PATIKA*web* : A WEB SERVICE FOR ACCESSING AND VISUALIZING PATHWAY DATA IN PATIKA DATABASE

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Emine Zeynep Erson

July, 2005

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. 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.

Prof. Dr. Özgür Ulusoy

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.

Assist. Prof. Dr. Uygar Tazebay

Approved for the Institute of Engineering and Science:

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

ABSTRACT

PATIKA*web* : A WEB SERVICE FOR ACCESSING AND VISUALIZING PATHWAY DATA IN PATIKA DATABASE

Emine Zeynep Erson
M.S. in Computer Engineering
Supervisor: Asst. Prof. Dr. Uğur Doğrusöz
July, 2005

After completion of Human Genome Project, there has been an exponential increase in the available biological data. Although there has been an enormous effort for creating ontologies, standards and tools, current bioinformatics infrastructure is far from coping with this data. The PATIKA Project aims to provide the community an integrated environment for modeling, analyzing and integrating cellular processes.

PATIKA project develops software tools providing access, visualization and analysis on the data in PATIKA database. In this thesis, we present analysis, design and implementation of PATIKA*web*, a Web-service having a user-friendly interface without requiring any registrations, installations. To achieve an enhanced data analysis , PATIKA*web* provides a multiple-view schema , compartments and compound graphs for visualizing molecular complexes, pathways and black-box reactions.

Querying component supports SQL-like queries and an array of graph-theoretic queries for finding feedback loops, common targets and regulators, or interesting subgraphs based on user's genes of interest. Constructed models can be saved in XML, exported to standard formats such as BioPAX, SBML or converted to static images. A highly interactive and user friendly querying interface is supported with PATIKA*web*.

Visual representation of complex information in pathway research is very important. The information should be presented with high coverage, while providing a user friendly interface. In this thesis we also present a new approach to visualize complex pathway information coping with the limitations introduced by ontology and graphical representation.

PATIKA*web* 's unique visualization and querying features fill an important gap in the pool of currently available tools and databases.

Keywords: Bioinformatics, pathway visualization, complex view management, Web service.

ÖZET

PATIKA*WEB* : PATIKA VERİTABANıNDAKİ YOLAK VERİSİNE ERİŞMEK VE GÖRÜNTÜLEMEK İÇİN AĞ HİZMETİ

Emine Zeynep Erson
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Assist. Prof. Dr. Uğur Doğrusöz
Temmuz, 2005

İnsan Genome Proje'sinin tamamlanmasının ardından, biyolojik sistem verilerinde üstsel bir artış oldu. Fazla sayıdaki ontoloji, standart ve araç yaratabilme çabalarına rağmen, mevcut biyo-enformatik yapısı bu verilerin karmaşıklığı ile mücadele etmekten çok uzaktır. PATIKA Projesi topluluğa hücresel işlemleri modellemek, çözümlemek ve birleştirmek için tümleşik bir ortam sağlamayı amaçlar.

PATIKA projesi PATIKA veritabanındaki veriye ulaşılmasını, çözümlenmesini ve görüntülenmesini sağlayan yazılım araçları sağlamaktadır. Bu tezde, PATIKA*web* isimli ağ tabanlı, kullanışlı, kayıt, yükleme gerektirmeyen yazılım aracının tasarım ve uygulanmasını sunmaktayız. Gelişmiş veri çözümleme olanağını sağlamak için, PATIKA*web* iki seviyede çoklu görüntü olanağı, moleküler komplekslerin, yolakların ve kara kutu reaksiyonların görüntülenmesi için bileşik çizge sağlamakta ve kompartmanları desteklemektedir.

Sorgulama bileşeni, geribildirimli döngüleri, ortak hedef ve düzenleyici veya kullanıcının ilgili genine dayalı ilgili alt çizgeleri bulmak için hem SQL benzeri sorguları hem de çizge kuramsal sorgu dizilerini destekler. Oluşturulan modeller XML formatında saklanabilir; BioPAX, SBML gibi standart biçimlere aktarılabilir veya sabit görüntülere çevirilebilir. PATIKA*web* fazla etkileşimli ve kullanışlı sorgulama arayüzünü destekler.

Yolak araştırmalarında, karmaşık bilginin görsel olarak gösterimi kritik bir rol oynamaktadır. Kullanışlı bir arayüz oluşturmaya çalışırken bilgi doğru olarak, mümkün olan en geniş kapsamla gösterilmelidir. Bu tezde, aynı zamanda karmaşık yolak bilgisini görselleştirmede ontoloji ve çizgesel gösterimden kaynaklanan sınırlamalarla mücadele eden yeni bir yaklaşım sunmaktayız.

PATIKA*web*'in benzersiz görselleştirme ve sorgulama olanakları ile, şu anda

mevcut araçlar ve veritabanları havuzunda önemli bir boşluk doldurmaktadır.

Anahtar sözcükler: Biyo-enformatik, yolak görsellenmesi, karmaşık görüntü idaresi, ağ hizmeti.

Acknowledgement

I would like to express my deepest gratitudes to my supervisor Assist. Prof. Uğur Doğrusöz, for his guidance and feedbacks during the preparation of this thesis. It has been a great experience and privilege for me to work with him and get benefit from his valuable mentorship.

I also would like to thank Prof. Özgür Ulusoy and Assist. Prof. Uygar Tazebay for reviewing the manuscript of this thesis and spending their valuable time.

During these two years I had the chance to work with a perfect team, PATIKA team. Friendships and supports of Asli Ayaz, Özgün Babur, Ahmet Çetintaş, Emek Demir and Erhan Giral were very valuable for me. They are, and will always be, more than friends for me.

Above all, I am very grateful for the endless love and support of my parents Leyla and Metin Erson, and my dearest sister Elif Erson. I feel stronger and happier with their love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Bioinformatics, which is a fast evolving field of modern science is defined as the computational analysis and processing of biological information. Roots of bioinformatics date back to 1950's, when the DNA structure was discovered by Watson and Crick and the encoding of genetic information for proteins was studied by Gamow [12]. Since then, high-throughput biological experiments caused accumulation of information and huge amount of data. Moreover in molecular biology, the discoveries are expressed in natural language rather than mathematical models as in physics. Defining qualitative and quantitative functionalities in molecular level presented computational challenges and that's how importance of bioinformatics is perceived [3]. Especially after 1990, when different types of high-throughput data became available, analysis of this data and higher order functionalities in cellular processes increased [7]. In parallel to these improvements, data access and analysis problems in molecular biology started to enter the scope of computer science approaches.

An era started in molecular biology, therefore in bioinformatics, when the Genome Project began in 1996. By 2003, 160 genomes were completely sequenced, and a lot more sequencing projects were in progress. The available data for DNA/protein sequences increased exponentially following these studies. The produced data were stored in databases like GenBank, EMBL (European Molecular Biology Laboratory nucleotide sequence database), DDBJ (DNA Data Bank

of Japan), PIR (Protein Information Resource) and SWISS-PROT [7]. Although the sequence information is almost completely discovered, functional attributes of genes are not fully understood. In other words, although we have the ingredients, we still do not have the recipe of how an organism functions. Therefore, the next logical step in Human Genome Project is discovering what these sequences mean in terms of their functions and possible interactions [17]. Due to the complexity of the problem, research on perceiving this information requires a different modeling approach. Bioinformatics, presenting computer science approaches provides the solution for this problem. Modeling the available data and its representation is one of the essential purposes of bioinformatics. Presenting available information with heterogeneous meta-data, especially for the functional genomics, is the solution produced by bioinformatics research [17]. A level of abstraction is required to manage and perceive the data. Therefore pathways are used, as the abstraction of molecular and cellular functional events, such as metabolic pathways and signaling transduction. Next step is to analyze this level of information for more complicated interactions and pathways. Analyzing pathways, brought new problems into consideration such as accessing and visualization of this data in an effective way. We attack this problem in this thesis, with a new software tool named PATIKA*web*, a promising solution to some of these issues.

1.1 Motivation

Accessing, analyzing and editing available information for metabolic/signaling pathway data became an important research in molecular biology. As new bioinformatics tools are developed, new requirements are emerged. The PATIKA project has been developing integrated visual environments for collaborative construction and analysis of various cellular pathways. These tools provided very extensive functionalities for data access, analysis and submission/integration. However even for the editing or the creation of a new pathway, the research process requires to access the current available data and analyze this data. Therefore easy access and enhanced analysis techniques like graphical visualization becomes a key step in pathway informatics. When we analyzed the use cases of PATIKA, we

have observed that most frequent use cases are the data access and analysis as opposed to data submission.

Another motivation of this research thesis is related to the preferences of the users about the convenience of the tools. For accessing and analyzing a specific gene or protein and related pathways, researchers do not want to spend time on registering for, downloading and setting up software. With these observations in mind, we have developed an easy-to-access tool PATIKA*web*, which is a Web based service. Main motivation for this tool is to provide a service to reach the data in PATIKA database via an extensive querying interface and visualize this data.

1.2 Background Information

A pathway is a network of interacting proteins to perform a specific metabolic or signaling task in living systems (See Figure 1.1). Signal transduction is the process converting one kind of signal into another by chemical modifications. Series of biochemical reactions produce metabolic byproducts, end-products or become part of other pathways themselves. Current knowledge in metabolic pathways is deeper than the signaling pathways [1].

Since the completion of the Human Genome Project, scientists have been generating huge amounts of data on cellular pathways. Therefore, many different databases started to host such data to simplify such complicated information. Although such databases contain different types of data, there is significant data overlap problem in these sources. KEGG[1], Enzymes and Metabolic Pathways database (EMP)[2], EcoCyc[3], SWISS-PROT[4], Gene Ontology (GO)[5], ExPASY[6]

[1]http://www.genome.jp/kegg/
[2]http://www.empproject.com/
[3]http://ecocyc.org/
[4]http://www.expasy.org/sprot/
[5]http://www.geneontology.org/
[6]http://www.expasy.org/

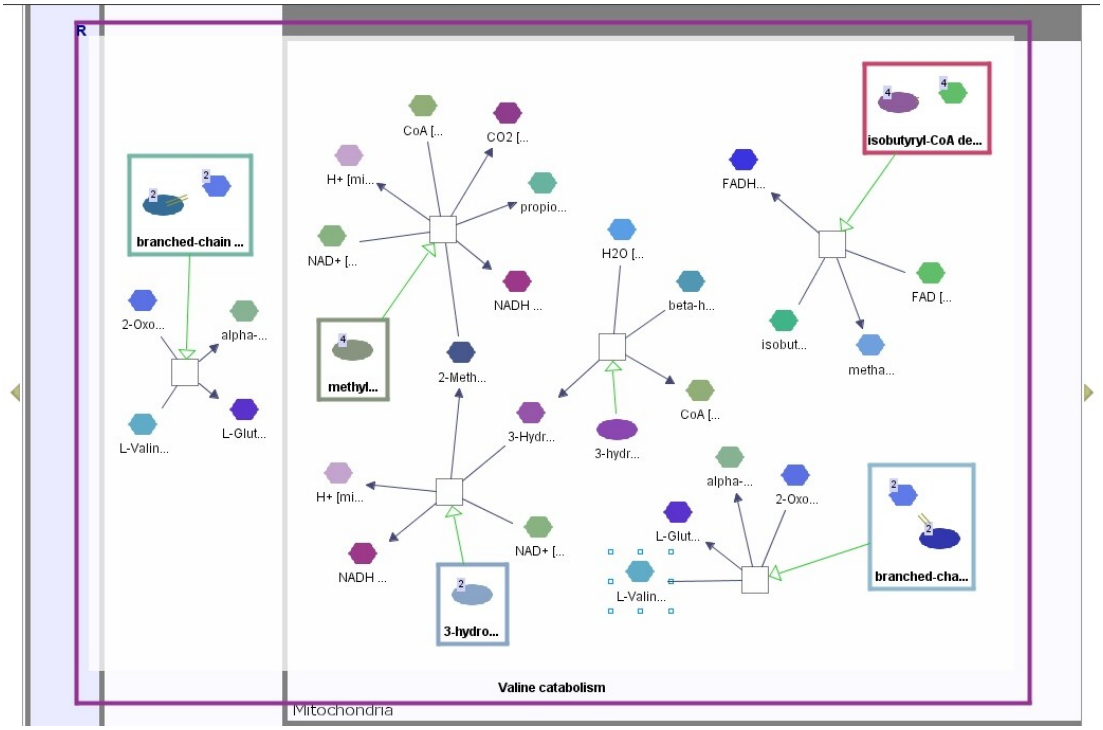

Figure 1.1: Sample pathway: Valine Catabolism, from PATIKA*web*

are only a few of examples. Some of these databases contain only metabolic pathway information like EcoCyc and EMP. Some others contain both metabolic and regulatory pathway data like KEGG. Researchers need to access the integrated information from these databases, while conducting their research. Accessing such data and integrating them will not be the feasible solution for most of the scientist. Therefore we developed a knowledge base, where we perform the data integration process and provide the integrated data from different databases. We used many public databases while integrating data for PATIKA database, including Entrez Gene, UniProt, PubChem, GO, Reactome and KEGG.

As sources available to reach data increased, the requirement for the portability of the data became an important issue. A standard way should be developed to exchange available data and newly produced data. BioPAX[7] and SBML (The Systems Biology Markup Language)[8] are such efforts. We use these standards in our software tools, to provide data portability.

[7]http://www.biopax.org/
[8]http://sbml.org/index.psp

Chapter 2

Related Work

As biological pathway data started to accumulate, software tools to access and visualize this data became indispensable. Therefore both commercial and academic research groups aimed to satisfy this requirement.

Initially the developed tools only presented still pictures of known pathways. *BioCarta*[1], which is a commercial company founded in 2000, developed this kind of a tool, where they provided still images of specific pathways. As new pathways are introduced into the knowledgebase of *BioCarta*, corresponding images have to be created manually. The necessary description of the pathway is then given by a summary text. *ExPASy*[2] is a similar tool providing a simple interface for textual queries to match any of the available biochemical data. Data is presented with map like still pictures. *KEGG* (Kyoto Encyclopedia of Genes and Genomics) provides hand-drawn still images[3]. *MetaCyc*[4] provides access to metabolic pathways for over than 300 organism. Similarly *EcoCyc*[5] is a database providing genome, metabolic pathway information for the bacterium Escherichia coli. Both *MetaCyc* and *EcoCyc* provide Web based tools for querying, editing and visualization for the data in their database [9].

[1] http://www.biocarta.com/
[2] http://www.expasy.org/cgi-bin/search-biochem-index
[3] http://www.genome.jp/kegg/
[4] http://metacyc.org/
[5] http://ecocyc.org/

One of the major drawbacks of all these tools is that it is required to create the pathway drawings as new data is introduced. That means that the integration of data and its representation in images require a significant amount of extra effort. Moreover still images cannot be modified, therefore they are only used for knowledge acquisition of one type. Research and development in this area started to develop new tools to provide dynamic visualization of pathways and to be able to integrate different databases at the same time.

Cytoscape[6] is a free, open source tool for general purpose modeling environment. The strongest point of *Cytoscape* compared to other tools is its plug-in adaptable architecture. Especially for data analysis, they provide different plugins like microarray data analysis. Integration of data, mainly annotations to the available graph is strongly supported. However, as it is mainly designed as a modeling tool, knowledge acquisition through *Cytoscape* is possible through plugins to load some data. They introduce new plugins as new types of data to load becomes available. For example, latest plugin release supports downloading protein-protein interactions. Another way to analyze the current data in *Cytoscape* is to save the data in the acceptable format and to reload it. Different layout algorithms are implemented to enhance the visualization quality. *Cytoscape Core* is implemented in Java with LGPL Open Source license. Graphical component of *Cytscape* uses yFiles Graph Library (Java Graph Layout and Visualization Library)[7]. One potential downfall for users is that *Cytoscape* requires a download and installation to run the software [14].

Another software tool for pathway visualization is *VisANT*[8], which is an open-source, online tool for access and visualization of bimolecular interactions. Currently VisANT obtains pathway data from KEGG database and draws information from Predictome database. Annotation and cross references are obtained by GenBank and SwissProt. The system provides navigation of data, manipulation and expansion of visualized pathways by basic graph operations like degree distribution, loop detection and shortest path identification [5].

[6] http://www.cytoscape.org/
[7] http://www.yworks.com
[8] http://visant.bu.edu/

Users need to point their browsers to the provided Web page and, choose to run the VisANT applet, run the VisANT Java Web-Start Application or download and install the stand-alone application. VisANT provides textual querying facility, where the user can enter the ORF IDs, GI numbers, or KEGG pathway IDs for an arbitrary number of genes. Saving the pathway locally or by protecting password to reach later remotely is possible in a simple delimiter based format. However not all the functionalities are available for the downloadable applet form. Registration is required for save and load operations. The system provides online structure by using J2EETM technology. They require a Web browser and Java Runtime Engine (JRE).

Another academic group developed a similar tool for pathway visualization. Center for Computational Genomics at the Case Western Reserve University, developed a system called *Pathways Database System* [10]. This system is an integrated system, composed of different software components. This system's database component provides pathway data extraction from different databases such as SwissProt and GenBank. They also provide a querying component, to access the data in their database. Their third main component is the visualization component. To achieve the visualization of the pathway data, they use their own graph editing library, *PathwayViz* in Java [13]. In their visualization scheme, they allow multiple level of abstractions to visualize data. To get full functionality for the system, users need to download different components, like browser and the viewer independently. Currently available components are metabolic/signaling pathway browsers, pathway viewers, pathway explorer, pathway editor, Java based viewer and an XML based Web service to make queries. Java based viewer component loads an applet from the Web site, and provides a tree of pathways, processes and molecular entities to be visualized. There is a limited number of entities provided to be visualized within this component. Limited editing operations are provided , like move, expand/collapse and find node/edge.

Observing the positive and negative points of these tools, Bilkent University Center for Bioinformatics started a project namely, PATIKA, which produce

software tools containing workpackages like data integration, analysis and visualization.

2.1 PATIKA Project

The Patika project aims to cope with the complex information in cellular processes and provide an infrastructure for this information. Having this road map, Patika project produces software tools with its own ontology, mapping to the data in its knowledge base visualizable, editable and analyzable in an editor [2]. Ontology is the formal specification of a concept, built to be portable among applications with different domains [8]. Patika team developed its ontology enabling integration of incomplete, complex and fragmented pathway information [1]. Throughout this thesis we will refer to the components of this ontology, therefore we will briefly cover these components.

- States and Bioentities: Macromolecules, small molecules are the actors of molecular level reactions. However these molecules have different states in the cell based on the localization or chemical modifications. Therefore we define these molecules as *bioentities* and define their *states* based on the variable conditions.
- Transitions: It is modeling for a functional process. They provide as avoiding hyper edges in graphical representation of a pathway.
- Compartments: Physical localizations of proteins play a significant role in pathway analysis as a potential implication of function. Therefore graphically cell compartments must be modeled as well.
- Molecular Complexes: Molecules performing in structural or functional groups are defined as complexes.

- Abstractions: Incomplete and complex information contributing to the network of higher level information must be represented. Abstracting a pathway information as a single processes or grouping similar process are required for complexity management. Abstractions are basically composed of states, transitions and possibly other abstractions. Visual representation of abstractions with the limitations introduced by the compartments and graphical invariants are also addressed in this thesis. We have five types of abstractions defined in PATIKA ontology as *Homology State, Homology Transition, Incomplete State, Incomplete Transition and Regular Abstraction.*

Considering the visualization of the pathway data specified with PATIKA ontology, we designed different levels of visualization. Analysis and representation of different levels of data requires separate handling. Therefore we defined two views as *Mechanistic Level View* and *Bioentity Level View.* Relations among bioentities, such as protein-protein interactions, are represented in Bioentity view, whereas data related to metabolic or signaling pathways are visualized in Mechanistic view. Visual representation of the ontological components can be seen in Figure 2.1 in both Bioentity and Mechanistic levels.

Figure 2.1 is a screenshot from PATIKA*pro* showing all ontological components.

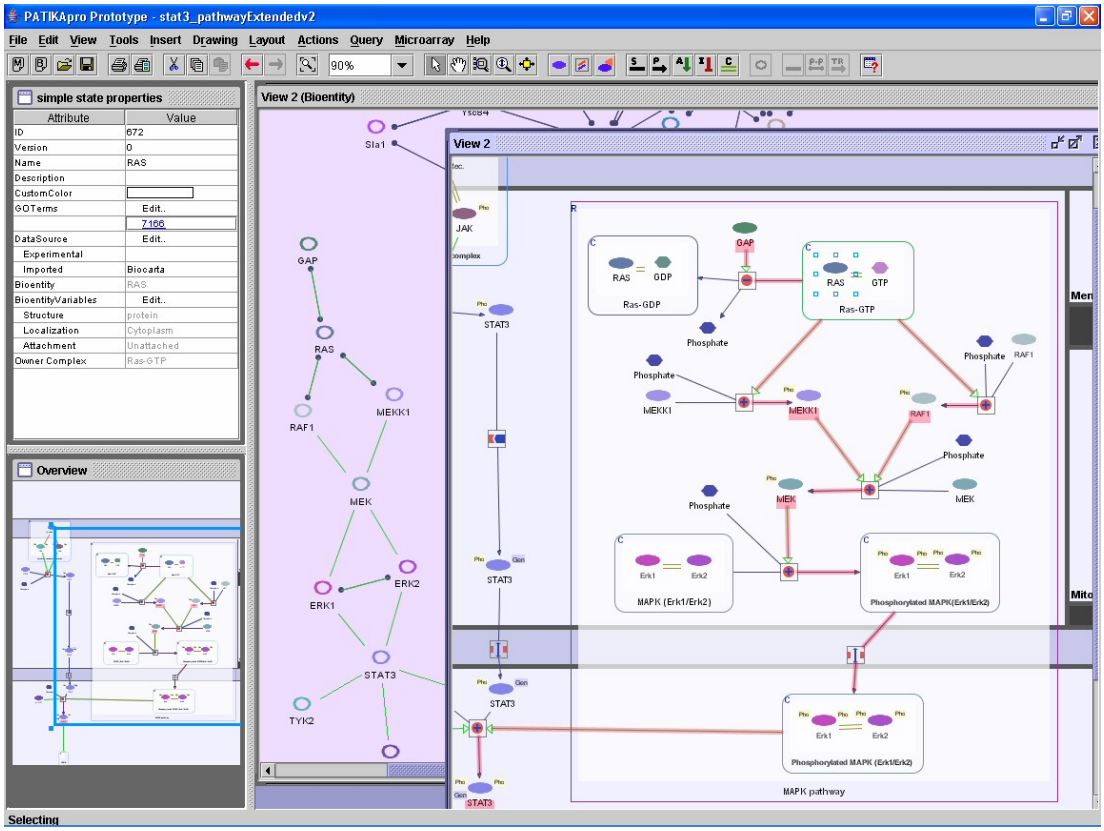

Figure 2.1: Screenshot from PATIKA*pro*

Chapter 3

Software System

3.1 Problem Statement

As we have stated previously in Section 1, data available in cellular processes at molecular level is accumulating very fast; and thus in parallel, researchers would like to reach this information rapidly. Presentation of the information is at least as important as the fast and rapid access to the data. Visual representation of the pathway information is preferred over the textual representation of the data by the researchers, due to the nature of the information.

We reconsidered these facts with the current user profile of Patika and other pathway visualization tools. Most of the researchers need fast and easy access to the metabolic/signaling pathway information to accelerate their research. Downloading, installing, executing a software tool, just to perform read only operations on a pathway database is not desirable in terms of time and effort for most of the researchers. We also foresee that, majority of the users will perform read-only access to our database, rather than write operations such as data submission. Therefore, we have decided to provide a Web service reachable through a Web browser.

In Patika the problem of pathway visualization and tools having extensive

visualization facilities had been attacked previously. For visualization component in PATIKA*pro* we have used *Tom Sawyer Visualization Java Edition* and customized it for extensive pathway visualization [15].

We need to build a framework where we can reuse the previously built visualization component of PATIKA*pro* and integrate it with the new Web based service, PATIKA*web*. Therefore our problem comes down to defining the requirements of PATIKA*web* based on the so far defined use case, design the system so that we can integrate the currently available approach to pathway visualization and define the technology to apply these architectures. We will address these issues in the next chapter.

Given the problem stated in Chapter 3.1, we have decided to build a Web-based service providing a read-only access to the PATIKA database, an interface to visualize this data and limited editing operations on the visualized data. However when we considered the use cases, we concluded that there is a considerable amount of memory and CPU requirement even for read only operations to be serviced by PATIKA*web*. Carrying the load of computationally heavy to the client side was not desired, since we do not want to put any necessary requirements on use of PATIKA*web* like CPU power or memory. The other alternative was to carry the logic that requires computation sensitive operations to the server side. This idea led us to the *thin client* concept.

A thin client is defined as a front-end having minimal software requirements and performing minimal computational operations. Internet and intranet are fundamentally based on the thin client paradigm [6] .All resource-requiring operations are performed at the server side, and results of the computations are transferred back to the client. Properly partitioning the resource-requiring processes between clients and servers is an effective way to distribute the computing resources. Based on the domain of the application the resource requirements change, but basic ones to consider are CPU cycles, memory, security, virtual memory, and high-speed data storage, etc. [4]. Another major advantage of thin clients is the concern for the developers and maintenance. Ability to have both scalability and centralized administration with thin client architectures is a

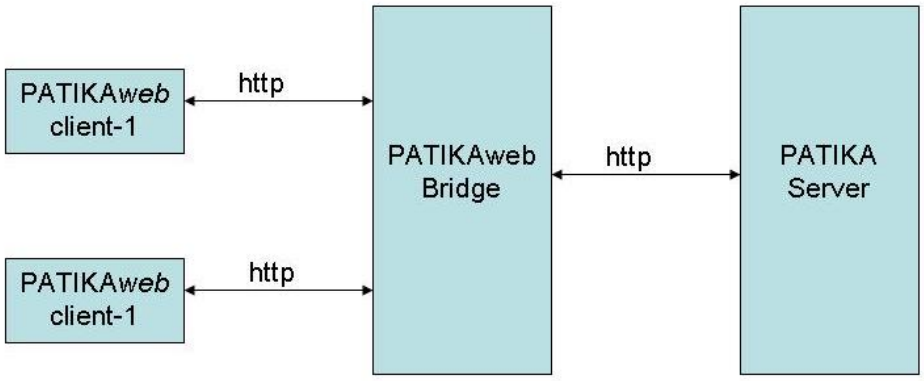

Figure 3.1: Modular architecture of PATIKA*web*

considerable benefit for fast changing resources and requirements [4].

Although thin-client architecture is advantageous for many perspectives, there are some drawbacks to consider. Client/Server architecture, which use the thin client approach rely on the bandwidth and network latency. It is experimentally shown that network latency becomes the bottleneck in such applications [11]. On the other hand, it is clear that network latency problem cannot be avoided for most of the *fat* clients as well. Improvements in the Web technology also helped to overcome these problems. Web components, plugins, like applets, JavaBeans, dynamic HTML pages are the technologies used extensively.

Web services are defined as the services available through Internet, accessible by standard Web protocols like http, using messaging data formats like XML[1]. Web services have a self-containing and modular structure to provide any application logic reachable through Internet[2]. Therefore we initially decided to define the modular structure of our Web service, PATIKA*web* and divide design steps based on the application logic.

In Figure 3.1, we present the multi-tier architecture for our Web-service. In this architecture, user, application and data components are tiered apart.

[1] www.w3.org/2003/glossary/subglossary/xkms2-req/
[2] http://www.ecots.org/

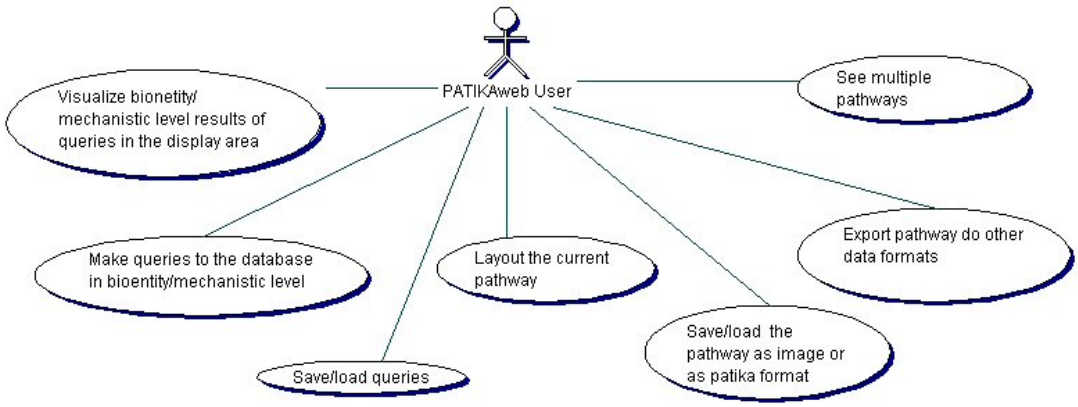

Figure 3.2: General use cases of PATIKA*web*

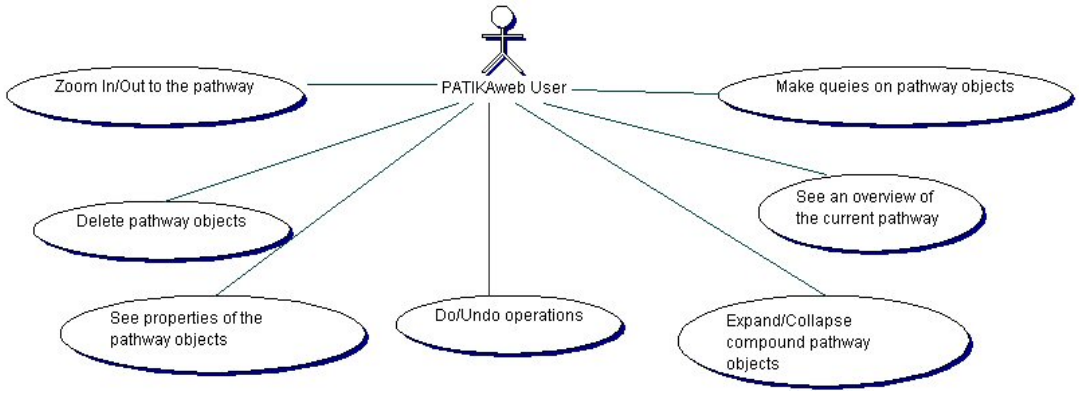

Figure 3.3: Editing use cases of PATIKA*web*

We have defined the functionalities of the components of the tiered architecture based on the requirements. For the requirements analysis phase, initially we have defined the use cases of PATIKA*web* in Figures 3.2 and 3.3.

Analyzing the use cases presented in Figure 3.2 and Figure 3.3, we identified the basic functionalities that we must provide as follows:

1. Pathway data model access by queries,
2. Visualization and editing of the current pathway model,

In the multi-tiered architecture, these functionalities are decomposed and designed in different components and mentioned in the following sections.

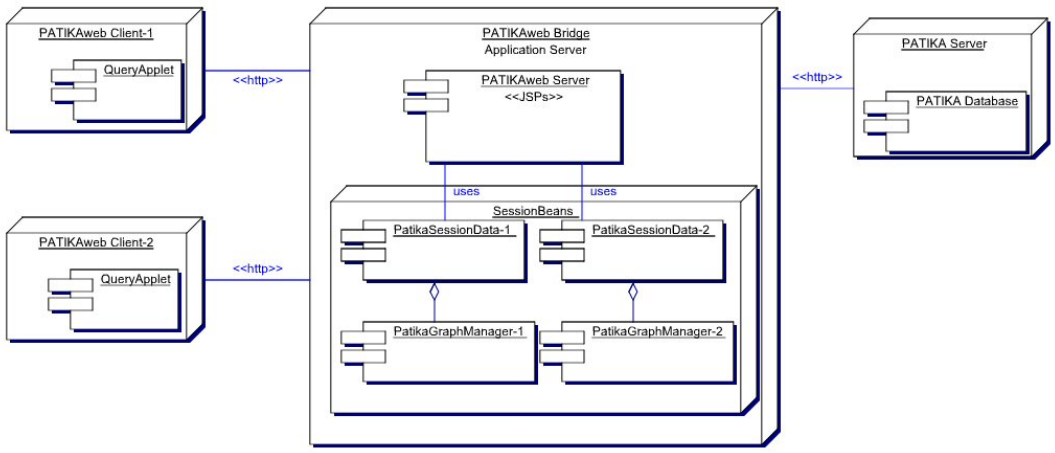

Figure 3.4: Deployment diagram of PATIKA*web*

We have decomposed our system according to the multi-tiered architecture and considered the following components in the design process as seen in Figure 3.1:

- PATIKA*web* Client Side Design: This component maps to the user tier in our architecture. In the client side, we focused on the user interface design of the Web page. Communication of the client to PATIKA*web* bridge over http is considered as well.

- PATIKA*web* Bridge Design: This component maps to the application tier in our architecture. In this part we designed the application logic. As we have defined in Section 3.1, we aim to reuse the code base that we have developed for previous pathway visualization tools, customized and integrated it with the new design developed for PATIKA*web*.

- PATIKA Server, Data Flow and Communication: This component maps to the data tier in our architecture. This part also covers the communication with PATIKA server.

Deployment of tiered architecture of PATIKA*web* , is detailed in Figure 3.4. In this figure we see that PATIKA*web* client communicates with PATIKA*web* bridge over http. We use the *Java Server Pages* (JSP) technology to handle the dynamic

web content. In the bridge, which behaves as the application server in the multi-tier architecture, we host the JSP's (Java Server Pages, see Section 3.5) and implement the application logic. Each client's session, in other words each client's application data is kept independently. The association of the user and the session information is managed by *JavaBean*s. Details of the components and their communication are given in the following sections.

3.2 PATIKA*web* Client

3.2.1 Analysis of Query Facility

The only way to analyze data in PATIKA*web* is by reaching data in PATIKA database through submission of queries on this data. As we have mentioned in Section 2.1, we support data in two levels, as mechanistic and bioentity levels. Therefore to facilitate the access of data in both levels, PATIKA*web* provides mechanistic and bioentity levels in both querying and visualization interfaces.

The queries to be submitted to PATIKA database should be as extensive and powerful as possible, to get full advantage of the integrated knowledgebase. In addition to the basic textual queries, we should also provide graph theoretical ones. Nesting of these queries is also critical, since the user may have a very specific interest in the pathway data. Having created a complex, recursive query, a user might like to save the query and load it later to continue working on it. Considering all these requirements, we realized that we needed a very extensive interface, where sophisticated user interaction is provided.

3.2.2 Analysis of User Interface

Considering the use cases mentioned in the Figures 3.2 and 3.3 and requirements pointed in Section 3.2.1, Section 3.3.1 and Section 3.3.1.1, we developed a service, where the users are able to reach an interface where they can query, visualize and

edit pathway data with a few mouse clicks. User interface of PATIKA*web* has the following components to provide these functionalities:

1. A drawing area for displaying pathway graphs: Results of the queries are displayed on this area. Only one pathway can be drawn at a time. We will refer to this area as canvas, from now on.
2. An inspection area: This is where the selected object properties are displayed
3. A query dialog: This is where the queries are created and submitted. The results are displayed either as a new drawing or merged into the existing one.
4. Menus and toolbars: To provide the limited editing functions on the pathway data, functional menus and toolbar components are provided.

To supply the requirements defined for the interface, we developed an interface as in Figure 3.5. As you see in this figure, we provide a canvas where we display the results of queries. This canvas displays both bioentity and mechanistic level graphs. We provide an overview window for the graph displayed in the canvas. Inspection of the pathway objects is facilitated with the inspector window. Considering the limitation of a Web browser by space, we planned to reserve the largest room for the the canvas (in the middle) , and located the overview window and inspector window to the remaining places as seen in Figure 3.5.

3.2.3 Detailed Design of PATIKA*web* Client

PATIKA*web* clients access the service supplied via their Web browsers. In these browsers, we provide highly dynamic visualization facilities over the displayed pathway information. Therefore we designed a mechanism where we can manage this dynamism both in the client side and at the server side. However we kept in mind that, our aim is to build a thin client, where most of the computation

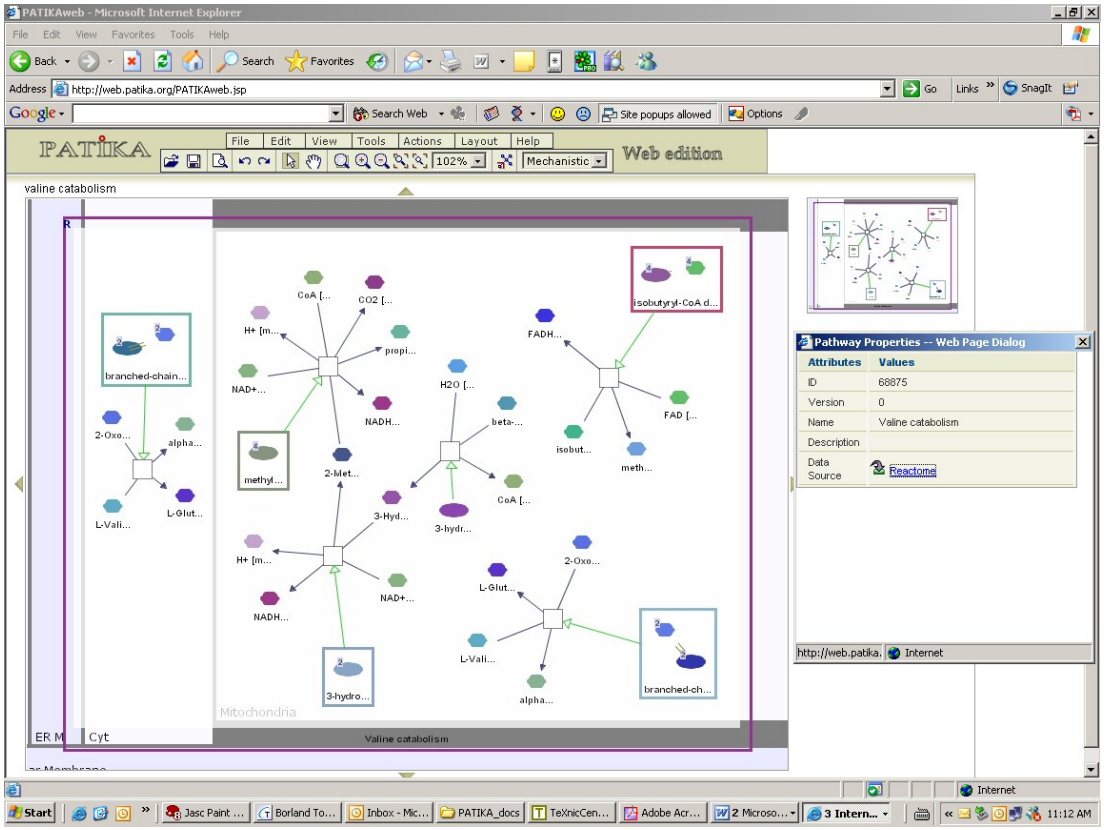

Figure 3.5: User interface of PATIKA*web*.

requiring operations will be performed on the server side. Consequently, what we tried to manage at the client side required basic scripting.

Simply what we have been mentioning so far is a dynamically created html page with client side scripting. Client side scripting is a strategy preferred lately, used to reduce both the CPU requirement at the server and the communication cost. JavaScript is the technology used for the specified cases.

We represent the pathway data as still images in *.jpeg* format in the canvas. Creation of this still image and transmission of this image is provided by *TSV JSP Edition* framework. We customized this functionality by adding limited editing operations on the image. All editing actions performed on the canvas, other operations like pathway object property requests are initially handled by JavaScript, where we can perform primitive operations. Computation requiring requests for the client action are submitted to the bridge.

The creation of the still jpg images are done by TSV JSP Edition tag libraries. Similarly creation of the overview image is done with these libraries [16].

We also provide the utility to export the locally visualized data to other exchange formats like BioPAX[3] and SBML[4]. Moreover the users are able to save the current pathway in PATIKA model, as a *pmdl* file, which is the PATIKA pathway model file in XML format, and load it later. All these use cases require, the facility to load a file from PATIKA bridge. Therefore load/save actions, that require access to the user's local machine are designed to be part of the html design. Since these operations are not part of the application logic, we considered them as the client side programming and designed the JSP pages accordingly.

3.2.3.1 Detailed Design of Query Dialog in PATIKA*web* client

Another component that we designed as part of the client side, is the query dialog. Once we investigated the requirements of querying facility in Section

[3]http://www.biopax.org/
[4]http://sbml.org/index.psp

3.2.1, we observed that the query interface required extensive capabilities such as nesting, saving, loading, etc. One alternative was to build these facilities with JSP technology, with a sacrifice in extensive user interface facilities. On the other hand, we had an alternative that enabled us to reuse the existing querying interface we have developed for PATIKA*pro*. This alternative was implementing the query dialog as an applet. By this way, we were able to adapt the old code base with minimal modification and moreover, we got full advantage of applet in interactive query creation and submission phase. However we admit that this choice has a drawback. Downloading an applet over an internet connection may be a slow operation, which is an issue we try to avoid. Considering the pros and cons of this approach, we decided that sacrificing speed over extensive querying facilities was feasible. Thus we designed query dialog as an applet and added the facility that enabled the clients to download the JavaTM Archive (jar) files and to execute the query applet at the client side.

In the query applet, we provide query saving, loading and executing actions. Interface of the query applet was previously designed. We have customized the design for PATIKA*web* as in Figure 3.6. *QueryApplet* contains the interface components for building the nested queries such as *EditorPanel* and *QueryForest*. *EditorPanel* is an abstract editor panel class for handling different types of queries such as field queries or neighborhood queries. The creation of the query object is handled at the these editors. Functionalities of the *QueryApplet* is handled by the *QueryAppletManager*. Saving, loading and execution of the queries are the responsibilities of the *QueryAppletManager*.

Sample sequence for the execution of a query can be seen in Figure 3.7. In this sequence diagram, the client executes the selected query, by sending the request to the corresponding data, which we will refer as *PatikaSessionData* in the PATIKA bridge with an http request. In this http request, PATIKA*web* client sends the composed query object in XML. *PatikaSessionData* receives the request for the execution of a query action with the query as a stream and behaves like a pipe and forwards the stream to *Proxy*. *Proxy* is responsible for the communications with PATIKA server. *Proxy* sends the query to the PATIKA server and waits for

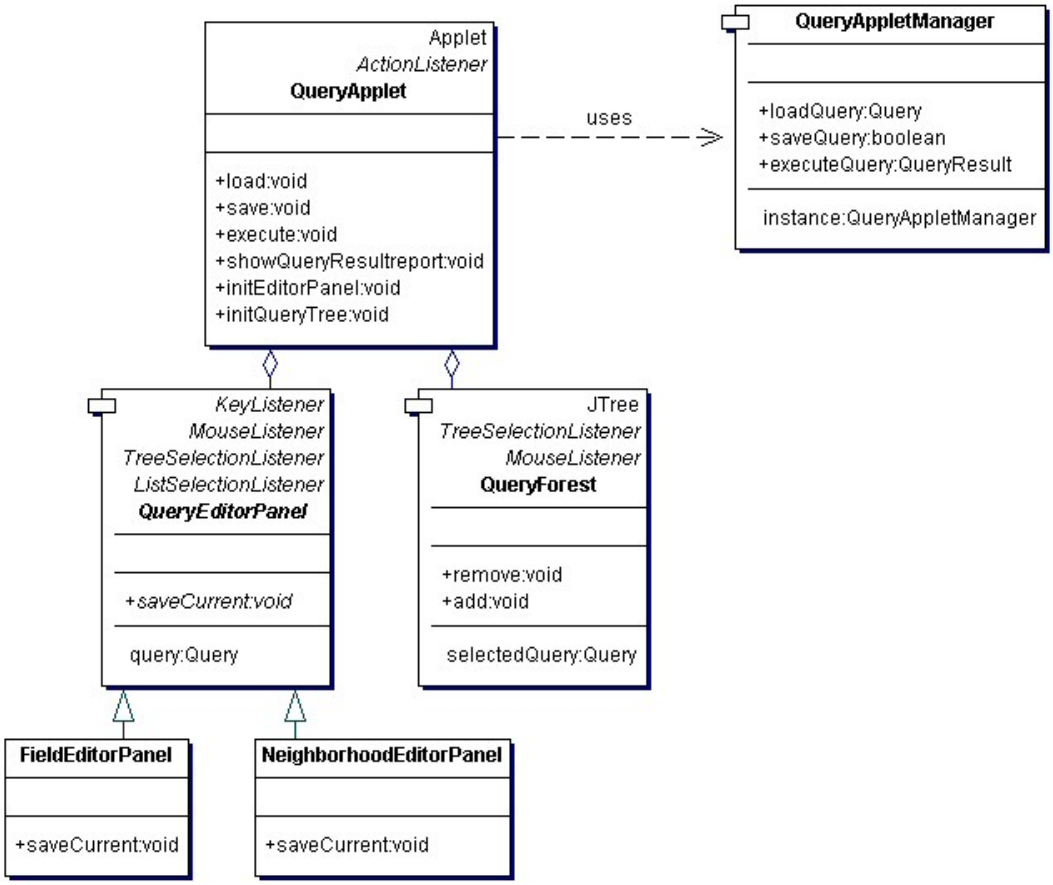

Figure 3.6: Class diagram for query applet

the query result. Once the result is received from the PATIKA server, *PatikaSessionData* informs the clients of the results and waits for a feedback. PATIKA*web* client may choose to see or ignore the query results. If the user wants to see the query result in the browser window, then the visualizable graph for the query result is processed. User has the option to merge the query result to the current view or visualize it as a new graph. All these visualization related operations are performed at the bridge and the graph is rendered as still image and sent to the client side format via http. Creation of the still image in jpeg format is handled by the TSV API.

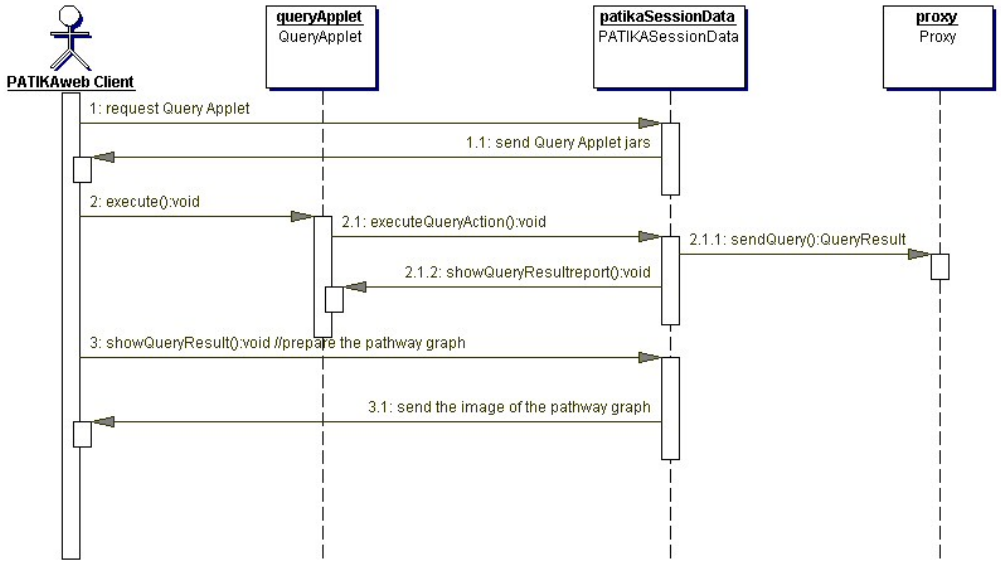

Figure 3.7: Sequence diagram for a query execution

3.3 PATIKA*web* Bridge

3.3.1 Analysis of Pathway Data Visualization and Editing

Once the requested model is retrieved from the PATIKA database, the user at the PATIKA*web* client side is able to visualize the result. In other words, PATIKA*web* client behaves like a graph window for the PATIKA*web* bridge.

Users are able to create and extend pathway models by performing queries both through an interface given and also through the object interface in the current view. As the obtained model gets complicated, the users want to save the current model. We provide saving capability in PATIKA model language, *pmdl* and allow loading it later. Moreover exporting the obtained graph to other standard data formats like BioPax[5] and SBML[6] are supported for enabling data portability.

Considering the read-only structure of our tool, we provide mechanisms to

[5]http://www.biopax.org/
[6]http://sbml.org/index.psp

increase quality of the visualization facility. We considered following components to increase this quality:

- **Bioentity and Mechanistic Levels:** Pathway data in bioentity and mechanistic levels should be visualized independently. We have to provide a visual representation of the cell model, while visualizing the mechanistic level data.
- **Layout of the pathway data:** Since we provide extensive querying facility, we need to present the result in the best possible way. Therefore, we apply a layout process on the pathway data prior and during the visualization.
- **Editing Operations:** Although we provide a read-only interface for pathway data in PATIKA database, to provide better visualization, we facilitate operations like zoom in/out, delete, pan/scroll, hit testing, overview, drag/drop, etc. Extending a pathway by submitting new queries are also be provided.
- **Complex pathway representation:** As mentioned in Section 1.2, pathway data can be hard to visualize in two dimensions with limited graphical representation facility. We need to represent this information with maximum coverage. This requirement can be thought of an independent component, therefore analysis and design of this component is studied separately in the following section.

3.3.1.1 Analysis of Complex Pathway Visualization

Incomplete and complex information contributing to the network of higher level information must be represented. Abstractions are introduced for this purpose in PATIKA ontology. We represent abstractions graphically with compound nodes having child graphs with the *TSV* API [15]. Compound nodes enable nesting child graphs, in other words provides us the framework to represent the compound pathway information for different levels of information.

Graphical representation of abstractions in PATIKA ontology is an important task. Since abstractions represent a cellular events/pathways, a molecular component can be part of two different cellular pathways. Representing two different pathways sharing an element, either requires making multiple copies of a molecule, or introducing a new graphical representation technique. Representing a molecule multiple times in a cell, within different pathways is misleading by giving wrong quantitative information. Therefore, we proposed different visual states of pathways, or abstractions as in our ontology.

We visualize abstractions as compound graphs with different visual states in our graphical representation scheme. These states and their meanings in graphs are as follows:

- **Expanded:** The default view of a compound graph is its expanded state. The child graph of the expanded node is visualized.
- **Collapsed:** A Collapsed compound node is represented as a single black-box like node, with the child graph folded. Collapsing a compound node conceals its child graph.
- **Hidden:** We define this state, as the abstraction being not visible. However in this state, we sill represent the members of the abstraction. Only we hide the information of abstraction owning the members. When an abstraction is hidden, its child graph becomes part of the parent graph and can be visualized.
- **Holo:** In this state we represent the pathway information without the compound node and child graph components. We present this information by adding a color information to the member of the abstraction. When an abstraction is in holo state, its child graph becomes part of the parent graph, as in hidden state; however we present the information of the abstraction with color coding.

You can visualize the abstractions in *Expanded* and *Holo* states in Figure 3.8. States with a green holo color are members of another abstraction. In Figure 3.9,

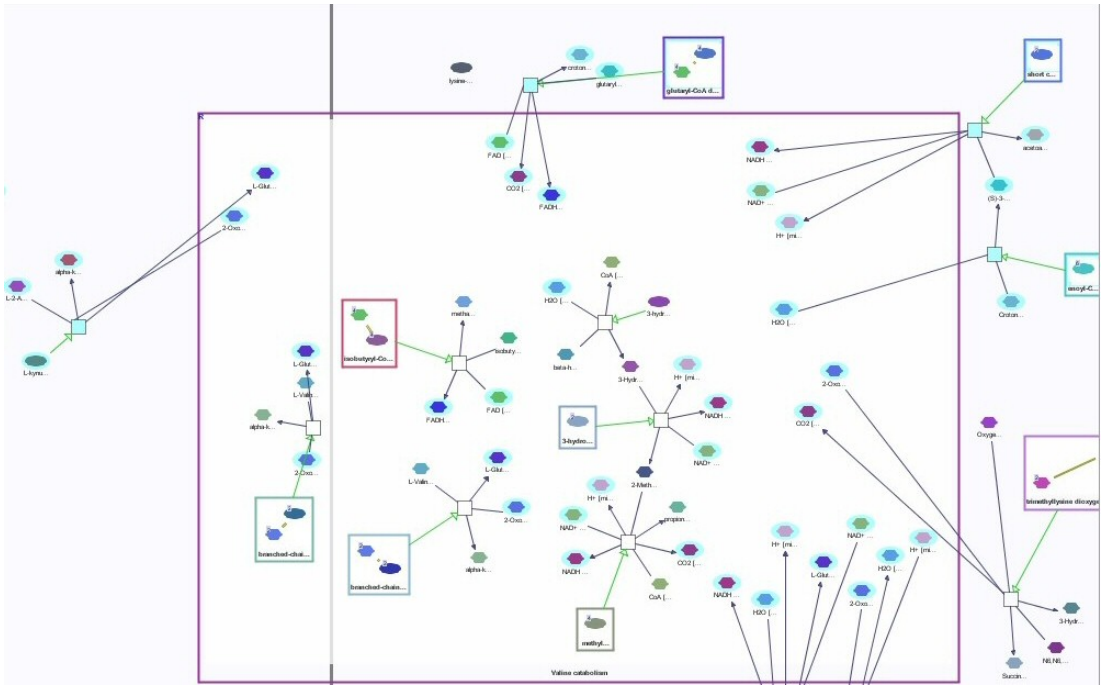

Figure 3.8: Abstractions in *Holo* and *Expanded* states

we can see the abstractions in *Collapsed* and *Expanded* states. Abstractions in the cytoplasm, drawn as black nodes are abstractions in collapsed state and the abstraction in the nucleus is in expanded state, where we can see its members in the child graph.

We define a concept of *relationship* among abstractions in terms of the relations among the members of the abstraction. *Abstraction A* and *abstraction B*, may share a subset of their elements. In this case we define these *abstraction A* and *abstraction B* as having *improper inclusion relationship*. On the other hand, *abstraction A* may contain completely *abstraction B* itself as a sub-component. In this case we define these *abstraction A* and *abstraction B* as having *proper inclusion relationship*. These definitions are critical when we try to find the best possible visual state in a pathway and present it.

As we have stated earlier in Section 2.1, we have five types of abstractions which are Regular Abstraction, Homology State, Homology Transition, Incomplete State and Incomplete Transition. We can represent all of these abstractions in expanded form. However due to the restrictions introduced by the cell model

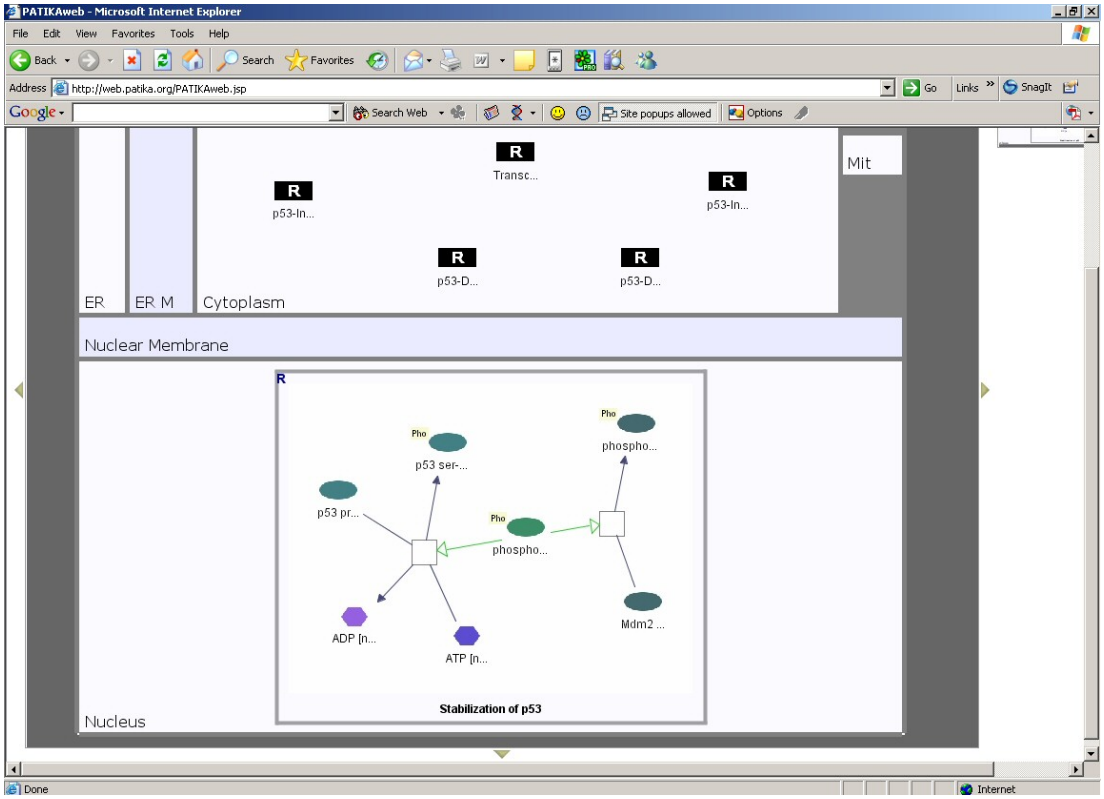

Figure 3.9: Abstractions in *Collapsed* and *Expanded* states

	Hidden	Collapsed	Holo	Expanded
Regular Abstraction (Spanning 1 compartment)	+	+	+	+
Regular Abstraction (Spanning many compartments)	+	+	+	-
Incomplete State	+	+	-	+
Homology State	+	+	+	+
Complex State	-	+	-	+

Table 3.1: Summary of the valid options for all types of abstractions

and biological constraints, not all visual states are applicable for all abstraction types. Allowed visual states for abstraction and complexes can be seen in Table 3.1.

3.3.2 Detailed Design of PATIKA*web* Bridge

PATIKA*web* bridge is the tier, where the application logic is implemented. Data access and visualization are the basic functionalities of our application. As we have mentioned earlier, we have a visualization approach for pathway visualization, developed earlier for other PATIKA tools. We have customized this code base to integrate it with the visualization scheme of PATIKA*web*.

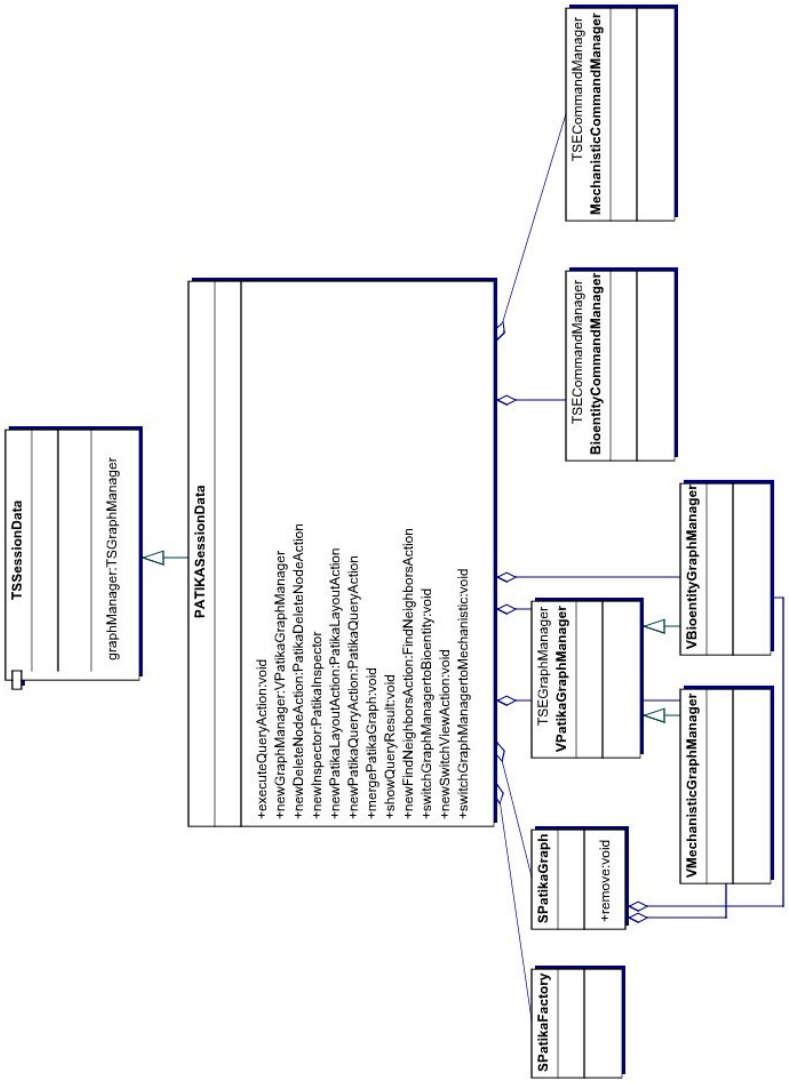

Figure 3.10: Bridge design

In Figure 3.10, the architecture of the session data can be seen. Session data, which is *PatikaSessionData* for our case, is instantiated as JavaBean, in the jsp file. As the user requests for the jsp file, this bean is instantiated in the bridge with the scope of session. This session bean keeps the data for that session, containing the current pathway data, query results and visual properties. Therefore, any action requiring a modification on any of these data, and consequently on the still image displayed on the canvas, is handled by the session data.

PatikaSessionData, is extended from *TSESessionData*, to customize it according to our preferences. Concept of two graph managers per session data, displaying query results, expanding/collapsing and performing other editing operations on the graph data are achieved with *PatikaSessionData*. Each *PatikaSessionData*, has one *BioentityGraphManager* and one *MechanisticGraphManager*. We carry the subject-view mechanism of PATIKA visualization scheme to PATIKA*web* as well. That's why each *PatikaSessionData* has one subject graph, *SPatikaGraph* instance (See Figure 3.10).

Although we limit the editing operations on the visualized pathway graph in PATIKA*web* , we allow operations like delete, expand/collapse, merge graphs and layout. Since these operations modify the graph topologically or visually, undo operations must be provided. However due to our *one subject and multiple view* mechanism, one operation in one view may require a modification in the other view. For example, deletion of a bioentity in the bioentity view causes removal of all of its states in the mechanistic view. If we want to perform an undo operation on bioentity view, we need to perform the corresponding undo operations on the mechanistic view. Command structure provided by *TSV JSP Edition* [16], does not provide this type of control. Therefore, we extended *TSECommandManager* into to sub-command managers as *BioentityCommandManager* and *MechanistiCommmandManager* (See Figure 3.10).

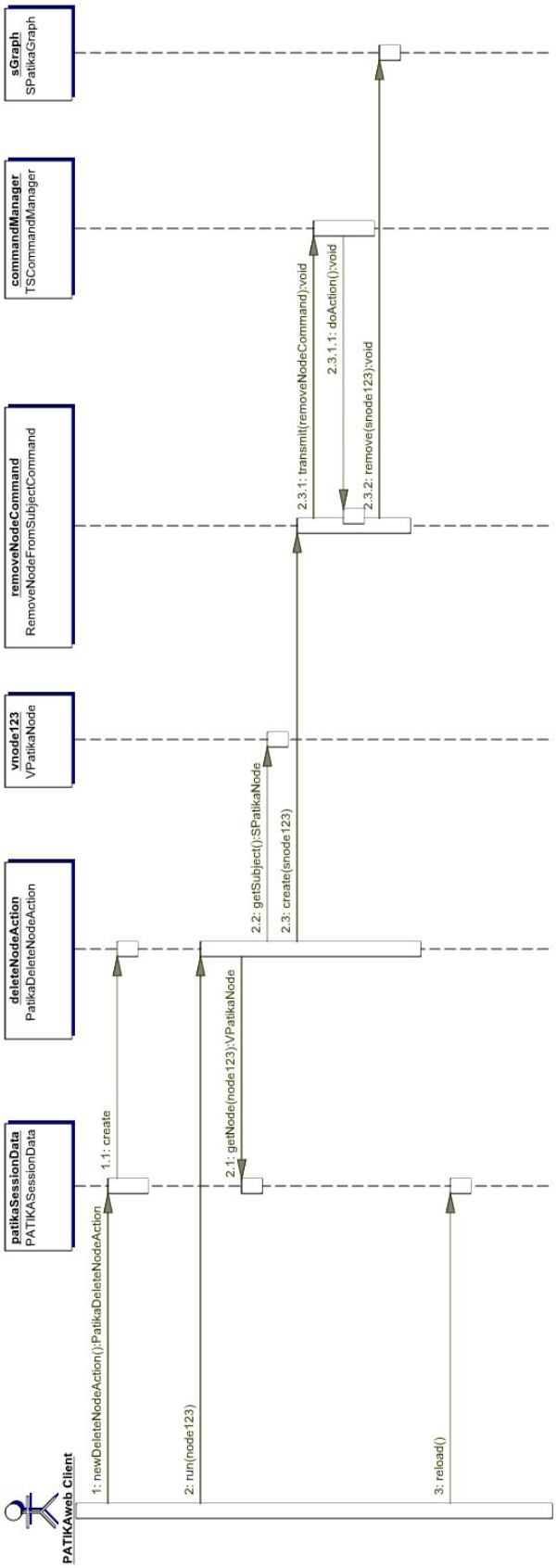

Figure 3.11: Sequence diagram for node deletion in PATIKA*web*

In our design every interactive operation requiring process on the pathway data, requires access to the PATIKA bridge. All of these editing operations has a corresponding *action class* (e.g. *PatikaExpandAction, PatikaLayoutAction, etc.*) at the bridge. As the user performs any of these editing operations, the request to perform that action at the session data is sent to the bridge. For example, in Figure 3.11, we see a sample sequence of a node deletion action. PATIKA*web* client performs the action on the browser, by calling the corresponding JavaScript. This script makes a request to the jsp file. As the jsp is compiled and the Java code is executed, *PatikaSessionData*, calls the corresponding method, *newDeleteNodeAction()*. This method creates and returns an instance of the action object. *PatikaDeleteNodeAction* transmits the command for the deletion of the node. As we are dealing with deletion of view-level objects and the command for node deletion requires the subject-level node, we need to access the node's subject at this point. We transmit the command to the correct command manager based on the type of the node, either *MechanisticCommandManager* or *BioentityCommandManager*. This command removes the node from the graph and adds the command to the undo stack, for enabling undo operations.

3.3.2.1 Complex Pathway Visualization Design

In PATIKA*web* the results of queries are visualized in the best possible way. While preparing the visual representation of the query result, one issue that we need to consider is the abstractions. As mentioned in Section 3.3.1.1,we allow abstractions to be visualized in four different visual states. Some of these visual states are not applicable for all abstractions, based on their relationships as defined in Section 3.3.1.1.

Visual state of an abstraction to be visualized in PATIKA*web* is decided in the PATIKA bridge and visualized in that state. Limited visual state transitions are allowed by user's choice. A PATIKA*web* user can only change an abstraction's visual state between expanded and collapsed. On the other hand an abstraction can be visualized in any of the visual states applicable for that abstraction, when it is initially visualized in the graph. The selection of the proper visual state of the

State in Editor	**Programmatic State**
Hidden	*ReadyToViz* or *NotReadyToViz*
Holo	*Visible_Holo* or *Invisible_Holo*
Expanded	*Visible_Expanded* or *Invisible_Expanded*
Collapsed	*Visible_Collapsed* or *Invisible_Collapsed*

Table 3.2: Programmatic states for abstractions' visual states

abstraction is done automatically. Based on the relationship of the abstraction with the other abstraction in the current graph, its visual state is decided. The order of visual state tested is like this: *expanded, collapsed, holo and hidden state.*

Programmatically, visual state transitions for an abstraction depends on the other associated abstractions in the view. We defined low level visual states matching with the user perceived visual states. User perceived visual states for abstractions correspond to the programmatic visual states of an abstraction as in Table 1. Complex molecules are compound molecules, although they are not abstractions ontologically and not all the visual states are allowed for complexes. Correspondences among them can be seen in Table 3.2.

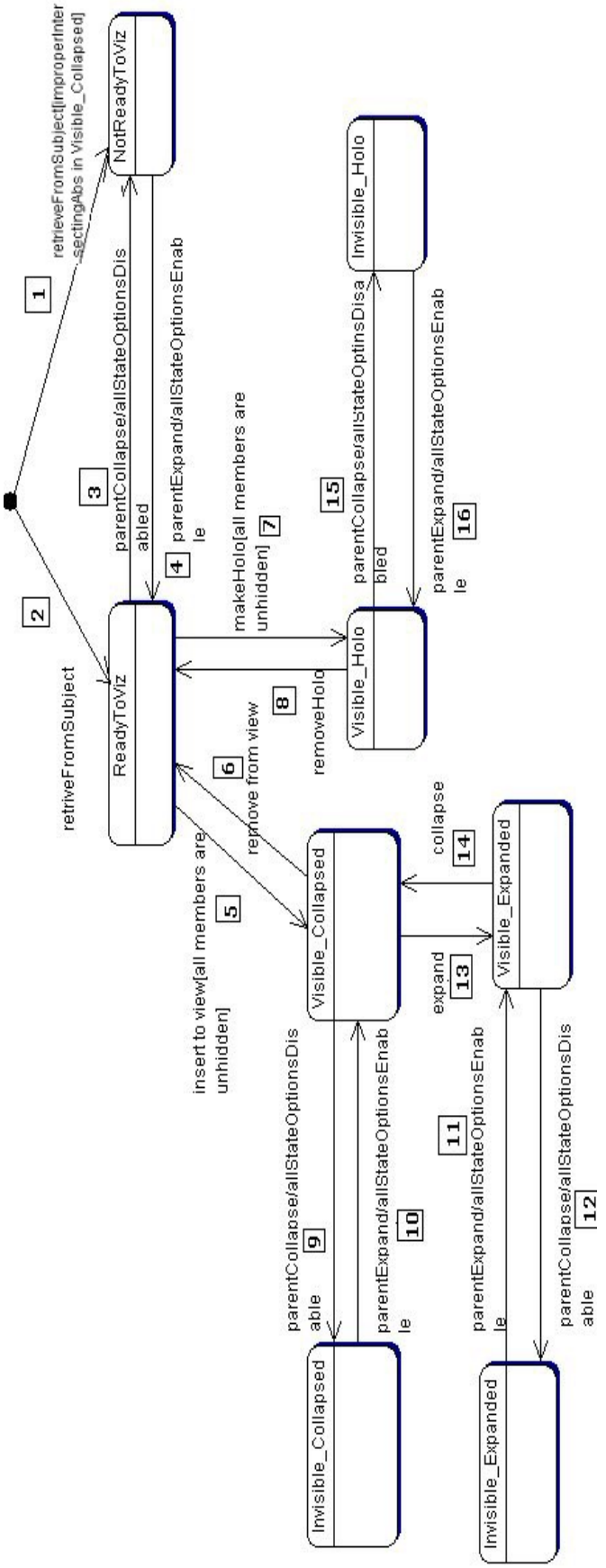

Figure 3.12: State-transition diagram of abstractions' visual states

Possible dependencies among abstractions are explained in the state-transition diagram (see Figure 3.12). As can be seen from the diagram, every user perceived state is divided into two sub states to manage the dependencies. One important design decision to point out is the order of transitions. An abstraction cannot pass directly from *holo* state to *expanded* state. Abstraction must first pass to *hidden* state, then to *collapsed* state and then to *expanded* state.

Dependencies among abstractions affect the visual state transitions of individual abstractions. There are two basic dependencies among abstractions in the same view: *proper* and *improper* intersection dependencies. In the proper intersection, members of one abstraction is a proper subset of members of another abstraction. We will refer the abstraction which contains all the members of another abstraction as the parent abstraction of the other. In the improper intersection dependency, two (or more abstractions) share one or more members but neither contains all members of the other. State transitions according to these dependencies are as in Figure 3.12. Each transition is labeled with a number and events that trigger these transitions, conditions that control these transitions and actions taken with these transitions are explained in detail below:

1. **Event:** Insert to view occurs.
   Condition: If abstraction A has an improper intersection with some other abstraction B and abstraction B is in *visible collapsed* state.

2. **Event:** Insert to view occurs.
   Condition: If there is no such dependency as in (1) in the view.
   Action: Disable *collapsed* state if another abstraction B having an improper intersection with current abstraction is in *holo* state.
   Action: Disable *collapsed*/*expanded* states if another abstraction B having an improper intersection with current abstraction is in *expanded* state.
   Action: Enable *collapsed*/*expanded*/*holo* states if all members are in unhidden states.

3. **Event:** Parent abstraction of abstraction A changes to *visible collapsed* state.

Event: Abstraction B having improper intersection with abstraction A changes to *visible collapsed* state.

4. **Event:** Parent abstraction of abstraction A changes to *visible expanded* or *ready to visualize* state from *visible collapsed* state.
   Event: Abstraction B (having improper intersection with abstraction A) changes to *visible expanded* or *ready to visualize* state from *visible collapsed* state.
   Action: Disable *collapsed* state if another abstraction B having an improper intersection with current abstraction is in *holo* state.
   Action: Disable *collapsed*/*expanded* states if another abstraction B having an improper intersection with current abstraction is in *expanded* state.
   Action: Enable *collapsed*/*expanded*/*holo* states if all members are in unhidden states.

5. **Event:** Visual state of abstraction A is set to collapse.
   Condition: All members of abstraction A must be in unhidden states.
   Action: insert() into the view is performed.
   Action: Enable all state options for this abstraction.
   Action: All member abstractions change their visible states to corresponding invisible states. If in *ready to visualize* state, changes to *not ready to visualize* state. Disable *hidden* state for member abstractions.
   Action: All abstractions having improper intersection with abstraction A, change their states to *not ready to visualize* state from *ready to visualize* state. Disable *collapsed*/*expanded*/*holo* state options for these abstractions.

6. **Event:** Visual state of abstraction A is set to *hidden*.
   Action: All member abstractions change invisible states to corresponding visible states. Remove from the view is performed.
   Action: All abstractions having improper intersection with abstraction A, change their states to *ready to visualize* state from *not ready to visualize* state. Remove from the view is performed.
   Action: Disable *collapsed* state if another abstraction B having an improper intersection with current abstraction is in *holo* state.
   Action: Disable *collapsed*/*expanded* states if another abstraction B having

an improper intersection with current abstraction is in *expanded* state.
Action: Enable *collapsed*/*expanded*/*holo* states if all members are in unhidden states.

7. **Event:** Visual state of abstraction A is set to *hidden*.
   Action: Disable *collapsed* state if another abstraction B having an improper intersection with current abstraction is in *holo* state.
   Action: Disable *collapsed*/*expanded* states if another abstraction B having an improper intersection with current abstraction is in *expanded* state.
   Action: Enable *collapsed*/*expanded*/*holo* states if all members are in unhidden states.

8. **Event:** Parent abstraction of abstraction A changes state to *visible collapsed*.
   Action: All state options for this abstraction are disabled.

9. **Event:** Parent abstraction of abstraction A changes state to *visible expanded* or *ready to visualize*.
   Action: Enable all state options for this abstraction.
   Action: All member abstractions change their visible states to corresponding invisible states. If in *ready to visualize* state, changes to *not ready to visualize* state. Disable *hidden* state for member abstractions.
   Action: All abstractions having improper intersection with abstraction A, change their states to *not ready to visualize* state from *ready to visualize* state. Disable *collapsed*/*expanded*/*holo* state options for these abstractions.

10. **Event:** Parent abstraction of abstraction A changes state to *visible expanded* or *ready to visualize*.
    Action: All state options for this abstraction are enabled.
    Action: All member abstractions change invisible states to corresponding visible states. *hidden* state is disabled for member abstractions.
    Action: For abstractions having improper intersection:

    - If they are in *not ready to visualize* state, change to *ready to visualize* state, disable *collapsed* state option, enable *expanded*, *hidden* and *holo* state option for these abstractions.

- If they are in *visible holo* state, for current abstraction disable *collapsed* and *expanded* state options; enable *hidden* and *holo* state options. Disable *collapsed* state for improper intersecting abstractions.

11. **Event:** Parent abstraction of abstraction A changes state to *visible expanded* or *ready to visualize*.
    Action: All state options for this abstraction are enabled.
    Action: All member abstractions change invisible states to corresponding visible states. *hidden* state is disabled for member abstractions.
    Action: For abstractions having improper intersection:

    - If they are in *not ready to visualize* state, change to *ready to visualize* state, disable *collapsed* state option, enable *expanded*, *hidden* and *holo* state option for these abstractions.
    - If they are in *visible holo* state, for current abstraction disable *collapsed* and *expanded* state options; enable *hidden* and *holo* state options. Disable *collapsed* state for improper intersecting abstractions.

12. **Event:** Parent abstraction of abstraction A changes state to *visible collapsed*.
    Action: All state options for this abstraction are disabled.

13. **Event:** Visual state of abstraction A is set to *expanded*.
    Condition: All members of abstraction A must be in unhidden states.
    Action: All member abstractions change invisible states to corresponding visible states. *hidden* state is disabled for member abstractions.
    Action: For abstractions having improper intersection:

    - If they are in *not ready to visualize* state, change to *ready to visualize* state, disable *collapsed* and *expanded* state option, enable *hidden* and *holo* state option for these abstractions.
    - If they are in *visible holo* state, for current abstraction disable *collapsed* state option; enable *expanded*, *hidden* and *holo* state options. Disable *collapsed* state for improper intersecting abstractions.

14. **Event:** Visual state of abstraction A is set to collapse.
    Condition: All members of abstraction A must be in unhidden states.
    Action: Enable all state options for this abstraction.
    Action: All member abstractions change their visible states to corresponding invisible states. If they are in *ready to visualize* state, change their state to *not ready to visualize* state. Disable *hidden* state for member abstractions.
    Action: All abstractions having improper intersection with abstraction A, change their states to *not ready to visualize* state from *ready to visualize* state. Disable *collapsed/expanded/holo* state options for these abstractions.

15. **Event:** Parent abstraction of abstraction A changes state to *visible collapsed*.
    Action: All state options are disabled for this abstraction.

16. **Event:** Parent abstraction of abstraction A changes state to *visible expanded* or *ready to visualize*.
    Action: Disable *hidden* state for all member abstractions .
    Action: All abstractions having improper intersection with abstraction A, must disable their *collapsed* state option.

Once we have identified the flow of the states and transitions, we have designed the system and integrated it within the existing view design as in Figure 5.13. As seen in this figure management of the visual states of each abstraction is handled by its view manager. State transition requests are handled in *AbstractionViewMgr*.

Each view manager of an abstraction keeps the lists of abstractions having improper relation or proper relations with itself. Upon changing a visual state of an abstraction, its view manager iterates over its dependency list, makes required modifications. Based on the limitations and relations, some visual states for the abstractions in the dependency lists are enabled or disabled. A sample sequence for expanding a *hidden* abstraction can be seen in Figure 3.14.

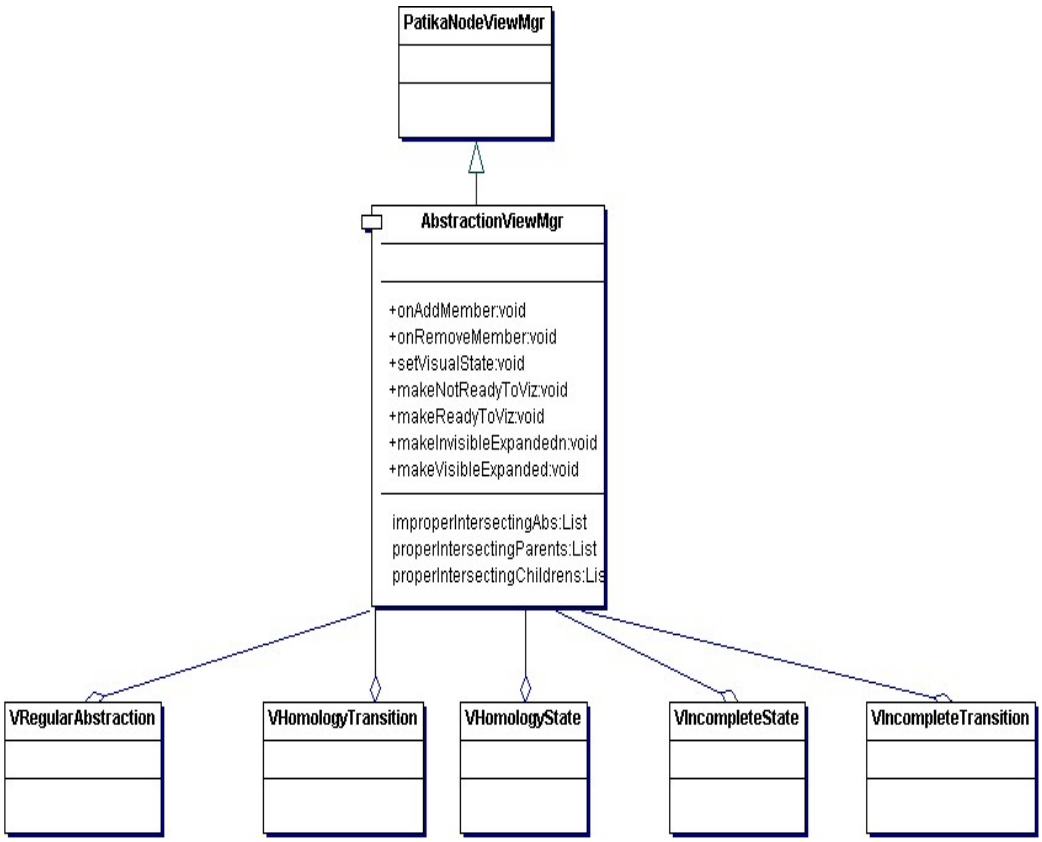

Figure 3.13: Design of abstraction view manager

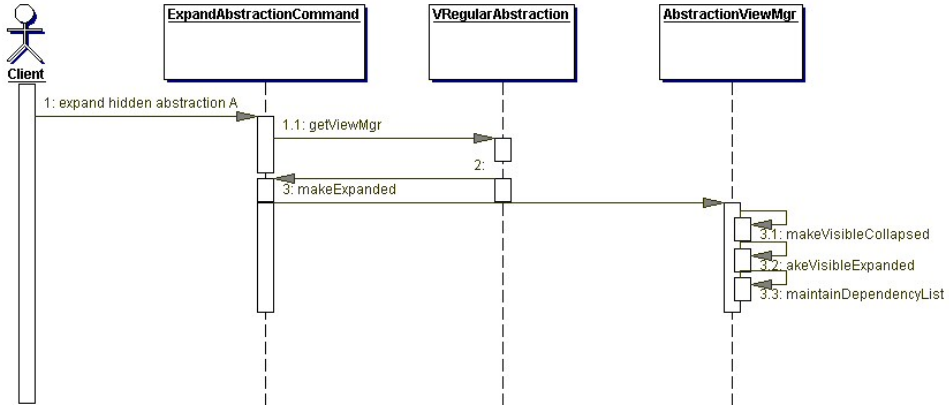

Figure 3.14: Sequence of expanding a *hidden* abstraction

3.4 PATIKA Server, Data Flow and Communication

PATIKA*web* server currently is the tier keeping the data. In PATIKA database we have pathway data integrated from different databases. We access to PATIKA server with PATIKA*pro* clients, in other words with *fat clients* and PATIKA*web* clients via the bridge. Therefore the interface and the protocols for data access are designed to be exactly same. Considering the modular structure for PATIKA*web*, the PATIKA*web* bridge as explained in Section 3.3.2, is designed to have the interface of a fat client for PATIKA server. Besides reusability, this modular and scalable architecture provided us the flexibility to modify the client or the application module, without the consideration of PATIKA server.

Communication of the PATIKA*web* bridge to the PATIKA database is done over http. Since read-only operations are allowed in PATIKA*web* this communication is basically required for data access. As the user sends the query to the bridge via http in XML, the bridge behaves like a pipe and forwards the XML representation to the PATIKA database using the http protocol again. At this point, bridge knows nothing about the query, as it only transfers the XML file. Up to this point, the client and the bridge together behaves like a fat client. When the server performs the query and returns the query result over http to PATIKA*web* bridge in XML file, bridge itself behaves like a fat client for the representation of the query. The query result object is extracted from the XML and then is processed to represent it with a pathway graph. Once the graphical representation of the query result is finished, only the still image is passed back to the client side via http.

3.5 Implementation Details

We have used different technologies in the implementation of PATIKA*web*. As we have implemented a three-tier architecture, each tier uses different components to achieve its task (See Figure 3.15). In the client side, we use JavaScript, HTML

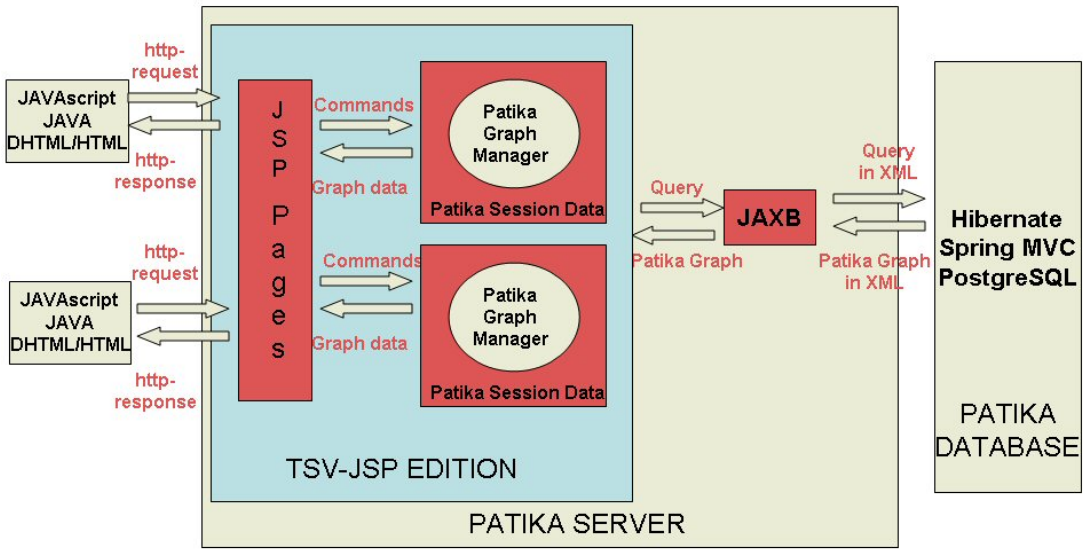

Figure 3.15: Architecture diagram of PATIKA*web* and their communication scheme

and DHTML to build the Web pages. These technologies provide us the dynamic content creation and limited event handling. We us DHTML for creating the menu and toolbar. In PATIKA*web* server, we use JSP technology within the *TSV JSP Edition* framework containing the graph editing toolkit. For the data flow, we use the XML format. We transfer queries from query applet at the client side to PATIKA*web* server and receive query results in XML format from PATIKA server. To handle marshaling and unmarshaling of these XML files we use Java Architecture for XML Binding (JAXB)[7] technology. In the PATIKA server PostgreSQL is used as the Database management system. For the object-relational mapping, Hibernate[8] is used. Control of the components in the server with MVC and managing the Hibernate are handled with Spring framework[9].

Prior to the design process, we investigated the *Tom Sawyer Software, JSP Edition*. This edition provided us the framework for a thin client, where we can present any graph visualization facility at a Web page. In this framework JSP

[7]http://java.sun.com/developer/technicalArticles/WebServices/jaxb/
[8]http://www.hibernate.org/
[9]http://www.springframework.org/

(Java Server Pages) technology is used. JSP technology facilitates the creation of dynamic Web pages[10], which is crucial in our service. JSP technology is an extension to the Servlet Technology, which are server side modules for interactive Web applications[11]. Moreover in the framework provided by *TSV JSP Edition*, *JavaBeans*, a reusable component[12] which we can integrate with our Web application to represent objects mapped to the clients' at the server side is also used.

Using these technologies, *TSV JSP Edition* provided us the basic skeleton that we will customize extensively for our Web service. Below is the sketch of *TSV JSP Edition* covering both the mentioned technology details and the provided framework :

1. Addressing the browser to main page, makes the request to the corresponding jsp page. This jsp page is first converted to Java code, compiled and loaded in the browser.

2. In the main jsp page loaded, there is html code and JavaScript and jsp code embedded.

3. When the main page is being loaded, an instance of the object is created that is defined as session bean in the jsp code part. The session bean instantiated in this framework is mapped with the graph that the user will be dealing with through out the session.

4. Every action on the main page is associated with a separate page, i.e has separate .jsp for every action. (Such as graph editing actions or file loading actions)

5. Every component that requires an action to be performed and be reflected to the browser, is linked to a JavaScript code.

6. Upon a click on a component on main page, corresponding JavaScript code forwards the browser to the corresponding jsp page.

[10]http://java.sun.com/products/jsp/
[11]http://java.sun.com/products/servlet/overview.html
[12]http://java.sun.com/products/javabeans/

7. In each of these .jsp pages, the session bean is used to perform corresponding actions. For example in *TSLoadFileAction.jsp*, *loadgraph()* operation is called on canvas with the given file name. (This filename is carried from the selection on the main page).

8. Every .jsp page includes other .jsp references to maintain other required actions. For example, *TSLoadFileAction* after loading the graph on canvas, needs to call *TSFitInCanvas.jsp* to fit the created image of the drawing in the drawing area. After the completion of the load action, main page must be reloaded. To be able to do this, *TSGoToMainPage.jsp* action page is included in all proper action pages.

9. For some actions such as zoom, scroll, or selection of "fit In canvas" explicitly, only the image displayed in the main page is changed. The whole page is not reloaded.

As we have been using *Tom Sawyer Software*'s visualization tools, we choose to use the same framework to increase the development time. This choice has also served our strategy to reuse the previous visualization solutions developed in *Tom Sawyer Software*'s Graph Editing Toolkits. Therefore we have built our design on top of the skeleton provided by *TSV, JSP Edition*. However due to the license agreements of *TSV JSP Edition*, we could not provide an open source or non-limited license agreement tool. We warn the users of PATIKA*web* with an opening page about the license agreements. PATIKA*web* is a freely available software through Internet for non commercial uses only.

We can think of the *Query Applet* as an independent component within PATIKA*web*. Therefore we considered different implementation issues for the query applet. As seen in the use cases specified in Section 3.2, PATIKA*web* clients should be able to save the queries that they have built in the query interface provided. Loading of these queries should be allowed as well. These scenarios require that a downloaded applet performs an access to the client's local machine. For obvious security reasons, any downloaded applet cannot perform a read or write access to the local machine. Only *trusted* applets can perform such operations. JDK 1.3 or higher provide the technology to create trusted applets.

Figure 3.16: Screenshot for the signed applet jars

Therefore we used the *keytool* to manage the keystores, certificates and *jarsigner* to sign the applet archives. These tools are available with JDK 1.3 or higher. When a PATIKA*web* client, downloads the query applet, a warning window pops up (See Figure 3.16) and asks the user whether they want to trust the application provider, "'Bilkent Center for Bioinformatics"'. If the user trusts the provider, the client downloads the applet jars, which is in total approximately 3MB, and then read and write operations to the client machine can be performed.

To access PATIKA*web* users should have one of the following browsers:

- Microsoft Internet Explorer 5.5 or more : It is the most frequently used browser among average Internet users.
- Mozilla Firefox 1.0 or more : Its usage ratio increased immensely lately.
- Apple Safari 1.0 or more: It is very popular in the biological community.

We considered the domain and the target user profile, deciding on the browsers that we will support. Biological community generally uses Apple Computers, which uses MAC OS and Safari Web browser. We decided on the browsers to support based on this fact and the statistics in Table 3.3 taken from W3 Schools[13].

[13]http://www.w3schools.com. W3Schools is a website for people with an interest for web technologies. These people are more interested in using alternative browsers than the average user

2005	**IE6**	**IE5**	**Firefox**	**Mozilla**	**NN 7**
July	67.0%	6.7%	19.7%	2.6%	0.5%
June	65.0%	6.8%	20.7%	2.9%	0.6%
May	64.8%	6.8%	21.0%	3.1%	0.7%
April	63.5%	7.9%	20.9%	3.1%	0.9%
March	63.6%	8.9%	18.9%	3.3%	1.0%
February	63.9%	9.5%	17.9%	3.3%	1.0%
January	64.8%	9.7%	16.6%	3.4%	1.1%

Table 3.3: Browser statistics month by month from W3 Schools

Chapter 4

Conclusions

Development of Patika*web* started with the realization of the requirement of an easy-to-access tool facilitating analysis on the pathway data. Then we clearly identified the requirements and use cases for this tool. Prior to the design step, based on the requirements we decided that we will build a Web based service with *TSV, JSP Edition*. In the design step, the biggest challenge was integrating the available design and codebase for pathway visualization to the new design of Patika*web*. In the implementation process, we have clearly split the components based on the multi-tier architecture and thus can be developed independently.

Patika*web* is now available for non commercial use through the address http://web.patika.org, where users can access to pathway data in Patika database and analyze this data with the unique and highly extensive visualization capabilities. Every non-commercial oriented researcher can access Patika*web* via their web browsers and get advantage of its functionalities.

Patika*web* currently serves the community with its unique and extensive capabilities, through an easy-to-access user friendly interface.

4.1 Contribution

Once we have decided on developing a thin client, we initially did some search on the available free graph editing packages like GraphViz[1], GINY[2], GVF[3], etc. Later we have decided on using *TSV, JSP Edition*. Unfortunately *TSV, JSP Edition* final release was not available those days and documentation was not complete. Therefore we analyzed the package and tried to understand their architecture. Once the structure of *TSV, JSP Edition* was clearly perceived, the next step was to design PATIKA*web*. Having discussed the requirement of PATIKA*web*with the team, we did the design to integrate the current architecture of PATIKA*pro* with the new architecture of PATIKA*web*. In the implementation step, we basically implemented the bridge side, where the main integration occurred. In the client side, we implemented the framework for the web service. Once the basic skeleton was ready, implementation of editing operations was much easier. We implemented some basic functionalities for the editing operations like save, load, export, etc. We had a senior project team responsible for the creation of the dynamic content of the PATIKA*web* at the client side. Rest of the PATIKA team also contributed to functionalities, which were to be imported from PATIKA*pro* or to be implemented specifically for the web service . During the development process, we continued to the development of PATIKA*pro* in parallel. Therefore, we did the integration of latest improvements in PATIKA*pro* to PATIKA*web* periodically.

We also managed the complex pathway visualization design and implementation. We have discussed and designed the visualization scheme and implemented the design for PATIKA*pro*. Later, the logic is imported to PATIKA*web*, with some minor modifications.

[1] http://www.graphviz.org/
[2] http://csbi.sourceforge.net/
[3] http://gvf.sourceforge.net/

4.2 Future Work

Due to the incompatibility of the browsers in handling some scripts, Patika*web* is currently stable on IE 5.5 or higher. With some minor bugs related to the DHTML's, users can also use Patika*web* with full functionality in Mozilla Firefox 1.0 or more. For both IE and Safari working on Mac OS, Patika*web* does not work properly either for the time being. The problems related to the operating system and the browsers must be considered to Patika*web* stable and consistent with all the browsers mentioned above. To be able to handle the inconsistency in browsers, failing functions must be implemented with browser dependent scripts. Although this is not a desirable solution for software development, inconsistency among the frequently used browsers force this approach.

Another problem, that we need to solve prior to the final release is related to the command structure. Since we have two different views, affecting each other with the editing operations, we want to supply a very strong do/undo mechanism. Currently we fail in performing do/undo operations in some specific scenarios. The design and implementation should be revisited to solve these problems and make the mechanism stable.

As the currently accessible version of Patika*web* is a beta release, for the version 1.0 release we aim to add some new functionalities like microarray support. Considering the modular architecture, any improvements and additional functionalities should be easy to integrate. One of the future improvements currently considered is adding a microarray analysis module to Patika*web*. The multi-tier architecture of Patika*web*, allows also any major changes and improvements, like the changes at the graph editing framework. In the future, we may build our own layout server with another third party graph editing toolkit and embed this structure within Patika*web*.

Bibliography

[1] 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*, 20:349–356, 2004.

[2] E. Demir, O. Babur, U. Dogrusoz, A. Gursoy, G. Nisanci, R. Cetin-Atalay, and M. Ozturk. Patika: an integrated visual environment for collaborative construction and analysis of cellular pathways. *Bioinformatics*, 18:996–1003, 2002.

[3] D. Endy and R. Brent. Modelling cellular behaviour. *Nature*, 18:391–396, 2001.

[4] J. Fulton and E. Kramer. Can you ever be too thin? *netWorker*, 1:19–23, 1997.

[5] Z. Hu, J. Mellor, J. Wu, and C. DeLisi. Visant: an online visualization and analysis tool for biological interaction data. *BMC Bioinformatics*, 5:17–25, 2004.

[6] M. Jern. "'thin"' vs. "'fat"' visualization clients. In *Proceedings of the working conference on Advanced visual interfaces*. ACM Press, 1998.

[7] M. Kanehisa and P. Bork. Bioinformatics in the post-sequence era. *Nature Genetics*, 33:305–310, 2003.

[8] P. D. Karp. An ontology for biological function based on molecular interactions. *Bioinformatics ONTOLOGY*, 16:269–285, 2000.

[9] P. D. Karp, M. Riley, M. Saier, I. T. Paulsen, S. M. Paley, and A. Pellegrini-Toole. The ecocyc and metacyc databases. *Nucleic Acids Research*, 28:56–59, 2000.

[10] L. Krishnamurthy, J. Nadeau, G. Ozsoyoglu, M. Ozsoyoglu, G. Schaeffer, M. Tasan, and W. Xu. Pathways database system: An integrated set of tools for biological pathways. *Bioinformatics*, 19:930–937, 2003.

[11] A. Lai and J. Nieh. Limits of wide-area thin-client computing. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM Press, 2002.

[12] C. A. Ouzounis and A. Valencia. Early bioinformatics: the birth of a disciplinea personal view. *Bioinformatics, Review*, 19:2176–2190, 2003.

[13] G. Schaeffer and Z. Ozsoyoglu. Pathwayviz visualizing biological pathways through an interactive graph. Technical report, CWRU, 2002.

[14] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome Research*, 13:2498–2504, 2003.

[15] Tom Sawyer Software Corporation. *Tom Sawyer Software Visualization Version 6.0 JSP Edition Developer's Guide*.

[16] Tom Sawyer Software Corporation. *Tom Sawyer Software Visualization Version 7.0 JSP Edition Developer's Guide*.

[17] G. Vernikos, C. Gkogkas, V. Promponas, and S. Hamodrakas. Genevito: Visualizing gene-product functional and structural features in genomic datasets. *BMC Bioinformatics*, 4:53–68, 2003.

Appendix A

Screenshots from PATIKA*web*

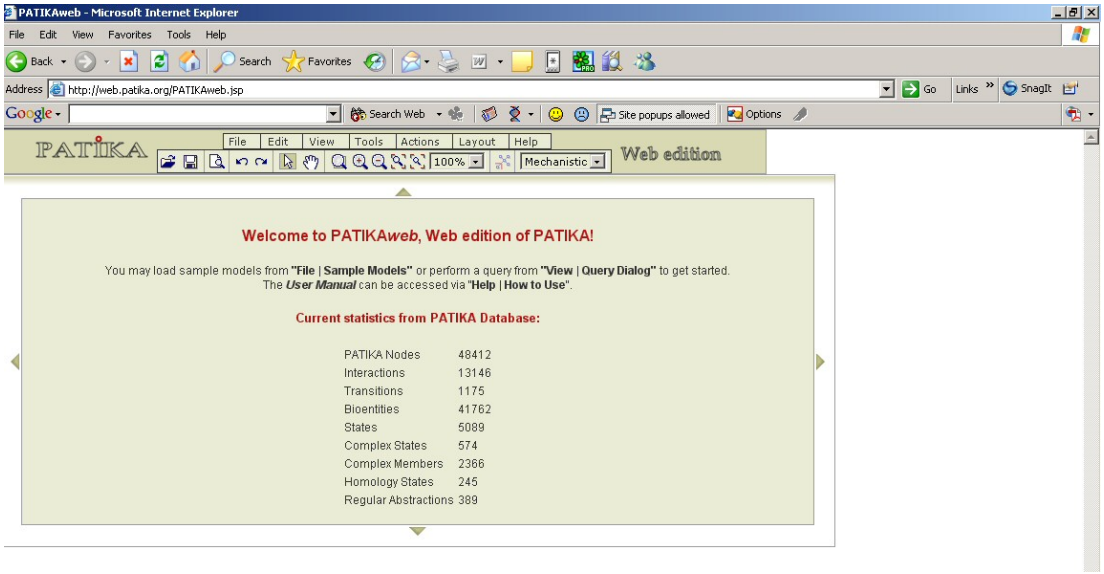

Figure A.1: Welcome page of PATIKA*web*

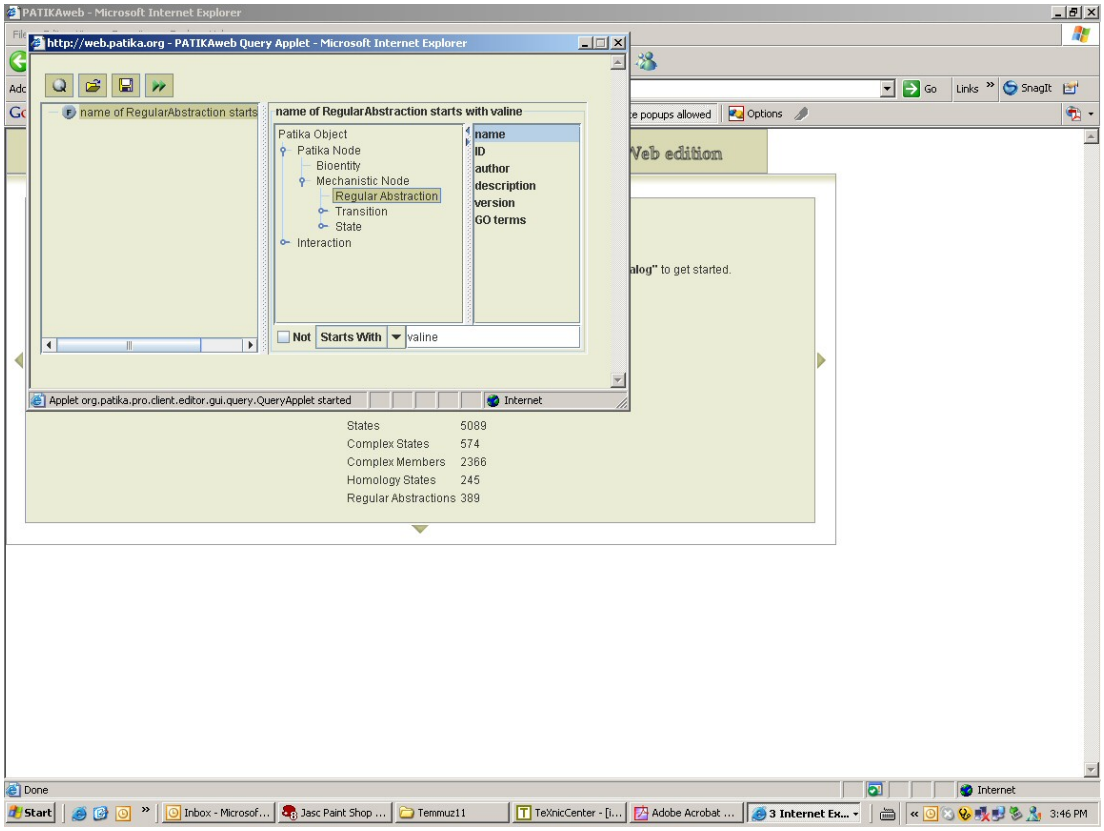

Figure A.2: Query applet

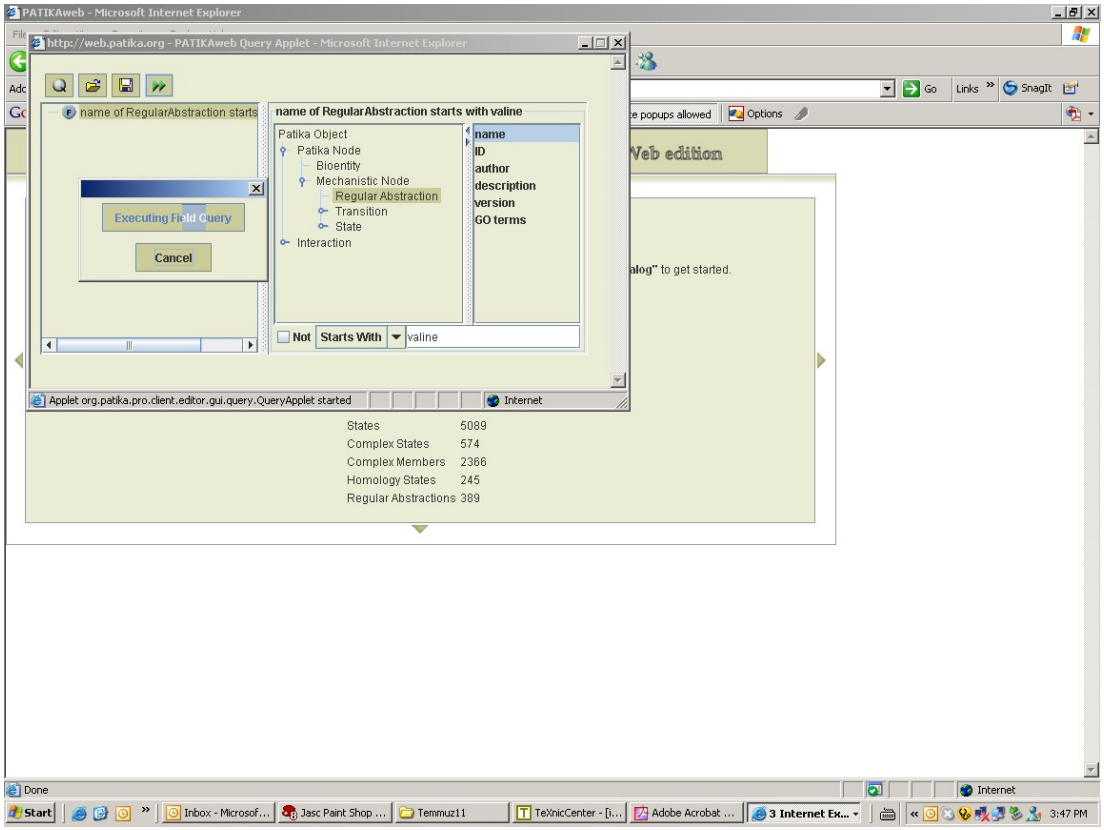

Figure A.3: Query is in progress

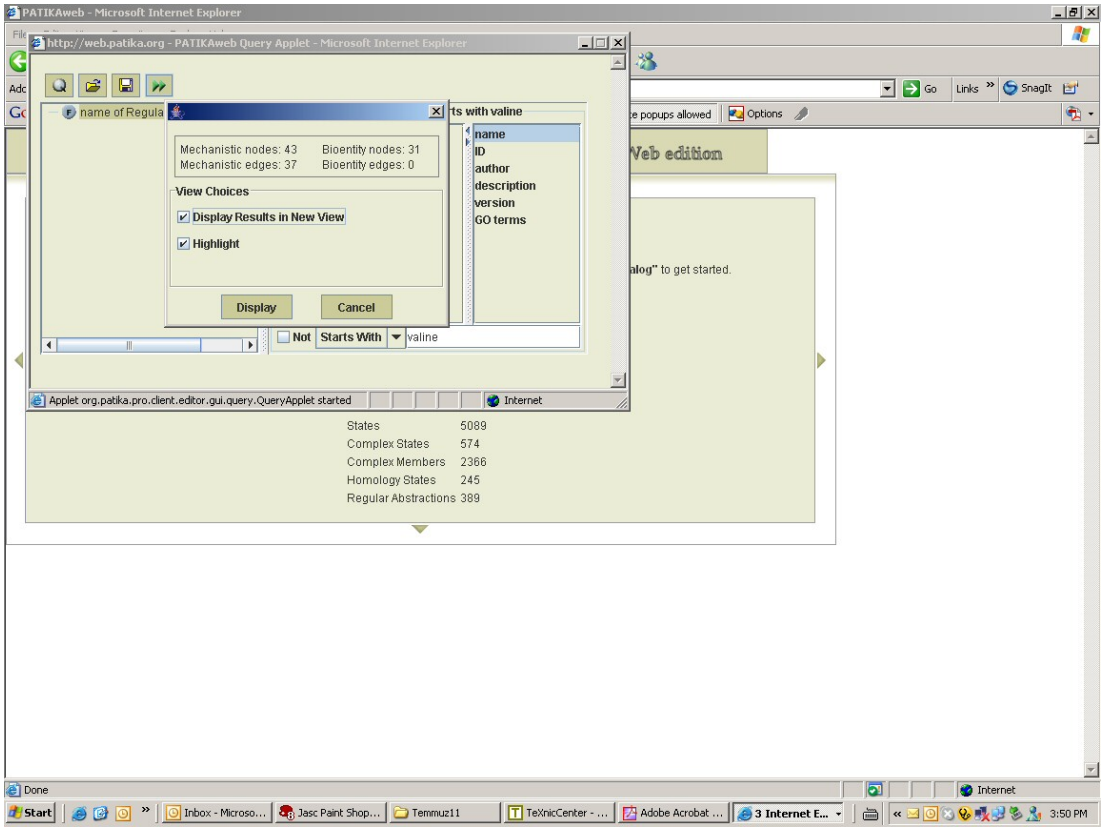

Figure A.4: Result report for the submitted query

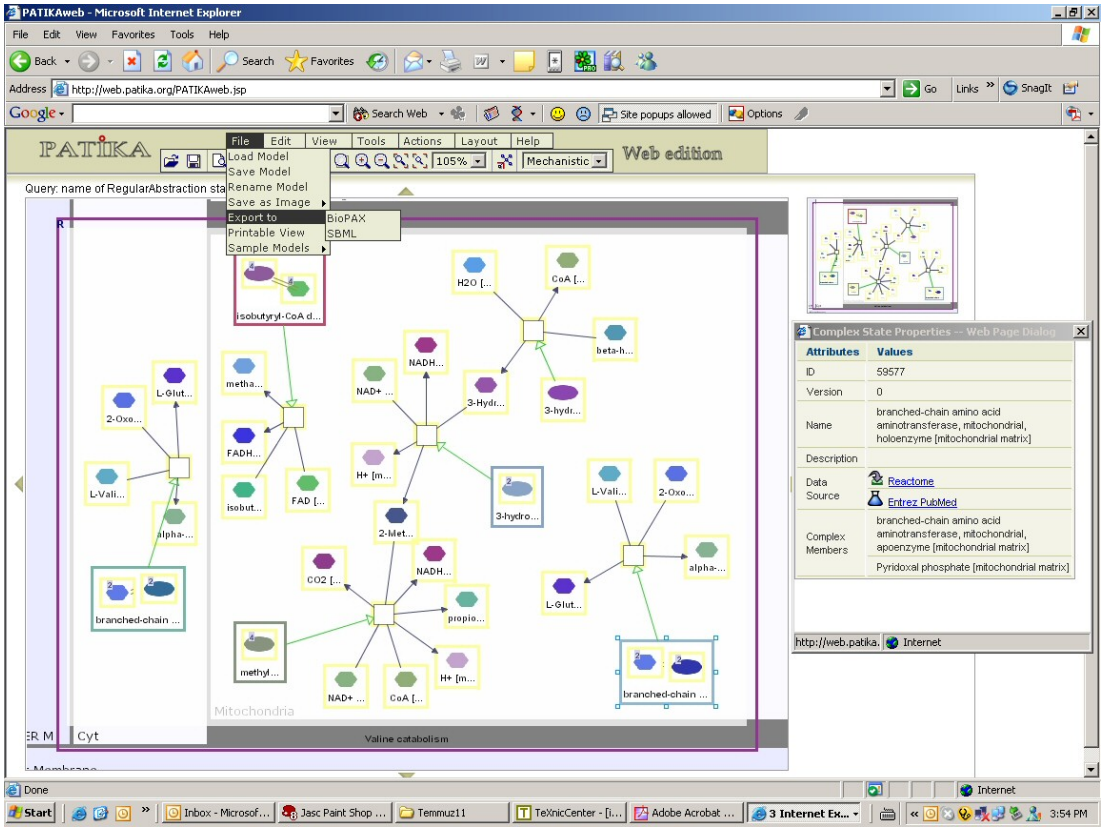

Figure A.5: Result of the query is visualized with the inspector window open for pathway object