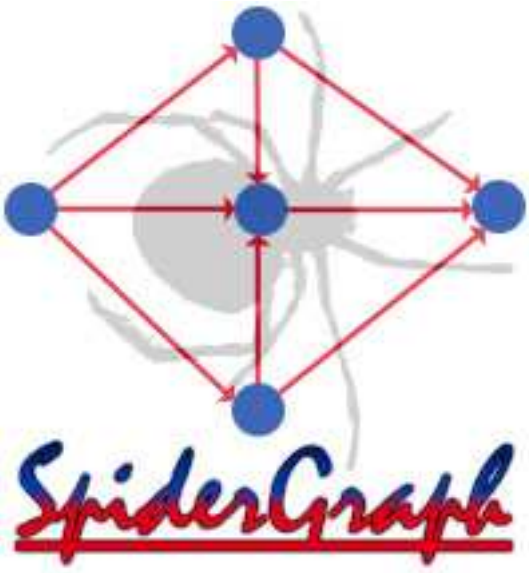
SpiderGraph
System Design Revised

Marc Cohen
Patrick DeMoss
Yevgen Voronenko
Frederick Walsh
Leland Weeks

2nd June 2003



Available online at http://www.mcs.drexel.edu/~uyvorone/cs452/

Contents

1 Introduction **6**

1.1 Abstract 6

1.2 Purpose 6

1.3 Document Scope 6

1.4 Design Goals 7

1.5 Overview of the Remainder of the Document 7

2 General Architecture **8**

2.1 Implementation Plan 8

2.2 Environment Components 9

2.3 Architecture 9

3 Graph Display **11**

3.1 XUL Design 11

3.2 graphDisplay file structure 11

3.2.1 Content 11

3.2.2 Skin 12

3.2.3 Locale 12

3.3 contents.rdf 12

3.4 graph.js 13

3.4.1 linksToURL(string url) 13

3.4.2 pruneURL(string url) 13

3.4.3 display() 13

3.5 navigatorOverlay.js 13

3.6 graph.png 13

3.7 navigatorOverlay.xul 14

3.8 results.html . . . 14
3.9 spidergraphOverlay.xul . . . 14
3.10 navigator.css . . . 14
3.11 spidergraph.xul . . . 14
3.11.1 MenuBar . . . 14
3.11.2 Buttons . . . 15
3.11.3 Display Area . . . 15
3.12 spidergraph.js . . . 17
3.12.1 close() . . . 17
3.12.2 reload() . . . 17
3.12.3 prune(nodeName) . . . 17
3.12.4 drawGraph() . . . 18
3.13 spidergraph.gif . . . 18
3.14 Summary . . . 18

4 Data Processing **19**
4.1 Overview . . . 19
4.2 Events . . . 19
4.2.1 Overview . . . 19
4.2.2 Preferences change . . . 19
4.2.3 Browser input . . . 20
4.2.4 Graph display interaction . . . 20
4.3 Classes . . . 21
4.3.1 Edge . . . 21
4.3.2 Node . . . 23
4.3.3 Graph . . . 25

5 Browser UI **28**

5.1 Overview 28

5.2 SpiderGraph Button 28

5.2.1 Visual 28

5.2.2 Implementation 29

6 Delivery and Installation **31**

6.1 Delivery Strategy 31

6.2 XPInstall 31

6.3 Considerations for Machine Dependent Code 31

7 GraphViz / DOT **33**

7.1 Description 33

7.2 Sample DOT Input 33

7.3 DOT Attributes 36

8 Mozilla **39**

8.1 Overview 39

8.2 Mozilla in Relation to SpiderGraph 39

A Index **40**

B Definitions, Acronyms and Abbreviations **43**

C Section Authors **44**

D Bibliography **45**

List of Figures

1 Implementation plan timeline 8
2 Top level data-flow diagram 10
3 graphDisplay file structure 12
4 Sample GraphDisplay window 16
5 graphDisplay file contents and dependencies 18
6 All the present Modern navigation buttons 28
7 MOZILLA browser with the close-up of SpiderGraph button 29
8 Current navigation interface 30
9 XPInstall dialog 31
10 Sample DOT input 34
11 Sample DOT output 35
12 Controllable node attributes in DOT 36
13 Controllable edge attributes in DOT 37
14 Controllable graph/subgraph attributes in DOT 38

1 Introduction

1.1 Abstract

The most commonly used navigation tools in modern web browsers are the *Forward* and *Back* navigation buttons. However, in many cases the default implementations of these buttons are unsatisfactory for the user.

SpiderGraph is intended to improve the standard navigation buttons by storing user click history in a graph data structure, and then displaying the history graph on demand to allow easy navigation to *all* previously visited pages. It will also offer the user the option of decorating the graph with other information, helping the user to quickly locate the desired location.

The SpiderGraph design proposed in this document will provide a specification for software implementation.

1.2 Purpose

SpiderGraph Requirements Specification [5] described the core functionality of the SpiderGraph web browser extension.

This document is intended to delineate system architecture and design of the SpiderGraph system web browser extension. It illustrates structural and functional design issues; describes system dependencies, collaboration and composition relations, and inheritance hierarchy.

It will serve as a basis for implementation of the first SpiderGraph release - **SG-1**.

1.3 Document Scope

This document provides a complete design of the SpiderGraph package, and specifies how the functional and non-functional requirements set in [5] will be satisfied.

Complete software architecture is presented, and subpackages, classes, and functions are described. Also user-interface prototypes are given.

Note, this document does *not* describe a working system, and thus might contain certain assumptions, which might be proven to be wrong in the implementation phase.

1.4 Design Goals

The ultimate goal of this design document is to build an interactive history graph extension for the MOZILLA web browser, that uses the existing graphing tool, but improves its graph layout capabilities by using domain specific information about web pages.

Target design shall possess the following characteristics:

- **Portability** - in this context, possibility of porting SpiderGraph to other web browsers
- **Extensibility** - ease of adding new graph transformations and improving rendering
- **Performance** - acceptable user interface response time
- **Ease of use** - intuitive user interface that fits well with the rest of the browser
- **Subsettability -** possibility of reusing subsets of SpiderGraph components

Perhaps the most useful subset of SpiderGraph would be the graph production and data transformation parts, which can be reused for the creation of other types of related graphs - such as web server statistics or directory and filesystem hierarchies.

1.5 Overview of the Remainder of the Document

The rest of this documents presents the overall SpiderGraph architecture, implementation and testing plan, and detailed component designs.

Sections 7 and 8 can be consulted for information regarding GRAPHVIZ and MOZILLA respectively.

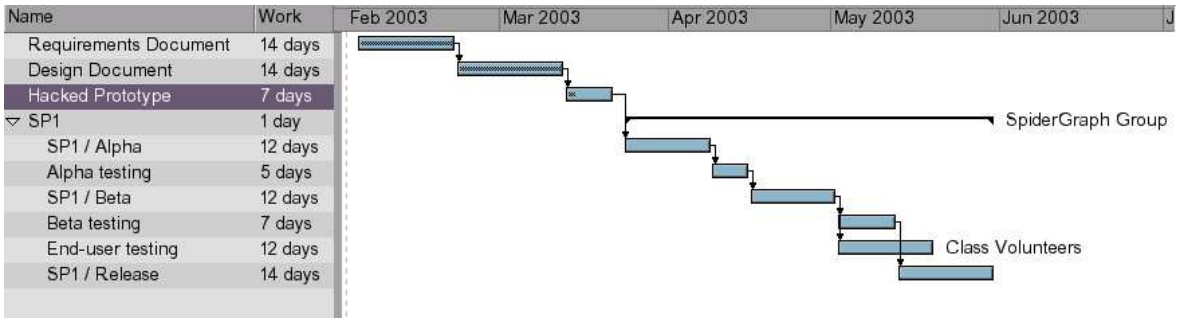


Figure 1: Implementation plan timeline

2 General Architecture

2.1 Implementation Plan

First release of SpiderGraph, known as **SP1,** will be completed by the end of May. The main development will start as soon as the first proof-of-concept prototype will be *"hacked"* together. It will proceed through four milestones:

- *Alpha* - all functional requirements must be satisfied
- *Beta* - all non-functional requirements must be satisfied, and most of the bugs revealed in alpha testing must be fixed
- *Initial Release* - all bugs revealed in beta and end-user testing must be fixed and initial requirements met.
- *Future Release* - Implementation of futre requirements listed in SpiderGraph's requirements document.

Figure 1 shows absolute deadlines for SpiderGraph milestones. All of the shown deadlines represent the latest possible date the subtask must be finished by.

Alpha and beta milestones will be followed by rigorous testing. The aim of alpha and beta testing is to reveal possible problems in the implementation. Alpha and beta testing will be performed by SpiderGraph team.

End-user testing is aimed at finding possible problems with the user interface, such as inconsistencies in naming or GUI layout. To find volunteers[1] for end-user testing, an

[1]It is not possible for the SpiderGraph team to perform this step, since we designed the UI, we would conclude it is the best.

announcement will be made in class, and also on the project web site.

2.2 Environment Components

SpiderGraph is not a stand-alone application. It is a browser extension and it coexists in a software environment consisting of the following meta-components:

- **Mozilla** (chrome/navigator) - web browser part of Mozilla
- **GraphViz** (dot) - dag layout module that produces graph images
- **SpiderGraph** - our package

SpiderGraph will be implemented as an add-on for the Mozilla web browser. It will add a new button to the navigator and also have its own window. Final stages of graph production will utilize dot, a graph layout and rendering engine, which is part of GraphViz package.

2.3 Architecture

Internally SpiderGraph will be organized into 4 subpackages:

- **Data Processing** - stores, manipulates, and exports graphs
- **Preferences** - loads and stores user modifiable parameters for all subpackages
- **User interface (UI)**
  - **Navigator UI** - extends Mozilla/navigator with SpiderGraph button, and adds data collection hooks.
  - **Preferences UI** - provides interface to the **Preferences** component

Diagram 2 shows how these components interact with each other.

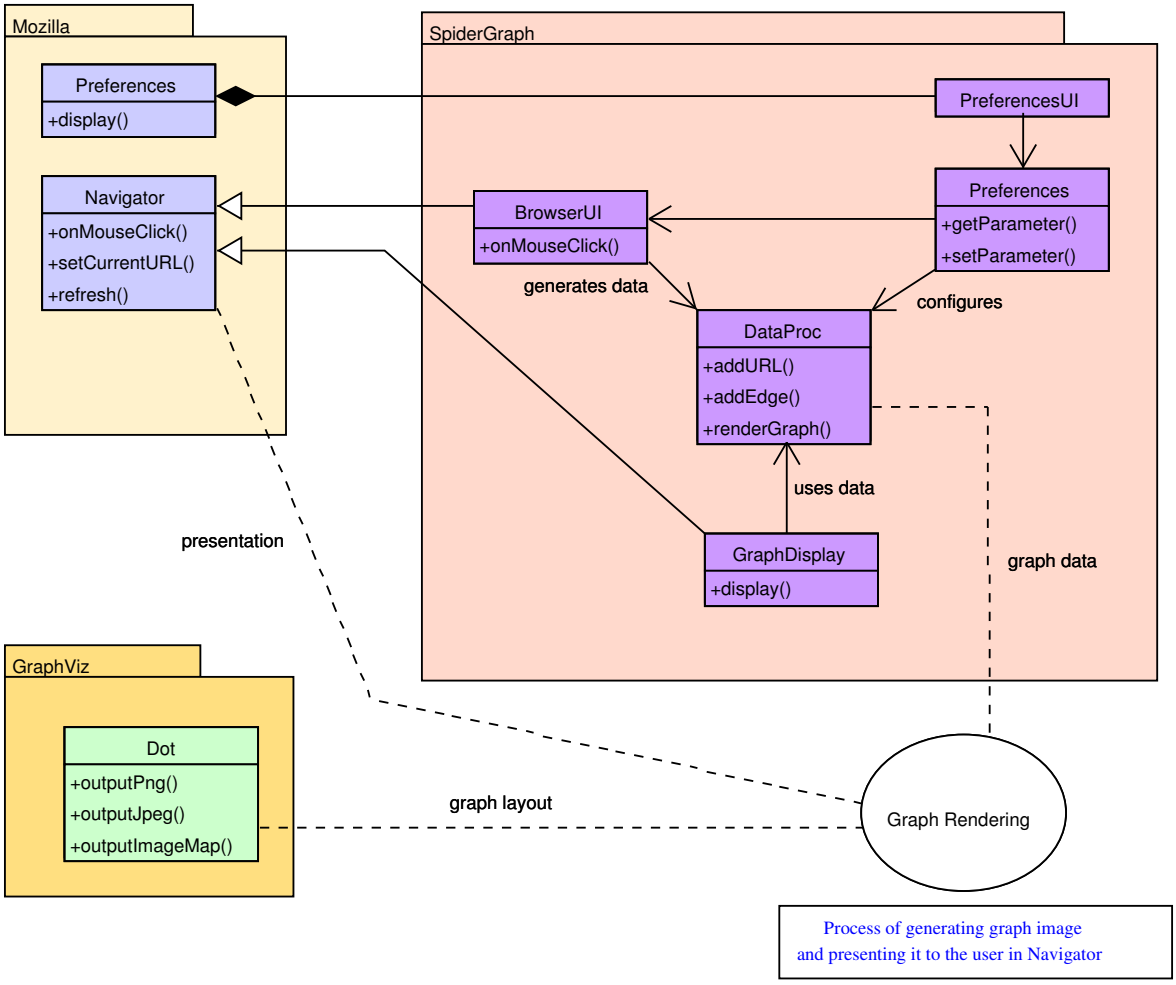


Figure 2: Top level data-flow diagram

3 Graph Display

3.1 XUL Design

The graphs generated by SpiderGraph will be displayed in a standalone window within the MOZILLA application. However, because the current MOZILLA web browser does not provide a suitable window for this purpose, it will be necessary to incorporate a new type of window into the web browser.

In order to maintain consistency within the design of the MOZILLA web browser, the layout and event handling for the new window will be written in XUL format. As such, the design of the window that SpiderGraph will use to display its graphs will adhere to standard XUL guidelines. More specifically, the design of the window will be broken down into the three standard XUL components, content, skin, and locale.

3.2 graphDisplay file structure

Together, the XUL design of the window will be incorporated into the MOZILLA web browser as a package named graphDisplay. That is, it will be placed inside the chrome directory located inside of the MOZILLA folder on the machine in which MOZILLA is installed. The overall file structure of the window design and incorporation into the MOZILLA directory structure is shown in the Figure 3.

3.2.1 Content

As is customary in XUL design, the content portion of the new window will contain information that is relevant to the layout and actions of the window. For example, the placement, size and shape of any buttons, in addition to their actions, will be housed in this portion of the window design. There will be two files that are created in order to do this. The first will be the XUL file that contains the declaration and layout of all objects within the window. This file will be written in XUL and named graphDisplay.xul. The second file will be named graphDisplay.js and will be written in JavaScript. It will contain the code that is executed when the user performs an appropriate action within the graph display window (ie clicks on a button).

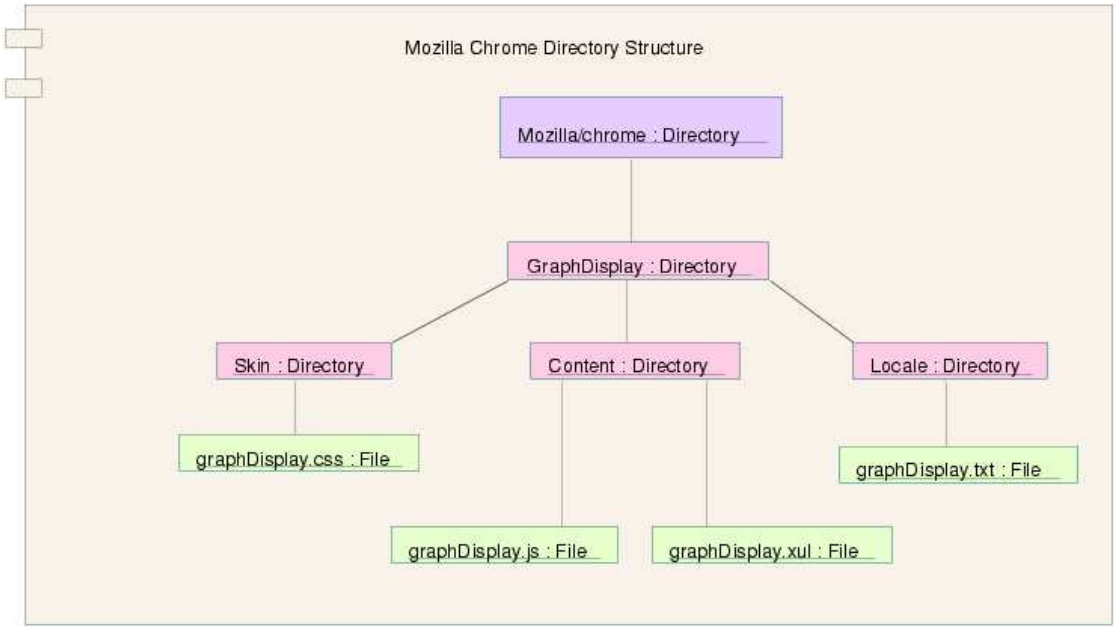


Figure 3: graphDisplay file structure

3.2.2 Skin

The skin portion of the window will contain all of the material responsible for the appearance of the window. More specifically, this portion of the window design will include any cascading style sheets and images, other than the graph, that are utilized by the window. The style sheet information will be stored in a file called graphDisplay.css.

3.2.3 Locale

Finally, the locale portion of the code will contain any text that is to be displayed outside of objects such as buttons or windows. More specifically, anytime text that is unrelated to the buttons or images is to be inserted in the window, it will be stored in a separate file and housed in the locale directory of the graph display window. This file will be named graphDisplay.txt.

3.3 contents.rdf

Initialization file allows the Mozilla browser to recognize SpiderGraph as a plug-in. Information tells Mozila about all of the interworkings and filetypes found in the Mozilla

package.

3.4 graph.js

Internal data structure which holds all properties of nodes, edges, and graph connectivity for the browsing graph displayed. There are many methods defined for this data structure, however, the most important are defined below.

3.4.1 linksToURL(string url)

Adds new URLs into the current graph.

3.4.2 pruneURL(string url)

Remove node with corresponding url from the graph data structure.

3.4.3 display()

Creates corresponding dot executable code from current graph data structure. In essense, it creates a graph that is visible.

3.5 navigatorOverlay.js

Catches user-caused browsing events which signals to graph.js to add the current URL to the current graph. navigatorOverlay.js also opens the SpiderGraph display window when the SpiderGraph button is clicked in the Mozilla interface. Additinally, code is supplied to parse URLs, curtailing their length, when they are displayed inside nodes.

3.6 graph.png

Image file generated by dot which represents the current browsing graph and is used by results.html.

3.7 navigatorOverlay.xul

Grabs spidergraph.gif and integrates image into a button onto the predefined space of the Mozilla browsing GUI. Essentially this creates the SpiderGraph button.

3.8 results.html

File generated by graph.js which contains the current browsing history graph. This html file is displayed inside the SpiderGraph window.

3.9 spidergraphOverlay.xul

Modifies Mozilla menubar to allow space for the SpiderGraph button.

3.10 navigator.css

Tells Mozilla how to allot space into the Mozilla menubar.

3.11 spidergraph.xul

This file will contain all of the layout information necessary to generate the window the users browsing history graph will be presented in. It will consist of a menubar, numerous buttons, and a graph display area.

3.11.1 MenuBar

The menubar will consist of only one entity, a File menu. However, there will be a total of four options under the File menu.

- **Close** - when selected, this option will close the window. Its event handler will be the function close() specified in the file graphDisplay.js

Future versions of SpiderGraph will allow the following features:

- **Open Graph** - this option will allow the user to select amongst previously generated graphs and have the desired one displayed in the display area of the window. Its event handler will be the function openGraph() specified in the file graphDisplay.js.
- **Save** - this option, when selected, will allow the user to save the graph that is currently on the screen within the display area. Its event handler will be the function saveGraph() specified in the file graphDisplay.js.
- **Save as** - this option allows the user to specify a name for the graph they wish to save. Its event handler will be the function saveAsGraph() specified in the file graphDisplay.js

3.11.2 Buttons

The window in which the users browsing history graph is displayed in will contain a total of three different buttons. They will reside in horizontal fashion along the top of the window just below the menubar. The three buttons will be named as follows:

- **Prune Graph** - When this radio button is selected as the current mode of the SpiderGraph window, nodes which are clicked upon by the user will be removed from the graph.
- **Browsing** - When this radio button is selected as the current mode of the SpiderGraph window, the links which are represented by the nodes will be brought up is a regular Mozilla browser window.
- **DrawGraph** - when pressed this button updates the graph.

3.11.3 Display Area

This portion of the window will contain the space in which the graph is actually presented. It will be presented in the middle of the window and be specified as an iframe object within spiderGraph.xul. Note also that the html file which will occupy the space will be results.html.

Figure 4 shows a sample display window.

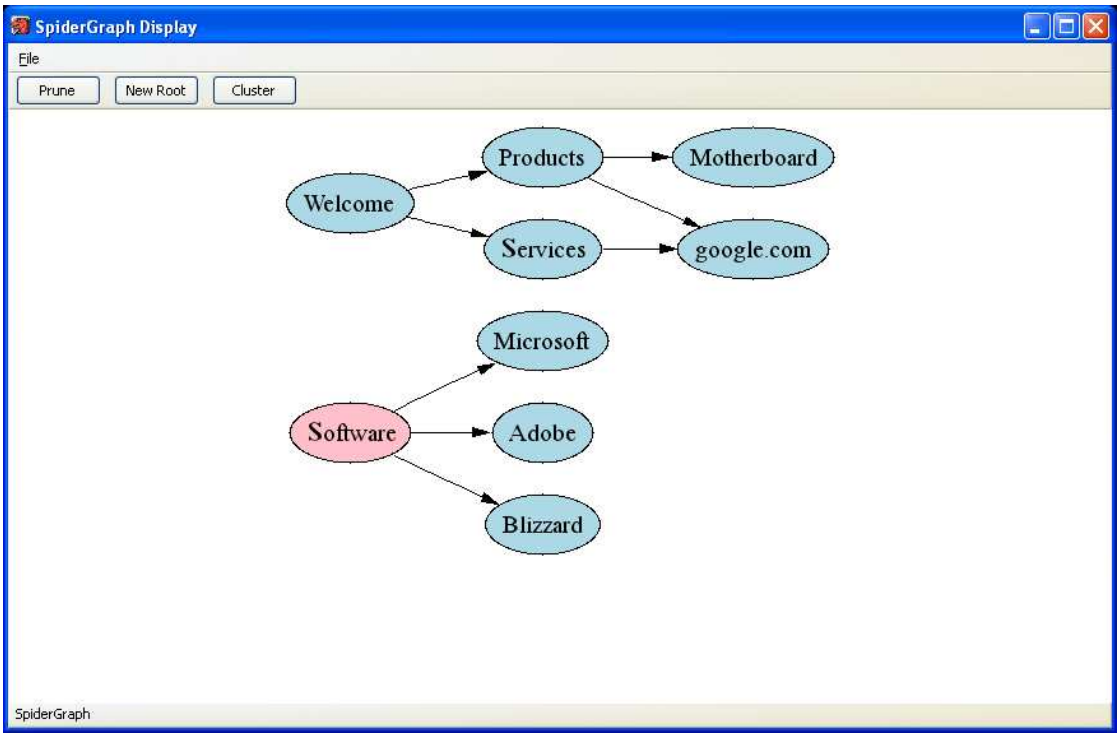


Figure 4: Sample GraphDisplay window

3.12 spidergraph.js

This file will contain all of the code that is executed when a menu item, button, or node within the active graph is selected. It will also contain the functions that allow communication between the display window and the processing component of SpiderGraph. All of the functions will be written in JavaScript.

Note: There is no function which tells the browser to point to a particular website when a node in the graph is clicked on because the graph is represented as an imagemap. As such, the data contained within the imagemap handles this aspect of functionality.

The list of major functions in addition to their inputs and outputs are presented below.

3.12.1 close()

- **Input**: None
- **Processing**: The SpiderGraph window is closed.
- **Output**: None

3.12.2 reload()

- **Input**: None
- **Processing**: The algorithm update the graph to display the most recent graph.
- **Output**: The graph is redrawn on the page.

3.12.3 prune(nodeName)

- **Input**: Name of node to remove from the graph.
- **Processing**: The graph is redrawn without the node which it it given as input. In addition, the node after the selected node, if one exists, becomes a new root.
- **Output**: None

3.12.4 drawGraph()

- **Input**: None
- **Processing**: This function goes ahead and calls the function disply(), from graph.js, which goes ahead and creates the png representing the browsing history of the user.
- **Output**: None

3.13 spidergraph.gif

The SpiderGraph image for the SpiderGraph button.

3.14 Summary

The following diagram summarizes the files and their associated content:

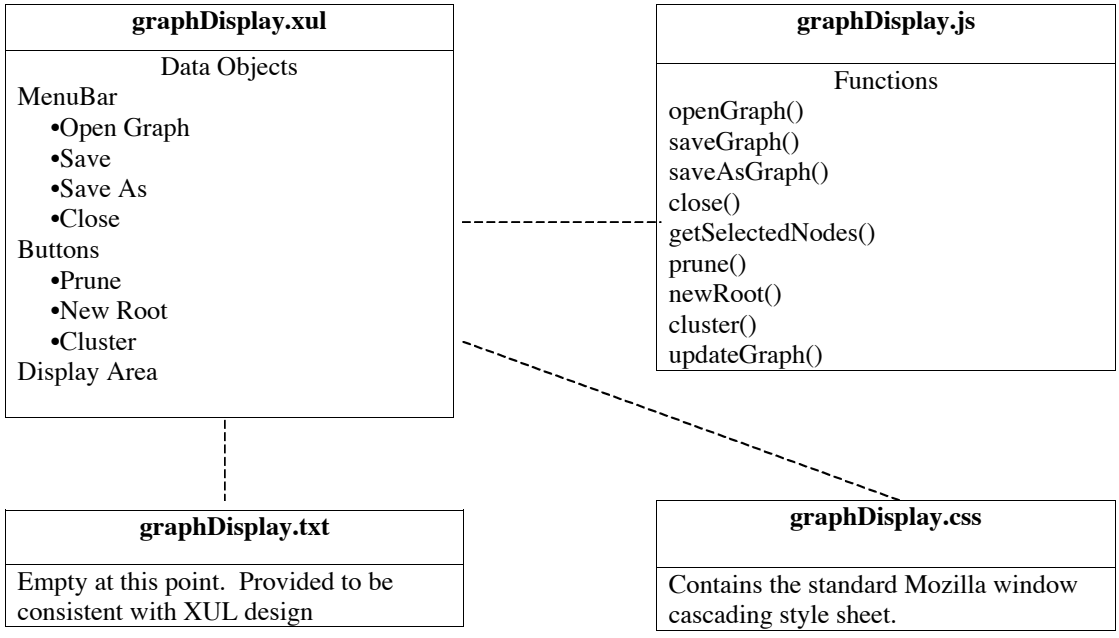


Figure 5: graphDisplay file contents and dependencies

4 Data Processing

4.1 Overview

The graph data will be held in a single data structure so as to facilitate manipulation and archival operations. All data needed to generate the browsing history graph is stored here, including node and edge properties.

The data structure will be implemented in JavaScript following an object-oriented design. Several objects comprise the primary Graph object:

- **Edge** - a representation of an edge in the graph.
- **Node** - a representation of a node in the graph.
- **Graph** - the entire graph, which is made up of one or more connected components.

4.2 Events

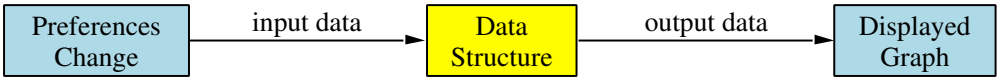
4.2.1 Overview

The data structure's major functionality exists to handle three specific events:

1. Preference change.
2. Browser input.
3. Display interaction (includes button activation and interaction with the graph).

4.2.2 Preferences change

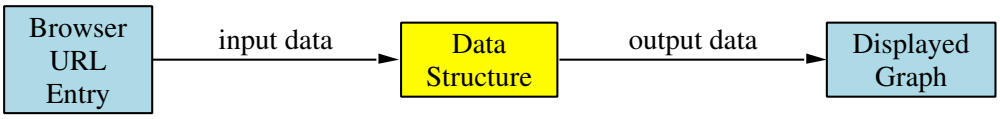
This event occurs when the user makes any changes to the set preferences held in the MOZILLA SpiderGraph menu option. Initially these preferences will be set to default values, but as the user makes changes to these values, the graph must update itself accordingly as soon as the changes are confirmed. The preference information is held in the UI component of the MOZILLA browser and is implemented in JavaScript.



The PrefsListener object will serve as an event listener, and pass all preference parameters to the Graph data structure once the user has made a change. The data structure will update itself according to the new user choices, and then pass this information on to the Graphviz utility for the new graph to be generated.

4.2.3 Browser input

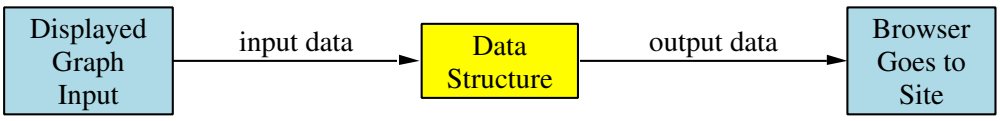
Any input from the browser which changes the current URL will trigger this event. Such user actions as a bookmark selection or a manual URL entry qualify as a Browser Input event. Data will be passed to the Graph data structure using the BrowserListener object, which retrieves the new URL from the MOZILLA browser.



The new URL will be recorded in the Graph data structure appropriately, and then the new data will be passed to the Graphviz utility so that the displayed graph can be updated.

4.2.4 Graph display interaction

This event occurs when the user interacts with any part of the graph display, such as activating nodes or buttons. The DisplayListener object listens to the display module for any user interaction, and when it is triggered it passes the information to the Graph data structure.



If the graph is altered, the data structure must update itself to correspond to the changes in the displayed graph. If a new URL is requested by the user using the displayed graph, then the Graph data structure must update itself appropriately, by incrementing that site's visit count for example, and then finally passing that new URL to the browser to visit.

4.3 Classes

4.3.1 Edge

Identification	EDGE
Purpose	The edge object represents a single edge in the graph, or a link from one website to another. It will hold a single pointer to a Node object to represent the node toward which it is pointing (not from). Also, all user preference information is stored in the object, such as arrow shape and thickness. Site tracking data such as link count (weight) and a variable to remember if the target site is special (Flash/Cookies/Java), are stored here as well.
Function	This object realizes the following functional requirements: • **3.3.1:** Directionality • **3.3.3:** Edge Style • **3.3.5:** Deriving Style for weighted edges
Dependencies	NODE Aside from expected edge attributes, this object carries a pointer to the Node object. This pointer represents a node which is pointed to by this edge.
Methods	Standard functions Edge exists as one of two objectes which complete what is essentially an adjacency list representation of a graph. Aside from the usual inspector and modifier methods, there are no special functions to consider.

Data	Internal data of this object will consist of a list of nodes, as well as various attributes which are used optionally by the user to give information about the edge. • *LinkedList <Node>* • *directionality* • *thickness* • *style* • *color* • *caption* • *arrow shape* • *visit count* • *weight* • *visible*

4.3.2 Node

Identification	NODE
Purpose	The Node object represents a single node in the graph, or a single website. It will hold a list of Edges to represent all edges pointing from this node and toward another, as well as all user preference parameters such as color and shape. Various site tracking data is also stored here, such as the visit count, site length, and URL.
Function	This object realizes the following functional requirements: • **3.2.1:** Node Labels • **3.2.2:** Node Style • **3.2.3:** Node Derived Style • **3.2.4:** Uniqueness • **4.2.3**: Inspecting nodes • **4.3**: Preferences
Dependencies	EDGE Aside from the expected node attributes used to define user preferences, this object holds a vector of Edge objectes. All edges which exist in this Node instance are graphically represented as edges leaving this node. The nodes to which these edges point are defined in the Edge object.
Methods	Standard functions The Node object complements the Edge object in the goal of representing a complete graph as an adjacency list. In addition to the corresponding inspector and modifier methods, each Edge from the vector of Edges is accessible using LinkedList methods.

Data	Internal data of this object is made up of a list of Edges, as well as basic optional node properties definable by the user: • *LinkedList <Edge>* • *shape* • *fill color* • *border color* • *label color* • *visit count* • *image count* • *length* • *MIMEType* • *URL*

4.3.3 Graph

Identification	GRAPH
Purpose	This is the primary object which holds all graph information. All individual connected components (each with a single root) are stored here as a single graph. Data in this object would be passed to the Graphviz utility for the generation of the actual graph display.

Function	This object enables events 1, 2, and 3 as defined in this section: Preference Change, Browser Input, and Display Interaction. This object realizes the following functional requirements: • **3.4.3**: Multiple roots • **3.4.4**: Automatically getting new roots • **3.4.5**: Manually creating new roots • **4.2.6**: Changing the Root • **4.3**: Preferences • **3.5.1**: Growth direction • **3.5.2**: New nodes • **3.5.3**: Stability • **3.6**: Pruning • **3.7.1**: Partial Equivalence Clustering • **3.7.2**: Grouping clustering • **3.7.3**: Reduction clustering • **4.2.3**: Inspecting nodes • **4.2.4**: Adding New Edges and Nodes • **4.2.5**: Inspecting the graph • **4.3**: Preferences
Methods	Retrieval and assignment to a specific node or edge within the graph is accomplished using appropriate methods from the LinkedList and internal methods from the Node or Edge objectes.

	updatePreferences() This method is used to tell the data structure that new preferences have been defined, and that the data needs to be updated with the new settings. *updateURL()* When a new URL is entered from the browser, this method is used to update the current URL to the one entered. The data structure will then generate a new graph and pass it to the display. *removeNode()* Nodes in the connected component are deleted entirely using this method. All edges leaving this deleted node, and all nodes and edges derived thereafter are also deleted. This method is meant to support manual as well as automatic deletion of nodes (pruning) in accordance with the Preference design. *addNode()* A node and corresponding edge will be added to connected component. This would usually be done automatically, as the user were to browse websites and add to the browse history. The growth of the graph happens with the use of this method.
Data	Internal data of this object is a vector of CNCTDCOMPONENT instances. • *LinkedList <CnctdComponent>*

5 Browser UI

5.1 Overview

MOZILLA's Graphical User Interface language is XUL, or Extensible User-Interface Language. It is a cross-platform language for describing user interfaces of applications and is modeled after XML, Extensible Modeling Language. Since MOZILLA is required to run on several platforms, the user-interface has several technical specifications and requirements.

MOZILLA's UI is divided into three layers: structure, style, and behavior. The structure layer identifies the widgets (menus, buttons, etc.) and their position in the UI relative to each other, the style layer defines how the widgets look (size, color, style, etc.) and their overall position (alignment), and the behavior layer specifies how the widgets behave and how users can use them to accomplish their goals.

5.2 SpiderGraph Button

5.2.1 Visual

A single button will be added to the current MOZILLA user-interface for the implementation for SpiderGraph. The spidergraph button is generalized to work with all themes.

Figure 6: All the present Modern navigation buttons

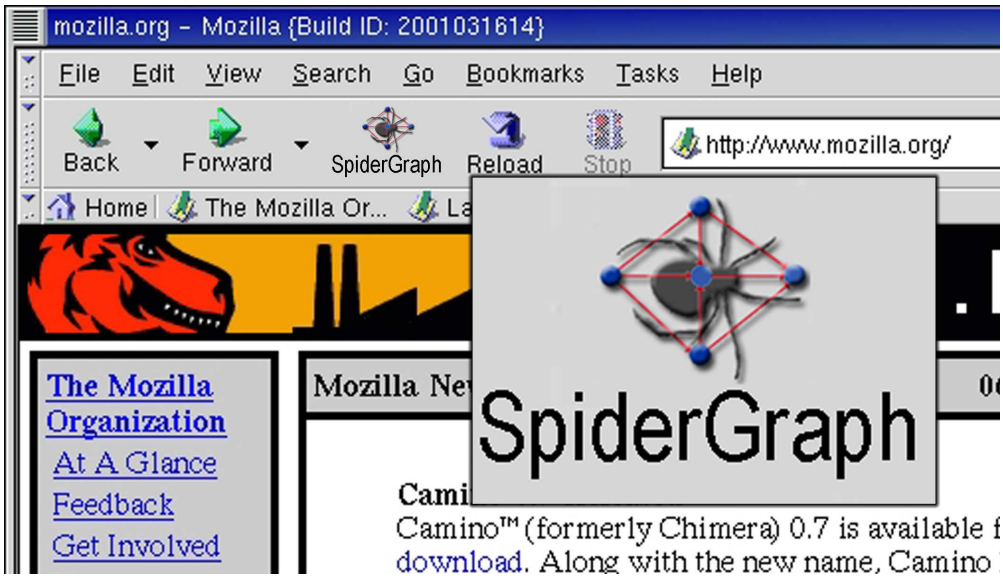


Figure 7: MOZILLA browser with the close-up of SpiderGraph button

A template for the new SpiderGraph button is shown in Figure 7. Note however, that the actual location of the button and its appearance will probaly be altered.

5.2.2 Implementation

Common practice in Mozilla is to use XUL to specify the graphical interface. This being the case, we will use XUL to add the button to the Mozilla toolbar. In order to mkae installation of SpiderGraph easier, we have decided to implement the button into the Mozilla web browser using an overlay. An overlay is simply a tool that allow one to add GUI components to existing GUI components without altering the code which specifies the original components. Event handlers for the button will be specified in JavaScript as is standard in XUL.

Specifically, a JavaScript function will be called each time the SpiderGraph button is clicked. The JavaScript will open the SpiderGraph window. Since the window is simply another MOZILLA browser window, the coding for the interaction with the button will be straightforward.

The main MOZILLA navigation interface, shown in Figure 8, is found in the *chrome* directory. The *chrome* directory is the main repository for the menu, navigation buttons,

and any customizable user interface components.

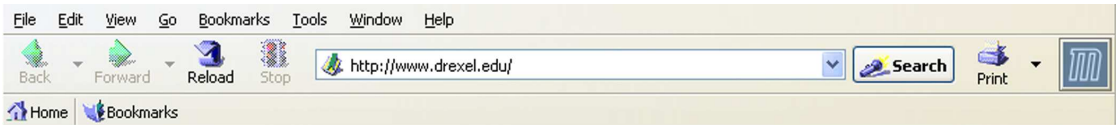


Figure 8: Current navigation interface

XUL makes use of tags to create user-interface elements. Here is a simple example of XUL code that will add a button to the MOZILLA web browser:

```
<button
    id="SpiderGraph"
    label="SpiderGraph"
    image="images/SGmodern.jpg"
    default="false"
    disabled="false"/>
```

It will be necessary to write interaction for the SpiderGraph button. The button may have an event handler with code that looks similar to the following:

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<window
    id="mozilla-window"
    title="Mozilla"
    orient="horizontal"
    xmlns:html="file://spidergraph.html"
    xmlns="file://spidergraph.html ">
```

Finally, by design, XUL has been created with install packages in mind. Therefore, our modifications will be able to be packaged into an installer, which the user may download and install.

The functionality and requirements of the button can be traced back to section 4.1, SpiderGraph Button, of the requirements document.

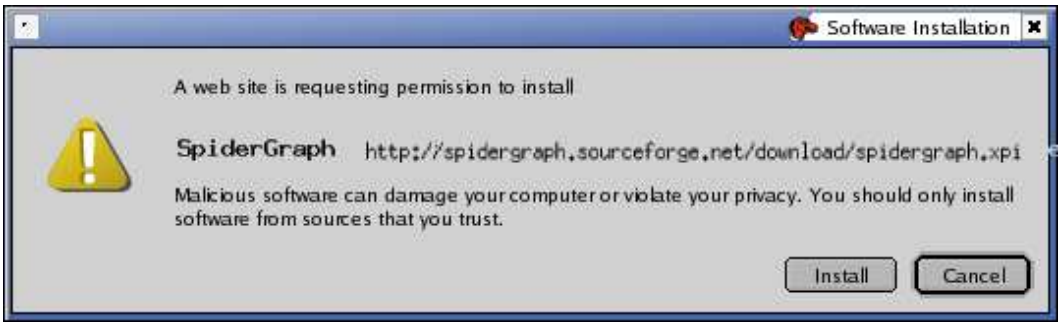


Figure 9: XPInstall dialog

6 Delivery and Installation

6.1 Delivery Strategy

SpiderGraph aims to be distributed on as many platforms as possible and have the easiest installation as possible. SpiderGraph will supply the following packages:

System	Package type	Package name
win32	xpi	spidergraph.xpi
linux	xpi	spidergraph.xpi

Since we are porting SpiderGraph to two architectures, the packaging process will be automated.

6.2 XPInstall

XPInstall, or "XPI", is a file format used by MOZILLA for plug-ins it shall be embraced as much as possible as it adds much needed functionality and usability. XPI is essentially a compressed archive that contains a small JavaScript program that registers modules within MOZILLA. Although it does have some shortcomings as it lacks the ability to install machine specific portions. Figure 9 shows a screenshot of XPInstall installer dialog.

6.3 Considerations for Machine Dependent Code

Certain portions of SpiderGraph must be machine specifically compiled, i.e. GraphView libraries. These portions will be kept in a directory of the users choice queried during

installation.

The install will be automated but it will contain two parts.

1. **Modifying current data:** This will consist of editing the "chrome.xml" so that SpiderGraph will be added to the user interface of MOZILLA, and other file the needs to by changed to allow for SpiderGraph. On the win32 this will be a simple batch script and on the Debian, Redhat, Unix and to a lesser extent Mac OS X it will be a simple shell script.

2. **Copying Data:** This consist of copying all of the xul, binaries and other data to the local hard disk so that it can be accessed.

7 GraphViz / DOT

7.1 Description

SpiderGraph relies heavily on dot, one of several components belonging to Graphiz, an open source graph-drawing utility program offered by AT&T. The overall process of SpiderGraph can best be described on a high level as follows. Initially, SpiderGraph extracts information from MOZILLA about a user's browsing history. The data processing unit than parses the information it receives from MOZILLA and organizes it in such a fashion as to allow dot to draw a graph representing the data. Dot than creates and returns an imagemap based upon the information that is spoon-fed to it. The resulting imagemap is than displayed back to the user via a specialized window in the MOZILLA web browser. As such, in order to fully understand the design of SpiderGraph, an elementary understanding of dot and some of its features is necessary.

Essentially what dot does is draw either directed or undirected graphs. As with all graphs, this requires that a set of vertices in addition to the edges between them be specified. In terms of SpiderGraph, all graphs will be directed. In addition, the vertices will represent web pages and the edges between them will represent the path taken by the user in order to view the pages.

7.2 Sample DOT Input

What follows is an example of a file that SpiderGraph may generate and pass along to dot. Its purpose is to introduce the syntax of the dot language.

NOTE: the double slash, //, represents a comment in the dot code and as such it is utilized here to explain what each line of code is doing in the file.

Although not all of the features available in dot and utilized by SpiderGraph have been utilized in this example, the general format and use of all features offered by dot and utilized by SpiderGraph have been demonstrated. For example, in the above code, if we wanted to add a URL to the node represented by "Software", the following statement would simply be inserted into the node declaration in the preceding line:

```
node [style = filled, fillcolor = "pink",
      color = "black", URL = "http://www.xyz.com"];
```

```
digraph G {   // begin graph declaration
   rankdir = LR;            //draw graph from left to right
   labelloc = "center";     //name of node will appear in the center

   //change the style node to pink with black border
   node [style = filled, fillcolor = "pink", color = "black"];

   //create node called Software
   Software

   //change the style node to pink with black border
   node [style = filled, fillcolor = "lightblue", color = "black"];

   Software -> Microsoft; // A->B mean an edge is drawn from A to B
   Software -> Adobe;
   Software -> Blizzard;

   edge[style = bold];  //make edge thicker
   Welcome -> Products; //note this starts a new root because no edges
                        //exist between the nodes below and above
   Welcome -> Services;
   Products -> Motherboard;
   Products -> "google.com";
   Services -> "google.com";
}
```

Figure 10: Sample DOT input

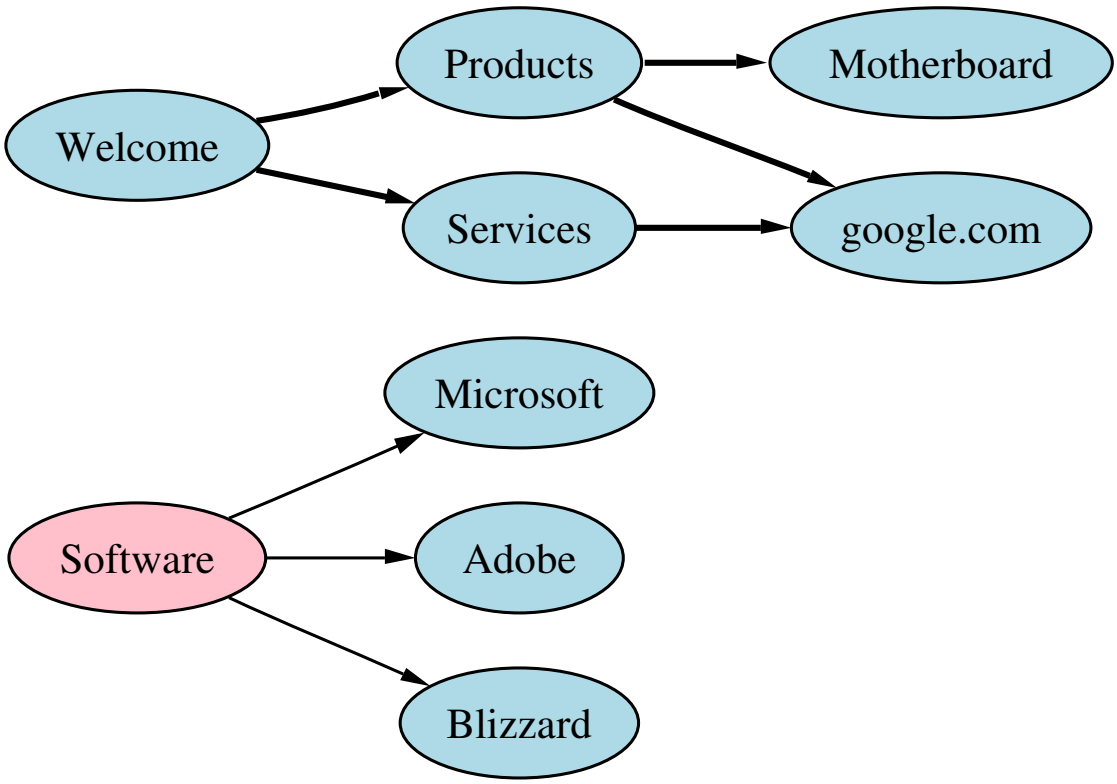


Figure 11: Sample DOT output

Name	Default	Values
bottomlabel		auxiliary label for nodes of shape M*
color	black	node shape color
comment		any string (format-dependent)
distortion	0.0	node distortion for shape=polygon
fillcolor	lightgrey/black	node fill color
fixedsize	false	label text has no affect on node size
fontcolor	black	type face color
fontname	Times-Roman	font family
fontsize	14	point size of label
group		name of node's group
height	.5	height in inches
label	node name	any string
layer	overlay range	all, *id* or *id:id*
orientation	0.0	node rotation angle
peripheries	shape-dependent	number of node boundaries
regular	false	force polygon to be regular
shape	ellipse	node shape; see Section 2.1 and Appendix E
shapefile		external EPSF or SVG custom shape file
sides	4	number of sides for shape=polygon
skew	0.0	skewing of node for shape=polygon
style		graphics options, e.g. bold, dotted, filled; cf. Section 2.3
toplabel		auxiliary label for nodes of shape M*
URL		URL associated with node (format-dependent)
width	.75	width in inches
z	0.0	z coordinate for VRML output

Figure 12: Controllable node attributes in DOT

7.3 DOT Attributes

DOT file has a graph (and possibly subgraphs), nodes, and edges. Each of these objects has controllable attributes. These attributes control the layout and appearance of the graph.

DOT attributes are summarized in the following three tables:

- Figure 12 shows the node attributes
- Figure 13 shows the edge attributes
- Figure 14 shows the general graph attributes

Name	Default	Values
arrowhead	normal	style of arrowhead at head end
arrowsize	1.0	scaling factor for arrowheads
arrowtail	normal	style of arrowhead at tail end
color	black	edge stroke color
comment		any string (format-dependent)
constraint	true	use edge to affect node ranking
decorate		if set, draws a line connecting labels with their edges
dir	forward	forward, back, both, or none
fontcolor	black	type face color
fontname	Times-Roman	font family
fontsize	14	point size of label
headlabel		label placed near head of edge
headport		n,ne,e,se,s,sw,w,nw
headURL		URL attached to head label if output format is ismap
label		edge label
labelangle	-25.0	angle in degrees which head or tail label is rotated off edge
labeldistance	1.0	scaling factor for distance of head or tail label from node
labelfloat	false	lessen constraints on edge label placement
labelfontcolor	black	type face color for head and tail labels
labelfontname	Times-Roman	font family for head and tail labels
labelfontsize	14	point size for head and tail labels
layer	overlay range	all, *id* or *id:id*
lhead		name of cluster to use as head of edge
ltail		name of cluster to use as tail of edge
minlen	1	minimum rank distance between head and tail
samehead		tag for head node; edge heads with the same tag are merged onto the same port
sametail		tag for tail node; edge tails with the same tag are merged onto the same port
style		graphics options, e.g. bold, dotted, filled; cf. Section 2.3
taillabel		label placed near tail of edge
tailport		n,ne,e,se,s,sw,w,nw
tailURL		URL attached to tail label if output format is ismap
weight	1	integer cost of stretching an edge

Figure 13: Controllable edge attributes in DOT

Name	Default	Values
bgcolor		background color for drawing, plus initial fill color
center	false	center drawing on page
clusterrank	local	may be global or none
color	black	for clusters, outline color, and fill color if fillcolor not defined
comment		any string (format-dependent)
compound	false	allow edges between clusters
concentrate	false	enables edge concentrators
fillcolor	black	cluster fill color
fontcolor	black	type face color
fontname	Times-Roman	font family
fontpath		list of directories to such for fonts
fontsize	14	point size of label
label		any string
labeljust	left-justified	”r” for right-justified cluster labels
labelloc	top	”r” for right-justified cluster labels
layers		*id:id:id...*
margin	.5	margin included in page, inches
mclimit	1.0	scale factor for mincross iterations
nodesep	.25	separation between nodes, in inches.
nslimit		if set to *f*, bounds network simplex iterations by *(f)(number of nodes)* when setting x-coordinates
nslimit1		if set to *f*, bounds network simplex iterations by *(f)(number of nodes)* when ranking nodes
ordering		if out out edge order is preserved
orientation	portrait	if rotate is not used and the value is landscape, use landscape orientation
page		unit of pagination, *e.g.* "8.5,11"
pagedir	BL	traversal order of pages
quantum		if quantum ¿ 0.0, node label dimensions will be rounded to integral multiples of quantum
rank		same, min, max, source or sink
rankdir	TB	LR (left to right) or TB (top to bottom)
ranksep	.75	separation between ranks, in inches.
ratio		approximate aspect ratio desired, fill or auto
remincross		if true and there are multiple clusters, re-run crossing minimization
rotate		If 90, set orientation to landscape
samplepoints	8	number of points used to represent ellipses and circles on output (cf. Appendix C
searchsize	30	maximum edges with negative cut values to check when looking for a minimum one during network simplex
size		maximum drawing size, in inches
style		graphics options, e.g. filled for clusters
URL		URL associated with graph (format-dependent)

Figure 14: Controllable graph/subgraph attributes in DOT

8 Mozilla

8.1 Overview

MOZILLA is an open source web browser first created in March of 1998 when Netscape released their portion of copyrighted code for the Netscape Navigator browsers and components to the open source community. However, Netscape could not release all of the source code for the browser so a community named mozilla.org was created to rewrite the missing portions of code, creating the first open source web browser. Since then, MOZILLA has been one of the largest ongoing projects in the open source community.

MOZILLA consists of several subcomponents. Components that need to be taken into consideration with the implementation of SpiderGraph include: SeaMonkey, User Interface, Help Viewer, JavaScript, Netscape Portable Runtime, Plug-ins, Necko/Gecko, XPCOM, and XPConnect.

8.2 Mozilla in Relation to SpiderGraph

SpiderGraph is plug-in for MOZILLA which allows for increased functionality in the navigation of webpages. When a user visits a new URL, it will be sent to the SpiderGraph navigational window, which is just a specialized MOZILLA window. SpiderGraph will use the data collected from the user and DOT to generate an image map in the specialized navigational window.

A Index

Index

abbreviations, 43
abstract, 6
acronyms, 43
addNode, 27
architecture, environment, 9
architecture, general, 8
architecture, internal, 9
attributes, 36
authors, 44

bibliography, 45
browser, input, 20
browser, user-interface, 28
buttons, 15

chrome, 9
close, 17
Cohen, Marc, 44
content, 11
contents.rdf, 12

data, collection, 9
data, export, 9
data, manipulation, 9
data, processing, 9
data-flow, diagram, 10
data-flow, top level, 9
deadlines, 8
definitions, 43
delivery, 31
DeMoss, Patrick, 44
design, goals, 7
display, 13
display, area, 15
dot, 9
DOT, attributes, 36
DOT, sample input, 33, 34
DOT, sample output, 33, 35

Edge, 19
environment, 9
events, 19
events, input, 20
events, interaction, 20
events, preferences, 19
events, processing, 19

Future Release, 8

Gecko, 39
getSelectedNodes, 17
goals, 7
Graph, 19
graph, buttons, 15
graph, displaying, 11
graph, export, 9
graph, manipulation, 9
graph, storage, 9
graph.png, 13
graphDisplay, 11
graphDisplay, files, 11
graphDisplay, summary, 18
graphDisplay.js, 17
graphDisplay.xul, 14
graphjs, 13

implementation, deadlines, 8
implementation, plan, 8

installation, 31
interaction, 20

JavaScript, 29, 39

linksToURL, 13
locale, 12

menu bar, 14
milestones, 8
milestones, alpha, 8
milestones, beta, 8
milestones, release, 8
Mozilla, 7
mozilla.org, 39

navigator, 9
navigator.css, 14
navigatorOverlay.js, 13
navigatorOverlay.xul, 14
Necko, 39
Netscape, 39
Node, 19

parameters, 9
plan, 8
prototype, 8
prune, 17
prune.URL, 13
purpose, 6

references, 45
removeNode, 27
results.html, 14

scope, 6
SeaMonkey, 39
SG-1, 6
skin, 12
software, environment, 9
SP1, 8
SpiderGraph, button, 28
SpiderGraph, subpackages, 9
spidergraph.gif, 18
spidergraphOverlay.xul, 14
subgraphs, 36
subpackages, 9

updateGraph, 18
updatePreferences, 27
updateURL, 27
user interface, 9
user-interface, elements, 30

Voronenko, Yevgen, 44

Walsh, Frederick, 44
Weeks, Leland, 44

XML, 28
XPCOM, 39
XPConnect, 39
XPInstall, 31
XUL, 11, 28

B Definitions, Acronyms and Abbreviations

clickpath The order in which a set of web pages or nodes are traversed.

cluster A set of related nodes grouped together or consolidated into a single node.

domain A group of web pages that belong to the same address or domain name.

edge A directional or non-directional line which shows the relationships between two nodes.

graph A diagram that exhibits a functional relationship between a set of elements.

GraphViz A graph layout and rendering package.

GUI Graphical User Interface

hypertext Computer based text retrieval system that enables a user to access particular locations in web pages or other electronic documents by clicking on links.

link A reference from some point in a hypertext document to some point in another document or another place in the same document.

Mozilla Web browser application.

navigate To traverse through the world wide web.

node A point or vertex in a graph.

prune To remove unnecessary or unwanted nodes and edges in a graph.

render Process of visualizing internal representation of graphical objects.

root The starting point or the first node of a graph.

RPM Red Hat Package Manager

SeaMonkey Codename for the current MOZILLA browser.

transitive The relation x to y and the relation y to z implies the relation x to z.

tree A graph in which there is only one route between any pair of nodes.

UI User-interface, in most cases refers to GUI

URL Uniform Resource Locator

W3C World Wide Web Consortium http://www.w3.org/

web browser An application used to navigate the world wide web.

web server A process running at a web site which sends out web pages in response to HTTP requests from remote browsers.

weight The varying degree of importance of an edge.

XPCOM Cross-Platform scheme for turning objects into discrete components.

XPConnect Binding between XPCOM and JavaScript.

XUL Exstensible User Interface Language. A fast and effective language to implement GUIs utilizing the Mozilla software package.

C Section Authors

- Section 1: Yevgen Voronenko
- Section 2: Yevgen Voronenko
- Section 3: Marc Cohen
- Section 4: Patrick DeMoss
- Section 5: Leland Weeks
- Section **??**: Frederick Walsh
- Section 6: Frederick Walsh
- Section 7: Marc Cohen
- Section 8: Leland Weeks

D Bibliography

References

[1] AT&T Research Labs , *"The DOT Language"*, http://www.research.att.com/~erg/graphviz/info/lang.html, February 2003.

[2] AT&T Research Labs , *"Welcome to GraphViz"*, http://graphviz.org/, February 2003.

[3] Andersen, A., Deakin, N., *"XUL Planet"*, http://www.xulplanet.com/, March 2003.

[4] Bergmann, S., *"phpOpenTracker"*, http://www.phpopentracker.de/, February 2003.

[5] Cohen, M., et al. *"SpiderGraph Requirements Specification"*, http://www.mcs.drexel.edu/~uyvorone/cs452, February 25, 2003.

[6] Institute of Electrical and Electronics Engineers, *"Hyperlinks"*, RFC 1866, Institute of Electrical and Electronic Engineers, 1998.

[7] Jupitermedia Corporation, *"JavaScript Source"*, http://javascript.internet.com/, March 2003.

[8] Kendall, S., *"Unified Modeling Language Dictionary"*, http://www.softdocwiz.com/UML.htm, March 2003.

[9] Mozilla Organization, The, *"Mozilla Hackers's Getting Started Guide"*, http://www.mozilla.org/hacking/coding-introduction", Febuary 2003

[10] Mozilla Organization, The, *"Mozilla Navigator Source Code"*, http://ftp.mozilla.org/pub/mozilla/releases/mozilla1.3b/src/mozilla-source-1.3b.tar.gz, March 2003

[11] Mozilla Organization, The, *"Tutorial: Creating a Mozilla Extension"*, http://www.mozilla.org/docs/tutorials/tinderstatus, March 2003

[12] Rational Software, *"UML Resource Center"*, http://www.rational.com/uml/, March 2003.