Università degli Studi di Udine
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Corso di Laurea in Tecnologie Web e Multimediali

Tesi di Laurea

Visualizzazione interattiva di bigrafi attraverso la libreria D3.js

Candidato
Andrea Simeone

Relatore
Prof. Marino Miculan

Anno accademico 2017-2018

Contatti dell'Istituto
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
http://www.dimi.uniud.it/

Alla mia famiglia.

Indice

Elenco delle figure

1

Introduzione

Negli ultimi anni la crescente disponibilità di sistemi di calcolo a costi ridotti, la crescente diffusione di sistemi software e la progressiva riduzione del costo dell'accesso alla rete unito l'allargamento delle ampiezze di banda disponibile hanno portato alla diffusione di sistemi mobili distribuiti. Di pari passo con la diffusione dei sistemi distribuiti sono stati proposti diversi modelli formali per la loro descrizione e definizione. Tra i vari modelli proposti in letteratura un modello ormai consolidato sono i *bigrafi* descritti da Milner in [7][6]. Questi permettono una trattazione intuitiva degli oggetti che compongono un sistema distribuito e di come essi interagiscono tra loro. I bigrafi grazie alla loro generalità si presentano come un meta-modello capace di descrivere sistemi di qualsiasi dominio si voglia: sono stati utilizzati per descrivere automi a stati finiti[3], π calcolo, reti di Petri[5] e anche sistemi biologici[2]. Punto di forza dei bigrafi è la capacità di unire al rigore formale del modello una rappresentazione grafica che ben rappresenta il sistema che viene descritto rendendo accessibile la comprensione dei processi che lo coinvolgono. Ad oggi, nonostante la sempre maggior diffusione del modello, lo sviluppo di sistemi per la realizzazione di rappresentazioni grafiche accurate dei bigrafi risulta ancora lacunoso: sono state proposte alcune soluzioni[1][1][4] che però presentano lo svantaggio di richiedere l'installazione di sistemi di sviluppo software e non permettono quindi la visione dei bigrafi da sistemi che non possiedono del software dedicato né tantomeno la visione da sistemi *mobile* o con potenza di calcolo limitata. Non è stato inoltre ancora sviluppato un formato che consenta la visualizzazione del bigrafo via Web: questa rappresenterebbe un'ottima soluzione in quanto consentirebbe di usufruire della rappresentazione grafica in un'ampia serie di dispositivi.

Questo lavoro di tesi si propone di fornire una soluzione a questa problematica realizzando un sistema di visualizzazione web basato sulla libreria *D3.js*[8]. La scelta è ricaduta su D3.js in quanto fornisce una serie di strumenti piuttosto potenti per creare visualizzazioni web legate a sorgenti di dati, dà la possibilità allo sviluppatore di modificare secondo le proprie esigenze i layout di base forniti e risulta inoltre, per le funzionalità offerte, non particolarmente *"esosa"* in termine di risorse hardware permettendo il suo utilizzo anche su dispositivi mobili o con scarsa potenza di calcolo. Per realizzare la visualizzazione si è definita inoltre la struttura dati per la descrizione dei bigrafi mediante l'utilizzo del formato *JSON* data l'ottima integrazione della libreria D3.js con questo formato; *JSON* inoltre ha

[1] Anche il Laboratorio di Sistemi Distribuiti dell'Università di Udine, all'interno di una libreria di implementazione di bigrafi e BRS, ha sviluppato un sistema di visualizzazione di bigrafi. Per maggiori informazioni: http://mads.uniud.it/wordpress/downloads/libbig/

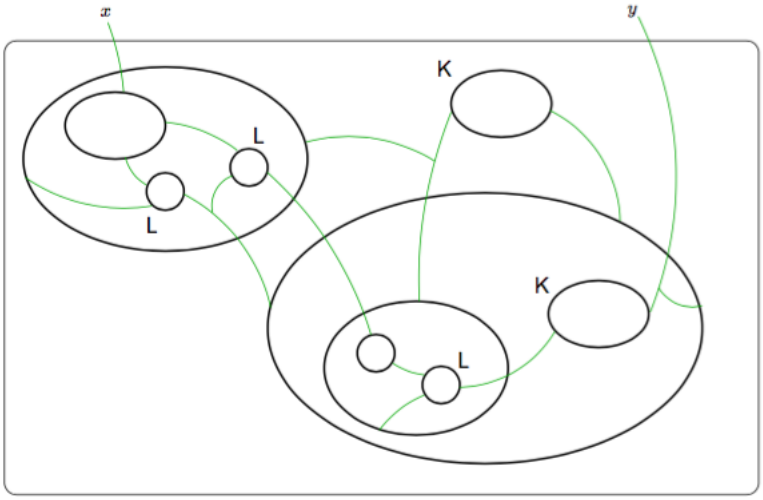


Figura 1.1: Esempio di bigrafo

raggiunto una maturità e standardizzazione che ne permettono un utilizzo sicuro e stabile nel tempo. La tesi proporrà una soluzione che comprenderà tutte le tipologie di grafi che si possono incontrare utilizzando i bigrafi: *place graph*, *link graph* e *bigrafo* come composizione di *place graph* e *link graph*.

Questo volume è così suddiviso: il capitolo 2 presenta una rapida introduzione ai bigrafi sottolineando in particolar modo le relazioni tra proprietà dei bigrafi e loro rappresentazione schematica. Il capitolo 3 presenta un'analisi preliminare del progetto con la definizione dei requisiti del sistema e l'architettura di riferimento. Il capitolo 4 introduce la libreria D3.js e la tecnologia SVG per la realizzazione di grafica vettoriale passando poi ad una descrizione dell'implementazione del software e la visualizzazione dei risultati ottenuti. Nel capitolo 5 infine si sono tratte le conclusioni sull'intero lavoro svolto proponendo alcuni possibili sviluppi futuri.

2

Bigrafi

In questo capitolo si farà una rapida analisi dei bigrafi, analizzando rapidamente le caratteristiche di questo particolare tipo di grafi, gli elementi che lo compongono e la loro rappresentazione grafica. Non verranno trattati argomenti quali le operazioni sui bigrafi nè in generale l'algebra dei bigrafi in quanto non interessanti per le finalità del lavoro svolto.

2.1 Bigrafi e loro componenti

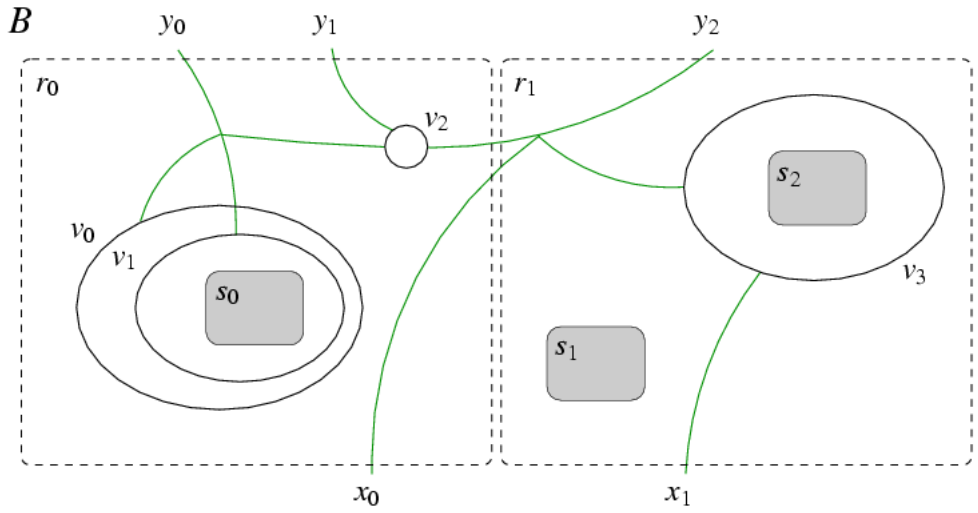
Un *bigrafo* è un particolare tipo di grafo in cui i nodi che lo compongono possono essere annidati tra loro e gli archi che collegano i nodi sono *iper-archi* ovvero consentono la connessione di un nodo di partenza a più nodi d'arrivo mediante l'utilizzo della stessa connessione. Pertanto, come il nome *bi-grafo* suggerisce, abbiamo a che fare con una struttura composta da due grafi completamente indipendenti tra loro che veicolano informazioni differenti, quali topologia dei nodi e collegamenti tra nodi. Inoltre si possono definire diversi tipi di nodi con significato e caratteristiche diverse rendendo il bigrafo un *metamodello* estremamente flessibile e utilizzabile in diversi contesti.

2.1.1 Tipologie di nodi

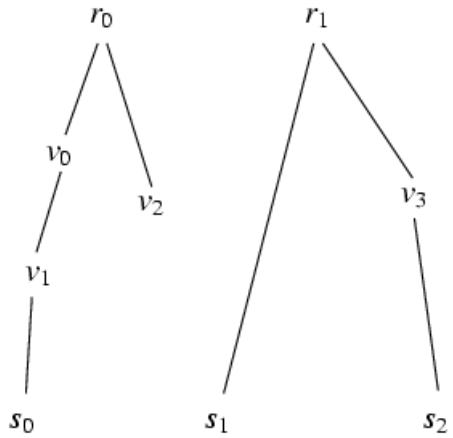
La grande capacità espressiva dei bigrafi rende possibile il loro utilizzo in diversi contesti, questo rende indispensabile definire dei significati e delle regole per ogni nodo. Per fare ciò si assegnano ad ogni nodo una segnatura e un controllo:

Definizione 2.1.1 (Segnatura e Controllo)**.** *Una segnatura è una coppia* (K, ar)*, dove* K *è un insieme di tipi di nodi chiamati* controlli*, e* $ar : K \to \mathbb{N}$ *è una mappa che associa ad ogni tipo di nodo (cioè ad ogni controllo) un numero naturale chiamato arietà.*

Con **arietà** intendiamo il numero di porte (e quindi di connessioni possibili) assegnato ad ogni tipo di nodo. Con **controllo** si intende l'assegnazione ad ogni nodo un tipo che lo definisce. Ogni nodo ha quindi un solo controllo assegnato (si può notare una similitudine con la programmazione orientata agli oggetti ed alla relazione tra classe e istanze di classe). Questo è molto interessante ai fini del lavoro qui descritto perché possiamo definire una primitiva grafica per ogni controllo (e quindi tipo di nodo) presente nel bigrafo. La coppia (K, ar) definisce quindi in maniera completa ogni nodo presente nel bigrafo assegnandoli un controllo e l'arietà.



(a) Bigrafo



(b) Place graph

Figura 2.1: Esempio di bigrafo e relativo place graph.

2.1.2 Place Graph

Il Place Graph è il grafico che specifica la topologia dei nodi e indica quindi come sono annidati i nodi. Il Place Graph può essere descritto come una foresta ed ogni albero della foresta ha come nodo iniziale la *radice*(o *regione*). Ogni radice è identificata da un numero che va da 0 a $n-1$ dove n è il numero di alberi. Ogni nodo può contenere un *sito* che è un nodo particolare che può contenere al suo interno un altro Place Graph permettendo la composizione di Place Graph. In figura 2.1 è possibile osservare un bigrafo e il relativo place graph. Per permettere la composizione tra bigrafi definiamo l'interfaccia del place graph che è descritta da:

- **interfaccia esterna**: è descritta da un numero naturale n che descrive il numero di *radici* presenti nel place graph. In figura 2.1 per esempio $n = 2$ (sono presenti 2 radici: r_o, r_1);

- **interfaccia interna**: è descritta da un numero naturale n che descrive il numero di *siti* presenti nel place graph. In figura 2.1 per esempio $n = 3$ (sono presenti 3 siti: s_o, s_1, s_2).

2.1.3 Link Graph

Il link graph è un *ipergrafo* (cioè un grafo i cui archi possono connettere più nodi) non orientato che descrive il collegamento fra i vari nodi del bigrafo. Il link graph descrive anche le le potenziali connessioni con altri bigrafi attraverso le *interfacce*. L'interfaccia del link graph è descritta da *interfaccia esterna* e *interfaccia interna*:

- **interfaccia esterna**: è composta dagli *outer names* che indicano le connessioni disponibili verso l'esterno offerte dal link graph: per convenzione sono poste nella parte superiore della rappresentazione grafica del link graph. In figura 2.2 l'interfaccia esterna è composta da $\{y_o, y_1, y_2\}$;
- **interfaccia interna**: è composta dagli *inner names* che indicano le connessioni disponibili verso l'interno offerte dal link graph: per convenzione sono poste nella parte inferiore della rappresentazione grafica del link graph. In figura 2.2 l'interfaccia interna è composta da $\{x_o, x_1\}$.

In figura 2.2 si può vedere il link graph relativo al bigrafo della figura 2.1 con outer names e inner names.

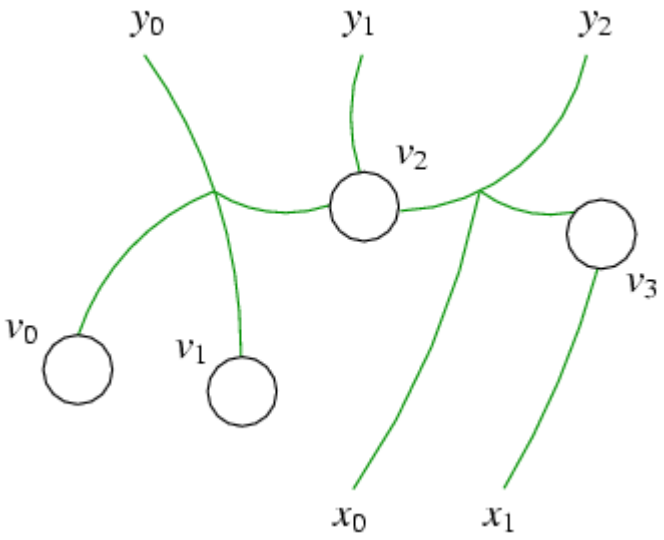


Figura 2.2: Link graph del bigrafo in figura 2.1

2.1.4 Bigrafo

Il bigrafo non è altro che l'unione di place graph e link graph. È importante notare che i 2 grafi condividono lo stesso insieme di nodi. I bigrafi vengono spesso descritti attraverso l'interfaccia che li caratterizza, un generico bigrafo B ad esempio è composto da:

$$B = \langle n, X \rangle \rightarrow \langle m, Y \rangle$$

stando ad indicare che B ha n siti, X inner names, m radici e Y outer names.
Il bigrafo in figura 2.1 sarà quindi descritto dalla seguente interfaccia:

$$B = \langle 3, \{x_0, x_1\} \rangle \rightarrow \langle 2, \{y_o, y_1, y_2\} \rangle$$

2.1.5 Composizione tra Bigrafi

I bigrafi nascono come strutture composizionali: è possibile definire bigrafi minimali e tramite operazioni di composizione definire bigrafi via via più estesi e complessi. L'operazione di composizione è possibile grazie alla presenza dei *siti* ovvero una particolare classe di nodi in cui è possibile inserire altri bigrafi. Non è però possibile inserire qualsiasi bigrafo: considerando due bigrafi A e B l'operazione di composizione $B \circ A$ è possibile se solo se l'interfaccia interna di B è uguale all'interfaccia esterna di A. Quindi, in generale se $A : I \to J$ e $B : J \to K$ sono due bigrafi con nodi e grafi disgiunti, dove $I = \langle k, X \rangle$, $J = \langle m, Y \rangle$, $K = \langle n, Z \rangle$ allora il bigrafo composto $B \circ A : I \to K$ non è altro che il risultato della coppia di composizioni $\langle A^P \circ B^P, A^L \circ B^L \rangle$ i cui componenti sono creati come segue :

- per la creazione del place graph $A^P \circ B^P : k \to n$, per ogni $i \in m$ si unisce l'iesima radice di B^P con l'iesimo sito di A^P;
- per la creazione del link graph $A^L \circ B^L : X \to Z$, per ogni $y \in Y$ si collega il link di B^L con outer name y con il link di A^L con inner name y.

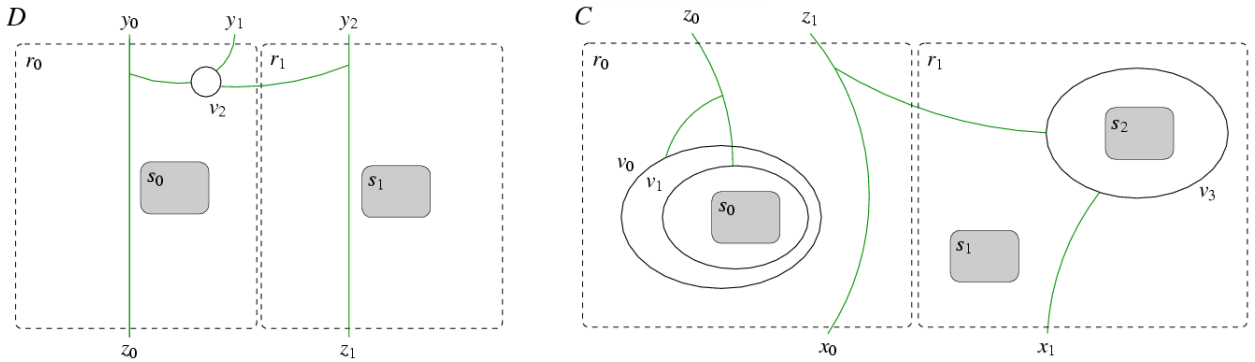
In questo modo i due bigrafi di partenza B e A si compongono in un unico bigrafo attraverso i nodi e i collegamenti messi in relazione dall'interfaccia comune J che cessa di esistere.
Per esemplificare il tutto consideriamo il bigrafi in figura 2.3: l'operazione di composizione dei 2 bigrafi C e D genera il bigrafo B presente in figura 2.4. Descriviamo con la notazione precedente definita i 2 bigrafi C e D:

$$D = \langle 2, \{z_0, z_1\} \rangle \to \langle 2, \{y_o, y_1, y_2\} \rangle$$
$$C = \langle 3, \{x_0, x_1\} \rangle \to \langle 2, \{z_o, z_1\} \rangle$$

Il risultato della composizione $D \circ C$ sarà uguale a:

$$B = D \circ C = \langle 3, \{x_0, x_1\} \rangle \to \langle 2, \{y_o, y_1, y_2\} \rangle$$

Si noti come l'interfaccia in comune tra C e D permetta la composizione dei 2 bigrafi e la sua scomparsa al termine dell'operazione stessa.



(a) Bigrafo D (b) Bigrafo C

Figura 2.3: Bigrafi C e D che compongono il bigrafo di figura 2.4

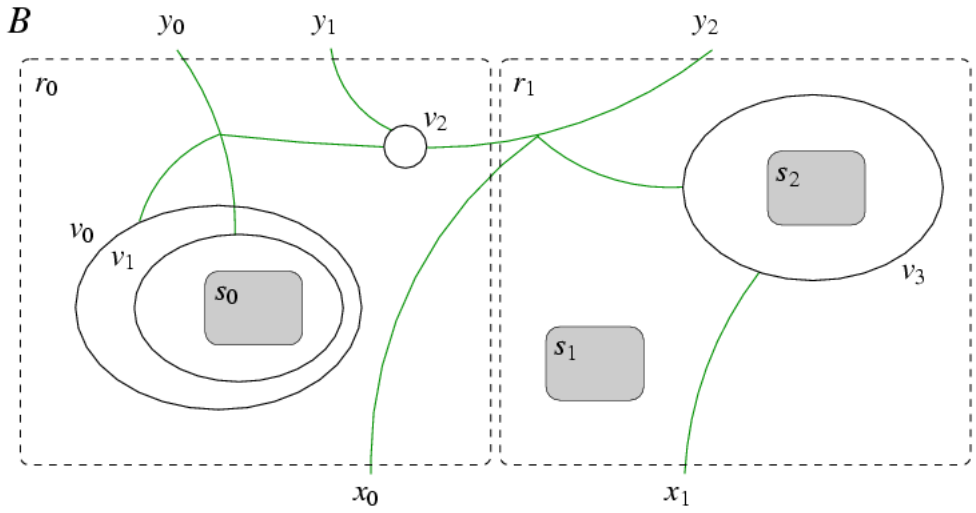


Figura 2.4: Bigrafo B risultante dall'operazione di composizione $D \circ C$

2.1.6 Giustapposizione di Bigrafi

Definiamo ora un'altra operazione per creare un bigrafo da altri 2 bigrafi di base. Si chiama *giustapposizione* o *prodotto* di bigrafi e consiste nell'affiancare un bigrafo ad un altro. Questa operazione è possibile solo se i due bigrafi sono *disgiunti*. Definiamo ora quando due bigrafi sono *disgiunti*:

$$F_i : \langle m_i, X_i \rangle \to \langle n_i, Y_i \rangle (i = 0, 1)$$

si dicono *disgiunti* se $|F|$, X_i e Y_i sono (rispettivamente) fra loro disgiunti.

Dati due bigrafi disgiunti:

$$F : \langle m_F, X_F \rangle \to \langle n_F, Y_F \rangle,\ G : \langle m_G, X_G \rangle \to \langle n_G, Y_G \rangle$$

la *giustapposizione* o *prodotto* $F \otimes G : \langle m_F + m_G, X_F \uplus X_G \rangle \to \langle n_F + n_G, Y_F \uplus Y_G \rangle$ è il bigrafo:

$$F \otimes G : (V_F \uplus V_G, E_F \uplus E_G, ctrl_F \uplus ctrl_G, prnt_F \uplus prnt'_G, link_F \uplus link_G)$$

dove $prnt'_G(m_F + i) = n_F + j$ ogni volta che $prnt_G(i) = j$.

3

Analisi e Progetto

In questo capitolo sarà presentata l'analisi preliminare svolta per la realizzazione del sistema software. Si prenderanno in esame la definizione dei requisiti del progetto e si fornirà un'architettura di riferimento per la realizzazione del nostro sistema software analizzando in dettaglio le funzioni di ogni singolo modulo e la interazione tra moduli.

3.1 Definizione dei Requisiti

In questa fase iniziale nello sviluppo del sistema software ci si è focalizzati su quelli che erano i principali bisogni degli utenti del nostro sistema, giungendo alla stesura di una serie di requisiti:

- Il sistema software deve fornire un mezzo per rappresentare e visualizzare i *bigrafi* opportunamente descritti da appositi file XML o JSON;
- deve utilizzare gli elementi grafici e lo stile di rappresentazione dei *bigrafi* utilizzato in letteratura;
- deve essere in grado di visualizzare in un'unica schermata i vari grafici che compongono i *bigrafi*, quindi i *place graph* e i *link graph* e il bigrafo nel suo complesso;
- deve utilizzare come strumento di visualizzazione un moderno browser web;
- deve poter permettere l'interazione dell'utente con il grafico realizzato permettendo azioni come *zoom*, trascinamento degli elementi grafici che compongono il grafico (*dragging*) e loro blocco all'interno dell'area di visualizzazione (*pinning*).
- la sua esecuzione non deve essere computazionalmente dispendiosa (sia lato client che lato server) in modo da permettere l'utilizzo del sistema anche attraverso hardware datato e dispositivi mobili (tablet e smartphone).

Fin dalla prima fase di definizione dei requisiti utenti si è capito che il software non doveva limitarsi ad una semplice visualizzazione della rappresentazione grafica del modello ma permettere una interazione tra utente e la rappresentazione grafica influenzando le scelte progettuali successive.

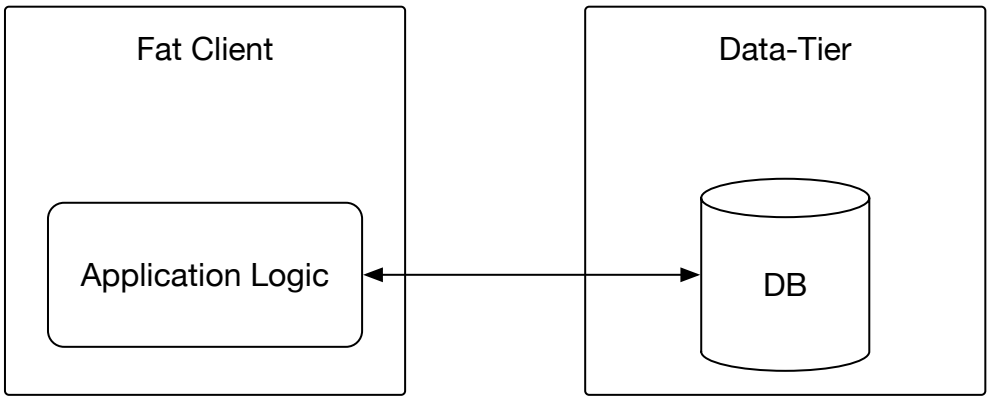


Figura 3.1: Esempio di sistema Two-Tier

3.2 Progetto: architettura di riferimento

La definizione dei requisiti dell'utente ha naturalmente influito nella definizione dei requisiti del sistema indirizzando innanzitutto le scelte architetturali del sistema, successivamente la scelta degli strumenti software più adatti alla realizzazione dello stesso. Fino dalle fasi iniziali della stesura dei requisiti software è parso naturale scegliere come architettura di riferimento per il nostro sistema quella **Two-Tier** (figura 3.1) (*"a 2 strati"* comunemente chiamata Client-Server) con **Fat Client** ovvero quella tipologia di client che contiene oltre alla visualizzazione e interazione con l'utente anche la parte relativa all'applicazione. Il modello *Two -Tier* è un pattern architetturale che prevede, per l'appunto, la suddivisione in 2 strati (detti *layer*) della nostra applicazione in cui ogni strato si occupa di un preciso aspetto all'interno dell'applicazione. I layer sono:

1. il **Presentation layer & Application processing** ovvero lo strato che si occupa di fornire le funzionalità richieste all'applicazione, della presentazione dei risultati della computazione all'utilizzatore del sistema e della interazione dell'utente con essa collezionando gli input dell'utente;

2. il **Data management layer** ovvero lo strato che gestisce i dati persistenti della nostra applicazione.

3.2.1 Descrizione dei moduli

Si procede ora nella descrizione dettagliata di ogni singolo layer.

3.2.1.1 Presentation & Application Layer

È il layer fondamentale della nostra applicazione, in quanto si occupa delle operazioni fondamentali del nostro programma:

- richiesta e ricezione via *HTTP* (HyperText Transfer Protocol) del file contenente la struttura dati in cui è descritto il bigrafo;

- parsing del file ricevuto e estrapolazione dei dati;

- creazione di una pagina *HTML* (HyperText Markup Language) con visualizzazione del bigrafo attraverso l'uso di *SVG* (Scalable Vector Graphics);
- offrire la possibilità all'utente di interagire con la rappresentazione grafica spostando gli elementi SVG liberamente nell'area di visualizzazione.

Dato il numero di operazioni svolte e la necessità di non gravare eccessivamente sulle risorse del client si è scelto di utilizzare il linguaggio *Javascript* che sin dalla sua nascita ha rappresentato il linguaggio più utilizzato nello scripting lato client e per la creazione della visualizzazione SVG di utilizzare una delle molteplici librerie di visualizzazione dati disponibili sul mercato quest'oggi. Per gli aspetti implementativi si veda il capitolo 4.

3.2.1.2 Data Management Layer

È il layer che fornisce all'applicazione i file opportunamente formattati che veicolano le informazioni relative ai bigrafi. È realizzato tramite un comune server web. All'atto del caricamento della pagina su client quest'ultimo invia una richiesta via HTTP di accesso al file che descrive la struttura del bigrafo da visualizzare, il server web processa la richiesta e risponde con l'invio del file desiderato.

3.2.2 Descrizione della comunicazione tra moduli

Viene di seguito brevemente descritto il meccanismo di comunicazione tra i moduli che compongono l'architettura software realizzata.

3.2.2.1 Richiesta del client

Al caricamento della pagina web di visualizzazione del bigrafo il client invia una richiesta asincrona HTTP/GET con il reclamo della specifica risorsa al server. Javascript offre la possibilità di effettuare facilmente la richiesta di accesso al file remoto mediante l'utilizzo della funzione $XMLHttpRequest$, analizziamo brevemente la struttura di una generica richiesta HTTP/GET:

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        var myObj = JSON.parse(this.responseText);
        document.getElementById("demo").innerHTML = myObj.name;
    }
};
xmlhttp.open("GET", "json_demo.json", true);
xmlhttp.send();
```

allo scatenarsi dell'evento *onreadystatechange* la funzione di callback verifica che lo status della richiesta sia quello voluto; verificato ciò si occupa di manipolare i dati ricevuti dalla richiesta. In figura 3.2 viene illustrato il semplice schema di funzionamento della richiesta dei dati al server web.

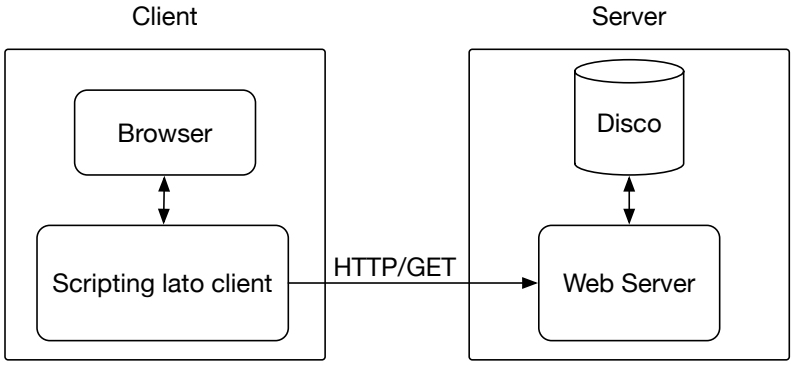


Figura 3.2: Schema di richiesta risorse

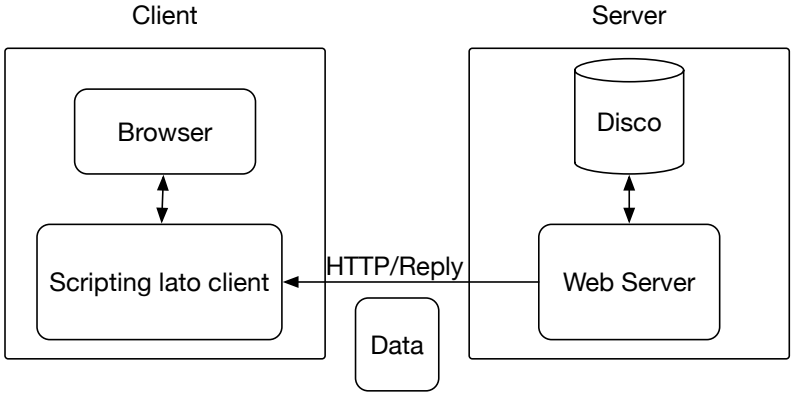


Figura 3.3: Schema di ricezione risorse

3.2.2.2 Risposta del server

Il server ricevuta la richiesta invio del file richiesto dal client provvederà alla sua ricerca su disco e al suo invio mediante HTTP. Il client alla ricezione dei dati, codificati in *JSON* o *XML*, eseguirà la funzione di callback con le relative operazioni di estrapolazione dei dati. Il client dovrà successivamente provvedere all'estrapolazione dei dati e alla creazione della pagina *HTML* contenente la visualizzazione del bigrafo. La figura 3.3 illustra schematicamente il processo di ricezione dei dati.

4

Implementazione

In questo capitolo sarà presentata l'implementazione del software di visualizzazione, descrivendo in dettaglio le scelte implementative, gli strumenti utilizzati, la tecnologia SVG, la struttura dati di riferimento e le visualizzazioni create.

4.1 Motore di rendering

Come già accennato nel capitolo precedente, in fase di analisi è stato necessario scegliere tra diversi strumenti di creazione e visualizzazione di grafi. Vista la crescente popolarità dell'infografica in diversi ambiti che vanno dall'editoria tradizionale al web sono state sviluppate diverse librerie per lo sviluppo di questa tipologia di visualizzazioni. La specificità dei bigrafi ha però presentato difficoltà insormontabili per la maggior parte delle librerie prese in esame: la quasi totalità di esse non permette una profonda customizzazione delle visualizzazioni fornite di default e il bigrafo in quanto risultato della composizione di link graph e place graph rappresenta un ibrido non realizzabile da molte delle librerie che sono state oggetto della valutazione.

A seguito di un'attenta analisi la scelta è ricaduta sulla libreria **D3.js** in particolare per i seguenti motivi:

- libreria matura ma al contempo in costante aggiornamento;
- possibilità di customizzare i layout di default dando grande libertà espressiva all'utilizzatore;
- rendering dei grafici mediante l'utilizzo di tecnologia *SVG* e *Canvas*;
- ottima integrazione con *JSON* e *XML*;
- funzioni per la selezione e interazione con gli elementi del *DOM* (Document Object Model) che non richiede l'utilizzo di altre librerie come *jQuery*.

Analizziamo ora nel dettaglio la libreria e le funzionalità che mette a disposizione dell'utilizzatore.

D3.js (abbreviazione di **Data-Driven Document**) è una libreria *Javascript* utilizzata per produrre visualizzazioni grafiche interattive basate su dati. Nasce nel 2011 dalle ceneri di **Protovis** per mano di Mike Bostock, dottorando all'università di Stanford, e del Stanford Visualization Group col fine di fornire una libreria altamente customizzabile per la visualizzazione di dati attraverso un moderno browser

Web. Negli ultimi anni ha conosciuto una rapida diffusione diventando una delle librerie di riferimento per la visualizzazione di dati: molte aziende utilizzano D3 all'interno dei loro prodotti software e colossi editoriali come *New York Times*[1] e *Guardian*[2] utilizzano D3 per la realizzazione di complesse infografiche visibili nei loro siti internet.

La libreria attualmente ha raggiunto una notevole stabilità e viene costantemente aggiornata (attualmente ha raggiunto la versione 4.13.0 ed è in RC la nuova versione 5) inoltre possiede una solida comunità di utilizzatori e sviluppatori che rilasciano costantemente nuovi pacchetti di visualizzazione o nuovi metodi creativi di utilizzazione delle capacità della libreria D3.

4.1.1 Caratteristiche di D3.js

La libreria D3 incorpora in una pagina **HTML** una serie di funzioni *Javascript* che permettono la selezione di elementi del **DOM**, creare elementi *SVG*, aggiungergli uno stile grafico, ed anche effetti come transizioni e animazioni. Questo permette di creare molto rapidamente, passando collezioni di dati alla libreria, rappresentazioni grafiche di molteplici tipi. La forza della libreria è inoltre quella di permettere la customizzazione delle visualizzazioni di default permettendo la creazione di rappresentazioni grafiche sempre nuove. Si analizza ora qualche aspetto interessante della libreria D3.js

4.1.2 D3.js Selezioni

Similmente a *jQuery*, D3 consente di creare o selezionare gli elementi del DOM e manipolarli in vario modo. Se ad esempio volessimo selezionare tutti gli elementi *g* contenuti nella pagina HTML e creare al loro interno un cerchio rosso con raggio 5px e con posizione random sarebbe sufficiente eseguire il codice seguente:

```
d3.selectAll("g")
    .append("circle")
    .attr("r",5)
    .attr("x",D3.randomUniform(0,length))
    .attr("y",D3.randomUniform(0,height))
    .style("fill","red");
```

Il selettore *D3.selectAll* provvederà alla selezione di ogni elemento *g* presente nel *DOM* e per ogni elemento selezionato applicherà le istruzioni che seguono il selettore. Altra particolarità da notare è la caratteristica concatenazione (chaining) dei metodi invocati tipica di *jQuery* e utilizzata anche da D3.

4.1.3 D3.js Data Binding

Punto di forza principale di D3.js è la capacità di caricare dei dati e tramite essi creare elementi grafici. Ad esempio è possibile caricare un set di dati e per ogni elemento del set creare un oggetto SVG con associate delle proprietà (ad esempio forma, colore ecc...) dipendenti da aspetti significativi dei dati caricati. Riprendendo l'esempio precedente passiamo a D3 un semplice array di numeri $array = [10, 15, 18, 30, 23]$ variamo il raggio del cerchio in relazione al numero passato:

[1] ad esempio: http://www.nytimes.com/interactive/2012/02/13/us/politics/2013-budget-proposal-graphic.html
[2] ad esempio https://www.theguardian.com/world/interactive/2013/feb/12/state-of-the-union-reading-level

```
d3.selectAll("circle")
    .data(array)
    .enter()
    .append("circle")
    .attr("r",function(d){return d;})
    .attr("x",D3.randomUniform(0,length))
    .attr("y",D3.randomUniform(0,height))
    .style("fill","red");
```

Per ogni elemento dell'array verranno eseguite le istruzioni successive al suo caricamento (determinato dall'istruzione *enter*), verranno quindi creati 5 cerchi rossi con raggio dipendente dal valore di ogni singolo elemento dell'array.

4.1.4 D3.js e i Layout

Al fine di rendere più comodo l'utilizzo della libreria, D3 mette a disposizione dell'utilizzatore dei *layout* di default che permettono la creazione di visualizzazioni complesse velocemente e facilmente. Per la realizzazione del progetto di tesi si sono usati due layout: *d3.tree* per la visualizzazione della struttura ad albero del place graph e *d3.forceLayout* per la realizzazione del link graph e del bigrafo.

4.1.5 D3.js version 4

Nel corso del 2016 D3.js è stato aggiornato alla versione 4 che ha portato ad una rivisitazione di molti aspetti della libreria tra i più interessanti il fatto che la libreria non è più monolitica ma è stata riorganizzata in un insieme di pacchetti che permettono di caricare solo le funzionalità della libreria che si utilizzano rendendo il caricamento e l'esecuzione del codice molto più veloce. Ogni singolo pacchetto può essere scaricato utilizzando il gestore di pacchetti di **Node.js** *npm* (node package manager) oppure inserendo manualmente nello script il riferimento remoto al pacchetto (facendo attenzione alle dipendenze tra pacchetti).

4.2 SVG - Scalable Vector Graphics

SVG (acronimo di *Scalable Vector Graphics*) è un linguaggio basato su **XML** sviluppato dal **W3C** (*World Wide Web Consortium*) per descrivere grafica 2D vettoriale. Ha conosciuto una gestazione piuttosto travagliata dovuta dal fatto che multinazionali come Microsoft e Macromedia avevano introdotto all'interno del W3C il linguaggio **VML**(*Vector Markup Language*) mentre Adobe e Sun Microsystem proponevano un linguaggio alternativo chiamato **PGML**(*Precision Graphics Markup Language*): SVG nasce dunque come compromesso tra 2 linguaggi differenti. Lo sviluppo di **SVG** è iniziato nel 1999 e la prima raccomandazione del W3C relativa all'uso del linguaggio **SVG1.0**[3] è del 4 settembre 2001. L'ultima versione del linguaggio **SVG1.1 Second Edition**[4] è del 16 agosto 2011. Una versione 2.0 è attualmente in sviluppo[5]. A oltre 15 anni dal suo primo rilascio **SVG** ha finalmente raggiunto un otti-

[3]La prima Recommendation del W3C di SVG 1.0 e visibile al link https://www.w3.org/TR/2001/REC-SVG-20010904/
[4]La Recommendation del W3C di SVG 1.1 è visibile al link https://www.w3.org/TR/SVG11/
[5]Attualmente è visibile una Candidate Recommendation di SVG 2.0 al link https://www.w3.org/TR/SVG2/

mo supporto da parte di tutti i principali Browser Web, rimangono criticità che riguardano sistemi che utilizzano versioni di browser *Internet Explorer* inferiori alla 11 e sistemi *Android* con versione inferiore alla 4 e che usano il browser di default.

4.2.1 Caratteristiche di SVG

SVG visualizza 3 tipi di oggetti:

1. **Immagini vettoriali**: ovvero l'immagine è descritta mediante l'uso di primitive geometriche che definiscono punti, linee, curve, poligoni a cui possono essere attribuiti colori, sfumature ecc... In caso di operazioni di ridimensionamento dello schermo l'immagine viene ridisegnata non generando alcun tipo di perdita di dettaglio;

2. **Immagini raster**: l'immagine viene vista come una griglia, ad ogni pixel (singolo elemento della griglia) viene assegnato un colore: in caso di operazioni di ingrandimento dell'immagine quest'ultima risulterà sgranata;

3. **Testi esplicativi** eventualmente cliccabili.

SVG consente di operare sugli elementi sopraelencati in vario modo raggruppandoli, modificando le caratteristiche grafiche, operando trasformazioni (creando quindi animazioni), e altro ma non dispone di uno $z - index$: non è possibile quindi determinare il livello di profondità degli oggetti disegnati (quindi l'ordine in cui vengono disegnati gli oggetti a schermo determina il risultato finale).

4.2.2 Gestione dell'aspetto del contenuto SVG

SVG consente diversi modi di gestire l'aspetto visuale degli elementi definiti al suo interno. I metodi generalmente utilizzati sono:

1. definire l'aspetto visuale per ogni elemento SVG: ogni elemento possiede gli opportuni campi per la definizione dello stile;

2. ottenere un SVG attraverso XML e XSLT: attraverso l'uso di XSLT (*eXtensible Stylesheet Language Transformations*) possiamo trasformare un documento XML in SVG che risulterà graficamente definito (a patto che XSLT sia completo di ogni descrizione necessaria alla completa trasformazione);

3. definire l'aspetto attraverso CSS: è possibile, come per l'HTML definire fogli di stile per gli elementi SVG.

Tutte le 3 alternative presentano dei vantaggi e degli svantaggi: ad esempio definire l'aspetto grafico di ogni elemento rende certa la portabilità dello stesso in quanto tutte le informazioni sono contenute nell'SVG ma, per grafiche di una certa complessità comporta un aumento delle dimensioni del file. D'inverso definire lo stile tramite CSS comporta una riduzione delle dimensioni del file oltre a semplificare la gestione della definizione dello stile (uso di classi, id ecc...) ma necessita che il foglio di stile, se non già integrato nella pagina venga fornito insieme alla pagina HTML che contiene SVG.

4.3 Descrizione dell'implementazione

Nei paragrafi successivi vengono illustrate le scelte effettuate per la creazione della struttura dati che descrive i bigrafi e le tecniche utilizzate per la creazione delle visualizzazioni dei Place Graph, Link Graph e dei bigrafi.

4.3.1 Definizione della Struttura Dati

Il primo passo nella realizzazione del sistema software è stato quello di definire la struttura dati per la corretta descrizione dei bigrafi da rappresentare. Per prima cosa si è scelto di utilizzare la codifica *JSON* (acronimo di JavaScript Object Notation) in quanto tecnologia ben integrata con la libreria D3, successivamente si è cercato il miglior modo di strutturare i dati per rappresentare le caratteristiche essenziali dei bigrafi. Si illustrano ora le codifiche di ogni singolo aspetto interessante dei bigrafi.

4.3.1.1 Codifica dell'Interfaccia

Per codificare l'interfaccia si è deciso di creare una coppia di array in cui vengono inseriti rispettivamente gli elementi atti a descrivere *inner names* e *outer names*.

```
"interface":[
   {"inner":["in1","in2","in3"]},
   {"outer":["out1","out2"]}
]
```

Figura 4.1: Rappresentazione dell'interfaccia in JSON

4.3.1.2 Codifica della Signature

A differenza di quanto illustrato nel Capitolo 1 e previsto dalla teoria dei bigrafi la codifica della *Signature* è stata strutturata per non prevedere una definizione dell'arità dei nodi ma definisce la loro rappresentazione grafica descrivendo forma e colore dell'elemento visualizzato. Oltre alle primitive grafiche previste dal SVG (*circle, ellipse,rect*) si è offerta anche la possibilità di creare anche poligoni regolari per arricchire le possibilità di visualizzazione dei bigrafi.

```
"signature":[
   {"id":"site","shape":"rect","color":"grey"},
   {"id":"root","shape":"rect","color":"white"},
   {"id":"node","shape":"ellipse","color":"white"},
   {"id":"leaf","shape":"circle","color":"white"},
   {"id":"user","shape":"polygon","sides":"3","color":"white"},
   {"id":"user2","shape":"polygon","sides":"5","color":"white"}
]
```

Figura 4.2: Rappresentazione della Signature in JSON

4.3.1.3 Codifica dei Nodi

Per la definizione dei nodi si è scelto di utilizzare la codifica standard utilizzata da D3.js per descrivere strutture gerarchiche (ad esempio per la visualizzazione di strutture ad albero). Questa scelta è stata particolarmente utile perché ha poi permesso l'utilizzo di tutta una serie di metodi offerti dalla libreria quali ad esempio la possibilità di conoscere il numero di discendenti di un dato nodo. La codifica ricalca quella illustrata in figura 4.4.

4.3.1.4 Codifica dei Collegamenti

La codifica dei collegamenti è stata influenzata dalla necessita di descrivere anche i collegamenti a ipergrafo. La libreria D3 non dispone di questo tipo di collegamento (la libreria si limita a definire connessioni a grafo semplice tra 2 nodi codificandole con la notazione *"source"*, *"target"*) e la codifica generalmente utilizzata dalla libreria non soddisfa i requisiti del progetto. Per descrivere il collegamento a ipergrafo si è deciso di descrivere i nodi raggiunti da un collegamento mediante la creazione di un array di nodi: ad esempio ogni nodo raggiunto dal collegamento c sarà inserito nell'array che lo descrive. In figura 4.3 viene illustrato il codice JSON che descrive i vari collegamenti, per omogeneità anche i collegamenti che interessano solo 2 nodi sono descritti con la stessa notazione.

```
"links":
[
   ["server1900","server992","server999"],
   ["server984","server983"],
   ["server984","server999","server911","server946","server913","server912"]
]
```

Figura 4.3: Rappresentazione dei collegamenti in JSON

4.3.2 Codifica e visualizzazione del Place Graph

Per la visualizzazione dei Place Graph si è fatto uso, vista la disponibilità di numerosi pacchetti di visualizzazione di strutture dati gerarchiche e alberi, di una visualizzazione fornita dalla libreria D3. In particolare si è fatto uso del *layout* **D3.tree** che permette di creare con facilità la visualizzazione ad albero di una struttura dati propriamente formattata.

I dati codificati in *JSON* hanno la struttura illustrata in figura 4.4; si noti come ogni nodo padre possiede un array di nodi figli al suo interno, le foglie dell'albero invece presentano il solo attributo *name* che li identifica. Una struttura dati di questo tipo passata a D3 comporta la creazione da parte della libreria di oggetti *nodo* per ogni elemento della struttura dati con al loro interno diversi attributi come l'altezza del nodo, la sua profondità, il suo ID, la sua posizione nello schermo e altro ancora facilitando la creazione della visualizzazione dell'albero. Ad esempio in figura 4.5 viene illustrato come per ogni nodo caricato dal file JSON viene calcolata la sua posizione a schermo (regolata attraverso il campo *attribute* del tag HTML *g* che lo contiene: questo viene fatto prima per ogni nodo radice (che viene restituito dalla funzione *d3.hierarchy*) e successivamente per ogni nodo discendente. Il risultato ottenuto è visualizzato in figura 4.6

```
{
  "name": "Eve",
  "children": [
    {
      "name": "Cain"
    },
    {
      "name": "Seth",
      "children": [
        {
          "name": "Enos"
        },
        {
          "name": "Noam"
        }
      ]
    }
    ...
```

Figura 4.4: Esempio di codifica di struttura dati gerarchica in JSON

```
var nodes = treeData.nodes;
  nodes.forEach(function(d){
   var shift = (margin.top+(height/nodes.length)*index);
   var  g = svg.append("g")
     .attr("transform", "translate(" + margin.left + "," + shift + ")");
   index++;
   var tree = d3.tree()
         .size([height/nodes.length,width]);
         var root = d3.hierarchy(d);
         tree(root);

   var node = g.selectAll(".node")
         .data(root.descendants())
         .enter().append("g")
         .attr("class", function(d) { return "node" + (d.children ? " node--internal" : "
             node--leaf"); })
         .attr("transform", function(d) {
           return "translate(" + d.y + "," + d.x + ")";
      });
```

Figura 4.5: Parte del codice di realizzazione dell'albero con D3.js

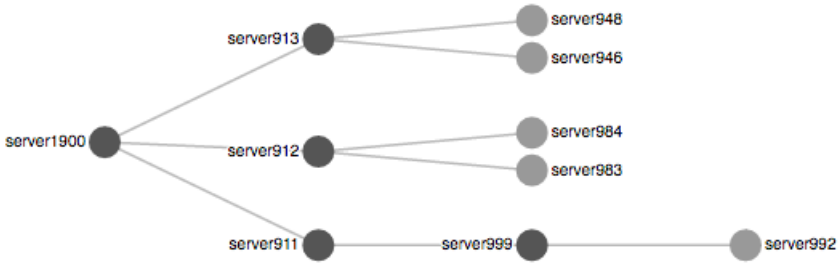


Figura 4.6: Visualizzazione dell'albero

4.3.3 Codifica e visualizzazione del Link Graph

La visualizzazione del Link Graph si è rivelata particolarmente impegnativa in quanto la libreria D3 non ha ancora sviluppato la possibilità di creare collegamenti tra nodi attraverso l'uso di **ipergrafi**. Un *ipergrafo* è una coppia (V, E) ove V è un insieme finito ed E è un insieme di sottoinsiemi di V di ogni cardinalità (quindi non necessariamente 2). Ad esempio se consideriamo la coppia di insiemi (V, E):

$$V = \{v_1, v_2, v_3, v_4\} \quad E = \{\{v_1, v_2\}, \{v_2, v_3, v_4\}, \{v_3\}\}$$

avremo come risultato l'ipergrafo in figura 4.7 Per ovviare all'assenza di una primitiva offerta dalla libreria è stato realizzato un pacchetto per la creazione di collegamenti con *ipergrafo*. Per realizzare la connessione si è resa necessaria la creazione di una nuova classe di nodi detti *"di collegamento"*: è stata verificata ad ogni sottoinsieme e la condizione $e \in E : |e| > 2$: se la condizione è vera allora un nodo *"di collegamento"* viene aggiunto al sottoinsieme di nodi e e viene creato un collegamento a grafo generico che la libreria *D3* fornisce tra i nodi contenuti in e e il nodo *"di collegamento"*. Il riconoscimento di questa classe di nodi *"di collegamento"* permette successivamente un differente stile di visualizzazione da parte della libreria creando l'effetto di collegamento con *"ipergrafo"*. Il codice in figura 4.8 illustra quanto descritto precedentemente.

Il pacchetto **D3-hypergraph** realizzato per questo lavoro è al momento l'unica proposta di collegamento con *ipergrafi* all'interno della libreria D3.js. Il pacchetto è stato reso pubblico ed è scaricabile[6] ed utilizzabile dalla comunità D3.js.

Per la visualizzazione del grafo (comprensivo di nodi, collegamenti e ipergrafi) si è utilizzato un layout già definito dalla libreria D3 chiamato **Force Layout**. Questo layout prevede una simulazione fisica del tipo *"massa-molla"* ed è possibile definire tutta una serie di forze all'interno della simulazione come, ad esempio, una forza repulsiva tra i nodi del grafo per evitare sovrapposizioni tra essi oppure la risposta elastica esercitata dai collegamenti tra nodi impostando il grado di elasticità delle connessioni. Questo tipo di layout consente quindi lo spostamento dei nodi all'interno dello spazio della simulazione e permette la modifica della configurazione del grafo come richiesto nei requisiti di progetto. In figura 4.9 è possibile apprezzare visivamente il risultato ottenuto.

[6]Per scaricare il pacchetto visitare la pagina https://github.com/AndreaSimeone/d3-hypergraph oppure cercare *D3-hypergraph* tramite **npm** (node package manager)

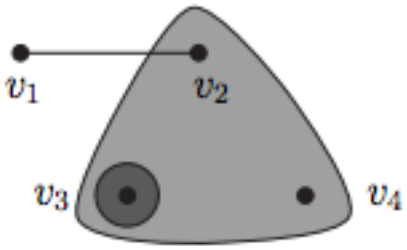


Figura 4.7: Esempio di ipergrafo

```
links.forEach(function(d) {
   //if link length >2 there's an Hyperlink: i need to create a connection node
   if (d.length > 2) {
   //connection node id creation
      var   id = 'ln';
      for(k = 0; k < d.length; k++) {
         id += d[k];
      }
   //connection node creation
      i = {id: id,link: true};
   //add the connection node to the node array
      nodes.push(i);
   //creation of the link from every node of the connection set to the connection node
      for (j = 0; j < d.length; j++) {
         hyper.push({source: d[j], target: i.id});
      }
   }else{
   //if link < 2 then the connection is the traditional one w/o connection node
      hyper.push({source: d[0],target: d[1]});
   }
});
```

Figura 4.8: Parte del codice per la realizzazione del pacchetto D3-hypergraph

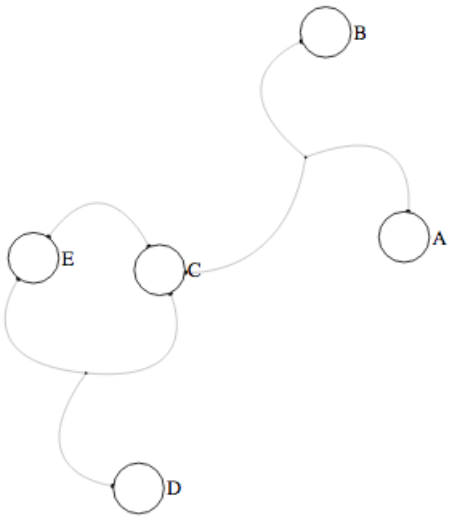


Figura 4.9: Esempio di grafo ottenuto con layout Force e il pacchetto D3-hypergraph

4.3.4 Visualizzazione Bigrafo

Ottenere la visualizzazione del bigrafo, data la sua specificità, ha richiesto un notevole lavoro di customizzazione dovuto dall'impossibilità di sfruttare layout D3 già pronti per la visualizzazione di grafi con gerarchie tra nodi. La soluzione pensata richiede l'utilizzo di più istanze del layout **Force** all'interno della stessa visualizzazione: ad esclusione dei nodi radice degli alberi presenti nella visualizzazione ogni insieme di nodi figli viene gestito da un differente layout Force con forze opportunamente calibrate in modo tale da permettere il rendering dei nodi figli all'interno dei nodi padri riuscendo così a creare la tipica visualizzazione dei bigrafi dove l'inclusione esprime gerarchia. Per la creazione dei collegamenti con ipergrafo si è utilizzato il pacchetto **D3-hypergraph** precedentemente utilizzato nella visualizzazione dei *Link Graph*. Si illustrano ora con l'aiuto del codice le fasi essenziali della creazione della visualizzazione.

4.3.4.1 Creazione e visualizzazione radici

Per visualizzare i nodi si è innanzitutto ordinato l'array contenente i nodi in base alla loro profondità; i nodi con profondità 0 sono le nostre radici. I nodi radice vengono quindi passati al *Force Layout* che gestisce la creazione e visualizzazione dei nodi radice. La dimensione dell'elemento grafico viene determinata dal numero di discendenti della radice tramite il metodo *node.descendants* fornito dalla libreria D3.

```
//ordino i nodi in base alla profondità
   nodes.sort(function(d,e){
       return d3.ascending(d.depth,e.depth);
   });
   ...
//acquisisco le radici del bigrafo
   var nodiRoot =[];
   for (var i=0;i<nodes.length;i++){
       if (nodes[i].depth==0){
      nodiRoot.push(nodes[i])
      }
   }
   ...
//creo visualizzazione dei nodi radice
   var rootNodes = svg.append("g").attr("class","root").selectAll("g").data(nodiRoot).enter()
   .append("g").attr("id",function(d){return d.data.id;});
```

Figura 4.10: Parte del codice per la visualizzazione dei nodi radice

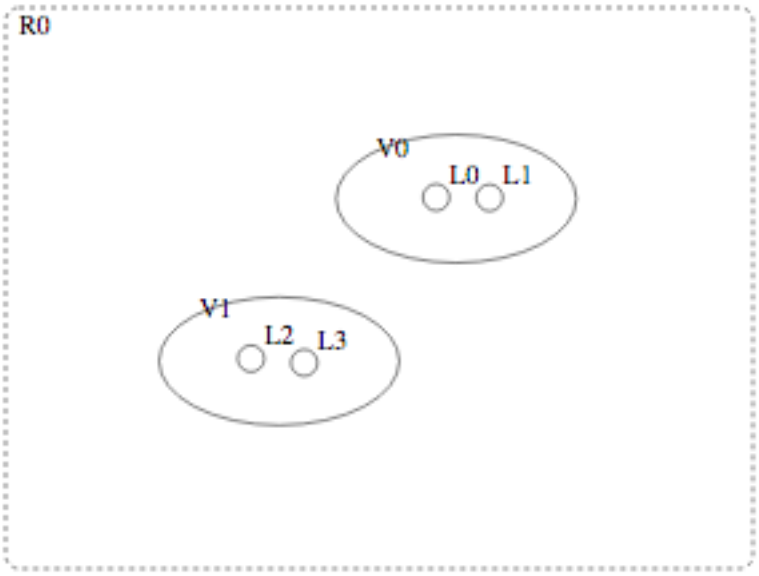
4.3.4.2 Creazione e visualizzazione altri nodi

Per la creazione e visualizzazione di altri nodi si è creata un'opportuna funzione *layoutCreation* che viene invocata per ogni nodo presente nell'array. Se il nodo ha figli la funzione crea un nuovo *Force Layout* con l'array contenente i nodi figli del nodo analizzato come argomento. La funzione inoltre analizza il riferimento di ogni nodo alla signature in modo tale da permettere la creazione dell'elemento grafico da visualizzare. La figura 4.11 illustra il codice *JSON* che descrive le gerarchie tra nodi e il risultato ottenuto con la creazione degli elementi grafici innestati all'interno dell'elemento grafico padre,

in particolare è possibile visualizzare un elemento radice (rettangolo tratteggiato) con 2 figli (elissi); ogni figlio contiene al suo interno i propri elementi figli visualizzati in questo caso con la figura cerchio (le foglie).

```
...
"nodes":[{
   "id": "R0",
   "type":"root",
   "children": [{
      "id": "V0",
      "type":"node",
      "children":[{
         "id": "L0",
         "type":"leaf"},
         {
         "id": "L1",
         "type":"leaf"
         }]
      },
      {
      "id": "V1",
      "type":"node",
      "children":[{
         "id": "L2",
         "type":"leaf"},
         {
         "id": "L3",
         "type":"leaf"
         }]
      }]
   }],
...
```

(a) file Json che descrive la struttura del bigrafo



(b) Esempio di inclusione tra nodi ottenuta con 4 layout Force

Figura 4.11: Esempio di inclusione tra nodi ottenuta con 4 layout Force

4.3.4.3 Creazione e visualizzazione connessioni tra nodi

Per realizzare la connessione tra nodi è stato creato un ulteriore force layout a cui sono stati passati tutti i nodi che compongono il bigrafo. La creazione di queste connessioni ha però creato diversi problemi: abitualmente in D3.js utilizzando il layout force le connessioni vengono realizzate tra nodi appartenenti allo stesso layout mentre in questo caso più layout contribuiscono alla visualizzazione grafica desiderata. In particolare gli effetti del layout che agisce sui nodi radice propaga i suoi effetti di spostamento all'interno dell'area SVG su tutti i nodi successori e così via. Un nodo foglia avrà quindi una sequenza di

trasformazioni determinate da tutti i suoi nodi antenati. Analizziamo in figura 4.12 il markup generato da un force layout in una visualizzazione standard dove non abbiamo gerarchie di nodi e il markup generato dalla visualizzazione del bigrafo: si nota come le trasformazioni che intervengono sul nodo

```
//markup generato da force layout
...
<g transform="translate"(100,100)>
   <circle r="20">
</g>
<g transform="translate"(200,100)>
   <circle r="20">
</g>
...
```

```
//markup generato da force layout multipli
...
<g transform="translate"(100,100)>
   <circle id ="n1" r="20">
   <g transform="translate"(200,100)>
      <circle id ="n2" r="20">
   </g>
</g>
...
```

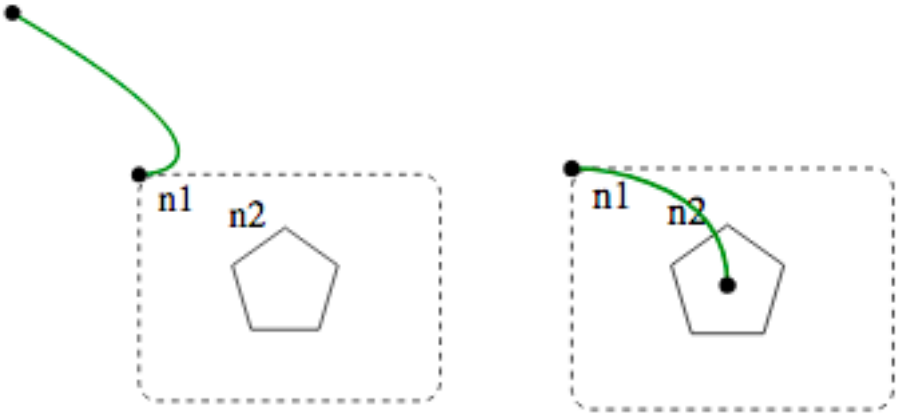
Figura 4.12: Differenze tra markup generato da force layout singolo e multipli.

con id *n2* sono 2 e la somma dei loro effetti determina il reale valore delle coordinate di *n2*: questo comportamento è necessario perché permette l'inclusione all'interno del nodo padre *n1* del nodo figlio *n2* (come desiderato dalla visualizzazione dei bigrafi in cui l'inclusione esprime gerarchia). Questa necessità determina un problema: due layout differenti gestiscono i 2 nodi che vengono rappresentati all'interno della libreria come 2 oggetti in cui le informazioni sulle coordinate sono accessibili mediante le proprietà *nodo.x* e *nodo.y* che però considereranno solo le trasformazioni provocate dal proprio layout force.
Se ad esempio andiamo a verificare le coordinate dei 2 nodi descritti dal markup di figura 4.12 si può notare un'anomalia: si nota come il nodo *n2* nel secondo esempio non considera la trasformazione

```
//lettura proprietà x e y dell'oggetto nodo di n1 e n2 generato da un singolo layout
console.log("n1.x:"+n1.x,"n1.y:"+d[1].y);
>n1.x:100 n1.y:100            //Ok
console.log("n2.x:"+n1.x,"n2.y:"+d[1].y);
>n2.x:200 n2.y:100            //Ok
//lettura proprietà x e y dell'oggetto nodo di n1 e n2 generato da 2 layout
console.log("n1.x:"+n1.x,"n1.y:"+d[1].y);
>n1.x:100 n1.y:100            //Ok
console.log("n2.x:"+n1.x,"n2.y:"+d[1].y);
>n2.x:200 n2.y:100            //Errore!
```

Figura 4.13: Visualizzazione delle proprietà x e y dei nodi

eseguita dal layout del nodo n1: se usassi le coordinate fornite dagli oggetti di *n1* e *n2* per creare una connessione tra i 2 nodi non otterrei quanto desiderato come illustrato in figura 4.14(a).



(a) Link senza trasformazione del nodo padre

(b) Link con trasformazione del nodo padre

Figura 4.14: Esempi di risultati ottenuti senza considerare la trasformazione eseguita dal nodo padre o invece considerandola

Per ovviare a questo problema, determinato dall'uso di molteplici layout e dalla necessità di inclusione di alcuni nodi su altri nodi, si sono create delle nuove proprietà dell'oggetto nodo generato dal layout chiamate $posX$ e $posY$ e una funzione di calcolo della posizione che tenesse conto, per ogni nodo, delle trasformazioni messe in atto dai nodi antenati. Guardiamo nel dettaglio la funzione in figura 4.15: la funzione legge il valore della trasformazione esercitata dall'elemento g che contiene il nodo analizzato e se il nodo ha un padre somma il valore di traslazione letto con il valore di traslazione a cui è stato sotto posto il nodo padre. Questa soluzione permette di ottenere il risultato corretto visibile in figura 4.14(b).

```
...
//leggo il parametro transform dell'elemento html
var string = absNode.getAttribute("transform");
if (string != null)
   translate = string.substring(string.indexOf("(")+1, string.indexOf(")")).split(",");
else{
   translate[0] = 0;
   translate[1] = 0;
}
//se il nodo ha un padre sommo il valore del trasform dei nodi predecessori al nodo in
    esame
if (e.parent){
   var parentNode = filterNodesById(filterNodes(nodesPos),e.parent.data.id);
   translation.x = parseFloat(translate[0]);
   translation.y = parseFloat(translate[1]);
   e.posX = translation.x + parentNode[0].posX;
   e.posY = translation.y + parentNode[0].posY;
} else {
   translation.x = parseFloat(translate[0]);
   translation.y = parseFloat(translate[1]);
   e.posX = translation.x;
   e.posY = translation.y;
}
...
```

Figura 4.15: Funzione per il calcolo della posizione di un nodo

4.3.4.4 Creazione e placing dei terminatori di link

Nella visualizzazione di *link graph* e *bigrafi*, in letteratura, per meglio definire i nodi collegati ad un dato link e visualizzare il numero di collegamenti che raggiungono un nodo vengono utilizzati dei terminatori di collegamento posti ai vertici del collegamento stesso detti *marker*(vedi figura 4.16). *SVG* prevede l'utilizzo dei marcatori e ha definito un tag specifico per la descrizione di questo elemento grafico: il tag <*marker*>. Questo tag definisce la grafica che verrà utilizzata come elemento grafico ai vertici del collegamento. La definizione della geometria del marker deve essere precedente all'utilizzo del marker e SVG prevede un apposito tag <*defs*> per la definizione di oggetti grafici che potranno essere successivamente usati all'interno della rappresentazione. Vediamo un esempio di utilizzo in e il risultato in **??**.

All'interno della visualizzazione del bigrafo la problematica principale è determinata dal corretto posizionamento del marker: SVG non è dotato delle primitive per la rilevazione di intersezioni tra oggetti all'interno della visualizzazione, inoltre le molteplici forme usate all'interno del bigrafo non consentono un calcolo dell'intersezione semplice. *D3.js* a differenza di altre librerie, come ad esempio *Cytoscape.js*[7] non offre il calcolo del posizionamento dei marcatori dPer la risoluzione di questo problema si è reso necessario l'uso di una libreria *Javascript* chiamata *kld-intersection*[8]. L'utilizzo di questa libreria all'interno di questo progetto, presenta un problema: la libreria è distribuita attraverso *npm* e i pacchetti *node* sono specificamente realizzati per l'utilizzo del codice *lato server* mentre l'applicazione di visualizzazione

[7]Per maggiori informazioni: http://www.cytoscape.org/
[8]libreria realizzata da Kevin Lindsey e scaricabile all'indirizzo https://github.com/thelonious/kld-intersections

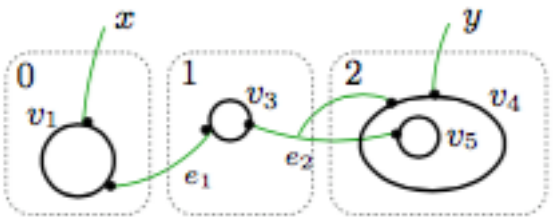


Figura 4.16: Esempio di Bigrafo con utilizzo di marcatori di collegamento

```
<svg width="120" height="120">
//definizione del marker a freccia
  <defs>
    <marker id="Triangle" viewBox="0 0 10 10" refX="1" refY="5" markerWidth="6"
        markerHeight="6" orient="auto">
      <path d="M 0 0 L 10 5 L 0 10 z" />
    </marker>
  </defs>
//utilizzo del marker a freccia al termine dell'elemento polyline
  <polyline points="10,90 50,80 90,20" fill="none" stroke="black"
      stroke-width="2" marker-end="url(#Triangle)" />
</svg>
```

Figura 4.17: Esempio di definizione e utilizzo di un marcatore a freccia con un elemento polyline

risiede nel client (come visto nel Capitolo 2). Per poter utilizzare la libreria è stata necessaria una riscrittura della stessa che permettesse l'accesso del codice che la compone direttamente senza l'uso di soluzioni software quali *Browserify* o *Require.js* che importano direttamente la libreria all'interno del codice della nostra applicazione pregiudicando però la separazione tra codice dell'applicazione e librerie utilizzate.

Risolto il problema di accesso ai servizi offerti dalla libreria il suo utilizzo si è rivelato abbastanza semplice: nella funzione che definisce ad ogni *tick* (unità di tempo della simulazione fisica del force layout) il ridisegno dei collegamenti viene invocata la funzione per il calcolo dell'intersezione tra l'oggetto nodo (e la sua relativa rappresentazione grafica) e il link (nel nostro caso una curva Bézier quadratica). La libreria ricrea al suo interno gli oggetti grafici di cui bisogna calcolare l'intersezione e restituisce un oggetto contenente lo status dell'intersezione (se c'è o non c'è intersezione) e un array contenente le coordinate delle intersezioni trovate; le coordinate restituite vengono quindi utilizzate per il posizionamento del collegamento al cui estremo è disegnato il marcatore a sfera tipico delle rappresentazioni grafiche dei

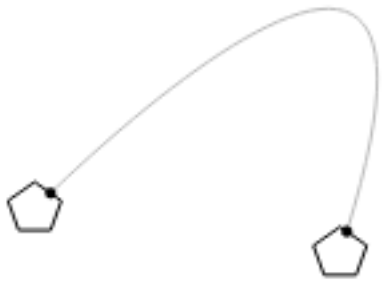


Figura 4.18: Posizionamento dei marcatori a seguito del calcolo delle intersezioni tra figure SVG

bigrafi presenti in letteratura. Il risultato ottenuto è visibile in figura 4.18.

4.4 Esempi di visualizzazione

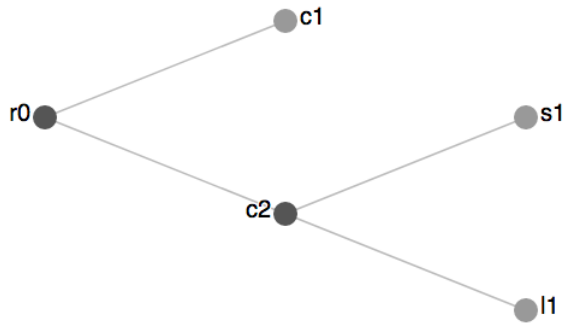
Di seguito vengono riportati alcuni esempi dei risultati ottenuti: per ogni visualizzazione viene fornita la struttura dati che la descrive, il *place graph*, il *link graph* e il *bigrafo* ottenuto.

4.4.1 Bigrafo con una radice

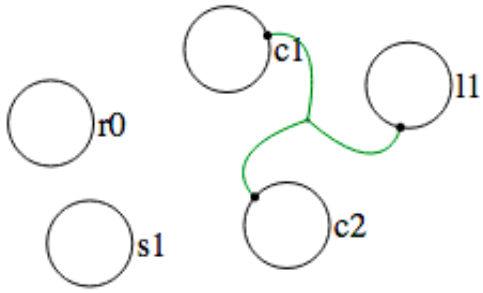
Viene presentato un esempio minimale con una radice e nessun inner e outer name. I risultati sono visibili in figura 4.19

```
{
"interface":{},
"signature":[{"id":"site","shape":"rect","color":"grey"},
        {"id":"root","shape":"rect","color":"white"},
        {"id":"node","shape":"ellipse","color":"white"},
        {"id":"user2","shape":"polygon","sides":"5","color":"white"}],
"nodes":[{
   "id": "r0",
   "type":"root",
   "children": [{
      "id": "c1",
      "type":"user2"
   },{"id": "c2",
      "type":"node",
      "children":[{
         "id":"s1",
         "type":"leaf"
      },{
         "id":"l1",
         "type":"leaf"
   }]}]}],
"links":[["c2","c1","l1"]]}
```

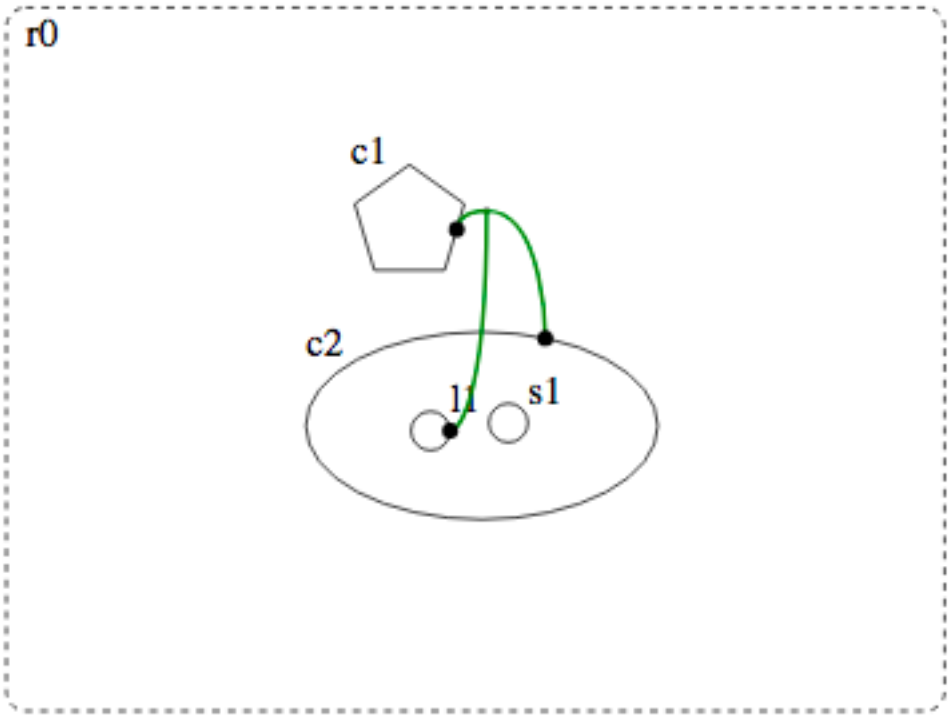
Listing 4.1: Struttura dati dell'esempio



(a) Place Graph



(b) Link Graph



(c) Bigrafo

Figura 4.19: Visualizzazione risultati esempio

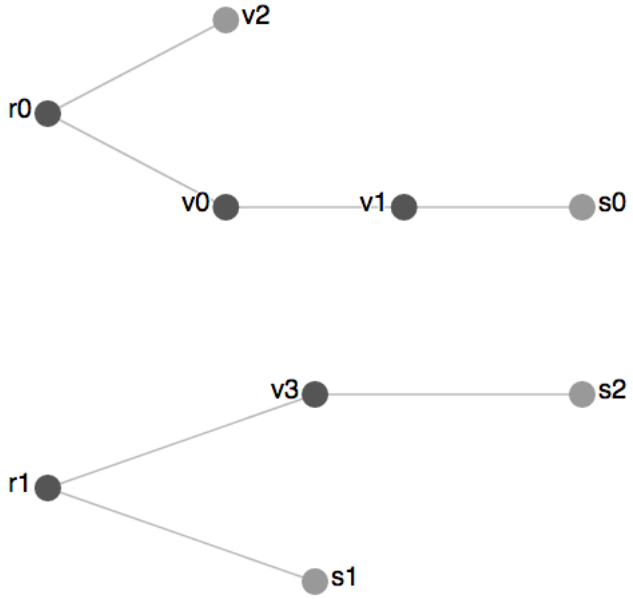
4.4.2 Bigrafo con due radici

Viene ora proposto un esempio più complesso con due radici e inner e outer names. Ripropone il rendering del bigrafo in figura 2.4. I risultati sono visibili in figura 4.20

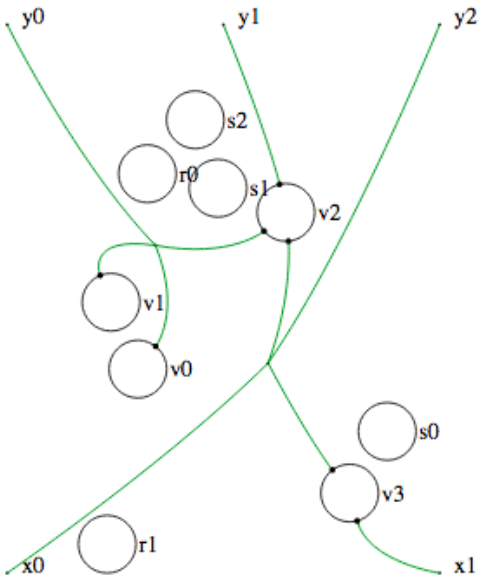
```
{"interface":{"inner":[{"id":"x0"},{"id":"x1"}],
              "outer":[{"id":"y0"},{"id":"y1"},{"id":"y2"}]},
"signature":[{"id":"site","shape":"rect","color":"grey"},
           {"id":"root","shape":"rect","color":"white"},
           {"id":"node","shape":"ellipse","color":"white"},
           {"id":"leaf","shape":"circle","color":"white"}],
"nodes":[{
   "id": "r0",
   "type":"root",
```

```
   "children": [{
      "id": "v2",
      "type":"node"},
      {
      "id": "v0",
      "type":"node",
      "children": [{
         "id": "v1",
         "type":"node",
         "children":[{
            "id": "s0",
            "type":"site"
      }]}]}]},{
   "id": "r1",
   "type":"root",
   "children": [{
      "id": "v3",
      "type":"node",
      "children": [{
         "id": "s2",
         "type":"site"
      }]},{
      "id": "s1",
      "type":"site"
   }]}],
"links":[["v0","v1","v2","y0"],
     ["v2","y1"],
     ["v2","v3","y2","x0"],
     ["x1","v3"]]}
```

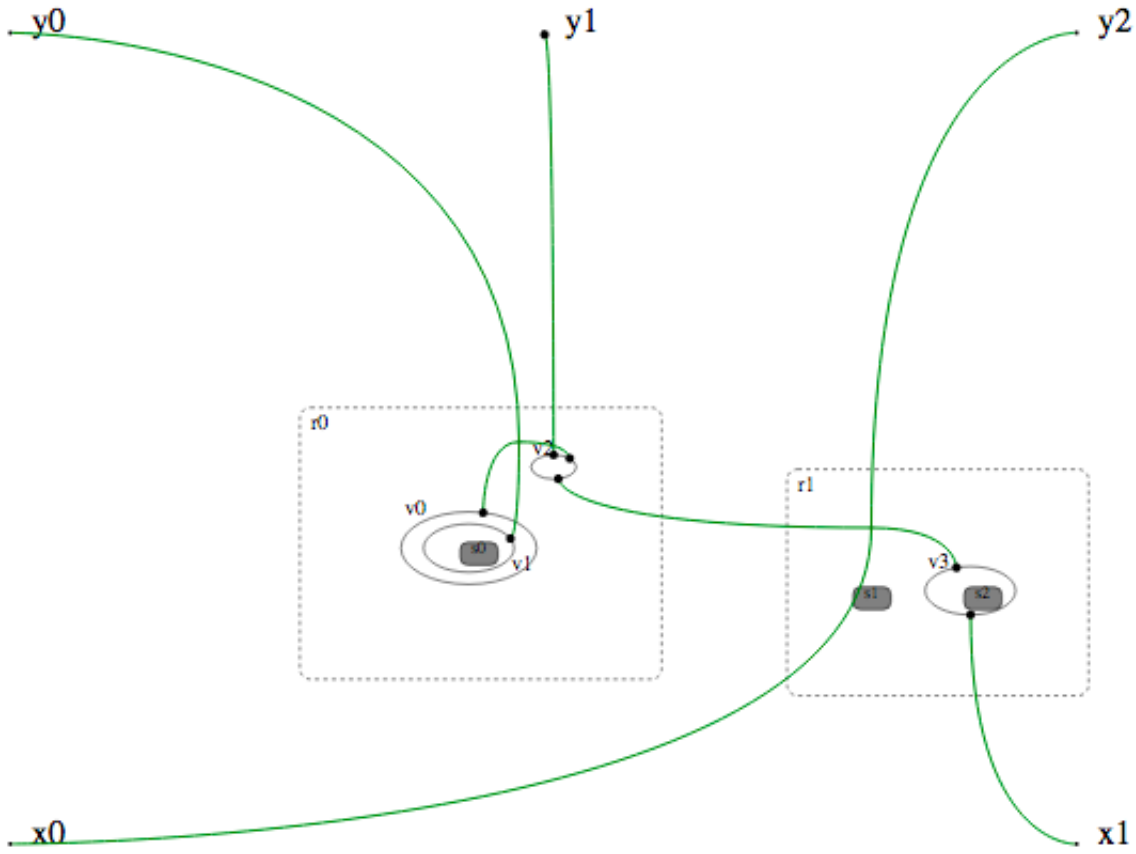
Listing 4.2: Struttura dati del secondo esempio



(a) Place Graph



(b) Link Graph



(c) Bigrafo

Figura 4.20: Visualizzazione risultati esempio

4.4.3 Bigrafo con tre radici

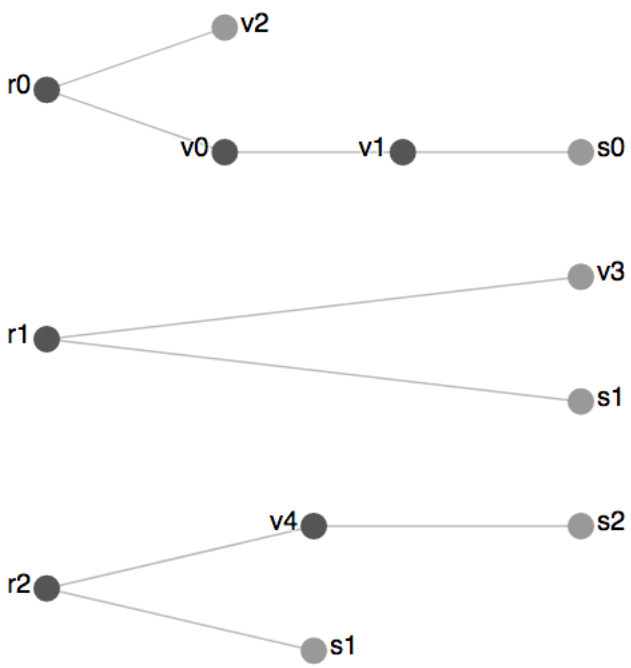
Viene ora proposto un esempio più complesso con tre radici, inner e outer names. I risultati sono visibili in figura 4.21

4.4.3.1 Struttura dati

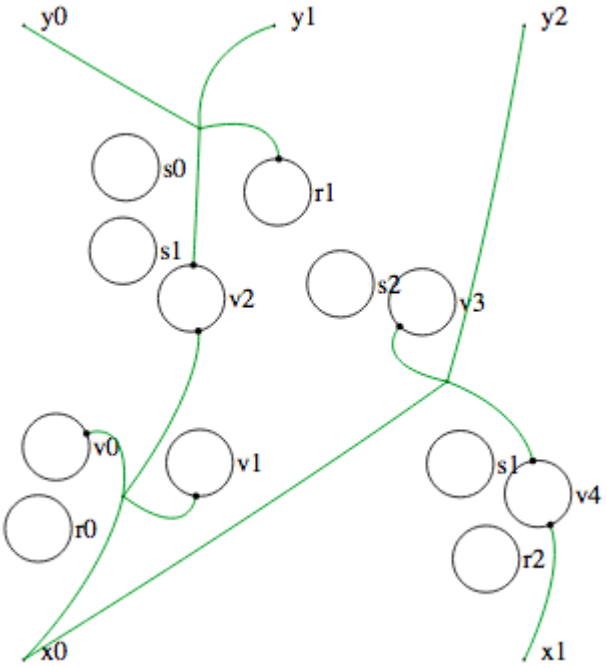

```
"interface":{"inner":[{"id":"x0"},{"id":"x1"}],
          "outer":[{"id":"y0"},{"id":"y1"},{"id":"y2"}]},
"signature":[{"id":"site","shape":"rect","color":"grey"},
        {"id":"root","shape":"rect","color":"white"},
        {"id":"node","shape":"ellipse","color":"white"},
        {"id":"leaf","shape":"circle","color":"white"},
        {"id":"user1","shape":"polygon","sides":"3","color":"white"},
        {"id":"user2","shape":"polygon","sides":"5","color":"white"}],
"nodes":[{
   "id": "r0",
   "type":"root",
   "children": [{
      "id": "v2",
      "type":"user1"},
      {
      "id": "v0",
      "type":"node",
      "children": [{
         "id": "v1",
         "type":"node",
         "children":[{
            "id": "s0",
            "type":"site"
      }]}]}]},
{
   "id": "r1",
   "type":"root",
   "children": [{
      "id": "v3",
      "type":"node"
   },{
      "id": "s1",
      "type":"site"}]
},{
   "id": "r2",
   "type":"root",
   "children": [{
      "id": "v4",
      "type":"user2",
```

```
      "children": [{
         "id": "s2",
         "type":"site"
      }]},{
      "id": "s1",
      "type":"site"
      }]}],
"links":[
   ["v0","v1","v2","x0"],
   ["v2","y1","r1"],
   ["v4","v3","y2","x0"],
   ["x1","v4"]]
}
```

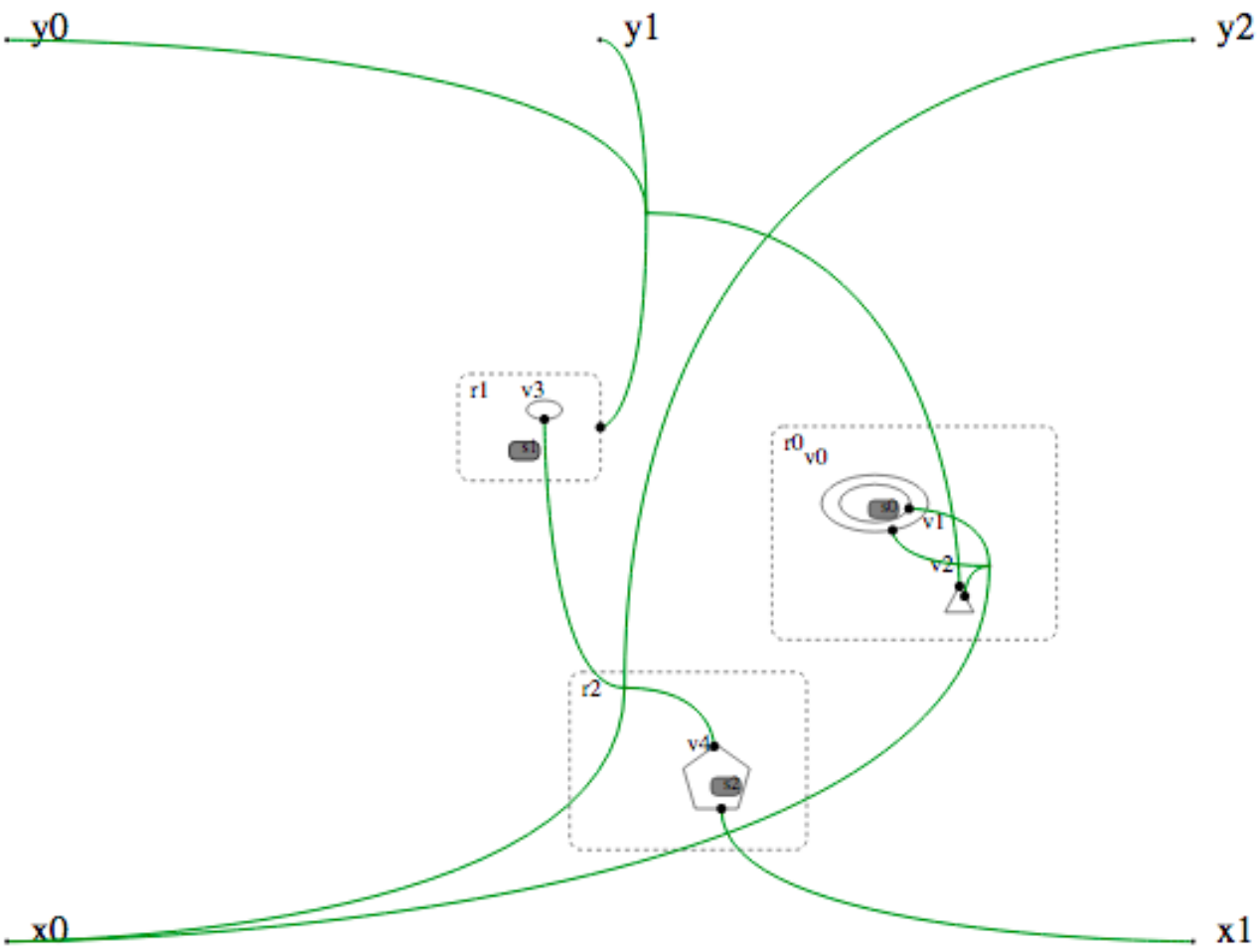
Listing 4.3: Struttura dati del terzo esempio



(a) Place Graph



(b) Link Graph



(c) Bigrafo

Figura 4.21: Visualizzazione risultati esempio

5

Conclusioni

In questa tesi è stata presentata l'implementazione di un sistema di visualizzazione del meta-modello dei bigrafi. Prendendo come riferimento il modello grafico proposto da Milner si è cercato di svilupparlo attraverso l'uso di un browser, SVG e D3.js. Inizialmente si sono definiti nel dettaglio gli aspetti di interesse dei Bigrafi (tipologie di nodi, ipergrafi e interfacce) e le diverse rappresentazioni grafiche da sviluppare in particolare Place Graph, Link Graph, e visualizzazione del Bigrafo come unione delle due visualizzazioni precedenti. Per fare ciò si è definita una struttura dati in JSON che definisse ogni aspetto interessante per la visualizzazione del bigrafo, in particolare la gerarchia tra nodi, la loro forma e il collegamento tra di essi. Successivamente si sono creati, attraverso l'uso della libreria D3.js, le rappresentazioni grafiche di Place Graph, Link Graph e del Bigrafo. In particolare per la visualizzazione di Link Graph e del Bigrafo si è usata una versione customizzata del layout Force di D3.js che permette l'interazione (drag e zoom) tra l'utente e la rappresentazione grafica del modello.

I requisiti descritti in fase di analisi sono stati soddisfatti, permettendo di ottenere un risultato finale corrispondente alle aspettative iniziali. Le azioni disponibili all'utente (zoom e dragging) sono state implementate mentre il pinning non è stato reputato interessante ai fini dell'interazione con il link graph e il bigrafo.

l progetto di tesi ha rappresentato una sfida importante data l'impossibilità di ottenere con gli strumenti offerti di default dalla libreria D3.js quanto era necessario al raggiungimento dei requisiti di progetto. Problemi come la creazione dell'ipergrafo, la gestione di force layout multipli e il posizionamento dei marcatori si sono rivelati tutt'altro che banali e la loro soluzione ha rappresentato un momento di crescita personale importante. Il progetto di tesi ha inoltre permesso la creazione di un nuovo pacchetto utilizzabile dagli utenti della libreria D3.js (d3-Hypergraph) e il futuro rilascio di un pacchetto per la gestione dei marcatori di collegamento sempre per D3.js.

Alcuni sviluppi possibili del lavoro svolto riguardano la possibilità di visualizzare anche la trasformazione di bigrafi del tipo BRS (Bigraphical Reactive Systems) o quantomeno permettere una modifica della visualizzazione del bigrafo al modificarsi del file JSON che lo descrive. Dotando il bigrafo di una palette di immagini sarà possibile creare visualizzazioni semanticamente più vicine al contesto preso in esame dal bigrafo.

Grazie al lavoro svolto in questa tesi è ora possibile visualizzare un bigrafo con estrema facilità utilizzando un moderno browser web, permettendo la visualizzazione dei bigrafi potenzialmente su qualsiasi dispositivo.

Bibliografia

[1] Giorgio Bacci, Davide Grohmann, e Marino Miculan. Dbtk: a toolkit for directed bigraphs. In *CALCO 2009 Conference Proceedings - Calco Tools*, volume 5728 di *LNCS*. Springer, 2009.

[2] Giorgio Bacci, Davide Grohmann, e Marino Miculan. A framework for protein and membrane interactions In *Proc. MeCBIC'09*. A cura di Gabriel Ciobanu, volume 11 di *EPTCS*, 2009.

[3] Søren Debois e Troels C Damgaard. Bigraphs by example. Relazione tecnica, Citeseer, 2005.

[4] Alexander John Faithfull, Gian Perrone, e Thomas T Hildebrandt. Big red: A development environment for bigraphs. *Electronic Communications of the EASST*, 61, 2013.

[5] Robin Milner. Bigraphs for petri nets. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*, pp. 686–701, 2003.

[6] Robin Milner. Pure bigraphs: Structure and dynamics. *Information and computation*, 204(1):60–122, 2006.

[7] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[8] Scott Murray. *Interactive Data Visualization for the Web*. O'Reilly Media, Inc., 2013.