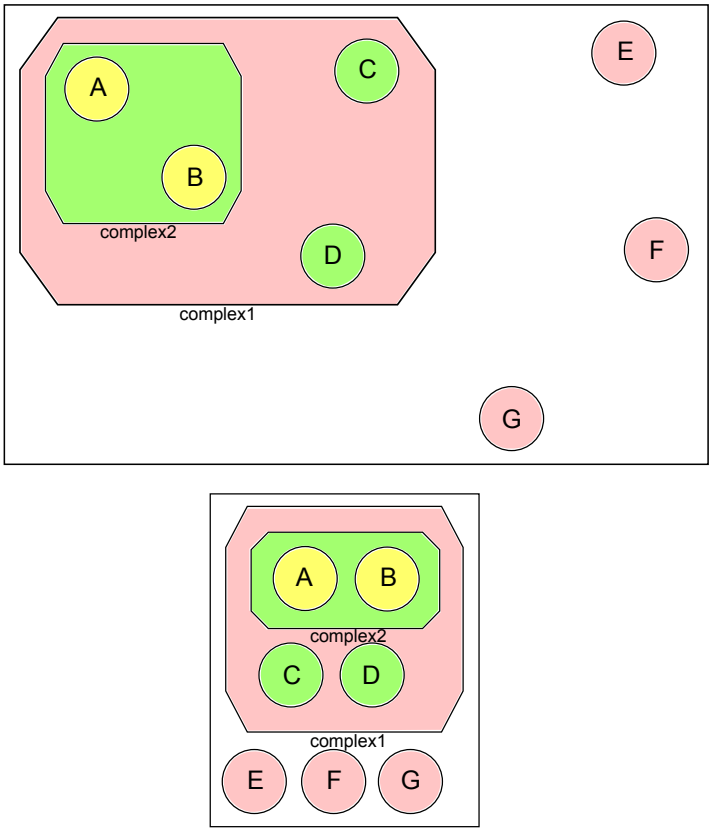

Figure 3.2: A disconnected graph laid out using CoSE (top) and SBGN with tiling (bottom). Disconnected molecules are shown using red, where complexes are represented by octagonal boundaries.

In order to handle the nested structure, a recursive approach is needed. In this recursive algorithm, processing order of complexes plays the key role. The processing should be performed in a bottom-up manner. Because, each time, bounding box of a complex is calculated using its child nodes. For instance, let $v \in V_G$ be a complex node and its depth $v_{depth} = d$, where $d > 1$ and $root_{depth} = 0$. In order to correctly tile the nodes at depth $d-1$, v_{width} and v_{height} must be known. If the ordering is not performed in a bottom-up manner, while

tiling complex members at depth $d-1$, v_{width} and v_{height} are going to be wrong. Because, their values are going to be as their initial values. Hence, tiling at depth $d-1$ is not going to be correct.

To achieve this bottom-up processing, depth-first search is performed on complex nodes of the graph. After finding all the complex nodes, rectangle packing is performed on each complex with respect to their orders.

Algorithm 8 Perform Layout

```
function LAYOUT()
    groupDisconnectedMolecules()
    complexOrder ←applyDFSOnComplexes()
    for all complex ∈ complexOrder do
        clearComplex()
    end for
    performLayout()
    for all complex ∈ complexOrder do
        repopulateComplex()
    end for
end function
```

3.1.1 Packing Complex Members

A user can specify tiling or polyomino packing method to pack the complex members.

Tiling is a straightforward but elegant method that makes use of a container for the objects [30]. The container is composed of strips that are put on top of each other. After sorting each node with respect to their areas in decreasing order, they are inserted into the container one by one.

Aspect ratio is taken into consideration while making the decision about whether to place the object horizontally or vertically by creating a new strip. This decision is made by checking if width of the container increases in case of placing an object into the shortest strip. If it has no effect on the width, the object can be inserted horizontally. However, if the width increases, it may be

Algorithm 9 Tiling pseudocode derived from [20]

```
function insertNode(container, node)
    if container if empty then
        insertNodeToRow(container, node, 0)
    else if canAddHorizontal(container, node) then
        insertNodeToRow(container, node, container.shortestRowIndex)
    else
        insertNodeToRow(container, node, container.lastStripIndex + 1)
    end if
end function
```

a good idea to place the object vertically. In this case, a new strip is created with that object. When a new strip is created, the container may get smaller by shifting the last element of the longest strip to the end of the new strip. Therefore, after each strip is created, a recursive check is performed to detect potential shifting operations.

Polyomino packing uses a grid structure and places the polyominoes on the empty grid cells [8]. Each polyomino consists of a number of polyomino cells. For instance Fig. 3.3 illustrates example of some sample polyominoes and their polyomino cells. Note that, if cell size is chosen carefully, a polyomino may result in less area when compared to the object's bounding box.

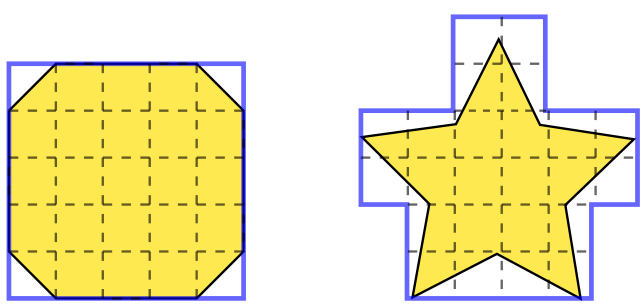

Figure 3.3: Polyomino representation and the polyomino cells of two objects.

In order to have a good cell size, a dynamical computation may be performed or a constant may be used. We use a predefined constant c_l and adjust the value with respect to specified aspect ratio.

$$step = \frac{2}{1 + aspectRatio} \tag{3.1}$$

$$cellSizeY = c_l * aspectRatio * step \tag{3.2}$$

$$cellSizeX = cellSizeX * step \tag{3.3}$$

After the polyomino cell sizes are calculated, polyomino cells are created. When the cells are created, a random permutation of them are performed and their bounding rectangles are calculated. Then, a grid is created and polyominoes are inserted to the grid one by one.

Algorithm 10 Polyomino Packing Body

```
function PACK(polyominoes)
    MAKEGRID(dim, 0)
    SORTINCREASINGSIZE(polyominoes)
    for all pmino ∈ polyominoes do
        PUTMINO(pmino)
    end for
end function
```

The grid structure is initialized as a small grid first. Since its dimension (dim) may be too small to store all polyominoes, extra cells may be needed at any time. In order to achieve this, an enlargement step is applied. If extra cells are needed, grid is enlarged by some constant $c_{enlarge}$ and already placed polyominoes are placed in the enlarged grid without losing their relative locations, before the new placement occurs.

In order to place a polyomino in grid, first, an unoccupied grid cell must be found. The check starts from center point of the grid and continues by increasing distance from center. A polyomino fits in a location only if all the grid cells corresponding to polyomino cells are unoccupied.

Fig. 3.4 illustrates the result of polyomino packing on a sample graph. In the sample graph, A, B and C are complexes. Members at different depths are coloured differently for convenience.

Algorithm 11 Placing A Polyomino Into Grid

```
function putMino(pmino)
    while !tryPlacing(pmino) do
        dim ← dim + c_enlarge
        makeGrid(dim, currentMinoes)
    end while
    for all cell ∈ pmino do
        x =← cell.x + pmino.x
        y =← cell.y + pmino.y
        grid[x][y] ← occupied
    end for
end function
```

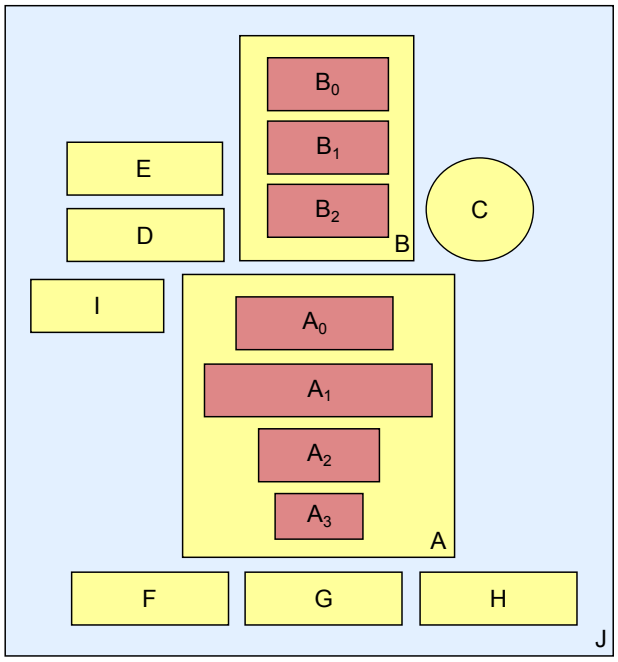

Figure 3.4: Polyomino packing result on a sample graph.

Initially, members of A and B are polyomino packed. Considering that all members of B have the same area size, one of them, which is B_1 in this case, is placed in the center and the other members B_0 and B_2 are placed around it.

Similarly, in order to pack the members of complex A, its members are sorted in decreasing order of their areas as A_1, A_0, A_2 and A_3. Then, A_1 is placed in the center, A_0 and A_2 are placed around it. In order to place A_3, an unoccupied cell is found, where the aspect ratio of A becomes as close as possible to desired aspect ratio. Finally, members of J are sorted as A, B, C, G, F, H, D, E, I and packed respectively.

3.1.2 Packing Non-Complex Members

CoSE uses gravitational forces to keep vertices close to the center as much as possible, but it may still not be sufficient to minimize the drawing area. Therefore, packing is performed on disconnected molecules to reduce the wasted area (Fig 3.2).

Considering that our layout algorithm supports complex member tiling, we make use of *dummy complexes* to pack disconnected molecules. This is achieved by grouping all disconnected molecules separately at each depth and assigning a dummy complex parent to them. This grouping should be performed at the very beginning of layout.

Fig. 3.5 illustrates tiling of non-complex members. In this figure, graph G is disconnected. C_1 and C_2 are two compound nodes, C_p is a complex node, $Z_1, \ldots, Z_8, A$ and B are simple nodes. In this graph, none of the nodes are connected to another node by an edge. Therefore, degrees of each node are 0. The graph manager stores 4 graphs, whose parent nodes are: root, C_1, C_2 and C_p.

C_p is the parent of A and B and its type is complex. Therefore, although A and B have no edges, considering that they have a complex parent, we do not refer them as "disconnected molecules". Hence, a dummy complex is created at

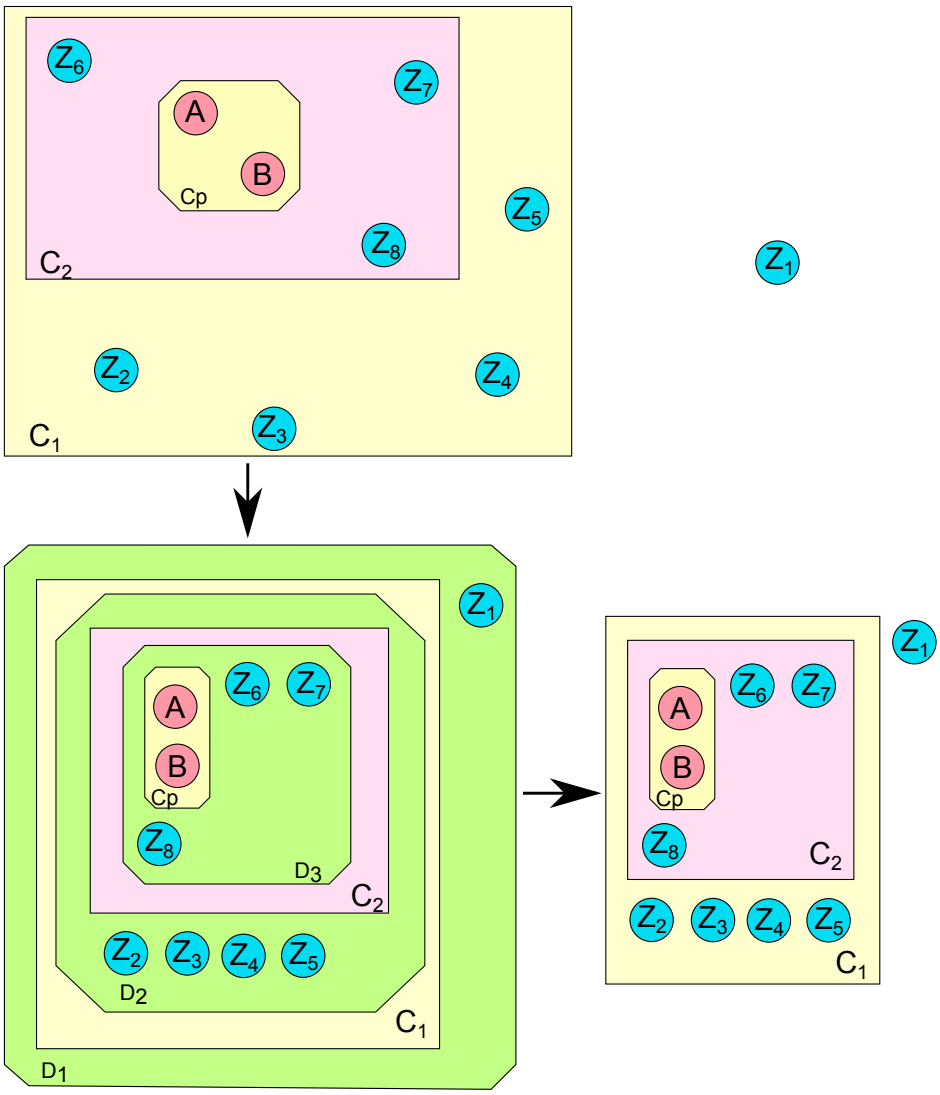

Figure 3.5: A dummy complex is created to group each disconnected molecules at each depth (large margins for dummy complexes are used to get a better understanding).

each level for each node group, except A and B. In this case, 3 dummy complexes are created:

- D_1 is created at the root level (depth = 0), stores C_1 and Z_1
- D_2 is created where depth = 1, stores C_2, Z_2, Z_3, Z_4 and Z_5.
- D_3 is created where depth = 2, stores C_p, Z_6, Z_7 and Z_8.

The algorithm packs the members of all complexes in the following order: C_p, D_3, D_2 and D_1. When members of a complex node is cleared out, its child graph is removed from the graph manager. Then, layout is performed with the remaining nodes. After layout is done, previously cleared children of complexes are reinserted into the graph manager. In the final phase, the dummy complexes are removed from the graph. Although the algorithm seems straightforward, there are some important cases to handle.

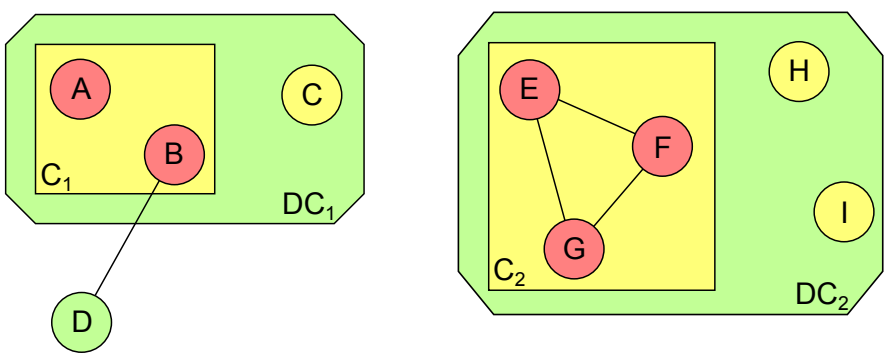

Figure 3.6: Two sample dummy complexes that have non-zero degree child nodes.

Fig. 3.6 illustrates the fact that, using node degrees for compound nodes is misleading. Therefore, for each compound node $v \in V$, we define a *graph degree* $GD(v) = y$, where y is the total number of edges of all child nodes of v. For example, $GD(C_1) = 1$ because of the edge between B and D. Similarly, $GD(C_2) = 3$. On the other hand, node degrees of C_1 and C_2 are 0.

If node degrees were used instead of the graph degrees, members of C_1 and C_2 would be cleared out from the graph before layout. As a result, their spring

forces would not be calculated and after repopulation, A, B, E, F and G would be placed incorrectly. To overcome this problem, if a compound node is encountered while identifying the disconnected molecules, its graph degree value should be used to get a properly tiled result.

Algorithm 12 Graph Degree Calculation of a Compound Node

```
1: function calcGraphDegree(node)
2:     degree ← 0
3:     if node has no child then
4:         degree ← node_degree return degree
5:     end if
6:     for all child ∈ node.childGraph do
7:         degree ← degree + node_degree + calcGraphDegree(child)
8:     end for
9: end function
```

3.1.3 Performing Compaction

After applying one of the packing algorithms explained above, compaction is applied to further reduce the wasted area.

In order to apply compaction, a visibility graph is constructed from the given set of nodes. A visibility graph is a *directed acyclic graph (DAG)*. It illustrates the visibility of nodes in different directions. For instance, in the case of bottom-to-top direction, an object on top can not "see" the object placed below it.

A node can see another node, if there exists a ray that intersects both of them without intersecting any other nodes in between. In order to find out whether or not there exists a ray between any given two nodes, a straightforward sweeping technique is used. Starting from the intersection area between p and q, walk on a line perpendicular to the desired direction. If a line can be found that does not intersect with any other nodes, the nodes are said to be visible to each other and the line segment between them is referred as a valid visibility edge. This visibility graph construction algorithm has a complexity of $O(N^3)$, where N denotes the number of objects.

Our algorithm applies compaction in two directions: first vertically and then horizontally. In order to find visibilities in different directions, each time the visibility graph should be constructed from scratch. After construction, using the DAG structure of visibility graphs, topological sort is applied to get the objects in order.

Algorithm 13 Compaction Procedure

```
function performCompaction(direction, nodes)
    visibilityGraph ←construct(direction, nodes)
    if visibilityGraph.edges is not empty then
        sortedNodeList ←topologicallySort(visibilityGraph.nodes)
        compactElements(sortedNodeList)
    end if
end function
```

Topological sort is defined on directed acyclic graphs and commonly used for task scheduling. The purpose of applying topological sort is to get an ordered list of nodes, where for each edge $e = (u, v)$ in the graph, u appears before v.

Compaction is performed by visiting all the nodes in the topologically sorted list. For each node u in that list, each incoming edge $e_i = (v, u)$ is checked. The node is translated by a distance of the shortest incoming edge length.

Fig. 3.7 illustrates compaction on a set of vertices. First, vertical visibility graph of given objects are constructed. Then, topological sort is applied and each node is labelled as its precedence value. Starting from the first element (0, 1, ..,6) each node is translated vertically, with respect to the empty space. After the translation is completed, horizontal visibility graph is constructed. Again, topological sort is applied, nodes are labelled respectively and horizontal compaction is performed.

The results show that, compaction does not improve the packing too much on compound graphs. Thus, in applications where running time is critical, compaction can be ignored.

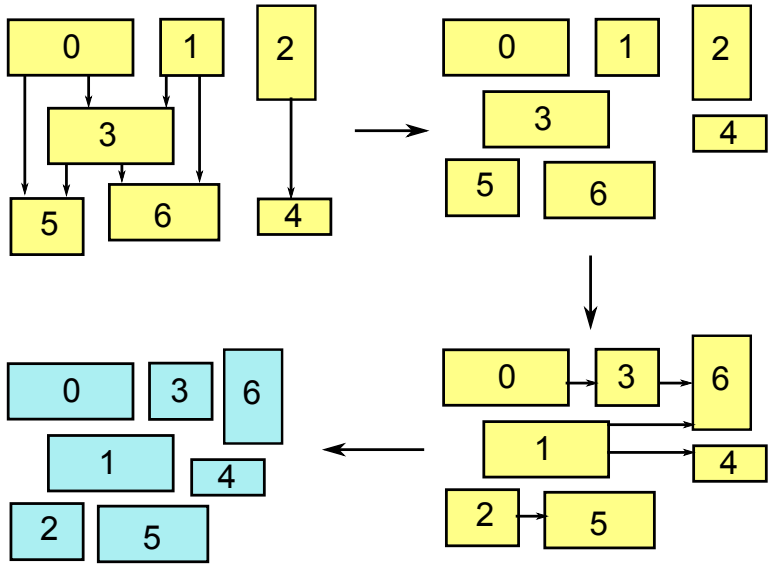

Figure 3.7: Vertical visibilities of a node set (top-left), vertical compaction result (top-right), horizontal visibilities (bottom-right) and result of horizontal compaction (bottom-left).

3.2 Handling Port Nodes

SBGN rules state that neighbours of a process node can only attach to the process from the port nodes placed vertically or horizontally on the opposite sides. Similarly, production nodes and substrate nodes should be grouped next to their corresponding port nodes to clearly visualize flow of the process. However, original CoSE algorithm will not respect this constraint as a general purpose layout algorithm (Fig. 3.8).

In order to satisfy this constraint in SBGN-PD layout, we introduce a new node type, called *SBGN process node*, which extends from the original CoSE node. We also introduce two port node types, where one of them is *input port* and the other one is *output port*. The ports are illustrated as small rectangular shaped nodes. Port nodes (p_{inp} and p_{out}) and their associated process node (p) are connected to each other by a new edge type, *rigid edges*. Unlike CoSE edges, a rigid edge does not exert spring force on its end nodes. Moreover, we introduce *dummy compounds* (p_{dummy}) to group each process node and its corresponding port nodes. Fig. 3.9 illustrates how SBGN-PD layout represents a process and

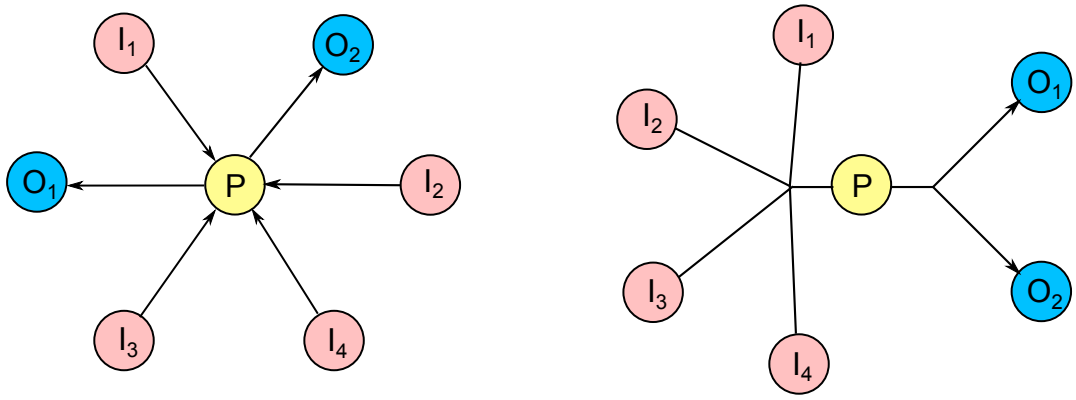

Figure 3.8: Process node handling in CoSE (left), how SBGN defines it to be (right)

its port nodes.

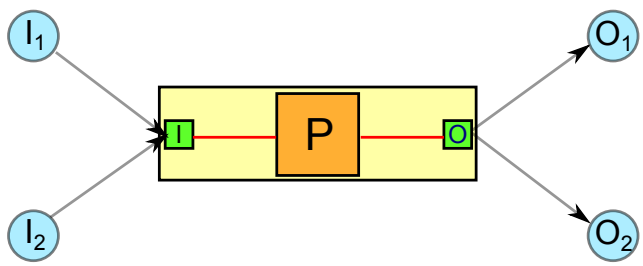

Figure 3.9: Illustration of a process node (orange) using dummy compound (yellow), port nodes (green) and rigid edges (red) in SBGN-PD layout.

At the very beginning of the algorithm, before the layout starts, a preprocessing is performed to create new nodes. After the nodes are created, connection of production and substrate edges between process nodes to the corresponding nodes revised as port nodes to corresponding nodes. Algorithm 14 illustrates this process.

The main purpose of using this structure is to keep those three nodes together and prevent port nodes to lose their relative positions. In order to ensure this, we use constraints for force calculations:

- Do not calculate spring forces for rigid edges.
- Do not calculate repulsion forces between a process and a port node.

- Do not calculate gravitation forces for process and port nodes.

Algorithm 14 Create Port Nodes

```
1: function CREATEPORTNODES(v, F⃗_disp)
2:     for all u ∈ V do
3:         if u.type is process then
4:             p ←NEWPROCESSNODE(u)
5:             p_inp ←NEWPORTNODE(input)
6:             p_out ←NEWPORTNODE(output)
7:             p_dummy ←NEWCOMPOUNDNODE(dummy_compound)
8:             CREATERIGIDEDGES()
9:             GROUPINTOCOMPOUND(p, p_inp, p_out, p_dummy)
10:            RECONNECTEDGES()
11:            p_dummy.owner ← u.owner
12:            remove u
13:        end if
14:    end for
15:    UPDATEBOUNDS(rootGraph)
16: end function
```

Because of the fact that CoSE does not allow force calculations for two nodes that do not belong to same owner graph, using the constraints above, it is ensured that repulsion and gravitational forces for the process and port nodes are calculated as 0. On the other hand, due to the substrate and production edges, spring forces are applied on the port nodes. However, the spring forces acting on port nodes are transferred to their dummy compound, before moving the objects. Therefore, moving those nodes are not going to disturb the relative positioning. Conversely, the forces exerted on the dummy compound are calculated as usual. Since, a compound propagates forces to its children, moving a compound means moving a process node and its corresponding port nodes. As a result, the port nodes and their associated process nodes move during the layout process without losing their relative positions.

3.3 Orienting Products and Substrates

SBGN supports a process node and its port nodes to be placed horizontally or vertically. However, CoSE does not support either ports or their rotation. In order to add a rotation feature, we introduce *rotational forces*.

Rotational forces are exerted only on process nodes (or dummy compounds in a roundabout way), by the corresponding production and substrate nodes. The main idea behind applying rotation is that, if a group of nodes persistently pull their associated process node in a certain direction, a decision is made to change the orientation of the compound node by either applying a 90-degree or 180-degree rotation. There are four possible orientations for a process node: right-to-left, left-to-right, top-to-bottom, and bottom-to-top, where the two parts of the orientation define the relative positions of input and output ports with respect to the associated process node, respectively.

Rotational force acting on a process node is calculated by using adjacent nodes of the corresponding port nodes. For each process node p in the graph, magnitude of the net rotational force is calculated by:

$$F_{rot}(p) = \sum_{i=1}^{n_c} angle(c_i, p_{inp}) - \sum_{j=1}^{n_p} angle(p_j, p_{out}) \tag{3.4}$$

where n_c denotes the total number of substrate nodes and n_p denotes the total number of production nodes for process p.

Fig. 3.10 illustrates rotational force calculation on a sample process node. Angles between substrate, production nodes and their corresponding port nodes are calculated. Applying the equation above, rotational force acting on process P is calculated as $F_{rot}(P) = (d_1 + d_2) - (d_3 + d_4)$.

Angle between a production or substrate node and its corresponding port node is calculated by:

$$\theta = \arccos \frac{|\vec{v_1}.x| \times |\vec{v_2}.x| + |\vec{v_1}.y| \times |\vec{v_2}.y|}{|\vec{v_1}| \times |\vec{v_2}|} \tag{3.5}$$

where $\vec{v_1}$ is the vector from target input point (t_{inp}) to input port (p_{inp}) and $\vec{v_2}$ is the vector from target node (A or B) to p_{inp}. It is also similar for the output nodes.

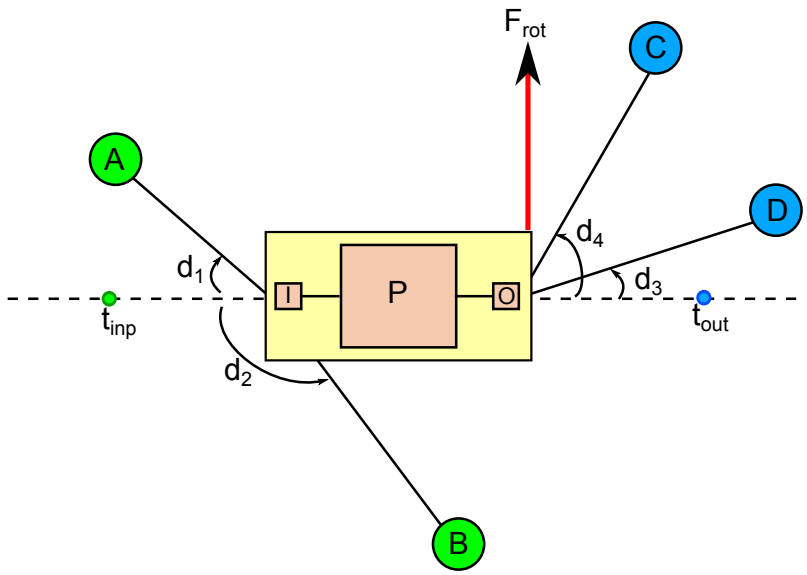

Figure 3.10: Rotational force acting on a sample process node.

An important point in angle calculation is that, absolute values of the angles should not be used and the estimated angle should not exceed 180 degrees. For instance, in Fig. 3.10, A and B pull the compound in opposite directions. In order to illustrate this difference, signed angle values should be used. The compound is intended to rotate clockwise due to the force applied by A, whereas B, C and D cause a counter-clockwise rotation.

Left test is performed to find the sign of an angle. This test indicates whether a given point is located on the left or right of a line segment. Given a target point p and a line segment l whose starting point is defined as l_s (t_{inp} or t_{out} in Fig. 3.10) and end point as l_e (p_{inp} or p_{out} in Fig. 3.10), left test is performed as follows:

$$(l_e.x - l_s.x) \times (p.y - l_s.y) - (l_s - l_e) \times (p.x - l_s) > 0 \rightarrow left\ turn \tag{3.6}$$

If the net rotational force acting on a process node exceeds a predefined rotation threshold, the associated compound can be rotated 90 degrees in clockwise or counter-clockwise direction, depending on the sign of the net force.

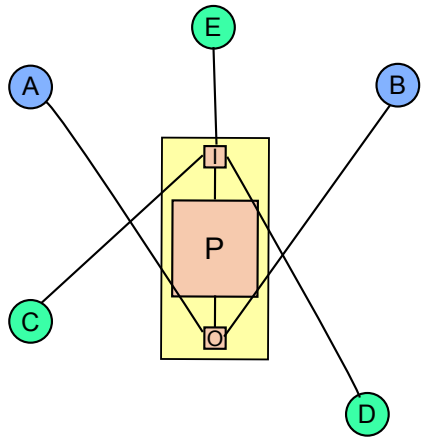

Figure 3.11: A sample layout, where net rotational force fails to detect 90-degree rotation.

However, supporting only 90-degree rotations may cause some problems. Fig. 3.11 illustrates the fact that 90-degree rotation may not always work and we need a support mechanism for detecting 180-degree rotations. 180-degree rotation is also referred as a swap operation, since we simply swap the port node locations of a process node. Swap operation has a higher priority over 90-degree rotation. Swap availability is detected by checking all angles of adjacent nodes. If the total number of obtuse angles is higher than a predefined swap constant, then the process ports are swapped.

Fig. 3.12 illustrates the sign scheme used for calculating angle values. This reference card should be interpreted as follows: for left-to-right case, if a consumption node resides on the left of its port node, the force is going to have a positive value. Similarly, if a consumption node is placed on the right area, the force is going to have a negative value.

Rotation and swap operations are applied while moving the nodes. Rotational forces are summed up for a number of iterations. Once every pre-defined fixed number of iterations, the sum is normalized and each process node is checked for whether or not they should result in rotation. After finding all process nodes that

desire to rotate, one of them is randomly selected and rotated.

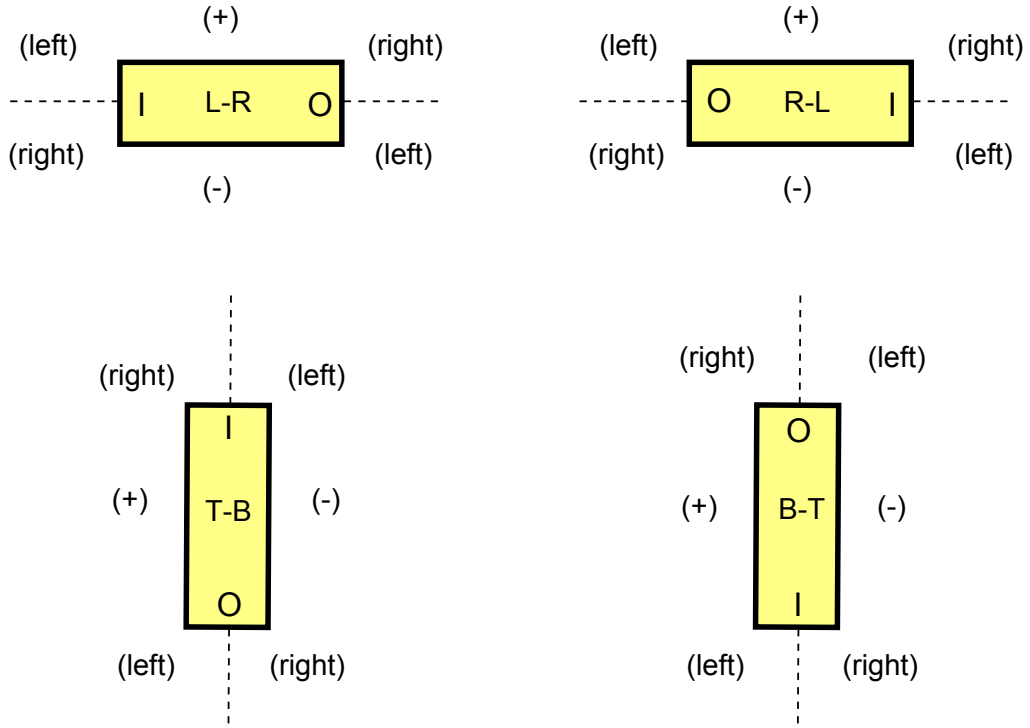

Figure 3.12: Reference card for rotational force signs for four possible orientations: left-to-right, right-to-left, top-to-bottom, and bottom-to-top.

Algorithm 15 Moving Nodes and Applying Rotation

```
function moveNodes()
    rotList ← ∅
    for all p ∈ processNodes do
        transferForces(p)
        resetForces(p)
        resetForces(p_inp)
        resetForces(p_out)
    end for
    if phaseNumber is 2 ∧ totalIterations % rotationPeriod is 0 then
        for all p ∈ processNodes do
            if isRotationAvailable(p) then
                rotList.add(p)
            end if
        end for
        randomlyRotate(rotList)
    end if
    CoSE.moveNodes()
end function
```

Chapter 4

Results

SBGN-PD layout is implemented using an open source project called *Chisio Layout (ChiLay)*, which is a graph layout framework with a number of layout algorithms implemented in JAVA [39].

Tiling and Polyomino Packing algorithms have been tested using Chisio Editor (ChiEd) [40]. ChiEd, based on ChiLay, provides a random graph creation option with specified values. Thus, we created some random graphs with different number of nodes with the following parameters:

- Minimum node dimensions: (10 x 10), maximum node dimensions: (200 x 200)
- Compound depth: Since we only create complex nodes and simple molecules, this parameter indicates the maximum depth of a complex node. This parameter has been set to 3, meaning that, a complex may have at most two complex parents.
- Number of siblings: This parameter indicates how many child nodes a node may have. It has been set to 5.
- Branch factor: 0.4

- Number of nodes: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 120, 150, 200, 300, 400

In Fig. 4.1, comparison of those two algorithms can be found on their adjusted fullnesses. It is seen that, polyomino packing has a clear advantage over tiling when the number of nodes is more than 60. Below that threshold, tiling gives more compact results.

On the other hand, when compaction is applied to polyomino packing, although a slight increase is observed, there is no significant increase. Tiling with compaction has not been included in the comparison because there is almost no difference to observe.

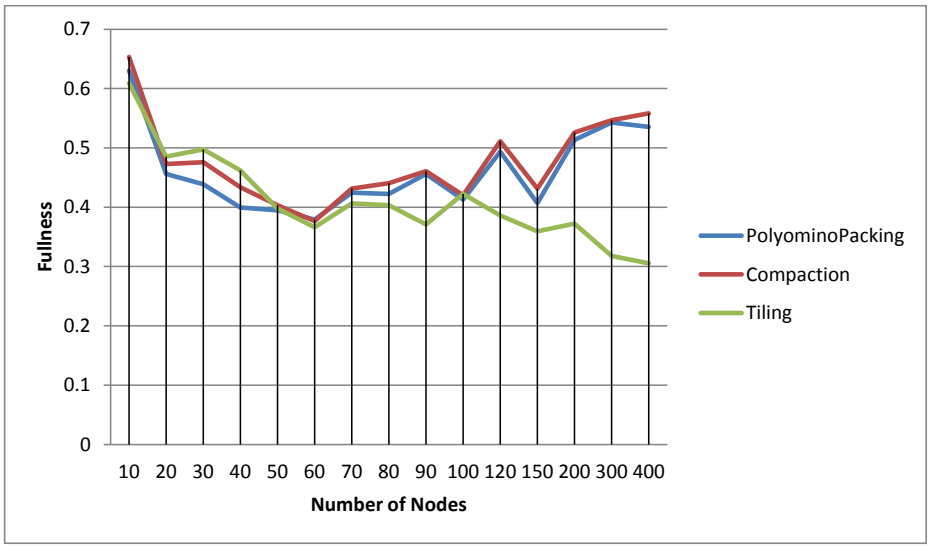

Figure 4.1: Comparison of tiling, polyomino packing without compaction and polyomino packing with compaction

The running time complexities of these two algorithms are not very relevant, because the process diagrams that we get do not have many complex members to pack. Considering that, tiling produces graphs with less wasted area for small graphs and its time complexity is smaller than polyomino packing (see section 2.5), we use tiling as our default packing method.

There are a number of parameters to be tuned in the algorithm. Each parameter has been tested with a number of values to find the most coherent set of

values. We performed the parameter tuning on a mixture of randomly selected different types of graphs. The test graphs were found using *Pathway Commons Network Visualizer (PCviz)* [41] and the graph information in SBGN-ML format is also queried using *Pathway Commons*' services [1].

The tests have been performed for the following seven parameters:

- Approximation Distance (ad): This parameter is used for location approximation of production, substrate and effector nodes. ad represents the radius of the approximation circle, centred at the process node. The corresponding nodes are randomly placed within this circle.
- Rotation Period (rp): Indicates the number of iteration count between each rotation operation.
- Angle Tolerance (at): This parameter is used to decide whether or not to accept neighbour node of a port node as "properly oriented". The wider the angle, the better the layout results are expected to be but actually orientation of production and substrate edges is going to be poor. Therefore, we do not accept values higher than 100 degrees.
- 90-degree Rotation (c_{90}): It is the minimum amount of total net force required to rotate a dummy compound node by 90 degrees.
- 180-degree Rotation (c_{180}): Indicates the minimum percentage required to swap port nodes of a process node, where the percentage value is calculated by dividing number of obtuse-angled production and substrate edges to the total number of edges of both port nodes.
- Approximation Period (ap): Indicates the number of iteration count between each location approximation operation. Prime numbers have been used to minimize the chance of finalizing layout right after the approximation. If this happens, the nodes are not going to be able to pull each other.

- Phase 1 Maximum Iteration Count (i_{p1}): This parameter denotes the maximum number of phase 1 iterations. Considering that, original CoSE algorithm has a maximum iteration count of 2500, we want to keep this value smaller.

The tests were performed on a number of random graphs. We tested each parameter by a set of numbers depending on our observation. We simply run the tests for each combination of those values. 10 runs have been performed for each graph with each value combination. The average of those 10 tests is reported as the average result, where the result indicates the percentage of edges that needs to be properly oriented to all edges in the graph, except for rigid edges since they were temporarily created.

Fig. 4.2 illustrates the result of testing each possible group of values. For each set of values, we sum the average results of each individual graph (s), where n_g denotes the number of graphs and report the average (s/n_g) as the average result.

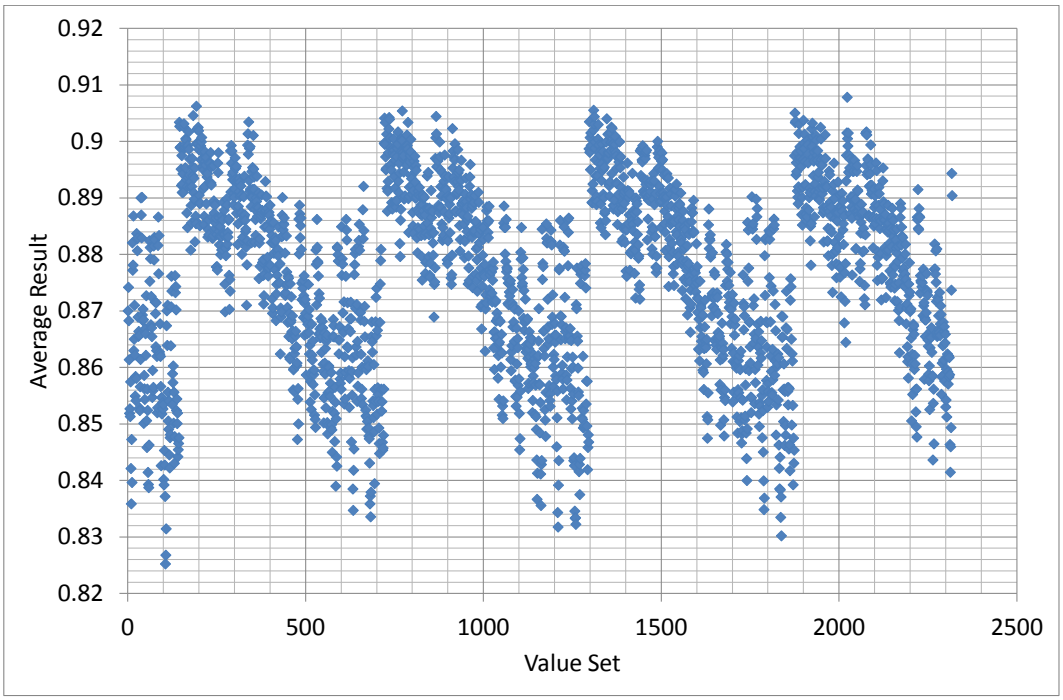

Figure 4.2: Percentage of properly oriented edge count to total number of edges to be oriented with different value sets.

Examining this figure, the most coherent group is found as $ad = 50$, $at = 100$, $c_{90} = 70$, $c_{180} = 0.5$, $rp = 2$, $ap = 211$ and $i_{p1} = 200$, resulting in a success rate about properly orienting production and substrate edges as 0.9 over 1.0.

The remaining charts are created by changing one parameter's value, where the rest of the parameters have fixed values at their optimal values found above.

Approximation distance has been tested by using 5, 20, 30, 50, 75 and 100. In Fig. 4.3, it is seen that when the approximation point is close to tightly connected nodes, better results are obtained. Increasing the distance causes low separation.

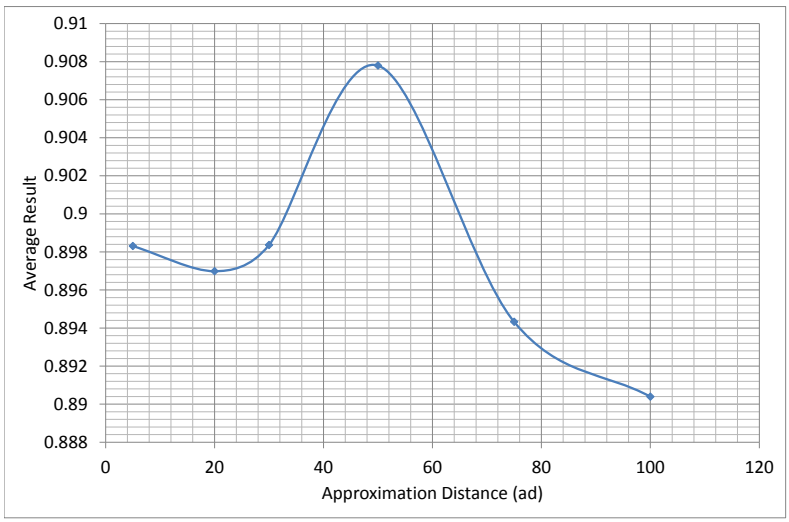

Figure 4.3: Effect of approximation distance on results.

Rotation period has been tested by using 1, 2, 5, 10, 20 and 30 as it is seen in Fig. 4.4. The plot indicates that if the rotations are not performed frequently, the results get worse as expected. Also, applying the rotations at each iteration causes problems. Because, the aim is to observe the insistence for rotation in one direction and then rotate. It is seen that, observing the rotational forces for 2 iterations is enough and is gives best results. Hence, the value is selected as 2.

Angle tolerance has been tested by using 90, 95, 100 and 105 as can be seen in Fig. 4.5. It is obvious that if the angle tolerance increases, the results are going to seem better on numerical values. However, there will be low separation due to allowing nodes to be placed far away from their associated port nodes. The tests

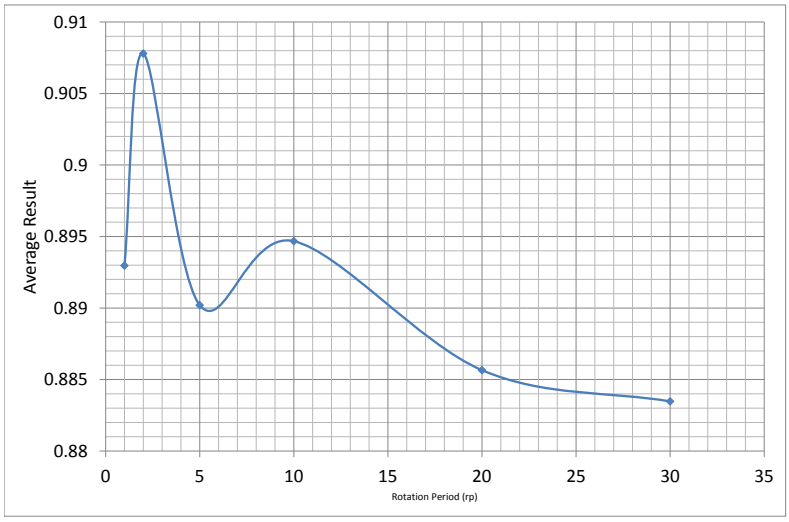

Figure 4.4: Effect of rotation period on results.

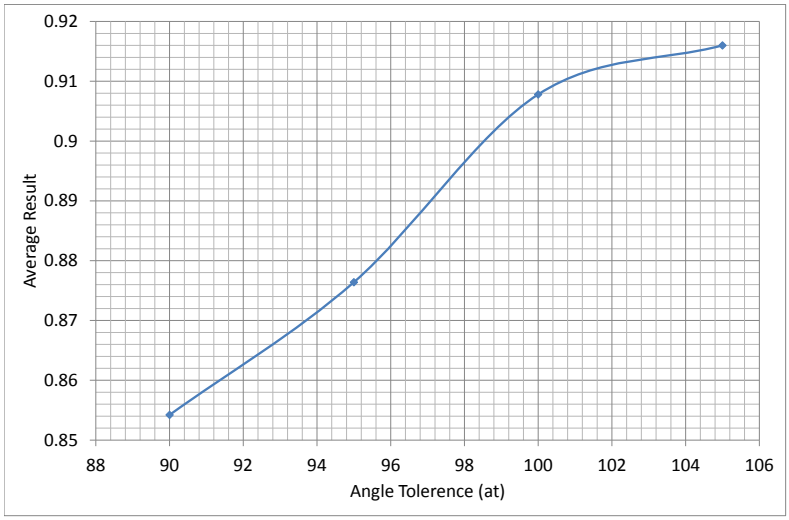

Figure 4.5: Effect of angle tolerance on results.

have been performed especially to visualize its effect. We tolerate production and substrate nodes that make at most 100 degrees with their associated port nodes.

90-degree rotation constant has been tested using 40, 60, 70, 90 and 100 as it can be seen in Fig. 4.6. The best results are obtained when a process node and its associated port nodes are rotated when the net force acting on them is higher than 70. However, if the rotation is performed almost for any values of the net rotational force, unnecessary rotations are applied. Similarly, if the expected net force is high, necessary rotations are neglected.

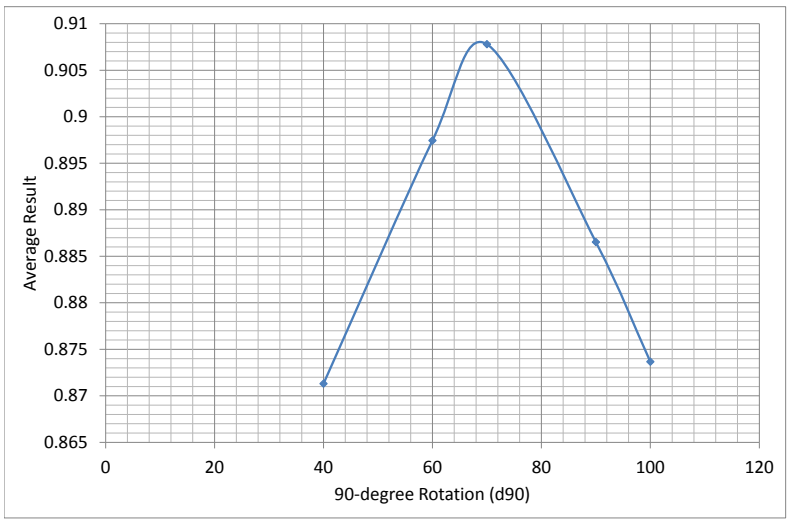

Figure 4.6: Effect of 90-degree rotation constant on results.

180-degree rotation constant has been tested using 0.4, 0.5, 0.6, 0.7, 0.8 as it can be seen in Fig. 4.7. If the process node is rotated when the percentage of its obtuse-angled edges to all edges is around 0.5. Therefore, c_{180} has been set to 0.5. Similar to c_{90} case, small values cause unnecessary swap operations and high values neglect the necessary ones.

Approximation period has been tested using the values: 101, 211, 307, 401, 503 and 701. Fig. 4.8 illustrates the fact that, changing the locations of substrate, production and effector nodes by force too frequently or too rarely decreases the quality of results. Thus, the parameter has been set to 211.

Maximum number of first phase iterations has been tested by using 100, 200,

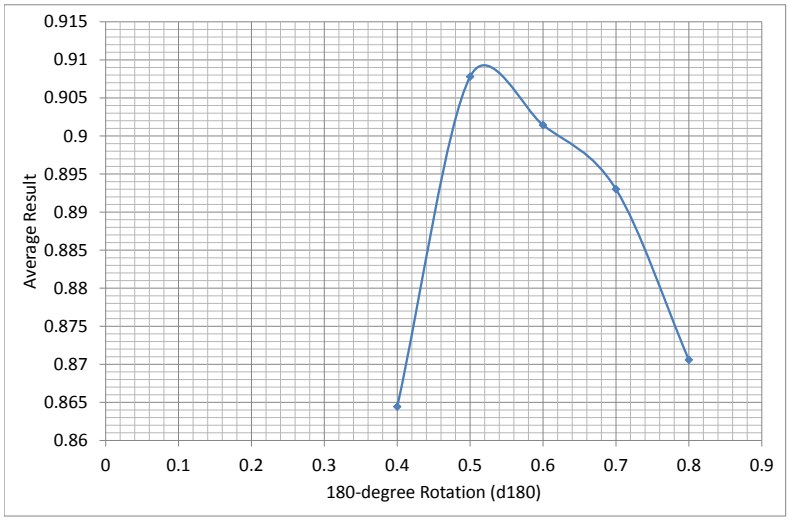

Figure 4.7: Effect of 180-degree rotation constant on results.

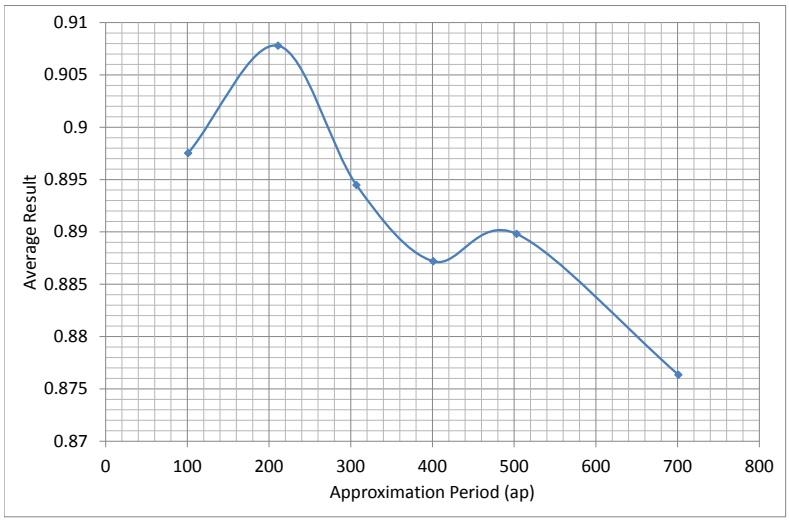

Figure 4.8: Effect of approximation period on results.

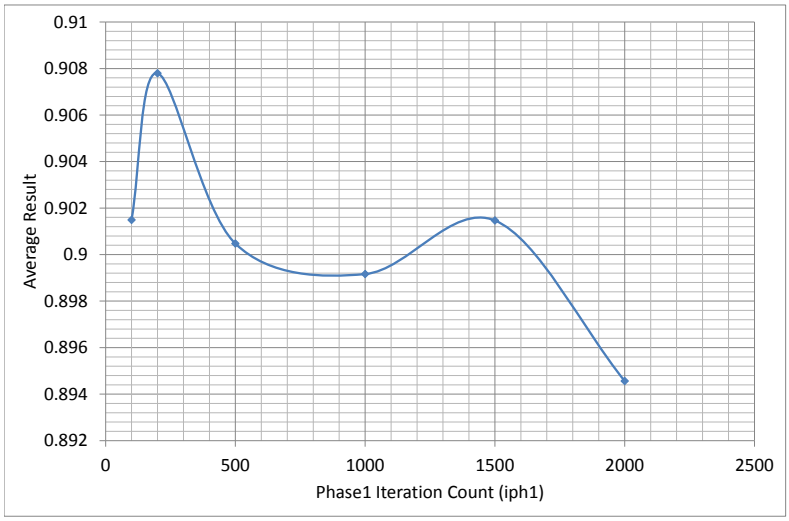

Figure 4.9: Effect of phase 1 maximum iteration count on results.

500, 1000 and 1500 as can be seen in Fig. 4.9. The plot indicates that when this parameter has a value around 200 iterations, the best results are obtained.

The results show that properly orienting substrate and production nodes is successful and has a clear advantage over CoSE. Fig. 4.10 illustrates the comparison between those two algorithms, where the percentage of properly oriented edges to total number of edges is reported as the average result.

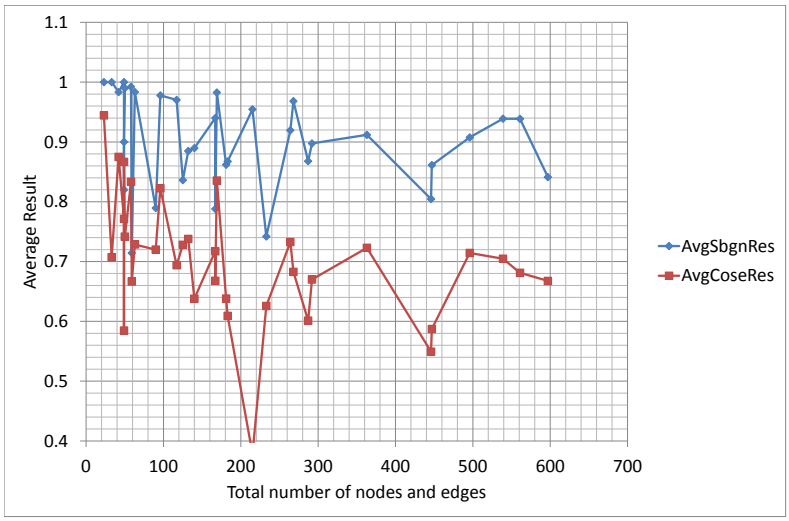

Figure 4.10: Comparison of CoSE and SBGN-PD average results.

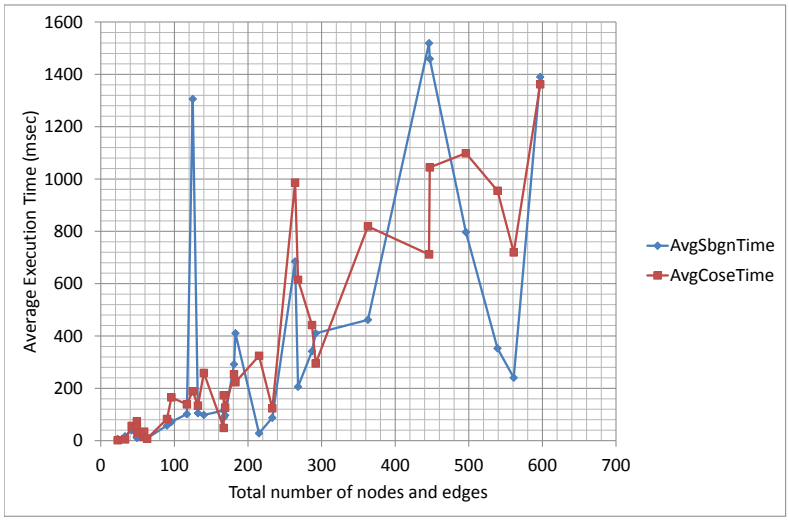

Figure 4.11: Comparison of CoSE and SBGN-PD average execution times.

In terms of execution times, SBGN-PD gives similar results to CoSE results for small graphs. However, as the iteration count required to complete phase 2 increases, the number of rotation operations increase and SBGN-PD layout requires more time to successfully orient the edges Fig. 4.11.

Considering the nested structure of the graph, it is better to examine the effect of node and edge count on the result by dividing the graphs into two types: compound and non-compound. Non-compound graphs are referred to the ones that do not include any complex or compound members (i.e., no nested structures). Compound graphs correspond to the opposite, respectively.

Fig. 4.12 shows the relation between total node and edge count on the average result for non-compound graphs. As expected, for small graphs, the algorithm is pretty successful and the success rate decreases as the graph gets bigger and more complex. However, almost all process description maps include complex and compound structures. Therefore, the number of non-compound graphs used for tests is small due to the difficulty of finding original non-compound graphs.

Fig. 4.13 illustrates the average execution time for non-compound graphs, respectively. The execution time is increasing rapidly as the graph gets more

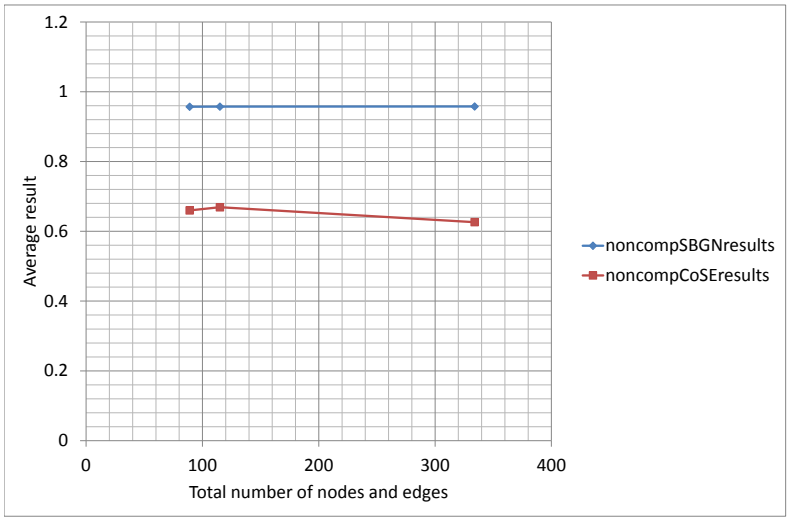

Figure 4.12: Effect of (N+E) on the average result for non-compound graphs.

complex. The slope of this line is calculated as 1.91 by using log-log plot. Log-log plot is used to find the characteristic of a line of the form $F_x = c_0 \times x^m$ and slope is calculated as follows:

$$m = \frac{\log(F_1/F_0)}{\log(x_1/x_0)} \tag{4.1}$$

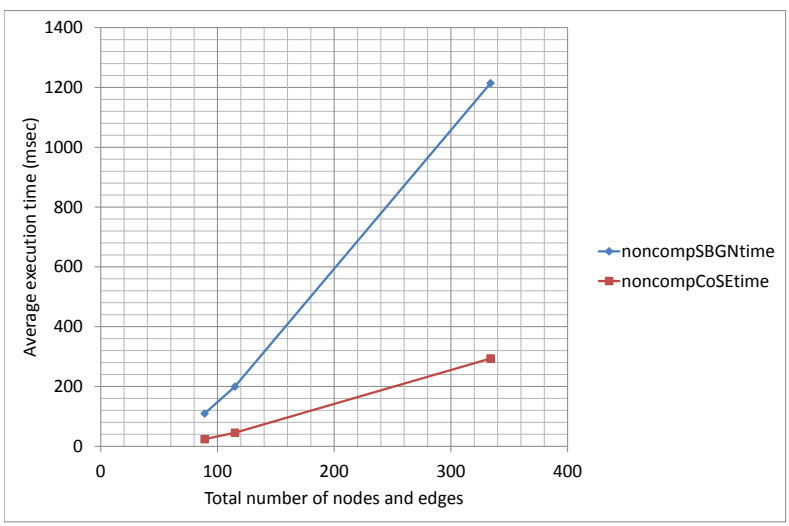

Figure 4.13: Effect of (N+E) on average execution time for non-compound graphs.

On the other hand, Fig. 4.14 illustrates the effects of total number of node and edge count on the average result, for compound graphs. Actually, compound

depth and branching factor affects the success of the algorithm. However, the result is not very clear as the graphs are selected randomly from Pathway Commons' database. It seems that the success rate is relatively small for the graphs that have more compounds with tighter relations to other nodes.

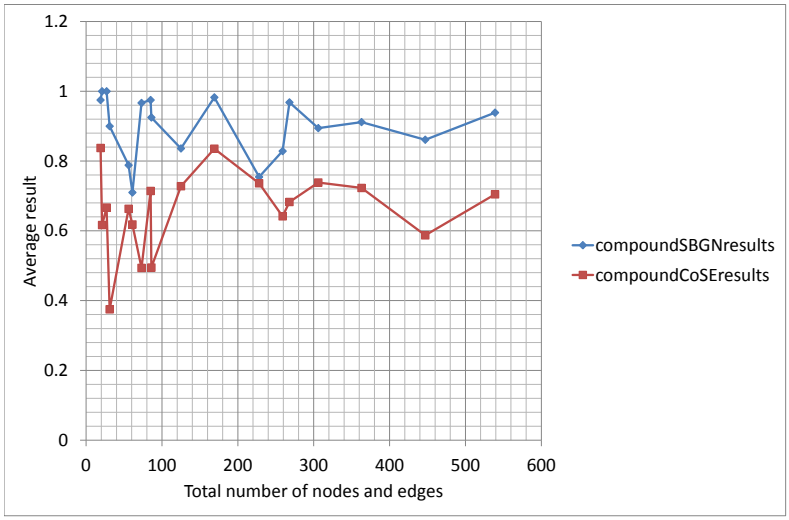

Figure 4.14: Effect of (N+E) on the average result for compound graphs.

It is similar for the average execution time (Fig. 4.15). The execution time has an increasing behaviour as we see in the non-compound graph case. However, some graphs may have less number of compound depths or more number of molecules to be packed, causing less lower execution time.

SBGN-PD layout uses tiling as its default packing method. Since, we do not use compaction with tiling, time complexity of packing operation is $O(VlogV)$, where V denotes the number of objects to be packed. In the worst case, all nodes are required to be packed. However, principally, this causes all members to be packed and cleared from layout, which means no forces are going to be applied. Considering that this packing operation is performed only once, at the very beginning of the layout, complexity of overall algorithm becomes $O(VlogV)$.

Creation of port nodes only requires a linear pass over all process nodes and it is only performed once. Location approximation for production and substrate nodes are performed only at some certain iterations. It takes a linear pass on

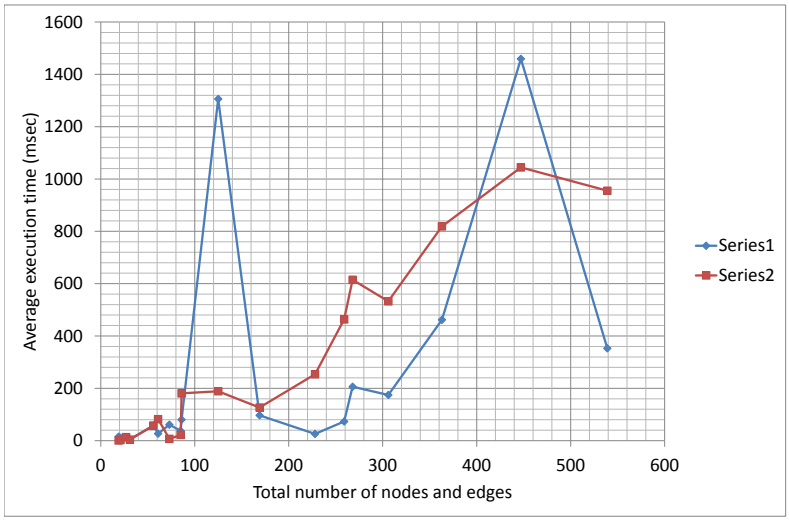

Figure 4.15: Effect of total number of nodes and edges on the average execution time for compound graphs.

process nodes to find all production, substrate and effector nodes. This approximation has a complexity of $O(c_0V)$, where c_0 is a constant and calculated as $c_0 = i_{ph2}/ap$.

In order to calculate rotational forces, all substrate and production edges are traversed. Therefore, this calculation requires $O(E)$ time and does not change the overall complexity. Hence, time complexity required by SBGN-PD layout is also similar to CoSE's and equal to $O(|E| + |V|^2)$.

4.1 Evaluation

SBGN-PD layout has two main promises: packing complex members, disconnected molecules and properly orienting production and substrate nodes.

Sample results for packing algorithms on a randomly created graph using ChiEd can be found in Fig. 4.16. This sample graph has 300 nodes and compound graph parameters have been set as the earlier introduced parameters.

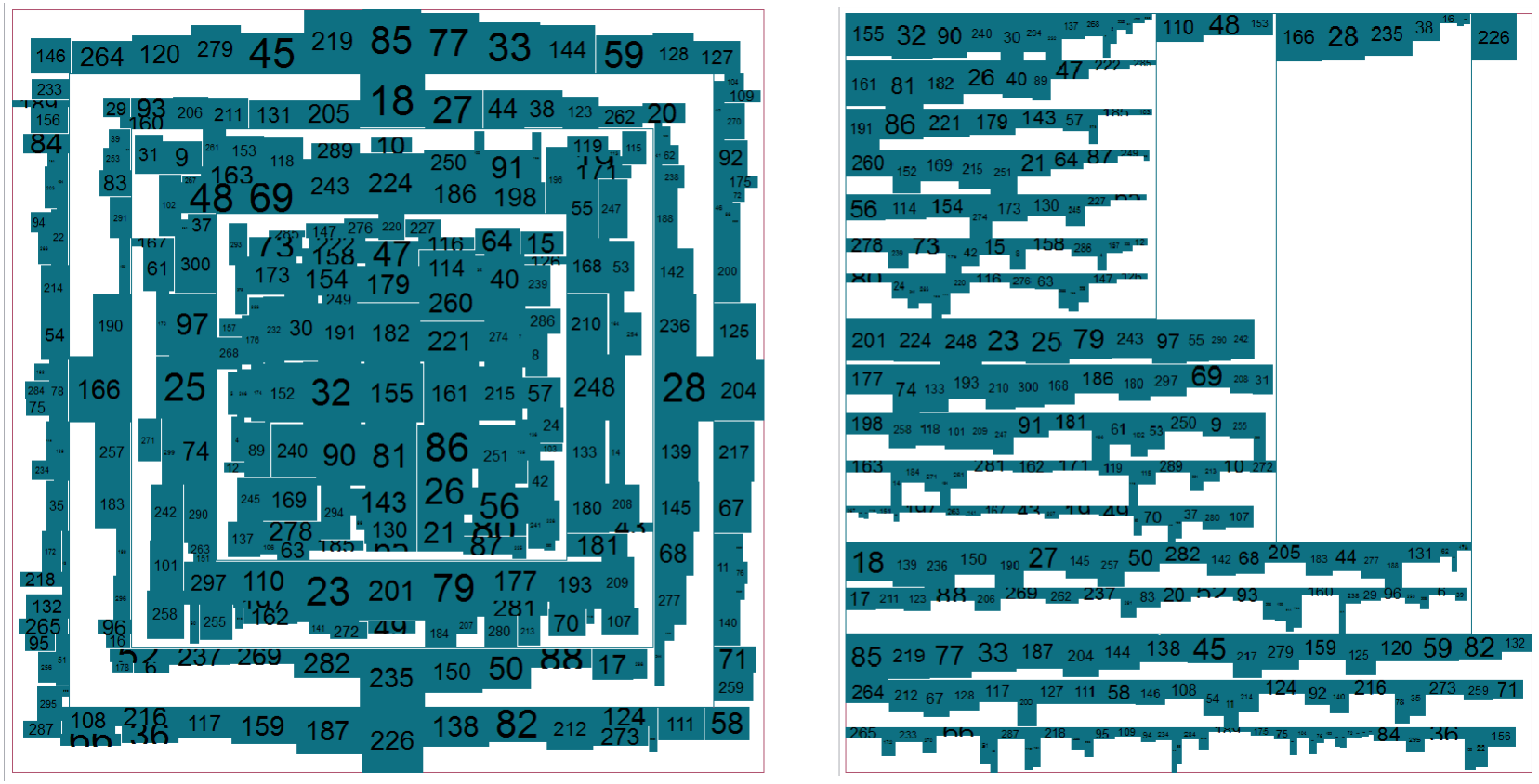

Figure 4.16: Comparison of polyomino packing (left) and tiling (right).

It is seen that both methods have been successfully implemented and can properly perform nested rectangle packing. This figure illustrates the fact that polyomino packing gives more compact results when the graph is huge. Please note that, tiling result (right) is scaled down to give a better view.

Figures 4.17 through 4.24 are some sample graphs laid out using SBGN-PD layout and visualized using SBGNViz, which is a specialized visualization tool developed for SBGN process description maps.

Results for orientation of production and substrate edges seem promising. The sample graphs are obtained using ChiBE, which is a visualization and free editing tool for pathway models represented by the BioPAX format using SBGN Process Description language [20]. In order to provide understandable results, relatively small graphs are chosen. Fig. 4.17 illustrates a successful layout result.

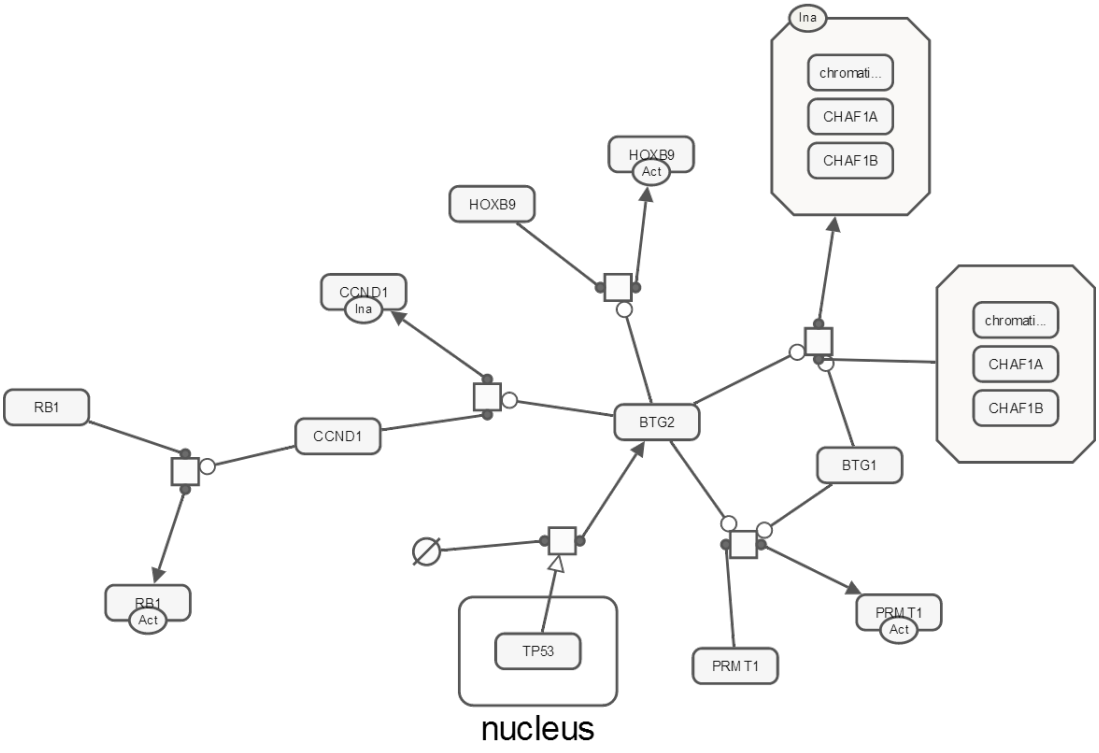

Figure 4.17: Btg family proteins and cell cycle regulation network laid out using SBGN-PD.

Fig. 4.18 also illustrates a successful layout operation on a real network. This sample network has a success rate of 1, which means that all of the process and substrate edges on this graph is properly laid out by making use of four different process node orientations. Additionally, the complex members are successfully tiled.

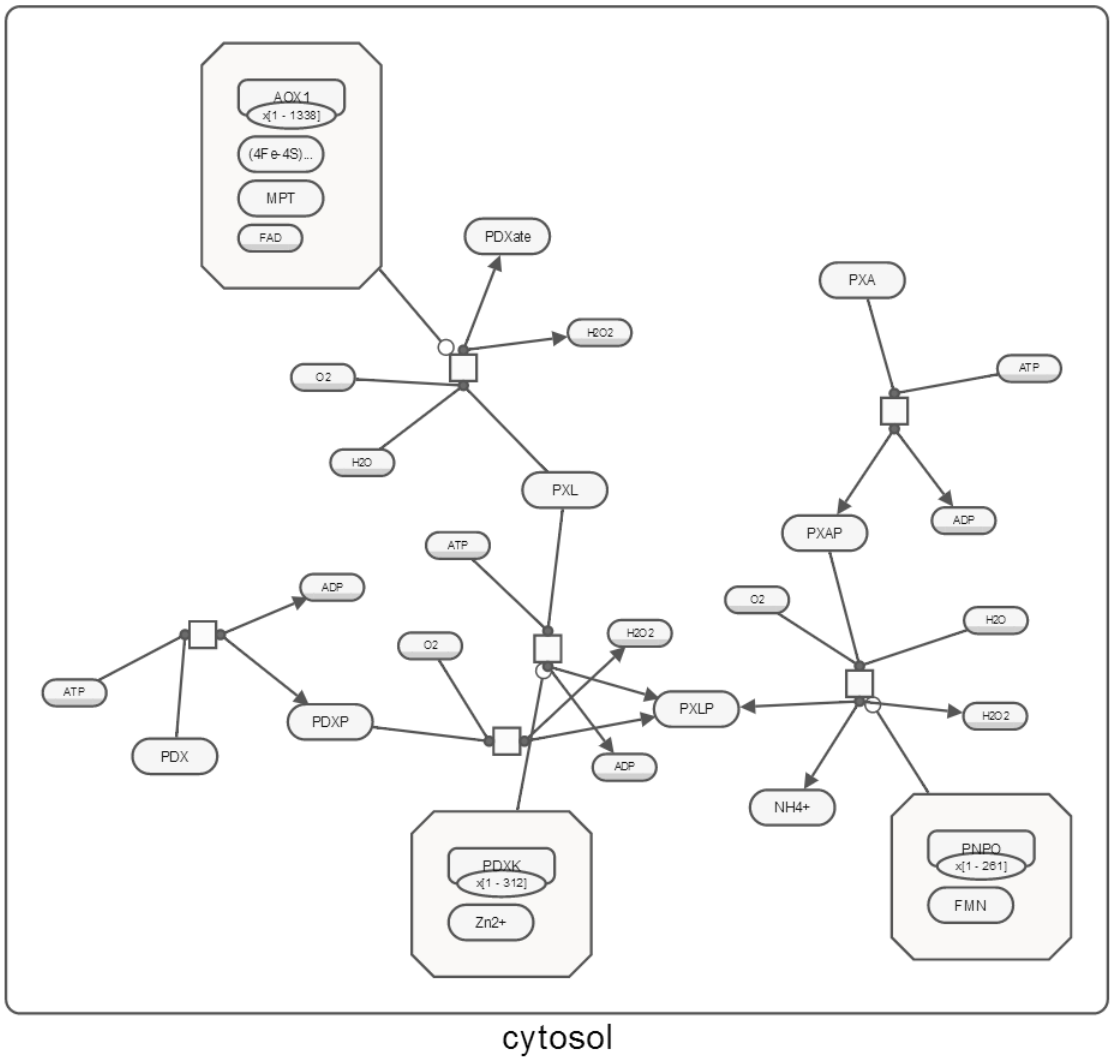

Figure 4.18: Vitamins B6 activation to pyridoxal phosphate network laid out using SBGN-PD.

The layout success does not depend only on the node count of the graph, but also the graph density. For instance, Fig. 4.19 illustrates the fact that, although there are a small number of nodes in a process, high connectivity makes it almost impossible to successfully separate the edges. Additionally, this figure also

illustrates the reason of using a graph degree. Although nucleoplasm node has 0 degree, it is not tiled with the other two nodes, due to its graph degree being greater than 0.

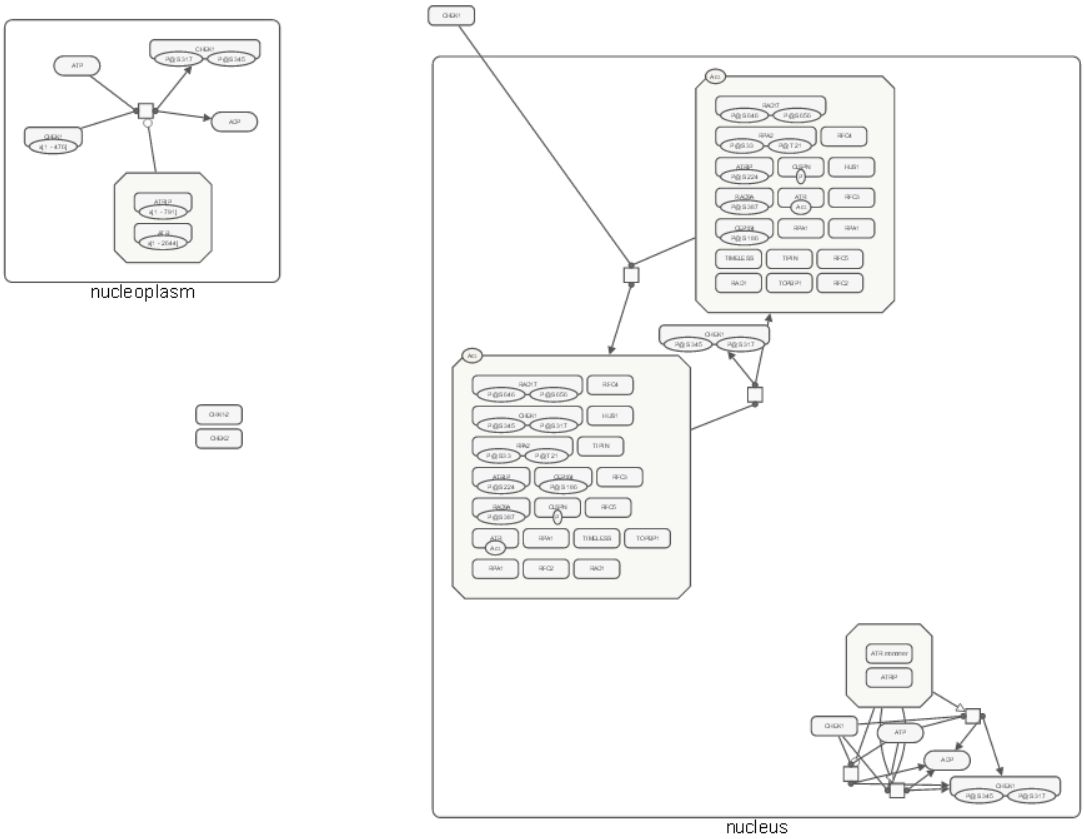

Figure 4.19: Process between ATRIP and CHEK1 laid out using SBGN-PD.

The following figures, Fig. 4.20, Fig. 4.21 and Fig. 4.22 are some graphs laid out using SBGN-PD. Those graphs illustrate the fact that, some substrate and production nodes of a process node may be placed in another compound node. During layout, the location of this compound node is determined with respect to the forces acting on it. Those additional forces may disrupt the proper orientation. This is a typical example, where multiple conflicting constraints can not be satisfied by any layout algorithm.

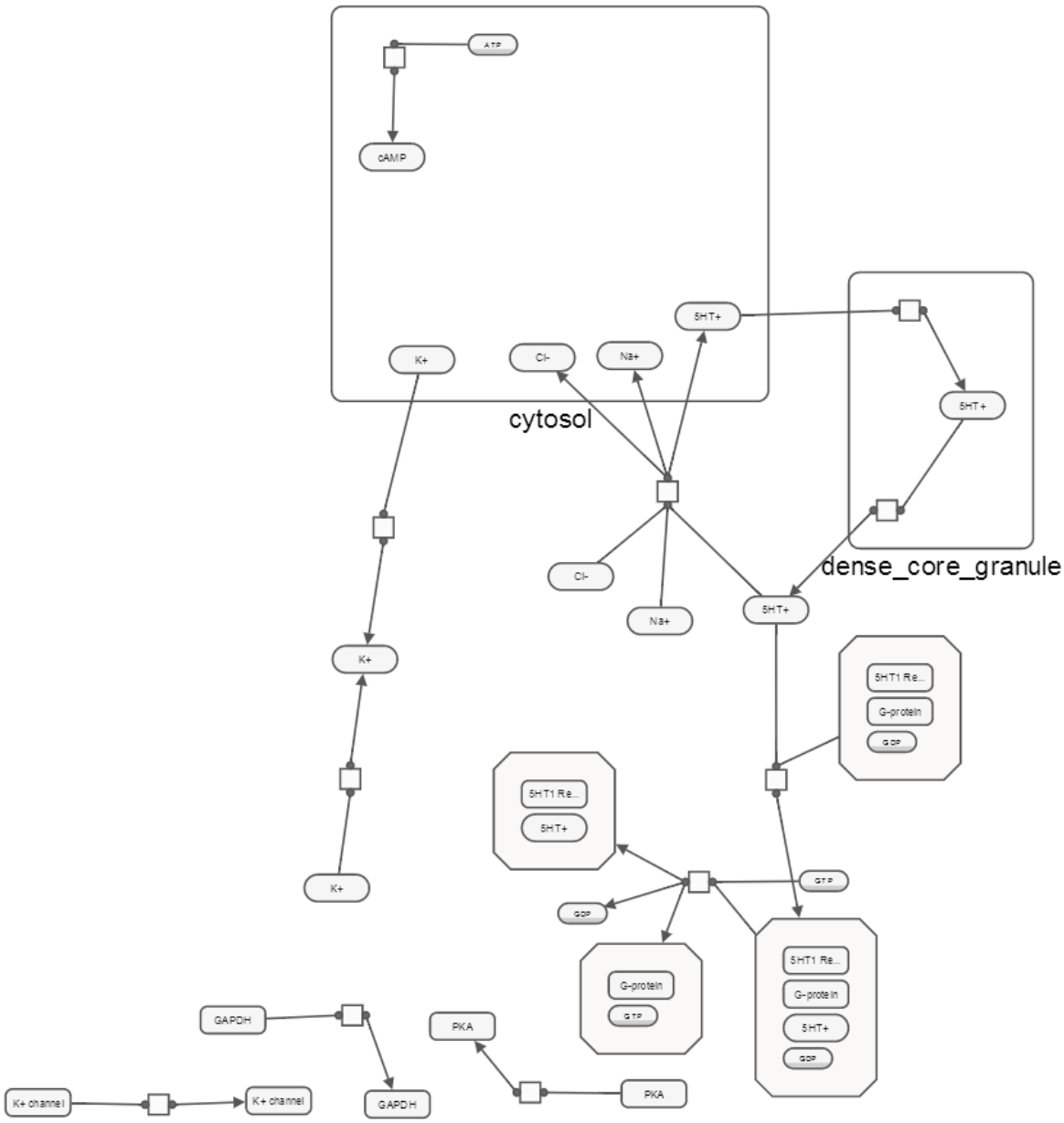

Figure 4.20: 5HT1 type receptor mediated signalling pathway laid out using SBGN-PD.

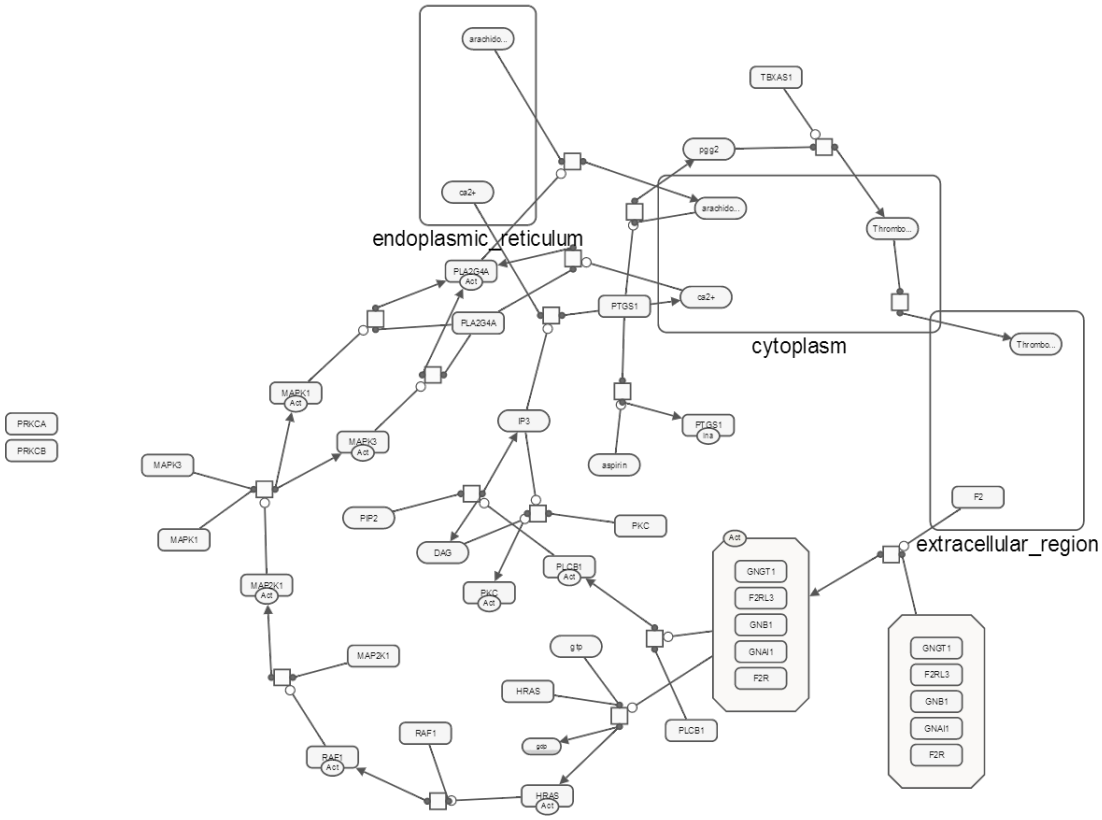

Figure 4.21: Aspirin blocks signalling pathway involved in platelet activation laid out using SBGN-PD.

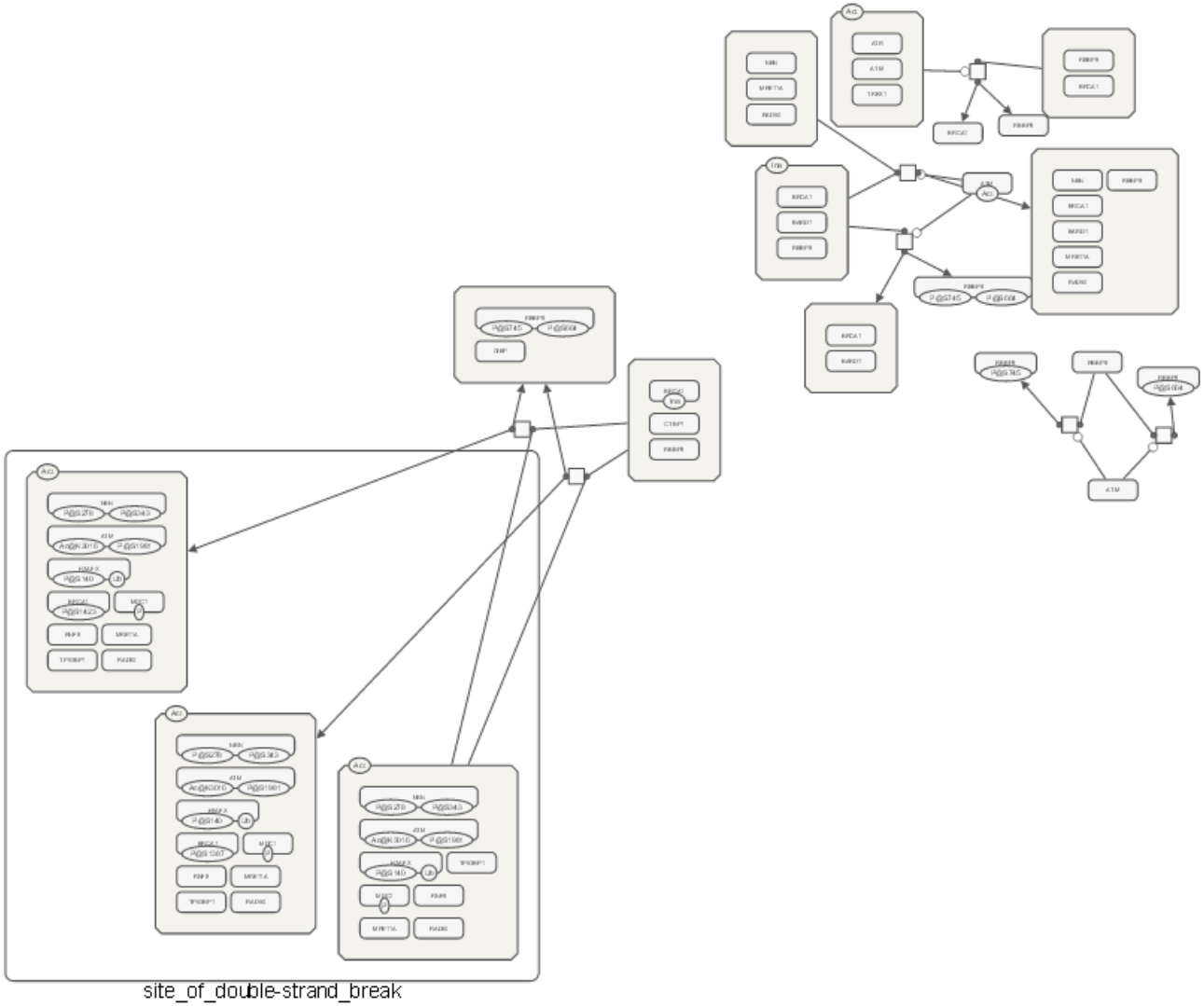

Figure 4.22: Process between ATM and RBBP8 laid out using SBGN-PD.

Fig. 4.23 and Fig. 4.24 illustrate the fact that if the product or substrate nodes of a process belong to another compound, our algorithm does not apply location approximation for the nodes that are owned by the other compound to avoid compound-compound collisions. Therefore, the layout produces long edges and the edges may not be properly oriented.

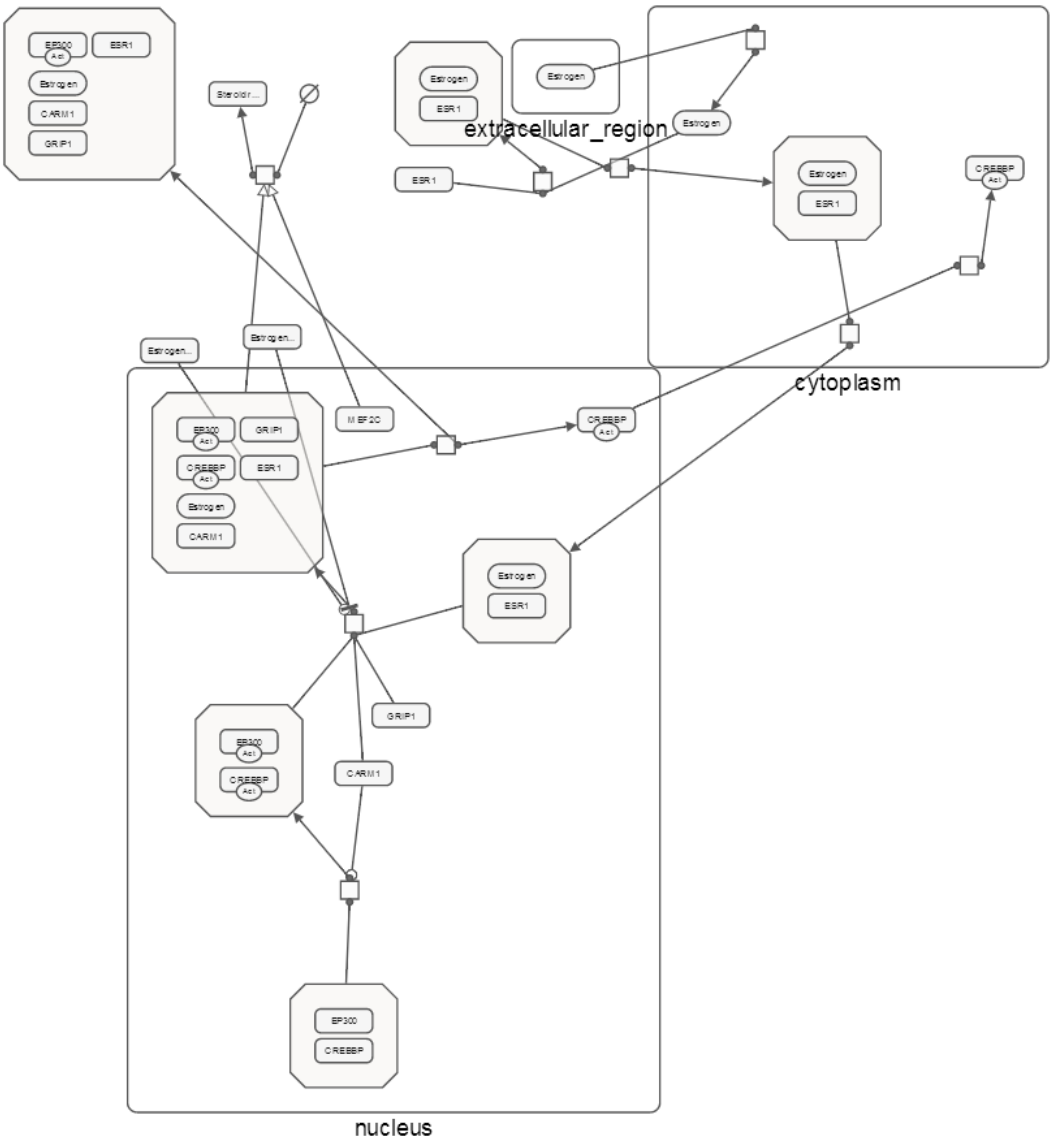

Figure 4.23: Carm1 and regulation of the estrogen receptor pathway laid out using SBGN-PD.

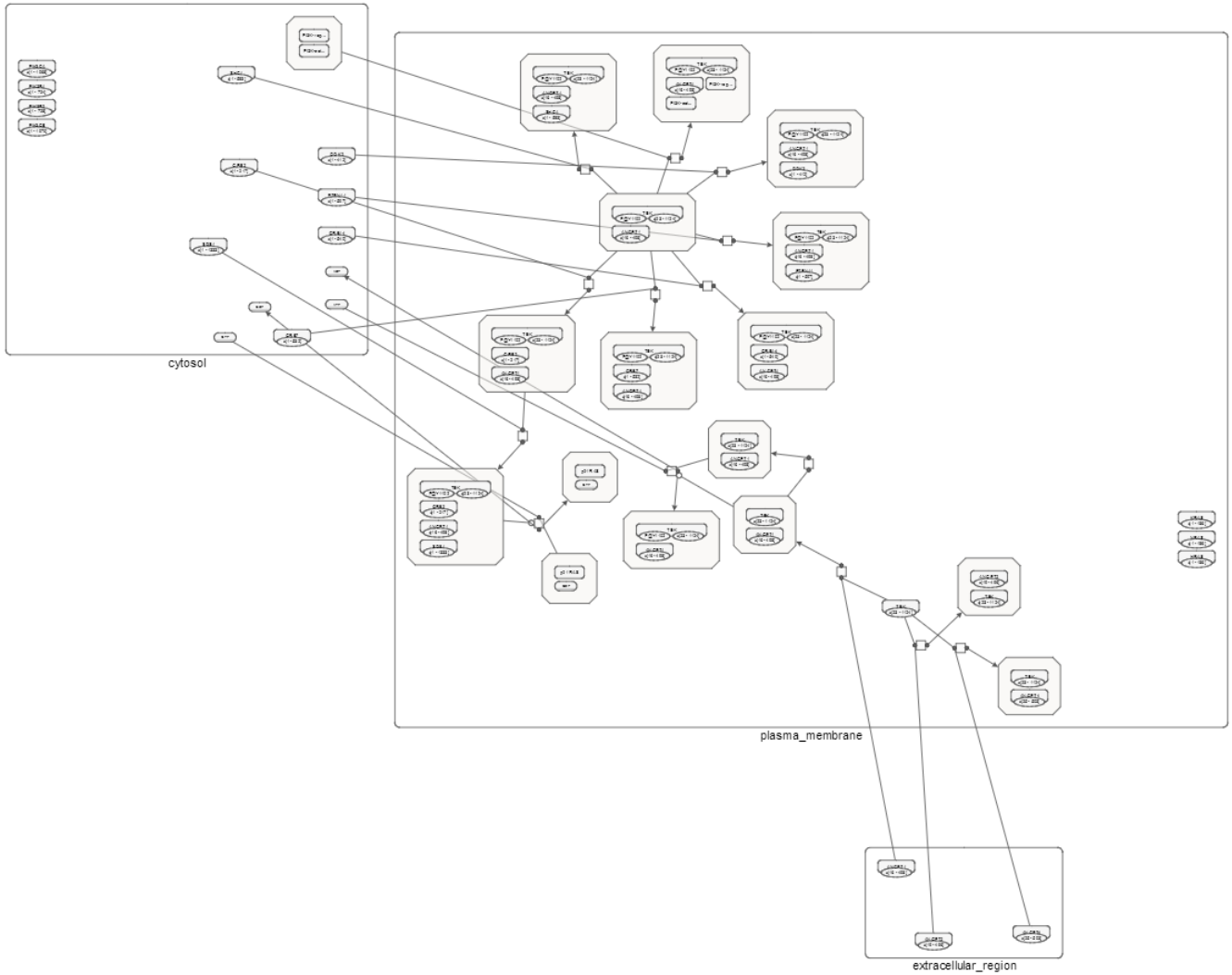

Figure 4.24: Tie2 signaling pathway laid out using SBGN-PD.

Chapter 5

Conclusion

The main motivation behind this study is to support Systems Biology Graphical Notation (SBGN) by providing a specialized automated layout algorithm for process description maps. This layout is called as SBGN-PD layout. Compound node support is crucial for representing nested hierarchy in process description maps. Therefore, considering its compound node support, we made use of a force-directed layout algorithm called Compound Spring Embedder (CoSE) [7].

SBGN-PD layout adds two main features to CoSE algorithm. The first enhancement provides properly packing of complex members and disconnected molecules by using two different rectangle packing algorithms: tiling and polyomino packing [30] [8]. The second one is supporting port nodes and providing rotation ability for process nodes by introducing a new force type. An important point to note is that those enhancements are added without disturbing the force-directed structure of the algorithm.

Our proposed layout algorithm has been integrated into Chisio Layout (ChiLay) [39]. It is also used by Paxtools, which uses ChiLay for laying out the graphs [42]. The graphs are obtained from Pathway Commons, a biological network database [1]. Additionally, SBGNViz, a visualization tool specialized for SBGN process description maps, is used to visualize the resulting graphs.

The results show that the improvements result in a considerably high success for separating substrate and production edges on the opposite sides of the process nodes. Additionally, successfully implemented packing algorithms provide aesthetically pleasing layouts by decreasing the wasted area. For future work, the problems addressed in section 4.1 can be further studied and execution time can be reduced.

Bibliography

[1] E. G. Cerami, B. E. Gross, E. Demir, I. Rodchenkov, z. Babur, N. Anwar, N. Schultz, G. D. Bader, and C. Sander, "Pathway commons, a web resource for biological pathway data.," *Nucleic Acids Research*, vol. 39, no. Database-Issue, pp. 685–690, 2011.

[2] A. Dilek, M. E. Belviranli, and U. Dogrusoz, "Visibioweb: visualization and layout services for biopax pathway models.," *Nucleic Acids Research*, vol. 38, no. Web-Server-Issue, pp. 150–154, 2010.

[3] "Sbgn reference card." `http://www.sbgn.org/Image:Refcard-PD.png`, Accessed in July 2014.

[4] C. Kucukkececi, U. Dogrusoz, A. Karacelik, and S. Ashyralyev, "Chied: Chisio editor."

[5] N. le Novere, "Systems Biology Graphical Notation: Process Description language Level 1," Oct. 2010.

[6] U. Brandes, "Drawing on physical analogies.," in *Drawing Graphs* (M. Kaufmann and D. Wagner, eds.), vol. 2025 of *Lecture Notes in Computer Science*, pp. 71–86, Springer, 1999.

[7] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir, "A layout algorithm for undirected compound graphs.," *Inf. Sci.*, vol. 179, no. 7, pp. 980–994, 2009.

[8] K. Freivalds, U. Dogrusz, and P. Kikusts, "Disconnected graph layout and the polyomino packing approach.," in *Graph Drawing* (P. Mutzel, M. Jnger,

and S. Leipert, eds.), vol. 2265 of *Lecture Notes in Computer Science*, pp. 378–391, Springer, 2001.

[9] J. Larkin and H. Simon, "Why a diagram is (sometimes) worth ten thousands words," *Cogn. Sci.*, vol. 11, pp. 65 – 100, 1987.

[10] Y. Lazebnik, "Can a biologist fix a radio? or,what i learned while studying apoptosis," *Cancer Cell*, vol. 2, pp. 179 – 182, 2002.

[11] N. L. Novre, M. Hucka, H. Mi, S. Moodie, F. Schreiber, A. Sorokin, and E. D. et. al., "The systems biology graphical notation," *Nature Biotechnology*, vol. 27, no. 8, pp. 735–741, 2009.

[12] E. Demir, M. P. Cary, S. Paley, and F. et. al., "The biopax community standard for pathway data sharing," *Nature Biotechnology*, vol. 28, no. 9, pp. 935–942, 2010.

[13] G. D. Bader, M. P. Cary, and C. Sander, "Pathguide: a pathway resource list.," *Nucleic Acids Research*, vol. 34, no. Database-Issue, pp. 504–506, 2006.

[14] "Sbgn software." `http://www.sbgn.org/SBGN_Software`, Accessed in June 2014.

[15] P. D. Karp and S. Paley, "Automated drawing of metabolic pathways," pp. 225–238, 1994.

[16] B. Genc and U. Dogrusoz, "A layout algorithm for signaling pathways.," *Inf. Sci.*, vol. 176, no. 2, pp. 135–149, 2006.

[17] M. Y. Becker and I. Rojas, "A graph layout algorithm for drawing metabolic pathways," *Bioinformatics*, vol. 17, no. 5, pp. 461–467, 2001.

[18] F. Schreiber, "High quality visualization of biochemical pathways in biopath.," *In Silico Biology*, vol. 2, p. 6, 2002.

[19] K. Wegner and U. Kummer, "A new dynamical layout algorithm for complex biochemical reaction networks.," *BMC Bioinformatics*, vol. 6, p. 212, 2005.

[20] O. Babur, U. Dogrusoz, E. Demir, and C. Sander, "Chibe: interactive visualization and manipulation of biopax pathway models," *Bioinformatics*, vol. 26, pp. 429–431, Feb. 2010.

[21] E. Demir, O. Babur, U. Dogrusz, A. Grsoy, G. Nisanci, R. etin Atalay, and M. Ozturk, "Patika: an integrated visual environment for collaborative construction and analysis of cellular pathways.," *Bioinformatics*, vol. 18, no. 7, pp. 996–1003, 2002.

[22] A. Funahashi, M. Morohashi, H. Kitano, and N. Tanimura, "CellDesigner: a process diagram editor for gene-regulatory and biochemical networks," *BIOSILICO*, vol. 1, pp. 159–162, Nov. 2003.

[23] P. Dabke, A. Cox, and D. Johnson, "Netbuilder: an environment for integrating tools and people.," *Computer-Aided Design*, vol. 30, no. 6, pp. 465–472, 1998.

[24] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 1998.

[25] U. Dogrusoz, Q. Feng, B. Madden, M. Doorley, and A. Frick, "Graph visualization toolkits," *IEEE Comput. Graph. Appl.*, vol. 22, pp. 30–37, Jan. 2002.

[26] P. Eades, "A heuristic for graph drawing," *Congressus Numerantium*, vol. 42, pp. 149–160, 1984.

[27] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs.," *Inf. Process. Lett.*, vol. 31, no. 1, pp. 7–15, 1989.

[28] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement.," *Softw., Pract. Exper.*, vol. 21, no. 11, pp. 1129–1164, 1991.

[29] S. G. Kobourov, "Spring embedders and force directed graph drawing algorithms," 2012. 23 pages, 8 figures.

[30] U. Dogrusoz, "Two-dimensional packing algorithms for layout of disconnected graphs," *Inf. Sci. Inf. Comput. Sci.*, vol. 143, pp. 147–158, June 2002.

[31] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition ed., 1979.

[32] A. Lodi, S. Martello, and M. Monaci, "Two-dimensional packing problems: A survey.," *European Journal of Operational Research*, vol. 141, no. 2, pp. 241–252, 2002.

[33] N. J. Nilsson, "A mobius automation: An application of artificial intelligence techniques," in *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, (San Francisco, CA, USA), pp. 509–520, Morgan Kaufmann Publishers Inc., 1969.

[34] J. claude Latombe, "Motion planning: A journey of robots, molecules, digital actors, and other artifacts," *International Journal of Robotics Research*, vol. 18, pp. 1119–1128, 1999.

[35] D. T. Lee, "Visibility of a simple polygon," vol. 22, pp. 207–221, May 1983.

[36] S. Kapoor and S. N. Maheshwari, "Efficiently constructing the visibility graph of a simple polygon with obstacles.," *SIAM J. Comput.*, vol. 30, no. 3, pp. 847–871, 2000.

[37] H. Edelsbrunner and L. J. Guibas, "Topologically sweeping an arrangement," *Journal of Computer and System Sciences*, vol. 38, no. 1, pp. 165 – 194, 1989.

[38] S. K. Ghosh and D. M. Mount, "An output sensitive algorithm for computing visibility graphs," in *FOCS*, pp. 11–19, IEEE Computer Society, 1987.

[39] "Chisio layout (chilay)." `https://code.google.com/p/chilay/`, Accessed in July 2014.

[40] "Chisio editor (chied)." `https://code.google.com/p/chied/`, Accessed in July 2014.

[41] "Pathway commons network visualizer (pcviz)." http://www.pathwaycommons.org/pcviz/, Accessed in August 2014.

[42] E. Demir, O. Babur, I. Rodchenkov, B. A. Aksoy, K. I. Fukuda, B. Gross, O. S. Sümer, G. D. Bader, and C. Sander, "Using Biological Pathway Data with Paxtools," *PLoS Comput Biol*, vol. 9, pp. e1003194+, Sept. 2013.