VISIBIO*web*: A WEB-BASED VISUALIZATION AND LAYOUT SERVICE FOR BIOLOGICAL PATHWAYS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Alptuğ Dilek

August, 2009

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.

Assoc. Prof. Dr. Uğur Güdükbay

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. Tolga Can

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

VISIBIO*web*: A WEB-BASED VISUALIZATION AND LAYOUT SERVICE FOR BIOLOGICAL PATHWAYS

Alptuğ Dilek
M.S. in Computer Engineering
Supervisor: Assoc. Prof. Dr. Uğur Doğrusöz
August, 2009

A biological pathway is a representation of biological reactions between molecules in a living cell. At present, there are hundreds of Internet-accessible databases storing biological pathway data. Exchanging, handling, and storing this data are crucial in terms of both providing understandability and allowing further enhancements on the gathered data. As a result of this necessity, many biological models were developed to cluster the data in a meaningful manner under a semantically reasonable hierarchy. As the amount and complexity of the data increases, visualization of pathways becomes inevitable. Graphs are inherently suitable for modeling pathways. The task of creating a visual representation for pathways dynamically requires methods from the area of graph visualization. As a result, many software systems, which can interpret the pathway data with a graph structure and visualize the constructed graph, emerged. However, many of these software systems are insufficient due to poor complexity handling of the underlying model, lack of visual standardization or long installation steps.

In this thesis, we introduce VISIBIO*web*, a new open-source and web-based visualization service for biological pathway models stored in BioPAX (Biological Pathways Exchange Language) format. VISIBIO*web* runs on Apache Tomcat server and is implemented in Java based on Eclipse GEF (Graphical Editing Framework). Google Maps API is used on the client side as the core component to visualize the representation constructed on the server.

VISIBIO*web* supports basic graph viewing functionalities such as zooming, scrolling, and selection of graph objects. The inspector window is provided to view the properties of the selected graph object. Once the view for the uploaded biological model is created, it can be stored as a static image. The biological models can also be persisted and embedded within other web sites just like Google

Maps. The layout information of the constructed graph is also provided in an XML-based format. The introduction of such a format is a good starting point to develop an official layout extension for BioPAX format.

Keywords: biological pathway, pathway visualization, graph visualization, graph layout, software system.

ÖZET

VISIBIO*web*: BİYOLOJİK YOLAKLAR İÇİN WEB TABANLI GÖRSELLEME VE MİZANPAJ SERVİSİ

Alptuğ Dilek
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Doçent Dr. Uğur Doğrusöz
Ağustos, 2009

Biyolojik yolaklar, canlı bir hücre içersinde moleküller arasında gerçekleşen biyolojik tepkimeleri temsil ederler. Günümüzde genel ağdan erişilebilen biyolojik yolak verisi içeren veri tabanlarının sayısı yüzler mertebesindedir. Bu verilerin değişimi, ele alınması ve saklanması, gerek anlaşılabilirlik gerekse de toplanan verilerin arttırılması açısından oldukça önemlidir. Bu gereksinimlerin bir sonucu olarak, toplanan verilerin anlamlı ve mantıklı bir düzende gruplanabilmesi için birçok biyolojik model geliştirilmiştir.

Verilerin miktarı ve karmaşıklığı arttıkça yolakların görsellenmesi kaçınılmaz bir ihtiyaç olmuştur. Çizgeler, yolakların modellenmesinde doğal olarak uygundurlar. Yolakların dinamik olarak görsel temsillerinin oluşturulması için çizge görselleme alanından yöntemler gerekmektedir. Sonuç olarak yolak verisini çizge biçiminde yorumlayarak görselleyen yazılım araçları ortaya çıkmıştır. Ne var ki, bu araçların birçoğu biyolojik modelin karmaşıklığının düzgün ele alınamaması, görselleme standardizasyonu eksikliği veya uzun yükleme adımlarının bulunması gibi nedenlerden ötürü yetersiz bulunmaktadır.

Bu tez çalısmasında, açık kaynak kodlu olan ve BioPAX formatında saklanmış yolak modellerinin web tabanlı olarak görsellenmesi servisi veren VISIBIO*web* geliştirilmiştir. Java prgoramlama dilinde Eclipse GEF kütüphanesi üzerine geliştirilen VISIBIO*web* Apache Tomcat sunucusunda çalışmaktadır. Kullanıcı tarafında, sunucuda oluşturulan modelin görsellenmesi için Google Maps API kullanılmaktadır.

VISIBIO*web* yakınlaştırma, kaydırma ve çizge nesnelerini seçme gibi temel çizge görüntüleme özelliklerini desteklemektedir. Kullanıcılara, seçilen çizge özelliklerini listeleyen inceleme penceresi sağlanmıştır. Yüklenilen biyolojik model

için oluşturulan görüntü sabit resim olarak kaydedilebilmektedir. Biyoljik modellerin görüntüleri tıpkı Google haritaları gibi saklanılarak başka web sayfalarının içerilerine konulabilmektedir. Oluşturulan çizgenin mizanpaj bilgisi kullanıcılara XML formatında bir dosya ile sunulmaktadır. Böyle bir formatın geliştirilmesi, BioPAX formatı için resmi bir mizanpaj ilavesi geliştirme açısından iyi bir başlangıç noktasıdır.

Anahtar sözcükler: biyolojik yolak, yolak görselleme, çizge görselleme, çizge mizanpaj, yazılım sistemi.

Acknowledgement

I would like to express my gratitude to my supervisor Assoc. Prof. Dr. Uğur Doğrusöz for his efforts in the supervision of this thesis. He was not only an academic advisor, but also an idol especially in human relations, work discipline, and ethics. It has always been an honor and pleasure to be his student.

I would like to thank to Assoc. Prof. Dr. Uğur Güdükbay and Asst. Prof. Dr. Tolga Can for showing keen interest to the subject matter and accepting to read and review the thesis.

This thesis cannot be completed without the advices and supports of Özgün Babur and Esat Belviranlı. I would like to thank them in name.

I would also like to express my gratitude to the Scientific and Technological Research Council of Turkey, TUBITAK, for their extensive support during my two years of M.S. study.

I would like to thank my Earth Angel, Merve, for her understanding and support during my thesis study. Her smile takes away all the problems and sadness in a moment.

Finally, I would like to thank my parents Ahmet and Afet, and my lovely sister Anıl for all their support during my whole life. Without their love I should not have been the person I am now.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A cell is a highly interactive biological unit, which tries to react to the changes occurring in its environment constantly, in order to survive. This task of survival involves interactions and reactions of biological molecules and environmental factors. *Biological pathways* are networks of series of such complex reactions and interactions at molecular level [30]. Thus, biological pathways are representations of knowledge about the cellular processes. Figure 1.1 is a diagram, which shows steps of interactions constituting an example cellular process in human cell.

Advances in genetics and microbiology resulted in the production of considerable amount of biological data in the last decade [13]. The data related with the cellular processes are modeled in the context of biological pathways. As the amount and complexity of data increase, analyzing the underlying biological model and extracting relevant information becomes harder.

1.1 Motivation

The increase in the amount of biological pathway data necessitates the use of software systems to store, process and analyze that data. Visual representation of the existing information is an important requirement to perform better analysis.

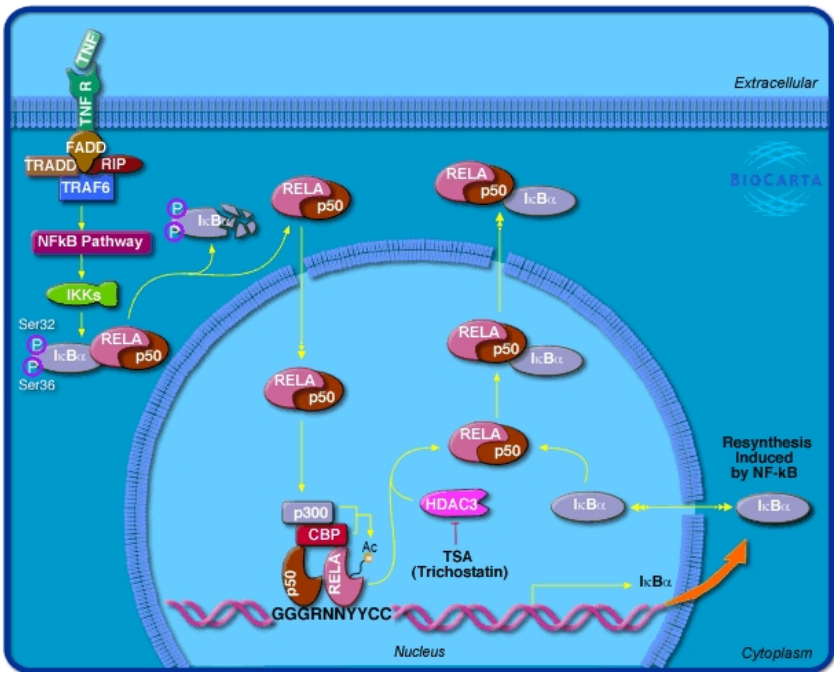

Figure 1.1: Acetylation and Deacetylation of RelA in the nucleus in a stylized diagram (courtesy of BIOCARTA [3]).

In visualizing the information at hand, graph data structure is suitable for the biological pathway domain. The biological entities become the nodes, and the interactions between them constitute the edges of the graph. Hence, the methods from the graph visualization area are applicable to biological pathway data. Using graphs as the visualization model can improve the analysis even further by the use of graph theoretical algorithms. With the help of such a representation, the biologists can infer new information, which is not apparent within the data gathered by experiments. As a result, a visualization component is a vital part of many biomedical applications used in the development of new drugs and diagnosis of diseases [11].

Many different biological pathway visualization software tools with different capabilities have been developed recently by a variety of research groups. The representation and interpretation of the biological information differ greatly between these groups and their tools. BioPAX [5] aims to resolve the lack of standardization for the representation of the data using a formal ontology. This makes data integration easier, but it is still far from achieving complete standardization.

There are so many types of pathway data available and different people focus on different aspects of the data. Hence, among many software tools, none has proven to be frequently used.

Properly handling the complexity of the underlying biological model is one aspect, for which existing software tools are insufficient. The biological model is composed of complex networks of interactions occurring in and between cellular locations. In order to model the biological requirements properly, the graph representation must support *compound graphs*. Compound graphs are graphs with nested child graphs inside. Multi-nesting is provided via having a compound graph having another compound graph as one of its children graph objects. Figure 1.2 is a simple biological pathway image from VISIBIO*web* showing the use of compound graphs. Supporting compound graphs not only means modeling them, but also means being able to layout the nested structure.

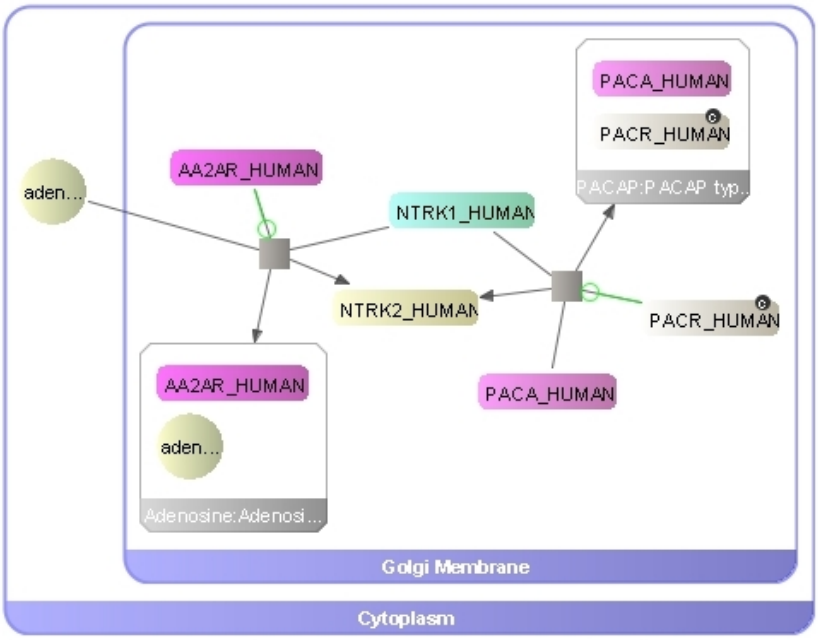

Figure 1.2: Cytoplasm containing Golgi Membrane and Golgi Membrane containing two other compound nodes, the complexes, resulting in a compound graph structure.

The lack of visual representation standardization was also limiting the use of existing software tools. The development of a visual standardization for biological pathways is difficult due to the existence of different levels of abstraction on the

data. However, an effort for the standardization of graphical notation used in diagrams of biochemical and cellular processes, named SBGN (System Biology Graphical Notation), resulted in a first release of a notation last summer. The existence of a standard notation is crucial in terms of sharing the knowledge between different research groups more accurately and efficiently [31].

Although some of the existing software tools have exciting and useful features, they do not attract the attention of the biologists as much as expected. Another reason for this is the requirement of long and possibly complex installation steps. The end-users seem to prefer easy-to-use and learn software systems possibly without the need of an installation step. A thin-client pathway visualization tool working inside a web-browser seems to be the best candidate to overcome this problem.

1.2 Results

With the motivation stated and the insufficiencies of the existing software tools, VISIBIO*web* fills an important gap in the area of biological pathway visualization. VISIBIO*web* is unique in satisfying all of the criteria explained earlier. It is one of the few software tools supporting SBGN and the only one that handles compound structures and works within a web-browser without the need of any installation process. It currently supports perhaps the most popular biological model format, BioPAX level 2. The detailed information about BioPAX is given in Section 2.2. VISIBIO*web* works inside a web-browser and its client side is built upon Google Maps API [17]. Hence, the canvas, in which the visualization is displayed, is a customized Google map. As a result, VISIBIO*web* canvas can be integrated within other web-sites easily just like an ordinary Google Map. This feature is named as *URL embedding* in VISIBIO*web* terminology. Moreover zooming, panning, and overview are provided via Google Maps API. Hit-testing (the mechanism to detect the graph object under mouse cursor), tooltips, inspector window, exporting the view to SVG and PNG image formats, saving the layout information of the view constructed, changing the canvas size functionalities are

also supported in VISIBIO*web*. Figure 1.3 shows VISIBIO*web* in action.

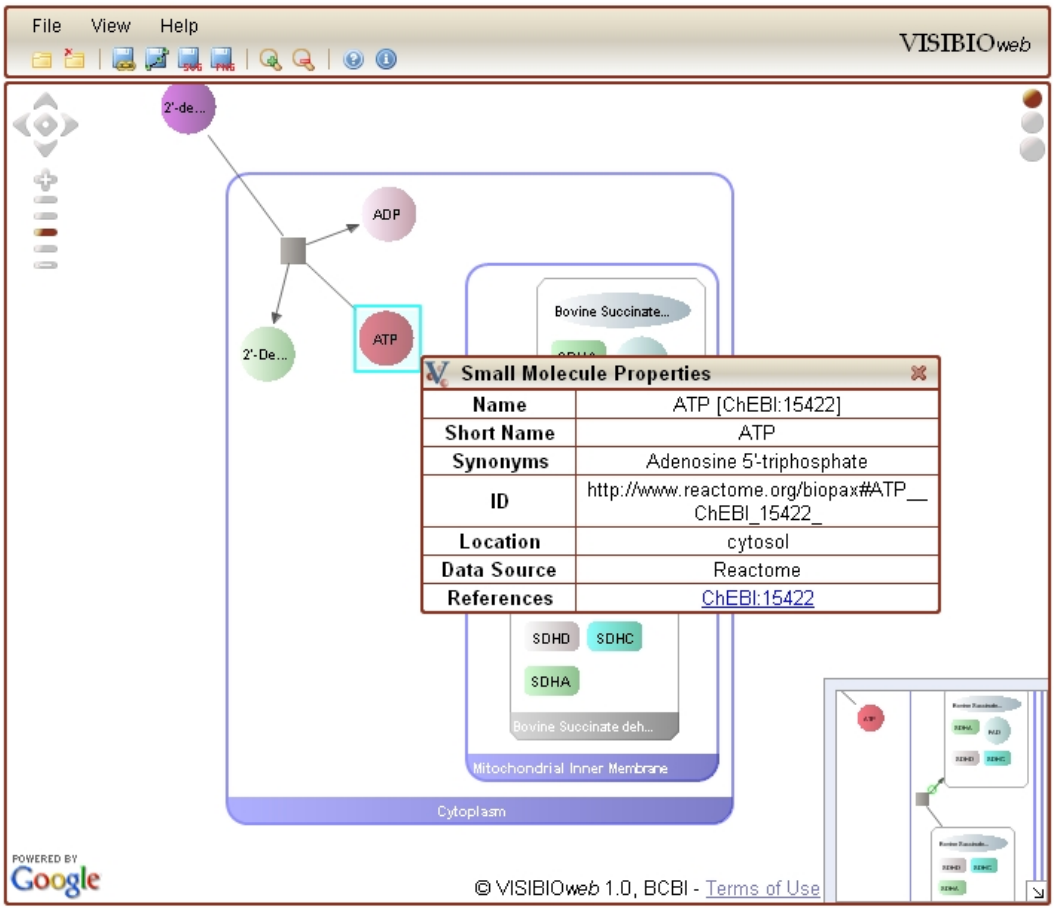

Figure 1.3: An overview of a sample BioPAX model in VISIBIO*web*.

The end users of VISIBIO*web* are able to customize the view to be created by changing the parameters provided in the load model menu. The compound visualization type (Pathway or Compartment), the partial model view selection (option to choose one or more of the pathways existing in the model to be loaded) selection and degree of separation (cloning the nodes which has a degree higher than that of specified) value are customizable options by the user to construct a more desirable view. In addition to display options, the users can also adjust the layout parameters to create a more useful layout for their purposes. The detailed information about the configuration of these parameters is given in the user's guide [35].

Ability to save the layout of the submitted model as a separate file in XML

format is an important feature of VISIBIO*web*. It provides the users with an opportunity to apply their own rendering on the graph constructed. Official BioPAX definition does not support a layout specification of the biological model. The XML schema we defined can be considered as an external layout extension for BioPAX models. It is not easy to define a standard for layout information. This is due to the reason that there is not a consensus about the mapping of biological model elements to the corresponding graph objects. For instance, in VISIBIO*web* an edge is created for a *Control* element in the BioPAX model, whereas another tool to visualize BioPAX models, ChiBE [1], prefers creating both a node and an edge in some cases. Hence, the construction of a graph even from the same model is subjective and can change considerably with respect to the level of detail desired to be visualized.

In order to support the storage and exchange of graph topology and geometry generated by VISIBIO*web*, an XML schema is defined. This schema is used internally to send the geometry information of the graph model constructed on the server side to the client side for the support of hit-testing and inspector window features. The schema might become a standard and be used as a layout extension for BioPAX model in the future. Acceptance of a formal layout extension for BioPAX should be an exciting milestone in the community. However, such a study requires great effort from many participants of the community and should take considerable amount of time. The XML file, which can be saved by the users of VISIBIO*web* is generated dynamically with respect to this schema. This file not only provides geometric information about the graph objects, but also provides the topology of the graph model constructed. The file also holds the identifiers of the model elements in the BioPAX model as reference. This is very helpful to preserve the relationship between the biological and graph models. The details of how this mapping is done and other information about the XML schema are given in Sections 3.1 and 3.2.

VISIBIO*web*'s success in the area of biological pathway visualization relies heavily on the interpretation of submitted biological model. The submitted model can contain different types of biological information. It is not trivial to extract the relevant information, which is in the scope of interest, and create a graph

model representation from it. VISIBIO*web* mainly focuses on the detailed molecular level interactions between biological entities. The detailed information about the biological pathway information visualized in VISIBIO*web* is discussed in Section 2.2. The technical details of the how parsing of a file in BioPAX format (.owl file) is performed and how the mapping to the graph objects is achieved are explained in Section 3.1.

One last achievement obtained during this study is the development of a design pattern, which provides extensibility to VISIBIO*web* for the integration of biological models other than BioPAX later on. The concept is named *Role Manager Structure*. The details of this concept is given in Section 4.2.2.1.

In the remainder of this thesis, the background and related work will be mentioned in Chapter 2. The methods, developed to achieve the results explained above, are explained in Chapter 3. The architecture and implementation details of VISIBIO*web* are given in Chapters 4 and 5 in order. Finally, the contribution and possible future extensions are mentioned in Chapter 6.

Chapter 2

Background and Related Work

This chapter starts with the terminology. Background information about the core components and concepts used during the development of VISIBIO*web* will follow. Discussion about different software tools developed to visualize biological pathway models will conclude the chapter.

2.1 Terminology

A node $v \in V$ (an edge $e \in E$), where $G = (V,E)$, is said to be a *member* of graph G; conversely G is said to be the *owner* of node v (edge e). A *compound graph* $C = (V,E,F)$ consists of *nodes* V, *adjacency edges* E, and *inclusion edges* F. It is required that the inclusion graph $T = (V,F)$ is a rooted tree, and no adjacency edge connects a node to one of its descendants or ancestors [14]. For the compound graph in Figure 2.1:

$V = \{a,b,\ldots,j\}$
$E = \{\{a,b\},\{a,g\},\{d,e\},\{d,g\},\{f,g\},\{f,h\},\{g,h\},\{i,j\}\}$, and
$F = \{bc,bd,be,cf,cg,ch,ei,ej\}$.

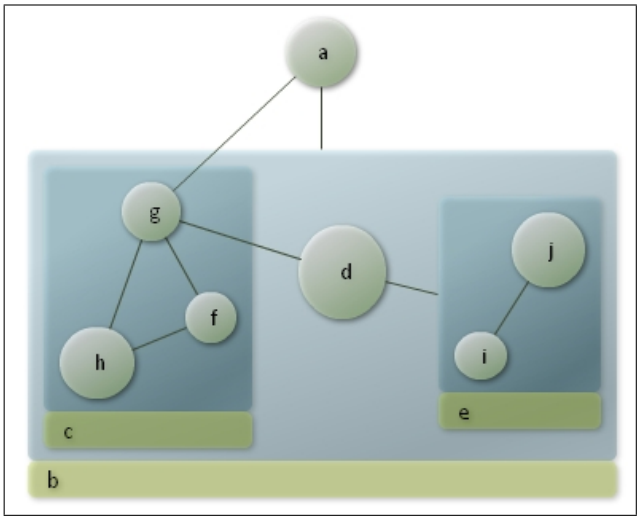

Figure 2.1: An example of a compound graph with multiple levels of nesting (3).

The node within which a graph is nested is called a *compound node*. A compound node is said to be *parent node* u of *child graph* G_i. A graph G_i is *owner graph* of nodes residing inside it. The graph at the top of the nesting tree is simply called *root graph*. In Figure 2.1:

$G_1 = \{\{a,b\},\{\{a,b\}\}\}$, $G_2 = \{\{c,d,e\},\{\{d,e\}\}\}$
$G_3 = \{\{f,g,h\},\{\{f,g\},\{f,h\},\{g,h\}\}\}$, and $G_4 = \{\{i,j\},\{\{i,j\}\}$.

The owner graph of nodes f, g, and h is G_3, which in turn is the child graph of its parent node c.

2.2 BioPAX

BioPAX is a biological data exchange format developed to enable integrity between different pathway resources. The existence of such an integrity is crucial to increase the data sharing between diverse organizations and the efficiency of computational pathway research. The details of BioPAX ontology is out of the scope of this thesis and it can be reached inside the level 2 documentation [6]. The major concepts, which are used in the creation of the graph model, are explained briefly in the remainder of this section.

BioPAX ontology is composed of many inheritance and composition relations among various concepts. Three are three top level classes in the ontology. Physical entity class can be considered as the base for all the biological molecules and structures inside the cell, such as DNA, RNA, protein, small molecules (e.g. ATP, H_2O, etc.) and various complex structures constructed by combination of these entities. Interaction class represents various types of interactions possible between the stated physical entities. Some example interaction types are biochemical reactions, catalysis, transport, etc. Pathway class is introduced to the ontology as the representation of biological pathways. An overview of the relations among these classes can be seen in Figure 2.2. As the diagram states, a pathway is composed of interactions. An interaction can contain physical entities (left and right participants of a reaction is an example of such a relation) and pathways resulting in a possible containment relationship between pathways.

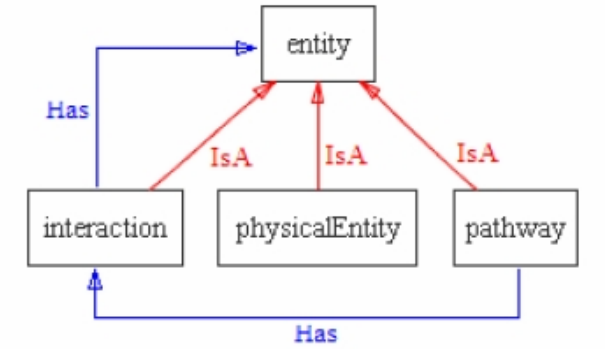

Figure 2.2: An overview of top level entities in BioPAX ontology [6].

There is not a clear separation between types of biological pathways. However, they can be grouped under three major types with respect to the content and detail of the information they contain. These groups are *Metabolic Pathways, Gene Regulation Pathways* and *Signal Transduction Pathways*. Metabolic pathways contain information about the chemical reactions at the molecular level. Gene regulation pathways represent the information about the activation and deactivation of genes at a more abstract level than that of metabolic pathways. The signal transduction pathways focus on the effects and transformation of signals inside the cell. BioPAX ontology is capable of modeling all these biological pathway types with the creation of relevant subclasses of the top level classes explained above. VISIBIO*web* parses the data related with the interactions of

biological entities at the molecular reaction level. Thus, the pathway information for which graph models to be created falls under the category of metabolic and signal transduction pathways.

The level of detail visualized in VISIBIO*web* is similar to the mechanistic level of PATIKA project [27]. Hence the *state* terminology is also an important concept, which constitutes the core of the view to be creaed, in VISIBIO*web* context just like in PATIKA. The existence of different special characteristics of a physical entity can be considered as the state of that entity. Some examples of such characteristics are the location, post-translational modification (for instance acetylation or phosphorylation of a protein) and the existence of a binding site (the region where other entities can bind). The state concept in BioPAX ontology is represented with *"Physical Entity Participant"* concept. Physical entity participants are key in VISIBIO*web*'s interpretation of BioPAX model. As a general rule, a new node is created for each physical entity participant. The details of the graph construction algorithm are given in Section 3.1.

2.3 Systems Biology Graphical Notation

Systems Biology Graphical Notation (SBGN) is an effort to create a standardization for the notation used in the visualization of biological data. Such a visual representation can add up to BioPAX's mission of increasing the data sharing within the community and enabling integrity of the data between different resources. As explained earlier in Section 1.1, BioPAX has achieved the standardization of the storage and representation of pathway data partially. However, there was still the problem of standardization for the graphical notation of the data. SBGN provides different types of graphical notations to satisfy the needs of the community [23].

The needs of the biologists in terms of the level of detail about the pathway data to be visualized were considered carefully while designing VISIBIO*web*'s components related with parsing and interpretation of BioPAX model. This

provided VISIBIO*web* to adapt to SBGN notation easily. The mechanistic level of information about the pathway data extracted in VISIBIO*web* is compatible with SBGN's *Process Diagram* language. Process diagrams in SBGN are used to represent casual sequences of molecular processes and interactions between biochemical entities along with their results. In more detail, each node in the diagram represents a given state of a biological entity. Thus an entity can be represented with different nodes for each of its different state in the same process diagram [25]. The existence of the state terminology, which is very similar to that of VISIBIO*web* also made SBGN a great choice for the graphical notation of the view. Figure 2.3 shows a sample process diagram from SBGN.

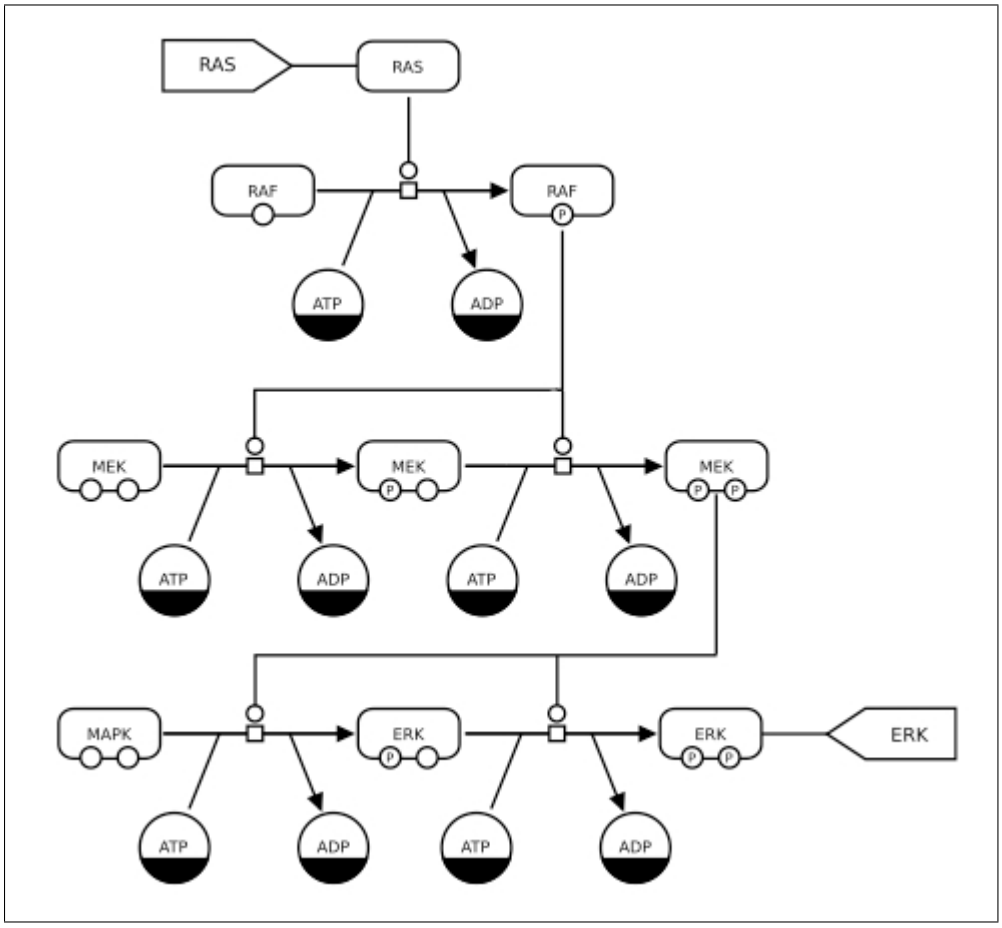

Figure 2.3: A sample process diagram from SBGN.

2.4 Graphical Editing Framework

VISIBIO*web* relies on Graphical Editing Framework of Eclipse (GEF) [16] to render the visual representation of the graph model constructed for the underlying biological model. A typical GEF application is composed of three major components: A model, figures and edit parts. These components form an application of well known MVC (Model-View-Controller) architecture in GEF context, as shown in Figure 2.4.

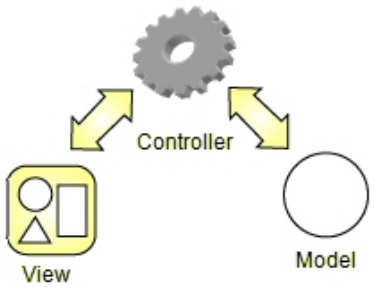

Figure 2.4: GEF MVC Architecture.

GEF is built on top of Draw2d and SWT [33] layer's of Eclipse. SWT is the provider of the widgets on which drawings of GEF figures are performed. Draw2d provides the rendering toolkit to display graphics including various basic figures and drawings. Figure 2.5 show an overview of GEF's dependency on these frameworks.

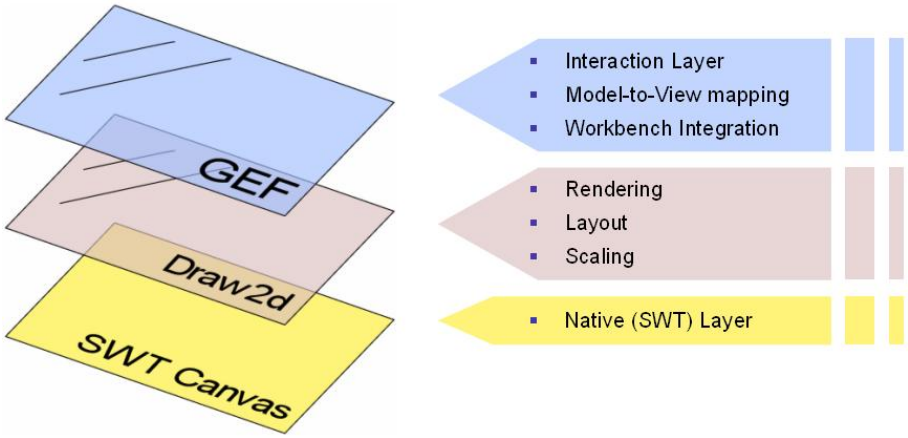

Figure 2.5: GEF is on top of Draw2d and SWT.

2.5 Google Maps API

Google Maps API provides developers a mechanism to embed Google Maps into their own web sites by using JavaScript. The API provides many useful services to create robust and easy-to-maintain maps. Some of these features are scrolling, zooming, overview support, and the ability to add overlays such as polygons on the map. The map itself is a useful canvas, which supports various user-oriented events such as mouse click, and mouse dragging. The API is freely available to non-commercial web sites.

There are various documents related with the API, from which many important features and implementation details need to be learned. It is necessary to have a basic understanding of how the API works in order to customize it. Google Maps API provides a display canvas, `GMap2` instance, inside which desired information is visualized. The content of this canvas is images, which are named as *tiles* in API terminology. Following remarks describe basic knowledge necessary to understand the customizations performed for VISIBIO*web*.

- *Pixel Coordinates:* Each individual tile is composed of 256x256 pixels and a point on a tile is referenced with an instance of `GPoint`, which holds numeric *x,y* values. Figure 2.6 is an illustration of a Google Map with just one tile. Top-left corner is *(0, 0)* and right-bottom corner is *(255, 255)*.

- *Earth Coordinates:* Google Map is designed and used to display maps. Thus, an Earth coordinate in terms of *latitude* and *longitude* is one of the coordinate systems used in the API. Any overlay, such as polygons in VISIBIO*web* case, that is put on top of a Google Map is placed by using latitude and longitude values stored as `GLatLng` JavaScript object of API. The API also provides methods to transform from pixel coordinates to Earth coordinates and vice versa. These methods are `fromDivPixelToLatLng(pixel:GPoint)` and `fromLatLngToDivPixel(latlng:GLatLng)`. These methods are useful in implementing hit-testing VISIBIO*web* pathway objects.

Figure 2.6: A single tile displaying the whole world. x values increase from left-to-right, whereas y values increase from top-to-bottom [18].

- *Tile Coordinates:* It is often the case that a Google Map will be composed of many tiles. Pixel coordinates determine the location of a pixel inside a tile. Similarly, tile coordinates are used to determine the position of a tile among many tiles existing in the map. Tile coordinates are referenced by unique *(x,y)* pairs. Figure 2.7 shows a Google map with 16 tiles, which are labeled with *(x,y)* pair values.

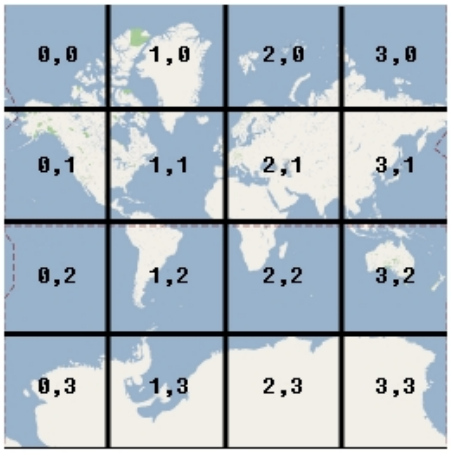

Figure 2.7: A Google map with 16 tiles showing the entire earth [18].

- *Zoom Levels:* Zooming in a Google Map means, retrieving the tiles for the same location of the map at a higher zoom level. This means, the tiles for each desired zoom level must be existing on the server in order for a Google

map to display them. Zooming one level in API means doubling in both x and y directions. In other words, if there are say 4 tiles at a certain zoom level, there are 16 tiles at a higher zoom level. There are 19 zoom levels in Google Maps API. Each zoom level displays a different level of detail. VISIBIO*web* uses 4 zoom levels for practical purposes.

- *Overlays:* Google Maps API provides different types of overlays, objects that can be placed on top of the map tiles. These objects hold latitudes and longitudes that determine their locations on the map. Thus, when a dragging or zooming operation occur, there is no need to update their pixel positions on the map. It is handled properly by the API. The only overlay type used in VISIBIO*web* is `GPolygon`, which consists of a collection of `GLatLng` instances. Figure 2.8 is an illustration of `GPolygon` instances on a Google map.

Figure 2.8: A Google map with 2 sample polygons rendered with red border color and pink fill color [20].

- *Events:* The API provides various events and corresponding event listeners in order to provide better ways of customization. Instead of mentioning all those events, it is better to mention important ones used in VISIBIO*web*. Events of `Gmap2` object commonly used in VISIBIO*web* are `mousemove(latlng:GLatLng)`, `click(overlay:GOverlay, latlng:GLatLng, overlaylatlng:GLatLng)`, and `zoomend(oldLevel:`

`Number`, `newLevel:Number)`. Events listened for instances of `GPolygon` objects are `mouseover()` and `mouseout()`. Details for these events can be found in [19].

2.6 Pathway Visualization Tools

The task of creating visual representation for biological pathway models have attracted the attention of many different researchers. Many different software systems with different capabilities have been developed to interpret and visualize pathway data. As stated in Section 1.1, none of these tools seem to satisfy the entire set of requirements and broad type of users. Table 2.1 shows the comparison of VISIBIO*web* with other popular pathway visualization software tools.

	BioPAX Support	Layout	Compound Support	SBGN	Availability	Tool Type
Cytoscape [10]	Yes	Automated	No	No	Open Source	Application
BiNoM [2]	Yes	Automated	No	Yes	Open Source	Application[1]
Reactome [29]	No	Automated	No	Planned	Open Source	Web
KEGG Tools [22]	No	Manual/Static	Limited[2]	No	Free	Web
BioCyc [4]	Export Only	No	No	No	Free	Application
VisANT [36]	Yes	Yes[3]	Yes	No	Open Source	Application, Applet
CellDesigner [7]	No	Yes	Yes	Yes	Free	Application
PathCase [26]	Yes[4]	Yes	No	No	TSS License[5]	Application
VISIBIO*web*	Yes	Yes	Yes	Yes	Open Source	Web

[1] BiNoM is a plug-in for Cytoscape
[2] Only cellular locations are represented as compounds; complexes are shown with simple nodes
[3] Compound support in the layout seems unreliable
[4] BioPAX support seems unreliable
[5] Tom Sawyer Software License is required

Table 2.1: VISIBIO*web* compared to popular biological pathway visualization tools

Chapter 3

Methods

This chapter includes various methods used in different parts of VISIBIO*web*. The parts of software, in which these methods reside, will constitute important components of VISIBIO*web*, thus we introduce them before the architecture.

3.1 BioPAX Handling

Parsing and processing the BioPAX model submitted to VISIBIO*web* server is the first step in constructing a viewable and inspectable representation of the underlying biological pathway data. BioPAX models are stored in .owl files. The end result of the BioPAX file parsing is a compound graph. As mentioned in Section 2.2, VISIBIO*web* is interested in state level information stored in a BioPAX formatted file. Thus, the view to be created for the resulting compound graph is similar to that of PATIKA's mechanistic level view. Users of VISIBIO*web* can configure the view to be created using the following display options: *Compound Visualization*, *Degree of Separation*, and *Allow Partial Model View*. Details about the usage of these options are discussed in user's guide [35].

In order to extract the relevant information from the file submitted, VISIBIO*web* relies on an external software library implemented in Java called Paxtools [28]. Paxtools performs the actual reading of an OWL file and constructs a model, which is the container for the BioPAX elements in the file, in the memory. This model is composed of Java objects, which represent the actual BioPAX elements. All relevant methods, which can be used to get the information related with particular BioPAX elements of interest, are provided in this model. Paxtools can also be used to validate a BioPAX formatted file and detect improper usages.

Once Paxtools constructs the BioPAX model in memory, it is the responsibility of VISIBIO*web* to process, extract, and store the relevant information in appropriate VISIBIO*web* objects. The mapping from the model constructed by Paxtools to VISIBIO*web* model, which is composed of nodes, edges, compound nodes, various roles and role managers, is a complicated process involving many steps. The complexity of the process originates from the fact that there is not a one-to-one relationship between objects of Paxtools model and those in VISIBIO*web* model object. Sometimes, more than one VISIBIO*web* model object are created for a single BioPAX element, whereas in other cases, many BioPAX elements are mapped to a single VISIBIO*web* model. There is even the case that VISIBIO*web* objects are created to represent certain fields of BioPAX elements, rather than the whole elements themselves. The details of this complicated process will be explained in the sequel. Here is a sketch of the overall process, before detailing each individual step.

Process BioPAX Model :
- detect and break cycles in the biological model
- if compartments are to be visualized then
- create compound nodes for cellular locations
- create nodes for PEPs
- create compound nodes for complexes
- if pathways are to be visualized hen
- process pathways and create compound nodes
- else
- process pathways

- process unreached elements
- redirect conversions
- apply degree of separation
- prune redundant compound nodes

3.1.1 Directed Acyclic Graph Creation

Pathways in BioPAX model are composed of other pathways, interactions, and pathway steps. Pathway steps contain interactions or pathways. Hence, there can be direct or indirect (through pathway - pathway step - pathway order recursively) parent-child relationship between pathways. These possible cyclic containments must be detected and broken by using manipulation functionality of Paxtools programmatically.

Thus, the first step in handling a BioPAX model is to create a *directed acyclic graph* (dag). A dag is a directed graph, where for every vertex v, there does not exist a non-empty directed path that starts and ends at v [12]. A cycle can occur in a BioPAX model between the instances of BioPAX Pathway class. Although this is not a scenario that frequently occurs, the cycles must be detected and broken in order to avoid a possible infinite loop problem while creating VISIBIO*web* model objects.

In order to detect the cycles, a directed skeleton graph is created by traversing the pathways of BioPAX model and creating nodes for pathways, pathway steps and interactions. The edges are drawn from the nodes representing pathways to the nodes representing their components (other pathways, pathway steps or interactions) and from the nodes created for pathway steps to the nodes created for their step interactions (pathways or interactions). A root node is introduced to the skeleton graph by adding an edge from that root node to every other node. The skeleton graph is then applied *Tarjan's algorithm* [34], where the strongly connected components of a directed graph are found. In a strongly connected component, there is a path from every vertex v to all other vertices of

the component and there is a path from every other other vertex to v. This means the cycles of the graph are contained in the strongly connected components of the graph. Hence, the strongly connected components, which have size greater than 1 has a cycle to be broken. Then, a breadth first search is applied to every such strongly connected component and cycles are broken via breaking the necessary containment relationships between pathways, pathway steps and interactions of the underlying biological model.

3.1.2 Processing Compartments

Compartments are the cellular locations, where the biological entities reside in. In BioPAX ontology, compartments are not represented as classes. They are stored as fields of instances of physical entity participant class, which was explained in Section 2.2. However, we would like to represent compartments as compound nodes in VISIBIO*web* if the user desires to visualize compartments as opposed to pathways as the compound nodes in the view to be created. Note that having pathway drawings including both comportments and pathways present difficulties for compound graph representation. Thus, the user must choose one or the other to represent with compound graphs.

All physical entities of a BioPAX model are traversed and the cellular location data is extracted for each of them. For every distinct cellular location, a compound node is created. The containment relationships between cellular locations are not stored inside the BioPAX model. However, we would like to nest the compartments of the model when possible. In order to achieve that, the compartment names are processed to apply the well-known nesting structure existing in a cell such as the cytoplasm contains the nucleus.

3.1.3 Processing Physical Entity Participants

In general, for every physical entity participant in a BioPAX model a separate node is created in VISIBIO*web*. However, there are a few exceptions to this rule.

The formal definition of physical entity participant in level 2 documentation of BioPAX is the following: "Any additional special characteristics of a physical entity in the context of an interaction or complex" [6]. Although this definition is very close to the state concept used in VISIBIO*web*, there is a subtle difference. In BioPAX, a new physical entity participant is defined for every separate interaction a physical entity involves. Hence, there might be situations where multiple physical entity participants represent the same state of that physical entity. In VISIBIO*web*, such physical entity participants must be represented with a single node, which means they must be merged. In order to tackle this situation, PEPs are checked for equivalence of state with `isInEquivalentState(pep)` method provided in Paxtools library. This method compares current pep with the one passed as parameter in terms of owner complex (if one exists), compartment and various modifications such as phosphorylation of a protein.

3.1.4 Processing Complexes

A complex is a physical entity which is a composition of other physical entities, which are bound to each other with non-covalent bounds. In BioPAX documentation, it is recommended not to define complexes recursively, which means a complex cannot exist in another complex. However, this recommendation is not always respected. Hence, nested complexes or empty complexes (complexes without any members) must be handled properly to satisfy the common use cases of BioPAX ontology. The handling process involves *flattening* the complexes. Flattening means adding the members of a complex, which is nested under another complex, to the top most complex in the containment hierarchy. Such complexes are not visually represented unless they are not empty, since it does not make any sense to flatten a complex without members. The flattening starts by iterating the members of top level complexes recursively. Here is the algorithm to flatten a complex node:

```
method FLATTENCOMPLEX(parent, complex)
1) members := complex.members
2) for m ∈ members do
3)   if m is instance of complex then
4)     if m.members.size is greater than 0 then
5)       call FLATTENCOMPLEX (parent, m)
6)     else
7)       node := create an empty compound node for m
8)       insert node to parent
9)   else
10)    node := create a node for m
11)    insert node to parent
```

Figure 3.1: The algorithm explaining flattening process applied to complex nodes.

3.1.5 Processing Pathways

Up until the point at which pathways get processed, not a single edge is created between the nodes created to represent PEPs. This is due to the reason, the interactions of a BioPAX model exist under the pathways of the model in general. Hence, processing the pathways involves also processing the interactions and creating edges between the nodes representing interactions and PEPs. The pathways are processed in a top-down manner just like the complexes. The pathway steps of top level pathways, ones which do not reside in other pathways or step interactions, are iterated and nodes are created for interactions encountered during this iteration. The creation of compound nodes for pathways is determined with respect to a user option. If pathways are visualized as opposed to compartments, then pathways are added to the graph model as compound nodes. An important point in visualizing pathways is to detect the *overlapping pathways*. Multiple pathways in a BioPAX might model might contain the same interaction(s) and sometimes the same pathway(s) as components. Among the overlapping pathways, only one of them can be visualized due to the geometrical restrictions in 2D drawings. The content of the remaining ones are added to the root graph. Although most of the details of overlapping pathway detection, interaction handling and edges creation are omitted, the pseudo code below shows the general approach for processing pathways:

```
method processPathway(parentNode, pathway)
1)  if a node is already created for pathway then
2)    insert parentNode to overlappingPathwaysList
3)  else
4)  compoundNode := create compound node for pathway
5)    interactions := {}
6)    pathways := {}
7)    for c ∈ pathway.components do
8)      if c is instance of interaction then
9)        add c to interactions
10)     else if c is instance of pathway then
11)       add c to pathways
12)     else if c is instance of pathway step then
13)       steps := c.steps
14)       for s ∈ steps do
15)         if c is instance of interaction then
16)           insert c to interactions
17)         else if c is instance of pathway then
18)           insert c to pathways
19)   for p ∈ pathways do
20)     call processPathway(compoundNode, p)
21)   for i ∈ interactions do
22)     call processInteraction(compoundNode, i)
```

Figure 3.2: The algorithm explaining processing of pathways.

3.1.6 Processing Interactions

There are different types of interaction classes in BioPAX ontology. Among these classes, VISIBIO*web* is interested in *control* and *conversion* types to visualize the mechanistic level interactions among biological entities. The implementation details of interaction processing is omitted not to complicate the understandability of the overall BioPAX handling. However, there are crucial parts, which must be emphasized for completeness.

In VISIBIO*web*, an edge is created for each control type interaction and a each node is created for conversion type interaction. Source node of the edge created for a control interaction is a node representing a PEP, and target node represents a conversion interaction. For conversion interactions, one or more edges are created along with the node to represent the conversion. The conversion interaction has left and right participants, which are PEPs in general. Hence, edges from nodes representing left participants to the conversion node and edges from the conversion node to the nodes created for right participants are created.

Another important issue is determining the cellular location of interactions if the compound type to be visualized is compartments. An interaction is placed in the compartment, in which the maximum number of participants of that interaction reside.

3.1.7 Processing Unreached Elements

The interactions, which are components of pathways, are processed while processing the pathways. However, there might be other interactions in the underlying biological model. Hence, these remaining interactions must be processed separately. Similarly, the insertion of the nodes representing the PEPs into the view is performed while processing the interactions. If there are isolated PEPs, which are not participating in any interaction, the nodes representing these PEPs must also be visualized. Such nodes are inserted to the view for completeness of the model.

3.1.8 Redirecting Conversions

A conversion is an interaction type which represents the biochemical reactions, transportations, transportations with biochemical reactions, complex associations and disassociations. Although not stated in official BioPAX documents, conversions are assumed to be from left-to-right direction by convention. Hence, the creation of all conversions are performed assuming the direction is from left-to-right. However, there are certain information in an OWL file, which can be used to identify the actual direction of the interaction. Sometimes, it is the case that both left-to-right and right-to-left directions are both valid and must be visualized. In such cases, the conversion node is cloned and the reverse edges (where source nodes become targets and vice versa) are created for the right-to-left direction of the conversion. In other situations, the right-to-left direction of the conversion must exist as opposed to what is created initially. In such cases, the edges are simply reversed. When the direction of a conversion is actually from left-to-right or it cannot be determined by the data in the file, no special treatment is needed.

3.1.9 Applying Degree of Separation

The degree of separation value is taken from the user as one of the display options provided in VISIBIO*web*. When a node representing a PEP has a degree that is greater than or equal to the degree of separation value, it is cloned. For every adjacency of the node, a clone node, which is of degree one is created. The process of cloning a node means creating an exact copy of the node in terms of role manager, role structure and the information it stores. The degree of separation option is not applied to compound nodes representing PEPs, hence complexes are out of the scope of the cloning process. The copy nodes are placed at the same owner compound structure of the original node. An application of degree of separation can be found in figures below. Figure 3.3 and Figure 3.4 shows application of degree of separation value 6 and 5 to the same biological pathway model respectively.

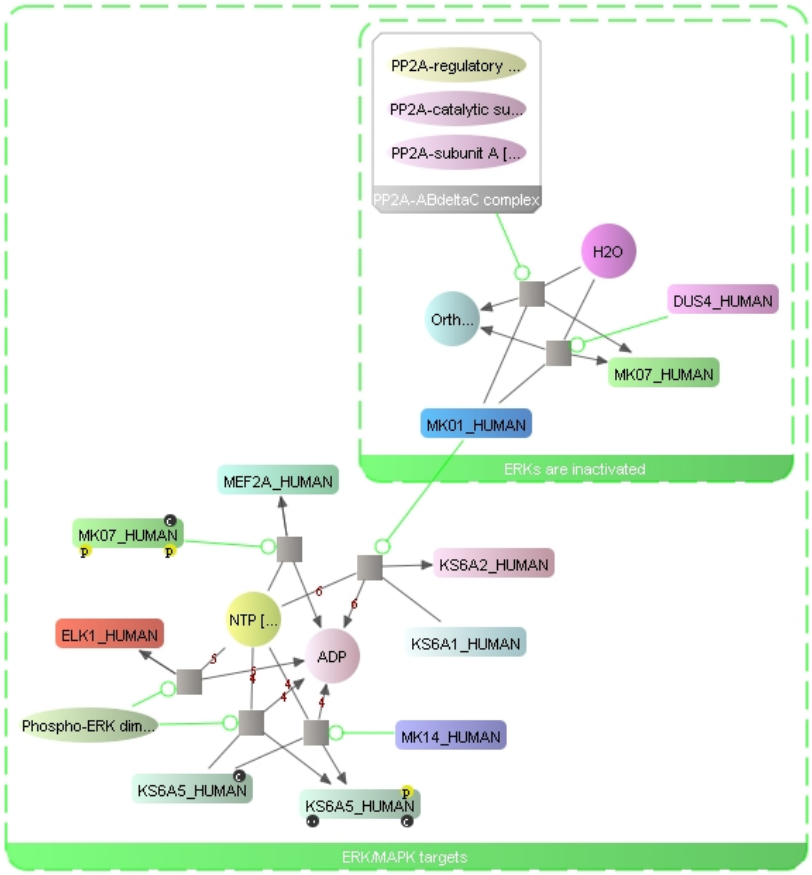

Figure 3.3: Sample model loaded with degree of separation value 6. ADP and NTP molecules are represented with single nodes.

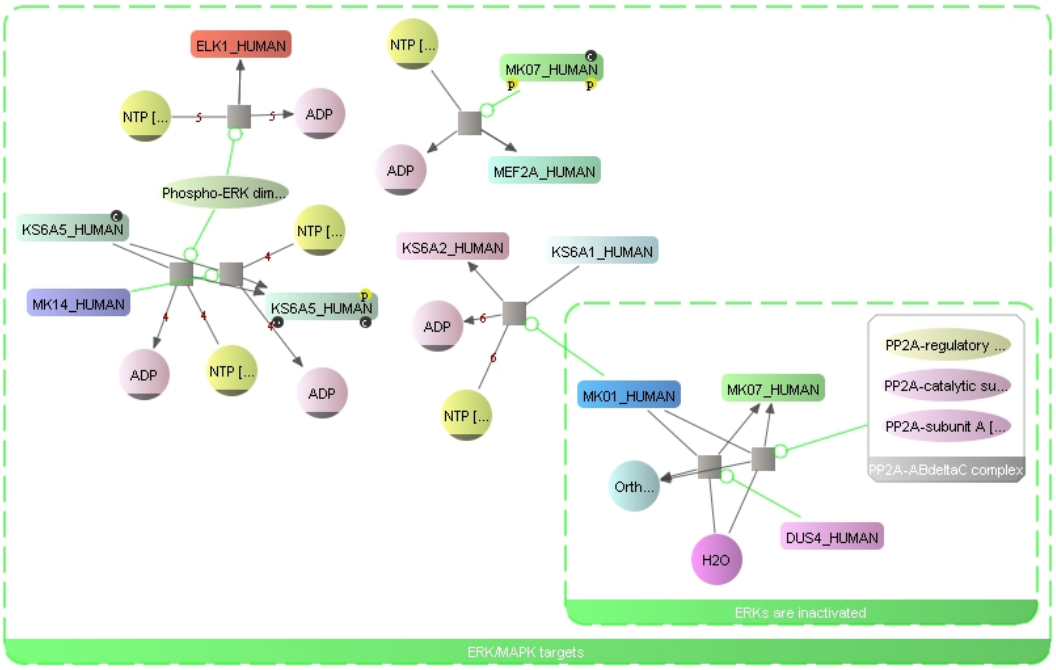

Figure 3.4: The same model in Figure 3.3 is loaded with degree of separation value 5. ADP and NTP molecules are represented with multiple nodes. Each node has a black marker at the bottom, which indicates it is a clone.

3.1.10 Pruning Compound Nodes

The very last step of processing an OWL file is pruning the redundant compound nodes of type compartment or pathway from the graph created. Note that complexes are also represented with compound nodes in VISIBIO*web* but they are out of the pruning process. Either the pathways or the compartments are displayed in the view constructed, hence it is sufficient to prune only the instances of the type that is chosen to be visualized. Pruning a compound node means removing the compound node from the constructed graph and inserting all of its content to the owner graph object of the compound node.

A compartment can be empty or can contain only other compartment as its only child. These situations can occur when the user chooses to apply partial model view and selects some of the pathways as the biological model as opposed to whole file. The pruning is performed by iterating over all the compound nodes created for compartments. Similarly, the overlapping pathways, which were

detected throughout the whole process, are pruned from the graph. The pruning operation is applied recursively for the pathways. All the parent pathways of an overlapping pathway is considered as overlapping, since we prefer not to display any information instead of displaying it as in complete.

3.2 Layout Extension

After a BioPAX model is handled as explained in Section 3.1 and a graph model to represent the biological model is created, a layout is applied on this graph. At the end of layout, the coordinates of the boundaries of nodes and the routing of edges are stored in an XML file. Providing users an XML file containing geometric information is useful, since user might desire to visualize the graph constructed by VISIBIO*web* with their favorite graph visualization tools. As a result, different view of the same graph can be created.

The need for generating an external file resulted from the lack of a geometry extension support in BioPAX. In other words, there is no proper location to store layout information in a BioPAX OWL file. Storing the identifiers of BioPAX elements, for which the graph objects are created, is useful in terms of providing the mapping between the biological and graph models. With respect to what is mentioned so far, an XML schema is developed to store geometrical and topological information of the graph along with references to the BioPAX file.

In this schema, the top most container is called a *view*. A view is composed of nodes and edges. The compound node element extends from the node element and has a child list that contains references to other nodes. The edge element holds references to source and target nodes. Both node and edge elements in the schema extends from the graph object element, which contains references to BioPAX element identifiers. In order to provide extensibility, custom data element is included in the schema. Custom data element allows the addition of any attribute type to the graph object element types. A sample usage of custom data internally in VISIBIO*web* is given in Section 4.2.4. Figure 3.5 displays

an overview of the schema in a diagram. A sample XML file, which is created through VISIBIO*web*, conforming to the schema provided is shown in Figure 3.6. Corresponding pathway view for this XML file is shown in Figure 3.7.

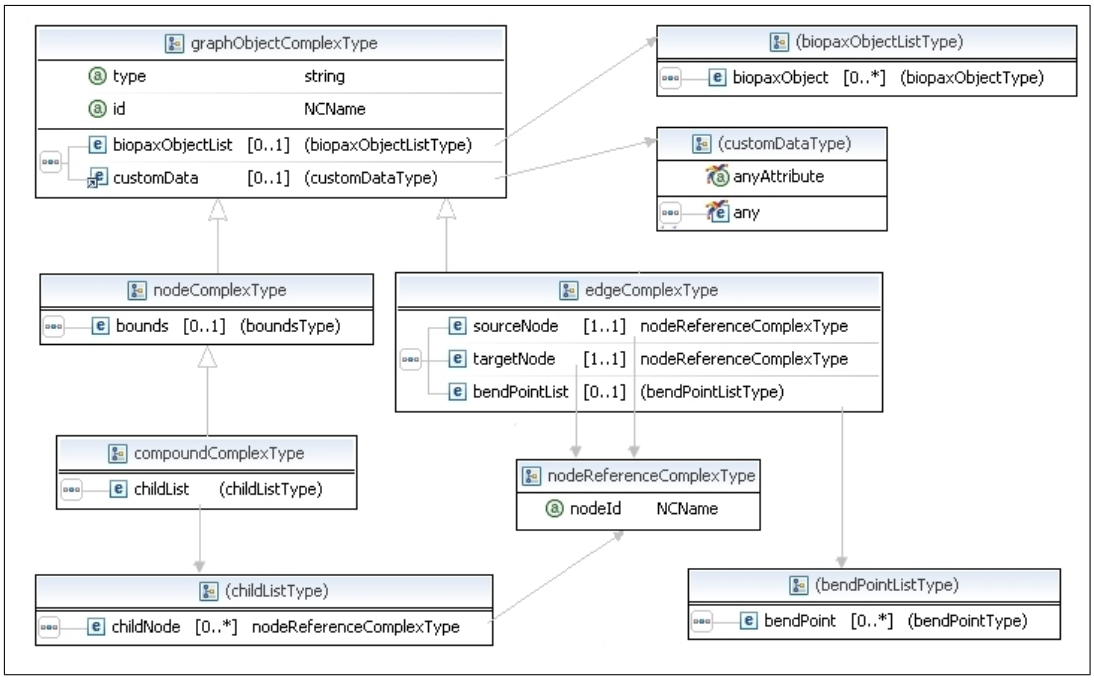

Figure 3.5: An overview of the schema showing important schema elements and their relations.

3.3 Rendering and Tile Generation

Once automatic layout is applied to the graph, what remains is to perform rendering. VISIBIO*web* relies on GEF, SWT and Draw2d libraries to render the graph and create images of the view. Since VISIBIO*web* is a web application, many of the important capabilities of GEF and SWT libraries cannot be benefited from. The rendering of the graph model is achieved by drawing the corresponding figures by using Draw2d on top of an invisible SWT canvas. After the drawing is performed, images of the view at each different zoom levels are created to be used on the client side inside the customized Google Map, which is VISIBIO*web*'s canvas. Each image at different zoom level is tiled into 256x256 sub-images, which are necessary for the customization of Google Maps API.

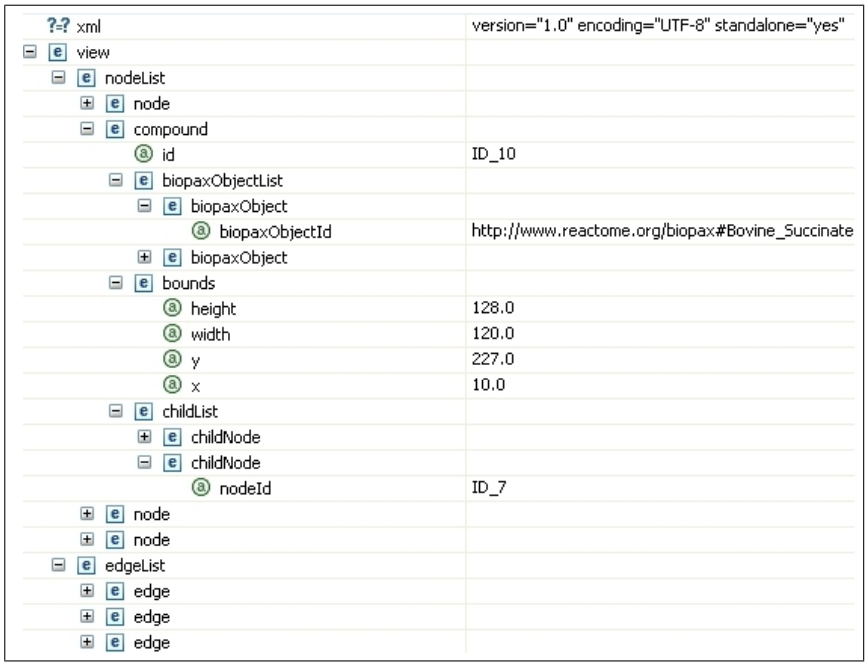

Figure 3.6: A sample XML file conforming to the layout schema.

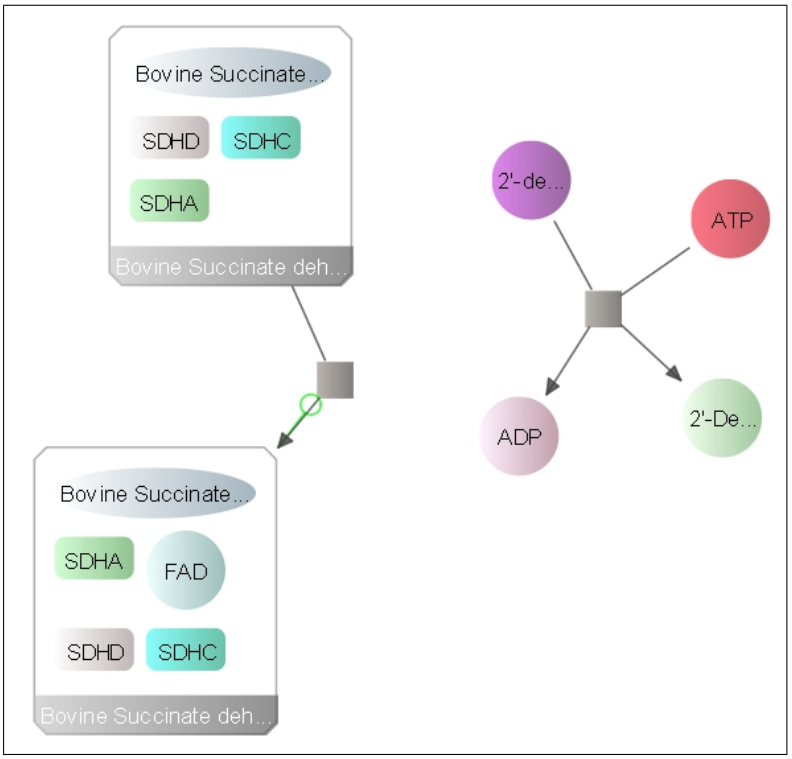

Figure 3.7: Corresponding pathway view of XML file shown in Figure 3.6.

3.4 Google Maps Customization

Although the common use of Google Maps API is to display maps as its name applies, it is not mandatory. In VISIBIO*web* case, we benefit from the API to create a graph visualization canvas embedded inside a web-browser. For this purpose, various customization steps are required. Since the API is very suitable for creating custom applications, these steps are not difficult to implement. To use the API, the first and most important step is to create tiles to be displayed in the map. A Google Map is composed of tiles, each having a size of 256x256 pixels. Once the tiles to be displayed are created, intelligent naming of these tiles are necessary to provide the web-browser access them by simple HTTP requests. More details about this process and other customizations including hit-testing, tooltips, and inspector window properties are given in Chapters 4 and 5.

Chapter 4

VISIBIO*web* Architecture

This chapter describes the architecture of VISIBIO*web*. Before detailing client side and server side architectures separately, we give a system overview.

4.1 System Overview

This section focuses on the overview of VISIBIO*web* architecture as a whole. Figure 4.1 represents the main components of VISIBIO*web* and their interactions. The communication between client and server side is initiated with a file upload event through a web-browser. As soon as this file arrives to the server, a complicated process involving components at server side starts. Requests from client side arrive at Apache Tomcat at the server.

Tomcat Server is the entry point for the execution of server side logic. Although not shown in Figure 4.1, Tomcat executes requested JSP (Java Server Pages) files and delegates the work to the *Session Handler* component. Session Handler component is responsible to manage all server side logic for a user till the end of session. Various files can be uploaded during a session. Session Handler uses services of other components, which are *BioPAX Parser*, *VISIBIOweb Core*, *Layout Manager*, *Tile and Image Generator*, and *XML Generator*; in order to

create the necessary outputs to be used by the components on the client side. The components on the server side are implemented with Java and JSP.

Client side architecture is mainly composed of user interface components. The most important component on the client side is *VISIBIOweb canvas*, which is a customized Google Map as mentioned earlier. VISIBIO*web* canvas is responsible to display the view constructed on server side properly. It also detects various user-oriented, interactive actions and events. The other visual components are *Inspector and Popups*, and *Menu and Toolbars*. These components are complementary elements for VISIBIO*web* canvas to provide additional functionalities and improve usability of the system. The last component on client side is *XML Parser* unit, which has the duty of parsing the geometry XML file sent from the server. An output of this component is polygons, which are provided by Google Maps API, added on top of VISIBIO*web* canvas. The other output is the information to be displayed in the inspector window for associated graph objects. The details of this process will be explained in Section 5.3.

Figure 4.2 is a sequence diagram showing the most common and basic use case scenario of VISIBIO*web*, loading an OWL file. The diagram contains only the most important classes and methods involved in the scenario, the others are discarded not to complicate the diagram any further. The diagram is useful in terms of revealing the interaction between components in both client and server side of the application. Scenario is initiated with a VISIBIO*web* user uploading a BioPAX model to be visualized. Tomcat server forwards the page request to `index.jsp`. Meanwhile, due to the start of a new session, `VWSessionListener` creates a new `VWAppMgr` that is responsible to manage the server side logic. Both of these classes are under Session Handler component. The application manager creates a `BioPAXParser` instance to create a graph model from the biological model in uploaded file. `VWAppWindow` communicates with VISIBIO*web* Core component to create the visual representation of the graph model. `CoseLayotPerformer` represents the Layout Manager component and is responsible to perform the layout. `VWLayoutXMLHandler` class is the most important member of XML Generator component mentioned above. `ImageSaver` and `VWTileCreator` classes constitute Tile and Image Generator component of the

server side. They are responsible to create the static images and tiles to be used on the client side. `VWAppMgr` communicates with the classes mentioned in order and return all material generated to `index.jsp`, which constructs an instance of `GMap2` (the name of the map class in Google Maps API) with URL of the tiles on the client side.

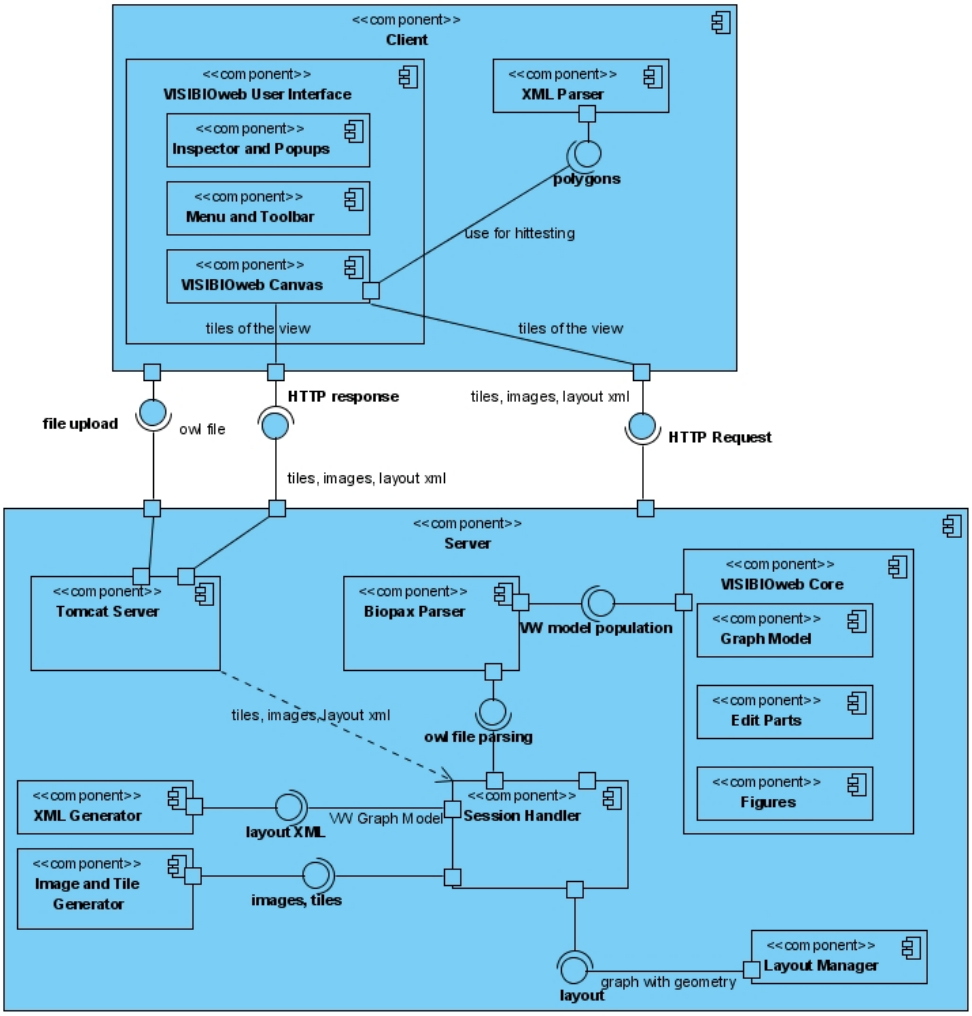

Figure 4.1: An overview of the main components in VISIBIO*web* and their interactions.

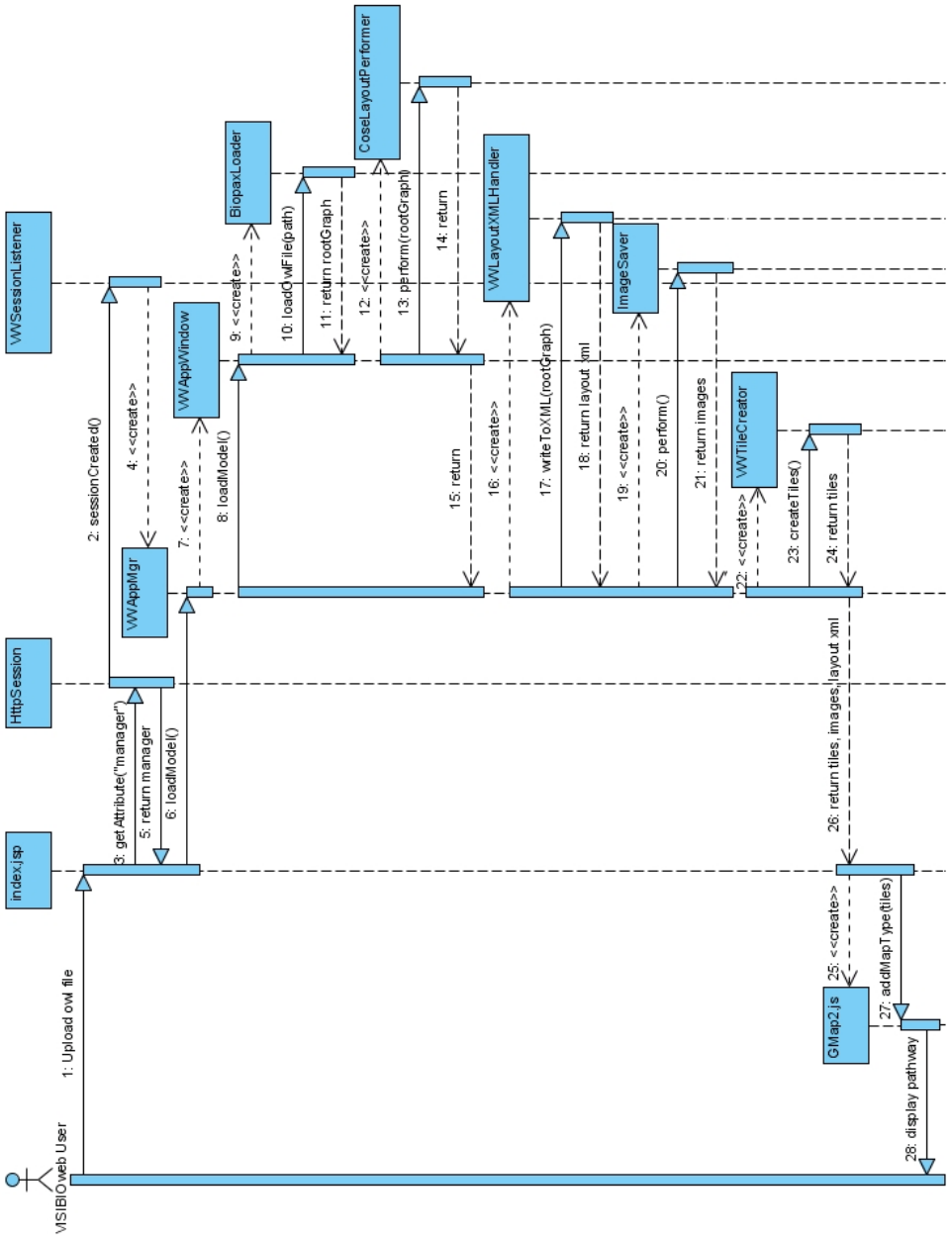

Figure 4.2: The most common scenario in VISIBIO*web* is loading an OWL file.

4.2 Server Side Architecture

Server side is composed of different components as shown in Figure 4.1 above. In order to ease understandability of the architecture only the most important parts are mentioned in following subsections.

4.2.1 Session Handling

In VISIBIO*web*, each user is assigned a session. When, the session for a user is initiated, an application manager is created for that user. This process is shown in Figure 4.2. Application manager is set as an attribute of the session. It is responsible for loading BioPAX model and generating all required information (tiles, svg and png formatted images, graph layout and topology information) for displaying the model on client side. Application manager saves all this information into a folder named with `modelName.timestamp` format on the server side. A user can upload multiple files during the same session, however information related with only the most recent model remains on the server unless the user persists the view for future use. This is needed to minimize the disk space requirement on the server. When the session is terminated the directory associated with session is deleted if not persisted.

4.2.2 VISIBIO*web* Graph Model

Since VISIBIO*web* is built on top of GEF, *Model*, *Figures*, and *Edit Parts* constitute the core components of the server side. GEF is explained in Section 2.4 briefly. The details of the interactions between the parts of MVC pattern such as command structure, and edit policies in GEF are out of the scope of this thesis. Figure 4.3 is a brief summary of these interactions. Edit parts are registered as listeners to the model elements via *Property Change Support* concept in GEF. The edit parts associated with model elements are notified about the changes of the properties such as location and dimension of a node and updates the associated

figure.

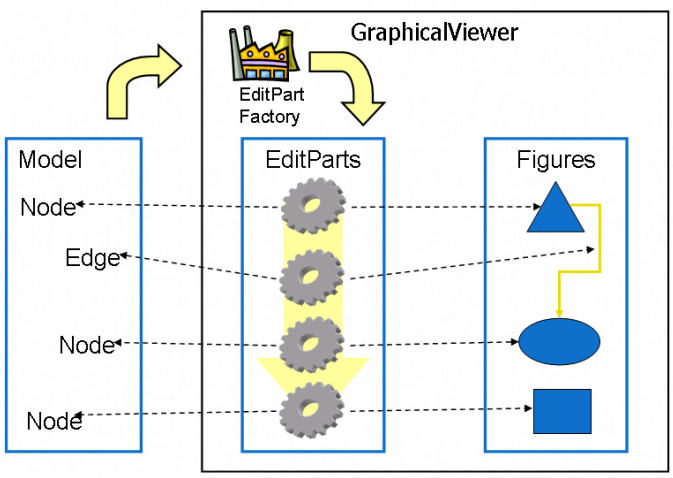

Figure 4.3: VISIBIO*web* MVC pattern application.

The model part in VISIBIO*web* is the graph data structure. Model part is designed to provide flexibility for the integration of support for biological formats other than BioPAX. Figure 4.4 is the class diagram of the graph model in VISIBIO*web*. Many of the simple accessor and mutator methods are not included in the diagram to increase readability. As shown in the figure, there is nothing related with BioPAX format; that is the graph model is independent of BioPAX format. BioPAX related data and constraints are stored in *Roles and Role Managers*, which will be explained later in this section. On top of the model hierarchy, `GraphObject` is introduced to provide property change support concept in GEF and role manager association. `Node` has in-edges and out-edges lists, where as `Edge` has associated target and source nodes. `CompoundNode` extends from node and implements `Compound` interface to support nesting of nodes in VISIBIO*web*. `RootGraph` also implements *Compound* interface and it is the top most structure in which all graph elements reside. `ModelFactory` is an abstract class to create instances of `GraphObject` and `RoleManager` classes. It must be overridden for each biological format separately. In VISIBIO*web* there is `BiopaxModelFactory` that extends from this class to create BioPAX related roles and role managers.

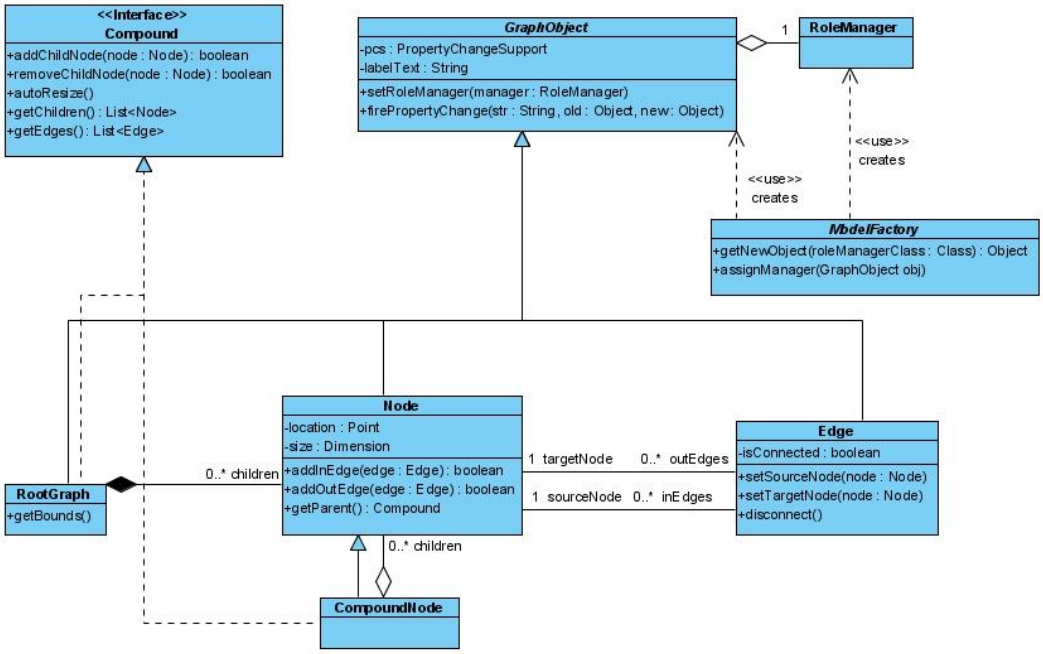

Figure 4.4: Graph model class diagram.

4.2.2.1 Role - Role Manager Concept

In order to benefit from the same graph model for different biological formats or even for a totally different application area, *Role Manager* concept is introduced to VISIBIO*web*. In this concept, the graph objects (node, edge, and compound node) are assigned so called *roles*, which determine the behavior of that specific graph object.

This concept was originally inspired from the Player-Role pattern, but the use is somewhat different. The player (a graph object in our case) does not change role during their life cycle. Instead, each graph object is assigned one or more roles that define its appearance and behavior. For instance there will not be two different classes extending from *Edge* for *Product* and *Catalysis* types in BioPAX. They will represented with the same object, Edge. The roles assigned to that edge determine the difference between being a *Product* or *Catalysis* in terms of behavior and appearance.

As described above, the roles assigned to a graph object will be the key point

in determining its behavior. As a matter of fact, the content of the role class has to answer various questions. For instance, for a node role, it must answer questions such as "Can it attach as source to a specific edge type?" or for a compound role "Can it contain a specific node type?". In order to increase usability of the structure, the design must support a graph object with multiple roles. In order to manage the multiple roles, *Role Manager* concept emerged. A role manager typically answers various questions by traversing the roles attached to it. Those questions must be handled in a way to handle necessary situations such as when the Boolean questions are to be ANDed or ORed.

In VISIBIO*web*, all BioPAX format specific constraints in terms of behavior and visualization are implemented in customized roles and role managers. For instance, for *Protein* entity type, `ProteinRoleManager` extending from `NodeRoleManager` and `ProteinRole` extending from `NodeRole` are introduced. BioPAX related information such as the sequence features, short names, and synonyms are stored in information classes extending from `Info` class.

Figure 4.5 is the class diagram showing the relations between role managers and roles. A very similar class hierarchy exists between the Role and Role Manager inheritance hierarchies. Moreover, the methods in role managers are the same as the ones in roles. For instance both `NodeRole` and `NodeRoleManager` classes have `isResizable()` method. This is not surprising, since the task of a role manager is to traverse through roles assigned and determine the behavior for the collection of the roles.

This concept is more meaningful and powerful for an interactive application than that of VISIBIO*web*. In VISIBIO*web* case, all the roles, role managers and graph object are created programmatically. However, in an interactive tool, a user might try to perform disallowed operations such as putting a pathway inside a complex in BioPAX domain. In such a case, the role manager for complex type will prevent the addition of a pathway inside.

An alternative approach to this concept is extending from the generic graph model displayed in Figure 4.4. In such an approach, domain-specific constraints would be included in modules containing graph structure specific information.

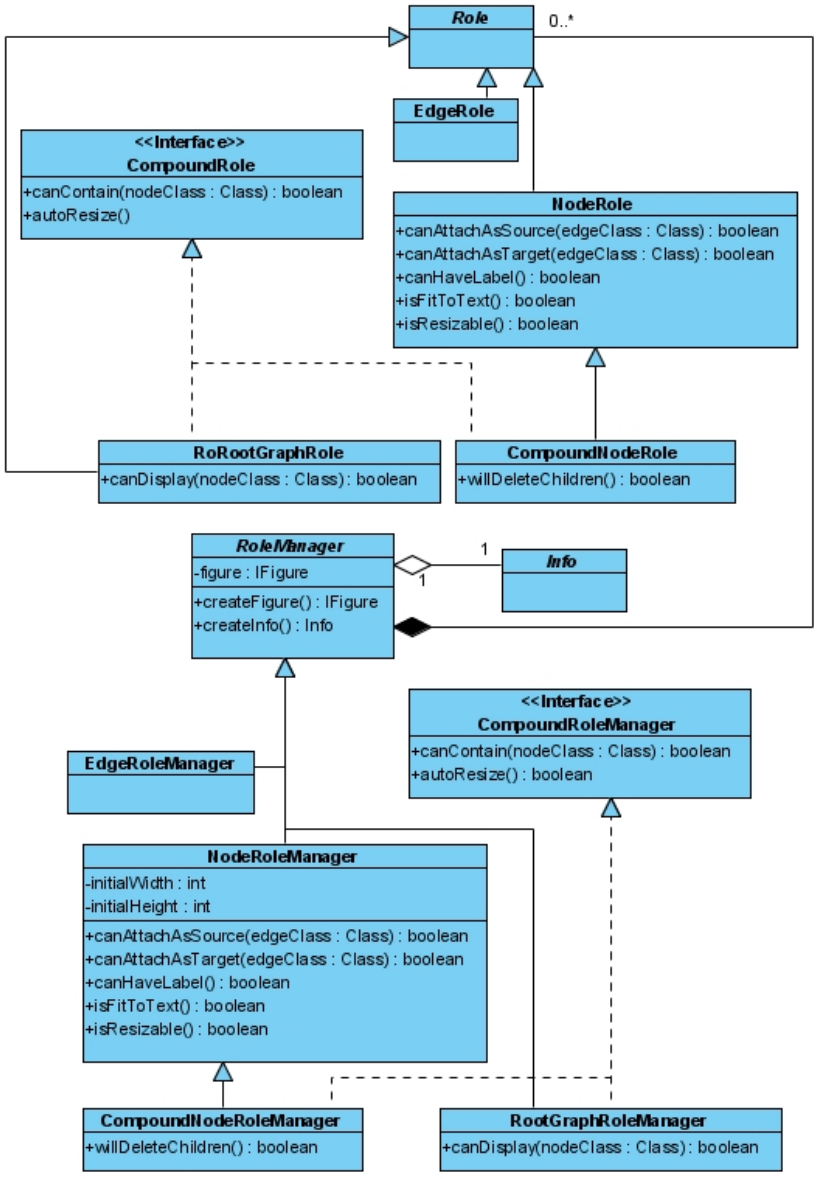

Figure 4.5: Role - Role Manager class diagram.

This would result in a violation of separation of concepts and duplication of code. Figure 4.6 is an illustration of such a case. The diagram shows modeling of another biological format, PMDL (PATIKA model), with conventional extension method and role manager concept. As the note in the diagram states, *Homology* concept related code is duplicated in the hierarchy at the left, since Java does not support multiple inheritance. If there is a change in that concept, the code must be changed in both *HomologyTransition* and *HomologyAbstraction* classes. However, the hierarchy at the right side provides better re-usability. *Homology* concept is modeled with its own role *HomologyRole*, which prevents duplication of code. *HomologyAbstractionRoleManager* has *HomologyRole*, and *AbstractionRole*; whereas *HomologyTransitionRoleManager* is composed of *HomologyRole*, and *TransitionRole*.

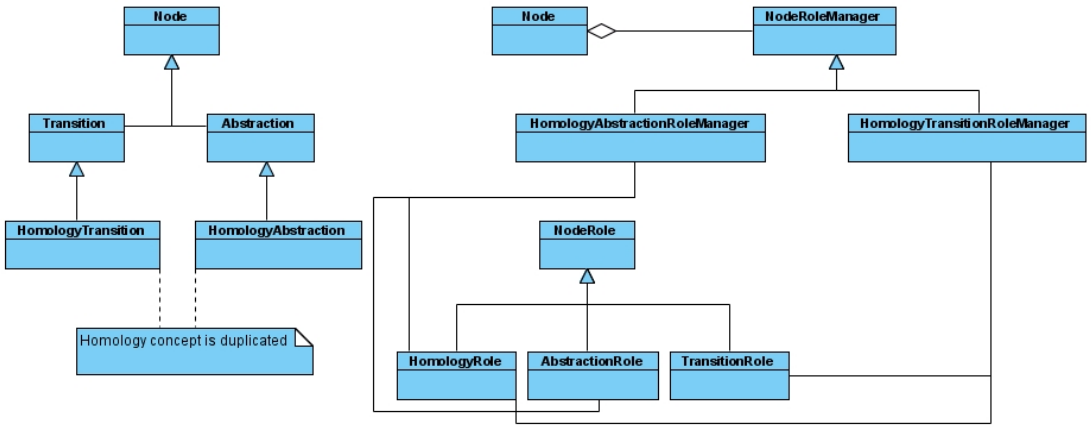

Figure 4.6: Role Manager vs. Graph Model Extension Comparison.

4.2.3 Layout

Layout is one of the most important components of the server side architecture. The success of layout component is critical in terms of generating esthetically nice views of the the graph constructed. For the layout of compound structure, the algorithm described in [14] is preferred. The algorithm is already implemented in Chisio (Compound or Hierarchical Graph Visualization Tool) [8] by i-Vis group [21]. The layout is performed on top of a structure named l-structure,

the details of layout component is explained in [9]. Before the layout takes place, l-structure is created from graph model in VISIBIO*web*. After the layout finishes, geometric information is transferred from l-level objects back to the VISIBIO*web* graph model objects.

Although the general structure of layout is preserved in VISIBIO*web*, a slight modification is performed for special treatment of compound nodes representing instances of *Complex* type in BioPAX. Complex type contains other biological entities such as Protein, DNA, RNA or Small molecule. Instead of applying a spring embedder based layout to complexes, it is better to apply "tiling" and obtain compound nodes having as tight boundaries as possible. Tiling means placing the members of the complex into it in rows and columns just like tiles. The layout design existing in Chisio is extended in order to perform tiling customization. Figure 4.7 shows the effect of tiling customization performed in VISIBIO*web*. The complex in this figure has tighter bounds than the one shown in Figure 4.8

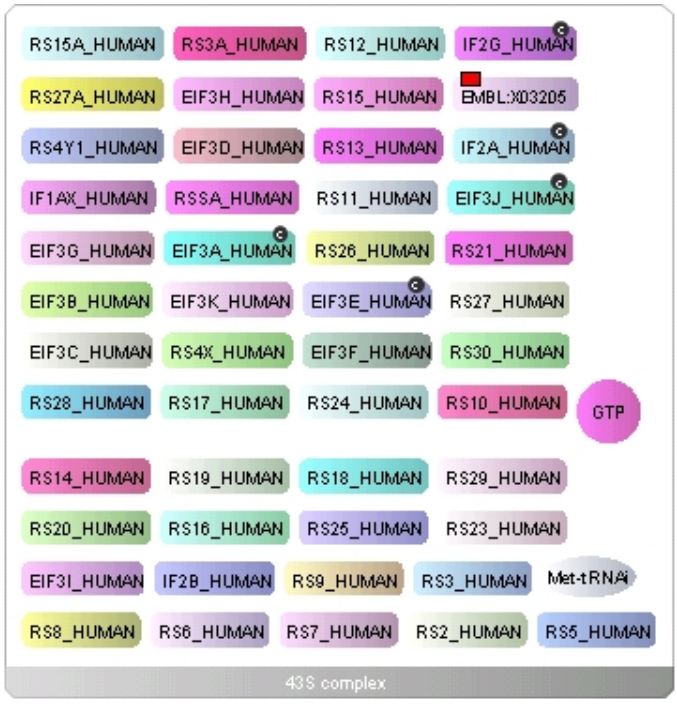

Figure 4.7: Application of tiling to a complex type compound node in VISIBIO*web*

Figure 4.9 illustrates the customization performed in VISIBIO*web* in order

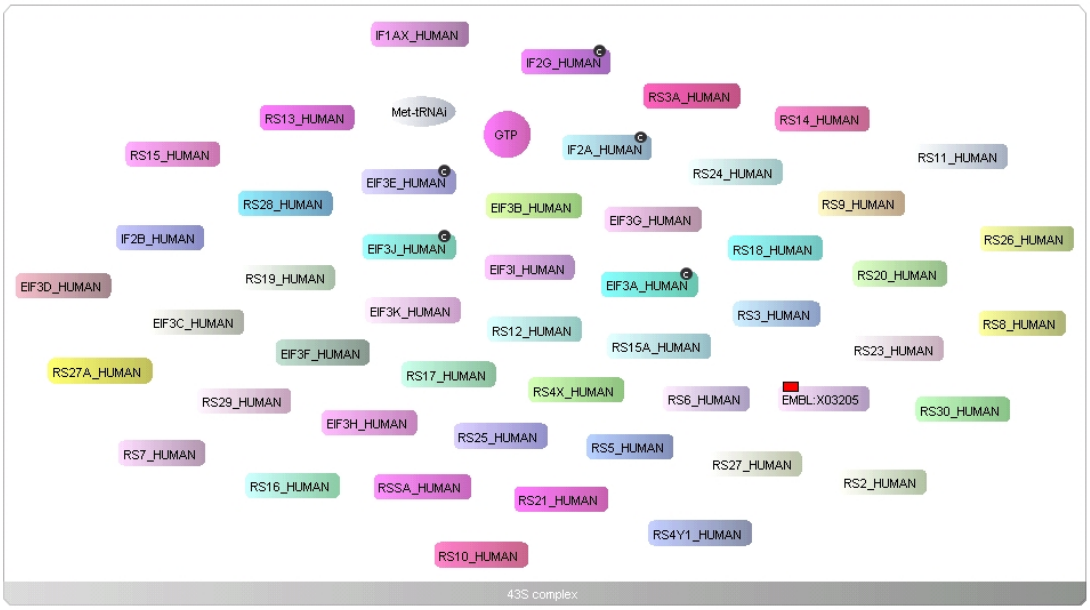

Figure 4.8: The same compound node in Figure 4.7 without tiling applied during layout

to apply tiling for complexes. The layout algorithm and node in l-structure, `LNode`, are extended and additional functionality are implemented in the sub-classes, `BioPAXLayout` and `LBioPAXNode`. The tiling logic is implemented in helper classes, `Tiler` and `MemberPack`.

The first task performed in *layout* method in `BioPAXLayout` class empties the complexes by storing the complex and member relationships in a map and crates instances of `MemberPack` and `Tiler` to perform tiling for members of the complex. After the complexes are emptied, they are treated just like simple (non-compound) nodes and the *layout* method of super class is executed. As the last step, the complexes are repopulated with their members, which are already applied tiling. In applying tiling, the aspect ratio of the complex is tried to be kept at 1, after the width of the complex becomes greater than that of a desired width value. The greedy approach applied to balance the height and width of the complex is explained below:

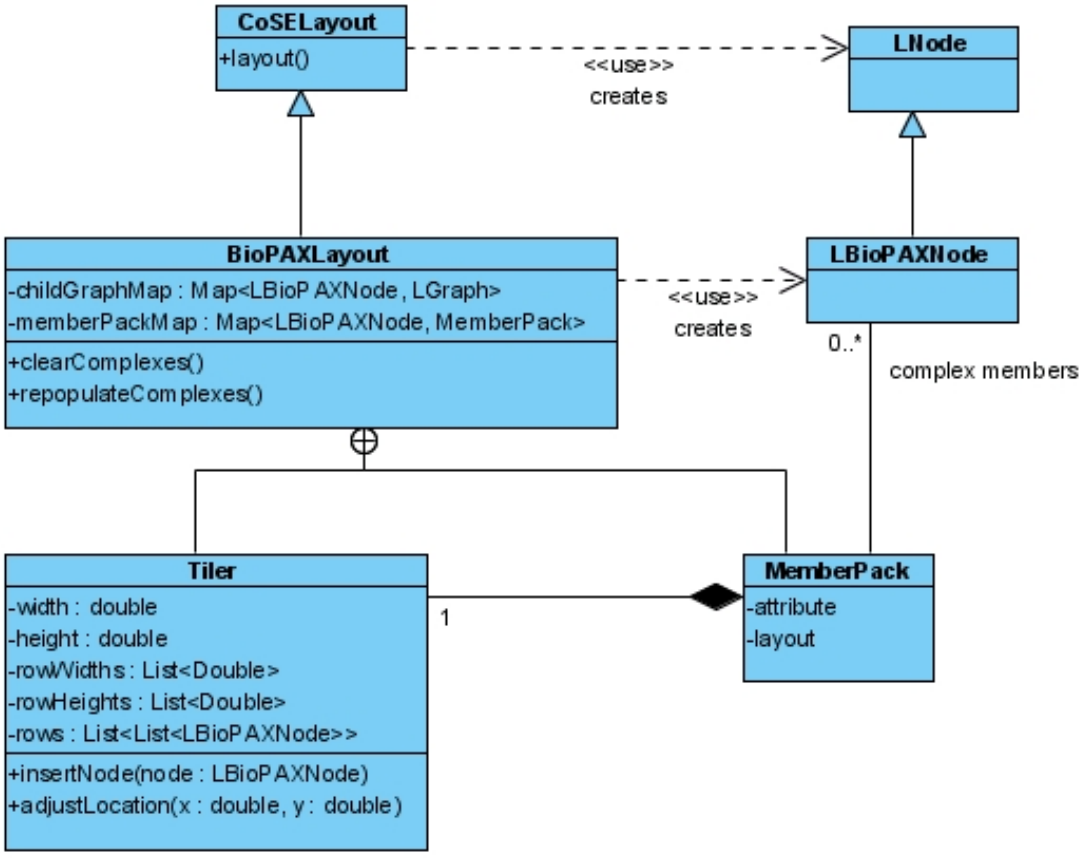

Figure 4.9: Class diagram illustrating the layout customization in VISIBIO*web*

```
method applyTiling(complex, complexMembers)
1) sort complexMembers in descending order with respect to their widths
2) rows := new list of lists of LBioPAXNode instances
3) for m ∈ complexMembers do
4)   if rows is empty then
5)     add m to rows(0)
6)   else
7)     shortestRow :=find row with smallest width
8)     newWidth := m.width + shortestRow.width
9)     if newWidth smaller than complex.width then
10)      add m to shortestRow
11)    else if newWidth smaller than desired complex width then
12)      add m to shortestRow
13)    else if newWidth smaller than complex.height + m.height then
14)      add m to shortestRow
15)    else
16)      newRow :=create a new row and add to rows
17)      insert m to newRow
18)  Update complex.width and complex.height
```

Figure 4.10: The algorithm explaining tiling customization applied to layout

4.2.4 XML Generation

The result of layout is stored in a separate XML file along with references to BioPAX element references as explained in Section 3.2. The same component generating this file is also responsible to create another XML file for internal use. This file is not provided to users and sent to the client side to detect the graph objects and display their properties in the inspector window. This XML file is also based on the same schema provided in Figure 3.5. It benefits from the *customDataType* element shown in the XML schema. The additional elements added as custom data are bounding polygon and property list. The bounding polygon is composed of four point elements, which store information about the rectangular area of the graph object on the client side. The property list is a collection of name, value pairs such as name of the represented BioPAX element, external references, and comments. This information is extracted from the uploaded file and stored in *Info* objects of the role managers of corresponding graph objects. Figure 4.11 illustrates the content of a sample extended XML with bounding

polygon and property list of a node element.

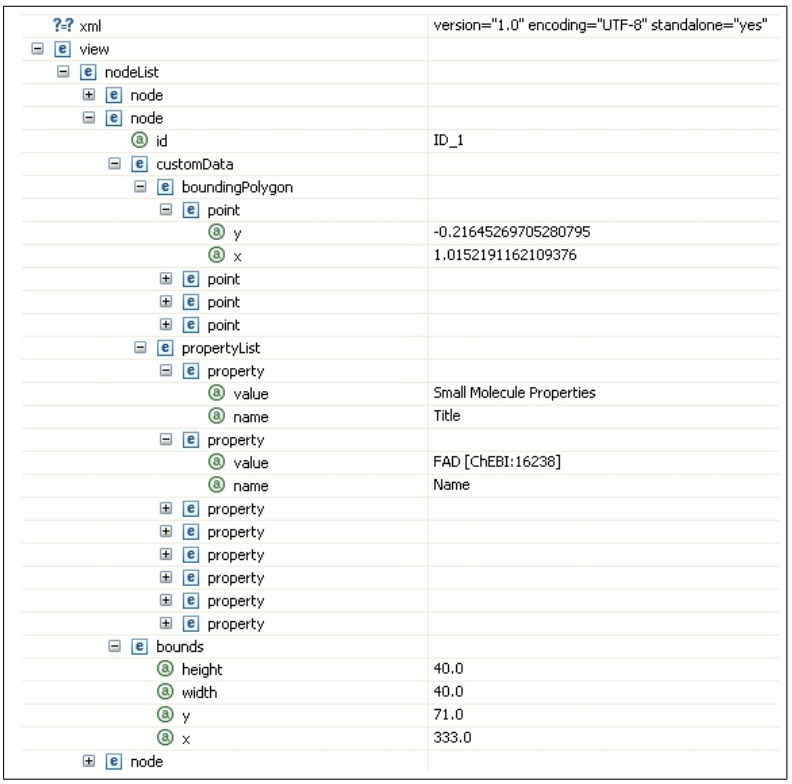

Figure 4.11: Extended XML to be parsed on the client side

As can be noticed from the XML, the bounding polygon element's point has very small double values as opposed to the bounds element of node, which has greater integer values as x and y. This is due to the reason that a transformation is applied for the boundaries of the graph objects. The algorithm to generate custom data for a graph object is given below:

4.3 Client Side Architecture

As explained earlier, the most important component in client side is VISIBIO*web* canvas. It is is built on Google Maps API, a pure Ajax compliant JavaScript library. Other components are built by using DHTML and JavaScript.

method CREATECUSTOMDATA(*graphObject*)
1) sort *complexMembers* in descending order with respect to their widths
2) *customData* := create new custom data element
3) *boundingPolygon* := create new bounding polygon element
4) **if** *graphObject* is instance of node **then**
5) *boundingPolygon* := *graphObject.boundary*
6) **else if** *graphObject* is instance of edge **then**
7) *boundingPolygon* := create a bounding rectangle to cover the edge
8) apply transformation to *boundingPolygon* (GEF to Google Map coord.)
9) insert *boundingPolygon* to *customData*
10) *propertySet* := $graphObject.roleManager.info.properties$
11) *propertyList* := create new property list element
12) **for** $propertyName \in propertySet$ **do**
13) *property* := create a new property element
14) *property.name* := *propertyName*
15) $property.value := propertySet.getValue(propertyName)$
16) insert *property* to *propertyList*

Figure 4.12: The algorithm explaining XML file generation

4.3.1 VISIBIO*web* Canvas

VISIBIO*web* canvas is a customized Google Maps API, which displays pathway views instead of maps. The customization process is in fact initiated by the tile generation component on the server side. The tiles are named intelligently in order the client side to receive correct tiles of the view. A wrapper JavaScript object called `VWMap` is created on the client side to manage Google Maps API's `GMap2` object's functionalities. Figure 4.13 shows attributes and methods of `VWMap`. The construction of the actual map is performed inside `constructMap` method, whose details are explained in Section 5. Among the attributes listed in the figure, the ones starting with a capital "G" are provided by Google Maps API.

4.3.2 XML Parsing

An XML file containing information about the boundaries of the graph objects and data extracted from BioPAX model is sent to the client side as explained in Section 4.2.4 earlier. This file is parsed at the client side and the information is

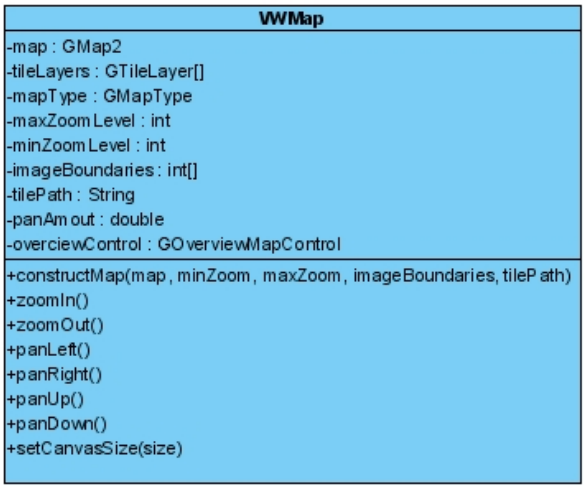

Figure 4.13: VWMap class diagram.

stored in JavaScript objects. A very simple design is used to store all information parsed Figure 4.14. `VWPolygon` objects are created with respect to bounding polygon element properties in the XML file. They are added on top of the map and used Google Maps API's event listener mechanism is used to perform both hit-testing and tooltips. These are explained in Section 5.3. `GraphObject` stores relevant information to be displayed in the inspector in terms of `Property` objects via holding reference to an instance of `PropertyList`.

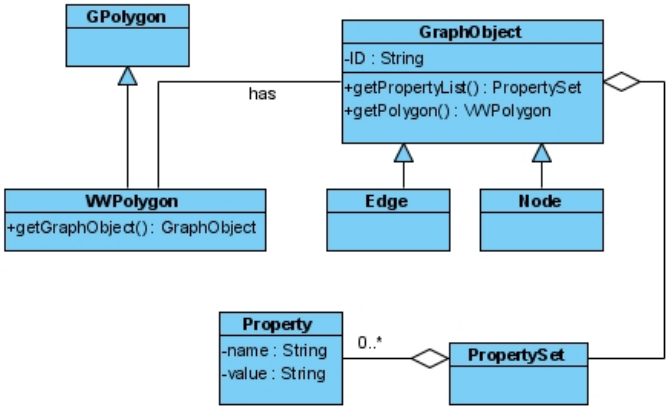

Figure 4.14: Class diagram showing client side model.

Chapter 5

Implementation Details

This chapter includes implementation details of the important components that make up VISIBIO*web*. It is not possible to discuss and examine the implementation of every component, so crucial ones among them are chosen for explanation. Implementation details about about more components of VISIBIO*web* are continued in Appendix A.

5.1 Tile Generation and Access

A view constructed for a biological pathway model is saved as 256x256 tiles for use on the client side as explained earlier in Section 3.3. As mentioned in Section 4.3.1 the tiles generated on server side are named properly for VISIBIO*web* canvas to retrieve correct set of tiles for visualization. This section explains how the retrieval of viewable set of tiles is triggered from the client side as well as required naming convention applied on the server side.

Google Maps is customized to display tiles generated for pathway views constructed instead of regular maps. This customization is achieved by overriding the `getTileUrl(GPoint point, Number zoomLevel)` method of `GTileLayer` interface provided by the API. The point passed as a parameter to this method

contains (x, y) pair as a reference to the coordinate of the tile, whose URL is needed. The second parameter is the current zoom level of the map displayed. Tile coordinates and zoom levels are explained earlier in Section 2.5. In order to benefit from this method, the URL of the tile to be displayed must be derivable from tile coordinates and the current zoom level. With this requirement at hand, the tiles are saved with respect to their tile coordinates and zoom level. To achieve this, the tile coordinates must be calculated on server side.

VISIBIO*web* tries to center the map to a specific latitude and longitude value and VISIBIO*web* zoom levels are map zoom levels 8-11 among 19 available zoom levels of Google Maps. This means zoom level 0 in VISIBIO*web* terminology is zoom level 8 in Google Maps. The tile coordinates are calculated on the server according to the number of tiles to be generated at a particular VISIBIO*web* zoom level. Once tile coordinates are calculated, an image file with `.png` extension for each tile is named by concatenating following strings: `Zoom Level_`, `<x tile coordinate value> x`, and `<y tile coordinate value>`. Figures 5.1 and 5.2 are tiles generated in VISIBIO*web* for a sample BioPAX model. The first one is at the smallest zoom level displaying entire model, where as the latter one is displaying portion of the model at a higher zoom level.

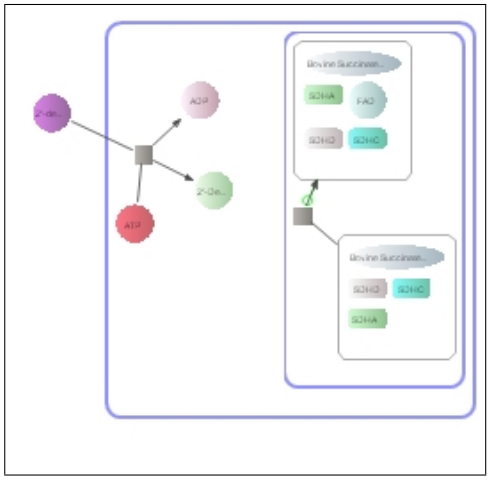

Figure 5.1: A tile generated by VISIBIO*web* at smallest zoom level with file name *0_128x128.png*.

Along with the tile name convention adapted on server side, the tile boundaries for each zoom level are also sent to the client side as an array for boundary

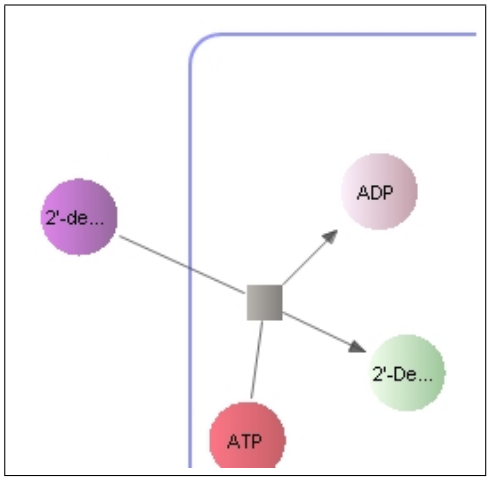

Figure 5.2: A tile displaying upper left corner of tile displayed in Figure 5.1 at a higher zoom level with file name *1_256x256.png*.

checking to be performed in requesting generated tiles. The array is composed of 16 values, each group of 4 holding tile boundaries for one VISIBIO*web* zoom level. This means, array indexes 0 to 3 are used for tile boundaries of VISIBIO*web* zoom level 0 and 12 to 15 are used for zoom level 4. Another information sent to the server to retrieve the tiles is the path to the tiles directory on server. The algorithm used in `getTileUrl(GPoint point, Number zoomLevel)` method to retrieve correct set of tiles in VISIBIO*web* is given below. The boundary array sent from server is named `tileBoundaries` and the tile directory path is named `tilePath` in the algorithm.

method GETTILEURL($point, zoomLevel$)
1) $index := (zoomLevel - 8)$
2) $minX := tileBoundaries[4 * index]$
3) $minY := tileBoundaries[4 * index + 1]$
4) $maxX := tileBoundaries[4 * index + 2]$
5) $maxY := tileBoundaries[4 * index + 3]$
6) **if** $point.x <= maxX$ and $point.x >= minX$
   and $point.y <= maxY$ and $point.y >= minY$ **then**
7) return $tilesPath$ + "/" + $index$ + "_" +
   ($point.y$) + "x" + ($point.x$) + ".png"
8) **else**
9) return URL of empty image on the server

Figure 5.3: The algorithm explaining accession of tiles from client side.

5.2 Dynamic Zoom Level Calculation

BioPAX files can be in varying size, which means amount of biological data stored inside .owl files are not same. This results in construction of graphs with different number of nodes and edges for different BioPAX models, hence varying size of graph boundaries.

Since a tile in a Google map is composed of 256x256 pixels, the number of tiles depend on the size of graph constructed. For instance if a graph with boundary of size 2400x1026 is created, 50 tiles must be generated to cover view to be constructed for the graph. If this boundary is used to generate tiles of default zoom level in VISIBIO*web* (which is zoom level 1), 13 tiles for smallest zoom level, 200 tiles for zoom level 2, and 800 tiles for zoom level 3 (highest zoom level in VISIBIO*web*) are needed. This makes a total of 1063 tiles. As the boundary of the graph increases, tile generation starts to be a bottleneck for the performance of server. As a result, it is not practical generating tiles with respect to constructed graph size only.

In order to tackle this performance problem, we introduce a new zoom level schema, dynamic server side zooming. This new zooming is not related with the zoom levels of Google map explained earlier. Dynamic server side zoom level is related with performing zoom operation on the view with the methods provided by Eclipse's GEF framework. If the boundary of a graph is greater than 512x512, the tiles for the default zoom level of VISIBIO*web* canvas is restricted. The graph is fit to 1024x1024 boundary via performing zooming. Hence the maximum number of tiles to be generated on the server side is 4 for the smallest zoom level, 16 for default zoom level, 64 for zoom level 2, and 256 for highest zoom level. Total tile number generated is then 340 at maximum. Since each view is fit to 512x512 or 1024x1024 size (if graph's boundaries exceed 512x512), server side zoom level applied on the views for different biological models are calculated dynamically. If this dynamically calculated zoom level is smaller than 0.001, than no tiles are generated and a warning message is displayed to the user to indicate the situation. We choose such an approach, since tiles generated with dynamically calculated zoom level smaller than 0.001 do not give any insight

about the biological model because of the nodes being too small visually.

5.3 Hit-Testing and Inspector Window

What is displayed inside a VISIBIO*web* canvas is a collection of .png images, which are named tiles, as explained earlier in Section 5.1. Since graph objects are part of these images, detection of them must be implemented via using event mechanism provided by maps API. This is called hit-testing in VISIBIO*web* terminology. In order to benefit from events of the API, `GPolygon` instances must be created for each graph object. These polygons need earth coordinates in terms of latitude and longitude values to be added as an overlay on to the VISIBIO*web* canvas. The coordinate values are sent from the server side to the client side inside an XML file as explained in Section 4.2.4. The polygon boundaries sent inside this XML file are already in Earth coordinates. A transformation from GEF coordinates to earth coordinates is performed on the server, before polygon boundaries are written to the XML file. A dynamic zoom level value, which is explained in Section 5.2, is used in applying the transformation.

Once an extended XML file containing earth coordinates for polygons to be created is constructed on client side, it is parsed and polygons, graph objects, and association relation between them are created with respect to Figure 4.14. Here is the algorithm used in parsing the XML file:

Note that it is important to create polygons for graph objects in the order: compounds, nodes, and edges. This is due to the reason that we would like to detect edges instead of nodes, and nodes instead of compound nodes in case of overlapping polygons. Compound nodes are written to XML file in a top-down approach, which means inner-most compound nodes are written later. This provides hit-testing mechanism to detect inner most polygon among overlapping polygon boundaries.

In order to perform hit-testing, VISIBIO*web* relies on `click(overlay: GOverlay, latlng:GLatLng, overlaylatlng:GLatLng)` event of `GMap2` object

method LOADGRAPH($xmlFile$)
1) $xmlDoc$:= create new document with respect to browser by using $xmlFile$
2) $compounds := xmlDoc.getElementsByTagName(compound)$
3) **for** $c \in compounds$ **do**
4) $polygon$:= create new polygon with $c.getElementsByTagName(point)$
5) $properties$:= create prop. list with $c.getElementsByTagName(property)$
6) $node := newNode(polygon, properties)$
7) $polygon.graphObject := node$
8) add $polygon$ to VISIBIOweb canvas
9) $nodes := xmlDoc.getElementsByTagName(node)$
10)**for** $n \in nodes$ **do**
11) $polygon$:= create new polygon with $n.getElementsByTagName(point)$
12) $properties$:= create prop. list with $n.getElementsByTagName(property)$
13) $node := newNode(polygon, properties)$
14) $polygon.graphObject := node$
15) add $polygon$ to VISIBIOweb canvas
16) **for** $e \in edges$ **do**
17) $polygon$:= create new polygon with $e.getElementsByTagName(point)$
18) $properties$:= create prop. list with $e.getElementsByTagName(property)$
19) $edge := newEdge(polygon, properties)$
20) $polygon.graphObject := edge$
21) add $polygon$ to VISIBIOweb canvas

Figure 5.4: The algorithm explaining parsing of the XML file on the client side.

provided by Google Maps API. A reference to currently selected polygon is held to display associated graph object's properties inside the inspector window. The algorithm for *click* event listener is given below. In the algorithm, if the clicked point on VISIBIO*web* canvas is inside a polygon, it is passed as an *overlay* argument to the event listener. In that case, if there is a previously selected polygon, it is hidden and the new polygon is made visible. In the meantime, the properties of the associated graph object is put inside the inspector window. Inspector window is displayed for the inspection of properties of the selected graph object.

method CLICKEVENTLISTENER($overlay, latlng, overlaylatlng$)
1) **if** *overlay* is not null **then**
2) $clickPoint := map.fromLatLngToContainerPixel(overlaylatlng)$
3) **if** *currentSelectedPolygon* is not equal to *overlay* **then**
4) **if** *currentSelectedPolygon* is not null **then**
5) hide *currentSelectedPolygon*
6) *currentSelectedPolygon* := *overlay*
7) show *currentSelectedPolygon*
8) clear inspector table
9) populate inspector table with *overlay.graphObject.properties*
10) show inspector window at *clickPoint*
11) **else**
12) **if** *currentSelectedPolygon* is not null **then**
13) hide *currentSelectedPolygon*
14) *currentSelectedPolygon* := null
15) clear inspector table

Figure 5.5: The algorithm explaining hit-testing and inspector window population.

Chapter 6

Conclusion

In this thesis, a new open-source, free, easy-to-use, and web-based pathway visualization tool for BioPAX formatted biological pathways is introduced. This tool, named VISIBIO*web*, runs on Apache Tomcat server and is implemented in Java. The visualization on the client side is performed by a customized Google map.

VISIBIO*web* provides basic graph viewing functionalities such as zooming, scrolling, and selection of graph objects along with the inspector window facility to display properties of the selected graph object. The view generated for underlying biological model can be saved as static images in SVG and PNG formats. The biological models can also be persisted and embedded within other web sites just like an ordinary Google map. The layout information is also available in an XML file. The generation of such a format with respect to an XML schema is a good starting point in developing an official layout extension for BioPAX format.

6.1 Contribution

There already exist software systems developed to visualize biological pathway models. However, none of these tools fully satisfy the needs of the pathway

community in terms of capability and usability. Some of them are not are not very good at managing the complexity of the underlying biological model due to lack of compound graph support. Whereas some others lack a visual representation standardization, which prevents them to be used frequently. VISIBIO*web* is one of the few software tools supporting SBGN standard notation. Among these software tools, only a few of them are web-based. Thus, many of them require long installation steps. However, the end-users prefer easy-to-use and learn software systems possibly without the need of an installation step. VISIBIO*web* is unique in satisfying criteria described above. Thus, it is expected to fill an important gap in the area of biological pathway visualization.

6.2 Future Work

Although current version of VISIBIO*web* is expected to fulfill needs of many users, there is still room for improvement for future versions. Possible future extensions are as follows:

1. VISIBIO*web* supports BioPAX model. Data interpreted in VISIBIO*web* is related with state level information of biological entities, however there is also more abstract level of data available in BioPAX model, which represent entity level relations as opposed to state level interactions. This data can also be visualized in VISIBIO*web* in upcoming versions. As mentioned earlier, VISIBIO*web* visualizes BioPAX Level 2 models. BioPAX Level 3, which is not officially released yet, should also be integrated to VISIBIO*web* as a future work.

2. New node types can be integrated to VISIBIO*web* to represent *control* interaction types better. Inclusion of a *control* node is sure to increase modeling capability of VISIBIO*web*.

3. Other biological model formats can also be supported in later versions. SBML (System Biology Markup Language) is the first one to be supported

in future versions, since it is popular in storing and exchanging data related with simulations of metabolism, and cell-signaling [32].

4. Current version of VISIBIO*web* relies on Google Maps API for client side visualization. However, to benefit from Google Maps on the client side, generation of tiles is required on the server side. This step is time consuming and current bottleneck of the system. Later releases of VISIBIO*web* may go with other client side visualization technologies to increase performance. Adobe Flex [15] might be a good candidate to perform client side visualization. Integration of Flex to the client side decreases the work load on the server side considerably. The only task of server side would then be parsing of BioPAX model, construction of graph model, layout service, and generation of layout XML. Possible transition from Google Maps to Adobe Flex for client side architecture can also provide support for interactive operations. Current version of VISIBIO*web* does not allow interactive graph editing facilities such as moving, deleting, and addition of pathway objects.

5. As explained in Section 5.2, the number of tiles to be generated is limited due to the performance reasons. As a result, the level of detail for large graphs decreases considerably. These large graphs generally constitute of many pathways. In order to get smaller graph boundaries and a better and more detailed view, collapsing mechanism might be used. Collapsing means, hiding the details of a compound node. It involves representing the compound node with a simple node in the view and hiding the children nodes. By collapsing some of the pathways in the view, the graph boundaries could be significantly decreased and an overview of the interactions between many pathways could be represented with better performance.

Bibliography

[1] O. Babur, U. Dogrusoz, E. Demir, and C. Sander. ChiBE: an analysis and visualization tool for BioPAX pathway models. private communication, unpublished manuscript, 2009.

[2] BiNoM - Biological Network Manager Cytoscape plug-in. `http://bioinfo-out.curie.fr/projects/binom/`, Accessed in August 2009.

[3] Biocarta - Charting Pathways of Life. `http://www.biocarta.com/pathfiles/h_RELAPathway.asp`, Accessed in August 2009.

[4] BioCyc - Collection of Pathway/Genome Databases. `http://biocyc.org/`, Accessed in August 2009.

[5] BioPAX - Biological Pathway Exchange. `http://sbml.org/Main_Page/`, Accessed in August 2009.

[6] BioPAX Level 2 Documentation. `http://www.biopax.org/release/biopax-level2-documentation.pdf`, Accessed in August 2009.

[7] CellDesigner - A modeling tool of biochemical networks. `http://www.celldesigner.org/`, Accessed in August 2009.

[8] Chisio - Compound or Hierarchical Graph Visualization Tool. `http://www.cs.bilkent.edu.tr/~ivis/chisio.html`, Accessed in August 2009.

[9] Cihan Küçükkeçeci. Chisio: A Visual Framework For Compound Graph Editing and Layout. Master's thesis, Department of Computer Science, Bilkent University, Ankara, Turkey, 2007.

[10] Cytoscape - Analyzing and Visualizing Network Data. `http://www.cytoscape.org/`, Accessed in August 2009.

[11] E. Demir, O. Babur, U. Dogrusoz, A. Gursoy, A. Ayaz, G. Gulesir, G. Nisanci, and R. Cetin-Atalay. An ontology for collaborative construction and analysis of cellular pathways. *Bioinformatics*, 20(3):349–356, 2004.

[12] R. Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.

[13] U. Dogrusoz, E. Z. Erson, E. Giral, E. Demir, O. Babur, A. Cetintas, and R. Colak. PATIKAweb: a Web interface for analyzing biological pathways through advanced querying and visualization. *Bioinformatics*, 22(3):374–375, 2006.

[14] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir. A layout algorithm for undirected compound graphs. *Information Sciences*, 179(7):980 – 994, 2009.

[15] Adobe Flex. `http://www.adobe.com/products/flex/`, Accessed in August 2009.

[16] GEF - Graphical Editing Framework. `http://www.eclipse.org/gef/`, Accessed in August 2009.

[17] Google Maps API. `http://code.google.com/apis/maps/`, Accessed in August 2009.

[18] Google Maps API Map Overlays. `http://code.google.com/apis/maps/documentation/overlays.html`, Accessed in August 2009.

[19] Google Maps API Reference. `http://code.google.com/apis/maps/documentation/reference.html`, Accessed in August 2009.

[20] Google Maps JavaScript API Example: Simple Polygon. `http://code.google.com/apis/maps/documentation/examples/polygon-simple.html`, Accessed in August 2009.

[21] i-Vis - Information Visualization Research Group of Computer Engineering Department at Bilkent University. `http://www.cs.bilkent.edu.tr/~ivis/`, Accessed in August 2009.

[22] KEGG Tools - Kyoto Encyclopedia of Genes and Genomes Tools. `http://www.genome.jp/kegg/download/kegtools.html`, Accessed in August 2009.

[23] N. Le Novere, S. Moodie, A. Sorokin, M. Hucka, F. Schreiber, E. Demir, H. Mi, Y. Matsuoka, K. Wegner, and H. Kitano. Systems biology graphical notation: Process diagram level 1. Available from Nature Precedings, 2008. `http://hdl.handle.net/10101/npre.2008.2320.1`, Accessed in August 2009.

[24] Apache Logging Services Project. `http://logging.apache.org/log4j/`, Accessed in August 2009.

[25] N. L. Novere, M. Hucka, H. Mi, S. Moodie, F. Schreiber, A. Sorokin, E. Demir, K. Wegner, M. I. Aladjem, S. M. Wimalaratne, F. T. Bergman, R. Gauges, P. Ghazal, H. Kawaji, L. Li, Y. Matsuoka, A. Villeger, S. E. Boyd, L. Calzone, M. Courtot, U. Dogrusoz, T. C. Freeman, A. Funahashi, S. Ghosh, A. Jouraku, S. Kim, F. Kolpakov, A. Luna, S. Sahle, E. Schmidt, S. Watterson, G. Wu, I. Goryanin, D. B. Kell, C. Sander, H. Sauro, J. L. Snoep, K. Kohn, and H. Kitano. The systems biology graphical notation. *Nat Biotech*, 27(8):735–741, August 2009.

[26] PathCase - Metabolic Pathways Database System. `http://nashua.case.edu/pathwaysweb/`, Accessed in August 2009.

[27] PATIKA - Pahtway Analysis Tools for Integration and Knowledge Acquisition. `http://www.patika.org/`, Accessed in August 2009.

[28] Paxtools - A library for accessing and manipulating BioPAX. `http://www.biopax.org/paxtools/`, Accessed in August 2009.

[29] Reactome - a curated knowledgebase of biological pathways. `http://reactome.org/`, Accessed in August 2009.

[30] P. Saraiya, C. North, and K. Duca. Visualizing biological pathways: requirements analysis, systems evaluation and research agenda. *Information Visualization*, 4(3):191–205, 2005.

[31] System Biology Graphical Notation A Visual Notation for Network Diagrams in Biology. `http://www.sbgn.org/`, Accessed in August 2009.

[32] SBML Systems Biology Markup Language. `http://www.biopax.org/`, Accessed in August 2009.

[33] SWT - The Standard Widget Toolkit. `http://www.eclipse.org/swt/`, Accessed in August 2009.

[34] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[35] VISIBIO*web* User's Guide. `http://www.bilkent.edu.tr/~bcbi/pvs/VISIBIOweb-1.0.UG.pdf`, Accessed in August 2009.

[36] VisANT - An integrative platform for network/pathway analysis. `http://visant.bu.edu/`, Accessed in August 2009.

Appendix A

Implementation Details Continued

A.1 Tooltip

Although Google Maps API provides various events and features for the customization of a map, tooltip support for `GPolygon` instances is not provided. However, tooltip is a useful feature, since the names of biological entities are typically quite long and they do not fit into the width of the nodes representing them. The width of a node is not extended to cover the name of the biological entity it represents. This is simply because non-uniform node dimensions affect layout in a negative way. Figure A.1 is an illustration of tooltip in VISIBIO*web*.

In order to implement tooltip support, a custom DOM event listener is added to the owner of the map container. Moreover, `mouseover()`, and `mouseout()` events of `GPolygon` are used. When the mouse moves over a polygon and stays in a 2x2 square of pixels range for at least one second, the tooltip of the graph object associated with that polygon is displayed. The tooltip is displayed inside an html *div* element inside VISIBIO*web* canvas. The position of the tooltip is determined via using the event object passed as parameter to custom DOM event listener.

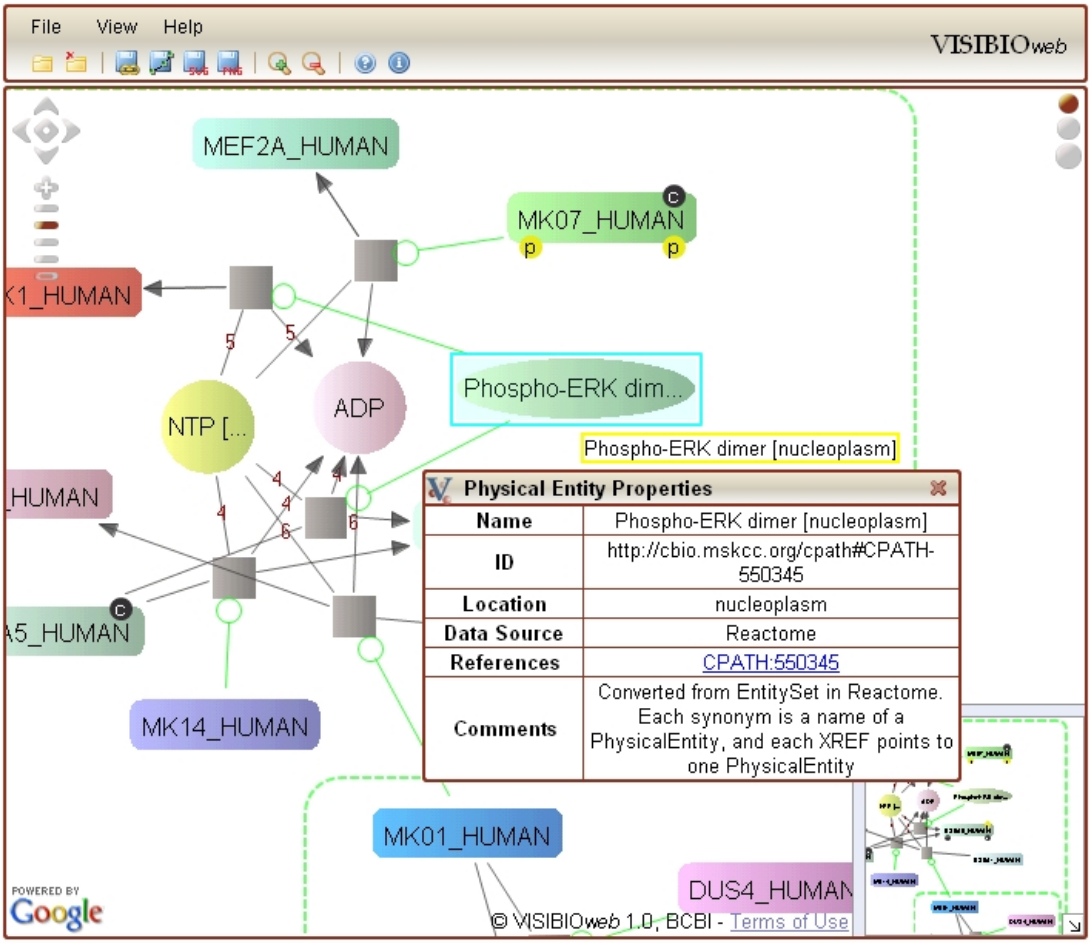

Figure A.1: Tooltip (inside yellow bordered box) for the selected graph object displays the full name of biological entity, which does not fit into the width of the node.

A.2 Exception Handling and Logging

Generation of view from biological model uploaded to the server is a complicated process, during which many exceptions and errors can occur. Proper handling of exceptions is crucial in terms of providing a reliable software. Different parts of code are applied try-catch blocks to catch various exceptions and display proper error messages. In VISIBIO*web*, a runtime exception class, named `VWException` is introduced. When an internal exception occurs, it is caught and an instance of `VWException` with a proper message is thrown instead. All uncaught exceptions including `VWException` instances are caught inside a JSP page, `exceptionHandler.jsp`, which is set as an error page. Necessary clean up is performed in this file and control is passed to `index.jsp` for displaying an error message. Figure A.2 shows an error message indicating a load problem on the server side.

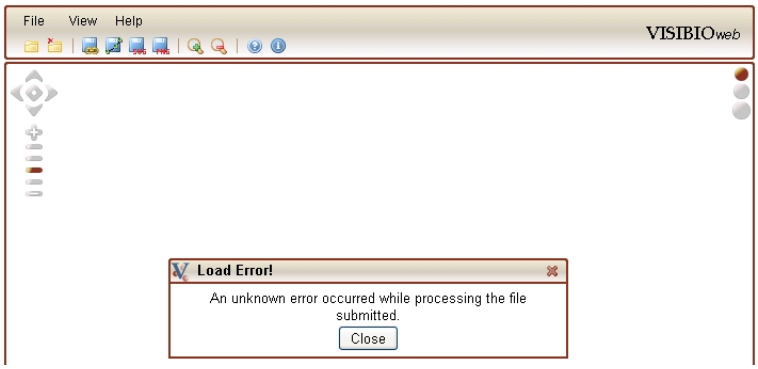

Figure A.2: An error message shown on the client side due to a problem on the server side.

It is not common to catch errors in Java, since they mostly occur as a result of abnormal conditions. However, errors of type `java.lang.OutOfMemoryError` are caught in VISIBIO*web*, if they occur while generating static SVG or PNG images. This is due to the reason that especially PNG generation requires large amounts of memory and a possible out-of-memory can result in an error message on the client side. Instead of displaying an error message, we prefer to display a warning message indicating SVG/PNG image might not be saved properly and provide the user with the view constructed for the submitted .owl file.

Logging server output is useful in terms of debugging and detection of exceptions explained above. Logging is performed via using Log4j library [24]. Logging is performed after critical parts of overall process including owl parsing, layout, tile generation, static image creation, and XML file generation. Exceptions are also logged to simplify determination of the problem on the server.

Appendix B

Sample Views from VISIBIO*web*

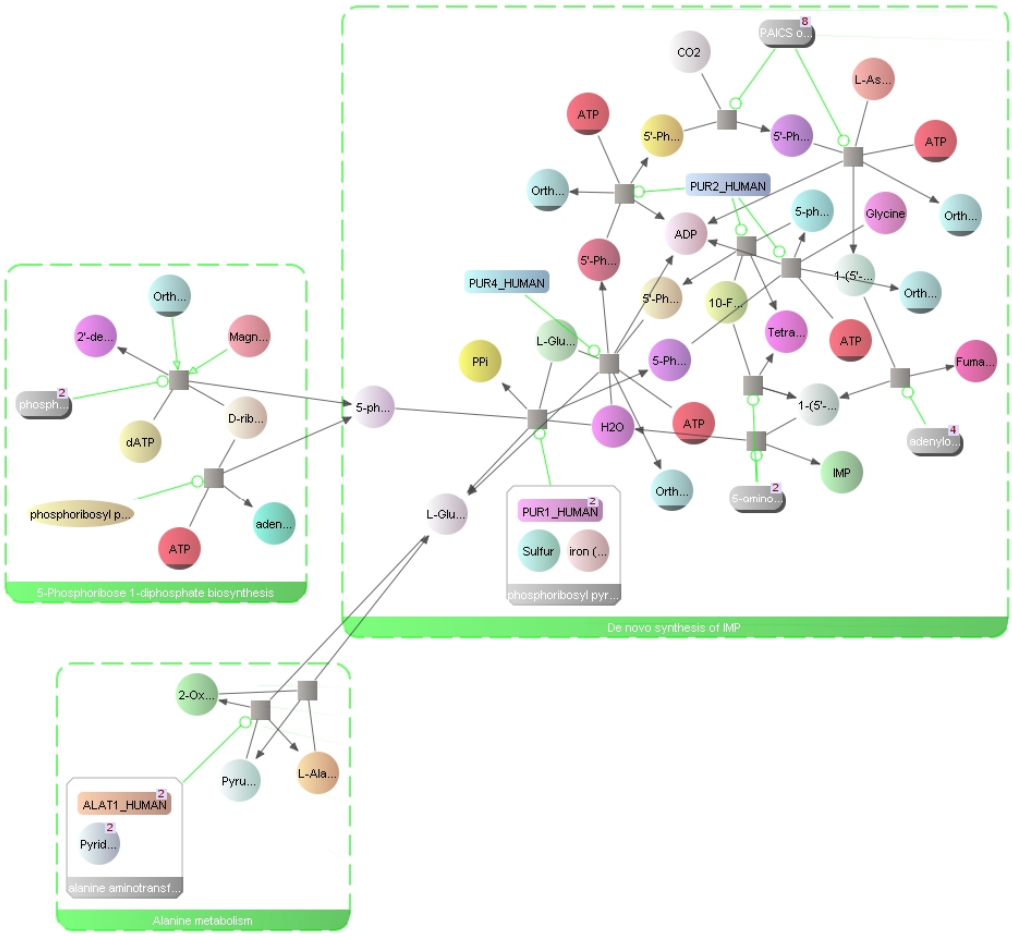

Figure B.1: A sample model containing 3 pathways.

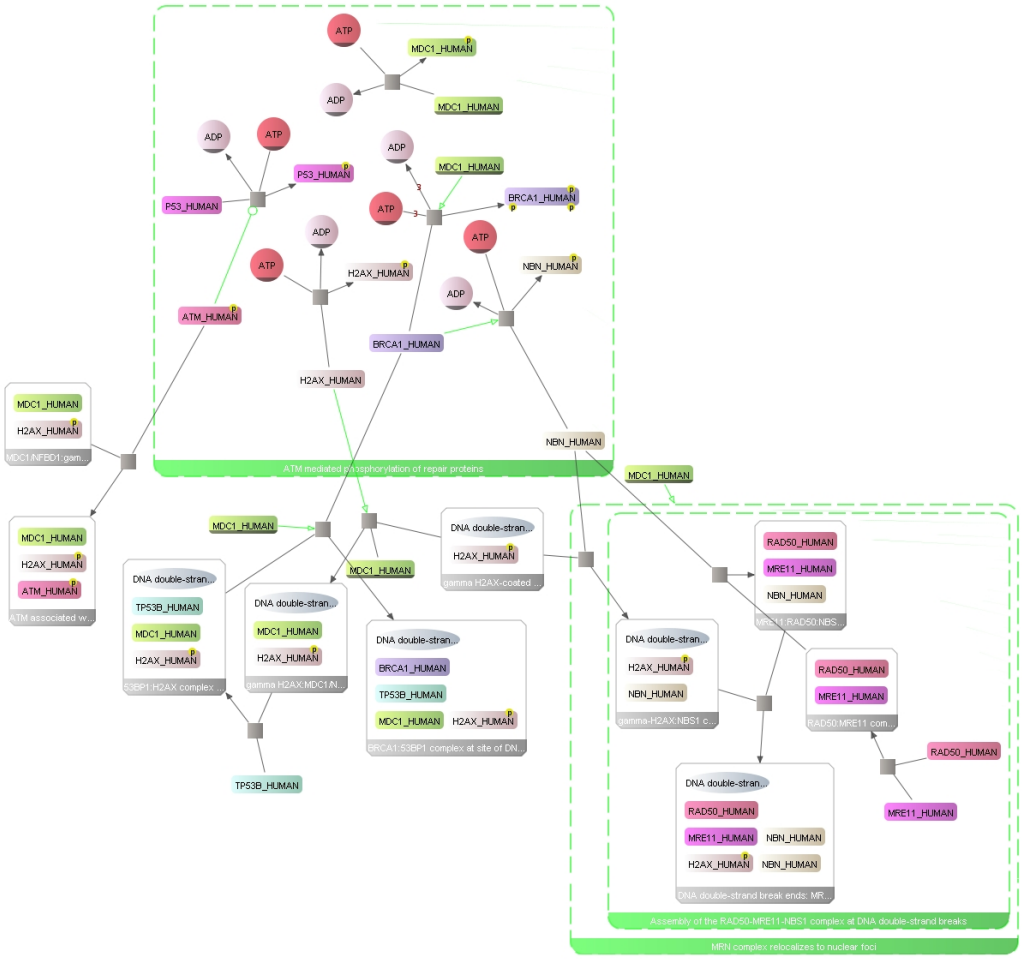

Figure B.2: A sample model containing 2 pathways and more interactions outside the pathways.

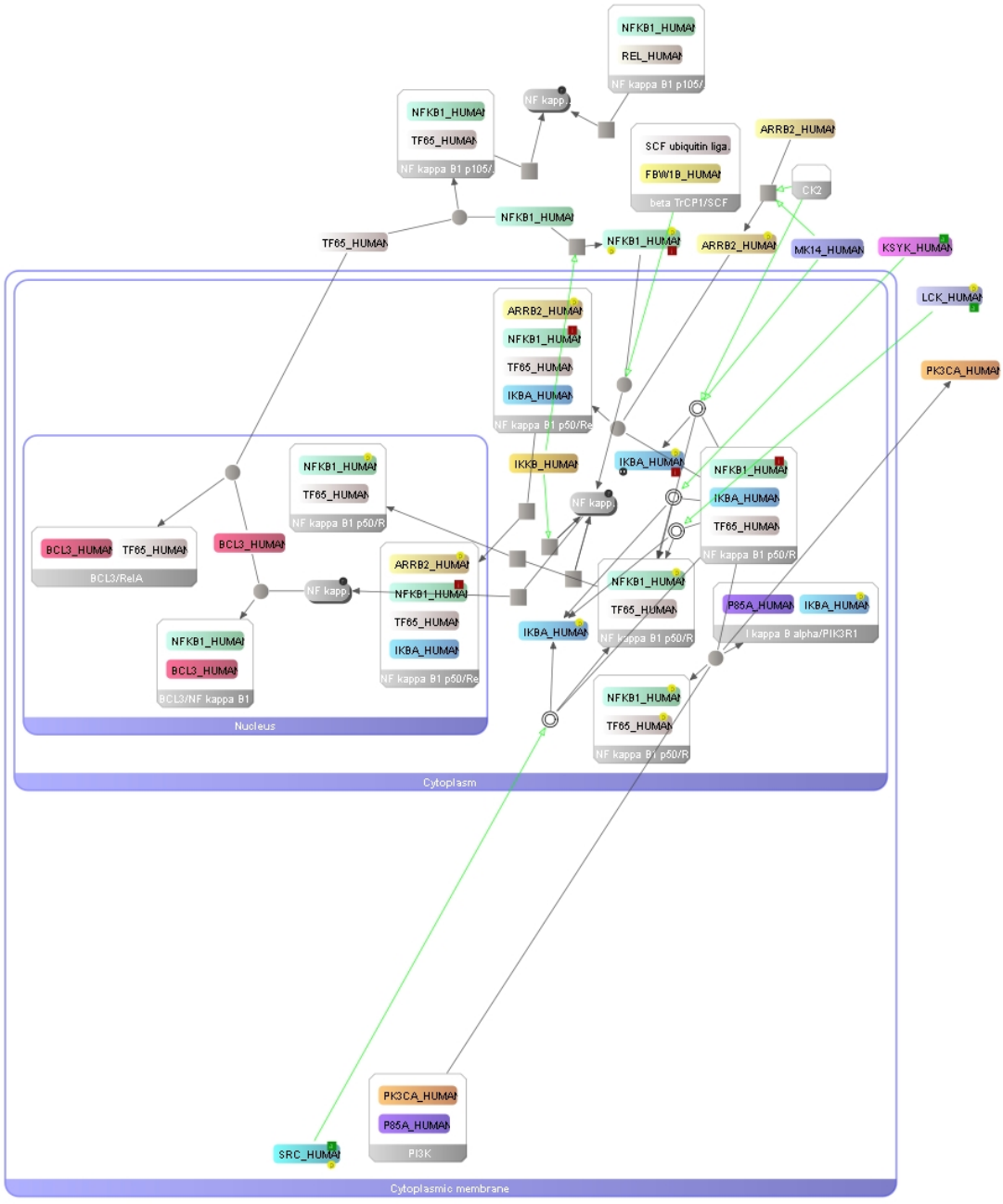

Figure B.3: A sample model displaying reactions inside 2 compartments and more interactions outside them.

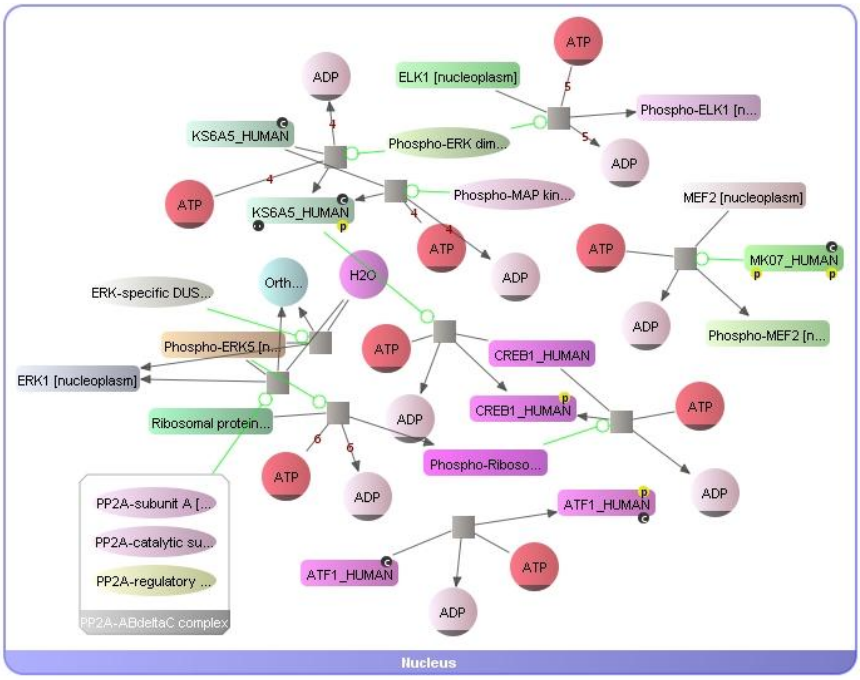

Figure B.4: A sample model displaying reactions inside nucleus.

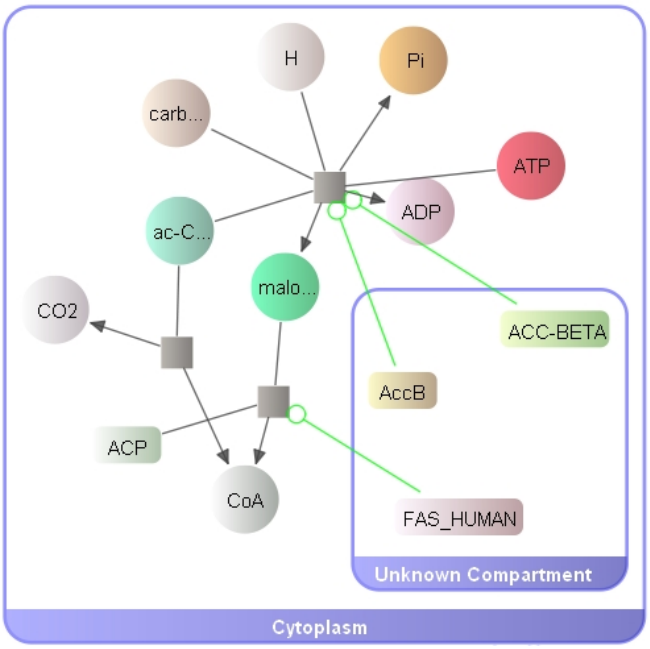

Figure B.5: A sample model displaying reactions inside cytoplasm and an unknown compartment.

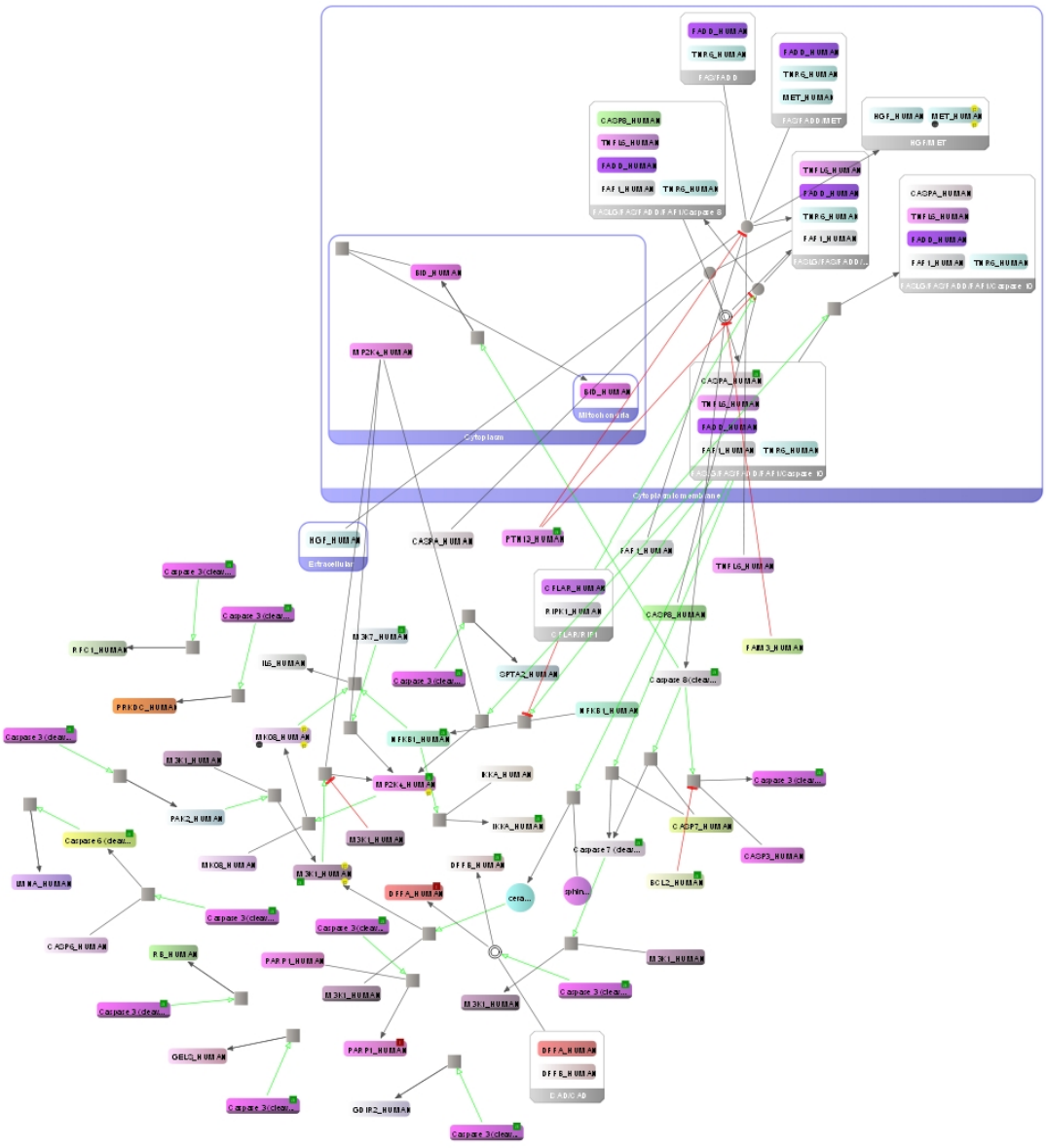

Figure B.6: A sample model displaying reactions inside 3 compartments and more interactions outside them.

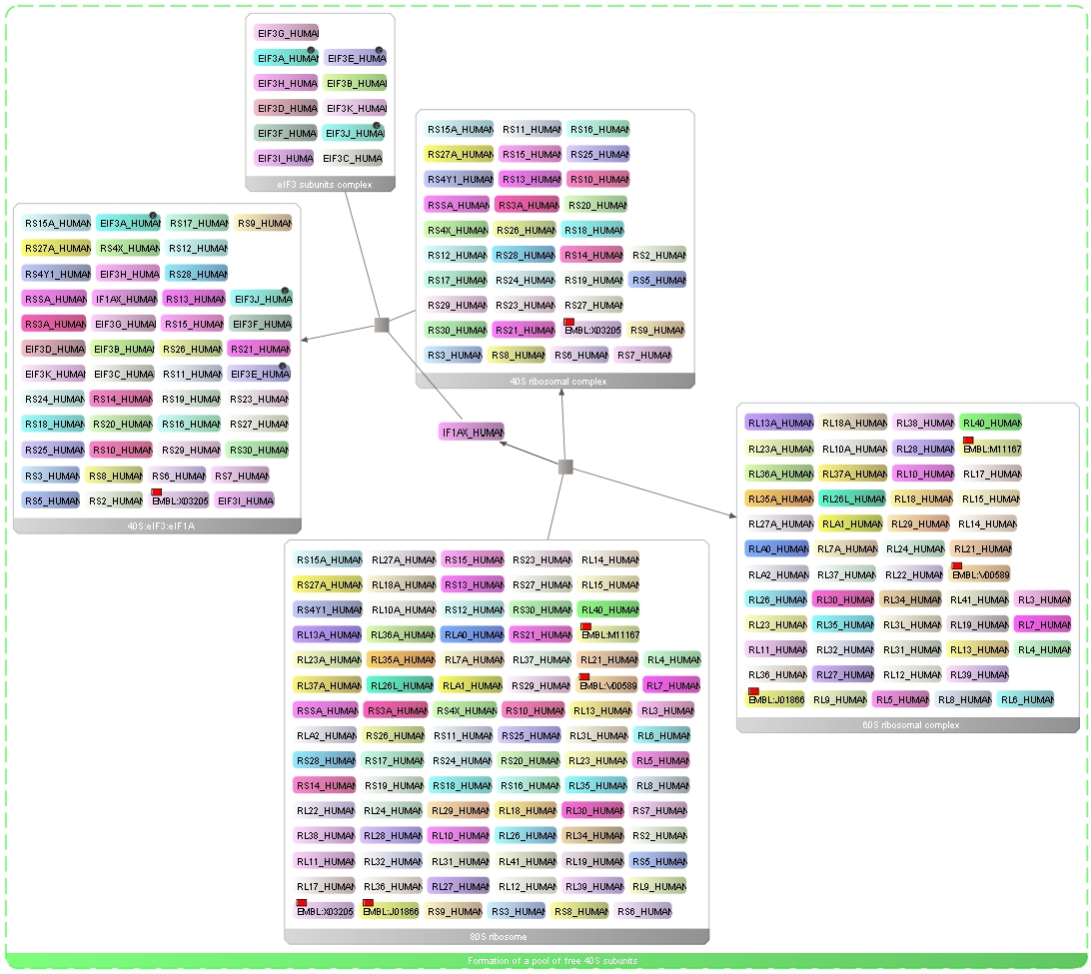

Figure B.7: A sample model displaying a pathway with 5 complexes each having many members.