

Università degli Studi di Salerno  
Dipartimento di Informatica



Corso di Laurea in Informatica

# TiveJS

Un'applicazione Javascript per il riconoscimento di  
linguaggi diagrammatici

A Javascript application for the recognition of  
diagrammatic languages

## **Relatori**

Prof. Gennaro Costagliola  
Dott. Mattia De Rosa

## **Candidato**

Dario Tecchia  
Matr. 0512102581

*Questa tesi la dedico alla mia famiglia.*

# Ringraziamenti

Innanzitutto vorrei ringraziare la mia famiglia per avermi permesso tutto ciò e per il sostegno datomi in questi anni. In particolare vorrei ringraziare mia madre.

Ringrazio il Professor Costagliola per avermi dato la possibilità di collaborare con lui, ringrazio anche il Dottor De Rosa per il supporto datomi durante lo svolgimento di questa tesi.

Ringrazio i miei cari amici e colleghi di informatica. Ringrazio i ragazzi del Centro sociale Allegretti, dimora del capitano Beppino Liberato.

Infine vorrei ringraziare Giulia, la mia ragazza, per avermi sopportato e sostenuto ogni giorno.

# Abstract

La comunicazione visiva è in molti casi più diretta ed immediata rispetto alla comunicazione verbale: disegni, foto e mappe sono esempi di frasi visive che necessitano di un contesto per essere descritte in modo naturale.

In questa tesi presento TiveJS, un'estensione della piattaforma draw.io, che sfrutta simboli e definizioni semantiche per il riconoscimento dei linguaggi diagrammatici e la traduzione di questi in altri linguaggi. Il tool applica delle definizioni semantiche ad un diagramma e restituisce una traduzione di quest'ultimo. La traduzione avviene attraverso due fasi principali: il riconoscimento del grafo e l'applicazione delle definizioni.

Il mio lavoro di tesi si basa su strumenti precedentemente sviluppati: LoCoModeler e TiVE. Precedentemente suddiviso in lato client e lato server, TiVE è stato re-implementato completamente in JavaScript, prendendo il nome di TiveJS, eliminando così la necessità del server.

# Indice

Ringraziamenti	ii
Abstract	iii
Elenco delle figure	vi
Elenco delle tabelle	vii
<b>1 Introduzione</b>	<b>1</b>
1.1 Motivazioni . . . . .	1
1.2 Organizzazione della Tesi . . . . .	2
<b>2 Lavori Correlati</b>	<b>3</b>
2.1 Draw.io e mxGraph . . . . .	3
2.2 LoCoMoTiVE . . . . .	4
2.2.1 LoCoModeler . . . . .	4
2.2.2 TiVE . . . . .	5
2.3 DrawSE . . . . .	6
<b>3 Linguaggi visuali</b>	<b>8</b>
3.1 Linguaggio verbale . . . . .	8
3.2 Componenti . . . . .	8
3.3 Vantaggi . . . . .	9
<b>4 Local Context</b>	<b>10</b>
4.1 Sintassi . . . . .	10
4.2 Semantica . . . . .	12

---

<b>5</b>	<b>TiveJS</b>	<b>15</b>
5.1	Funzionamento e Implementazione . . . . .	15
5.1.1	SGPath . . . . .	16
5.1.2	Progettazione di sentenze visive . . . . .	17
5.1.3	Riconoscimento grafo . . . . .	18
5.1.4	Applicazione definizioni . . . . .	20
5.1.5	Applicazione Regole Semantiche . . . . .	23
5.2	Tecnologie Utilizzate: JavaScript . . . . .	28
5.2.1	Punti di Forza di JavaScript . . . . .	28
5.2.2	Principali Applicazioni . . . . .	29
5.3	Librerie utilizzate nel Progetto . . . . .	29
<b>6</b>	<b>Contesti d'utilizzo</b>	<b>30</b>
6.1	Entitiy Relationship . . . . .	30
6.2	Flowchart . . . . .	30
6.3	Tree . . . . .	30
<b>7</b>	<b>Conclusioni e Sviluppi Futuri</b>	<b>31</b>
	<b>Appendice A Codici</b>	<b>32</b>
	<b>Bibliografia</b>	<b>38</b>

# Elenco delle figure

2.1	Schermata di draw.io . . . . .	3
2.2	Schermata di LoCoModeler . . . . .	4
2.3	Schermata di TiVE . . . . .	5
2.4	Switch per la selezione della modalità in drawSE . . . . .	6
3.1	Esempio di sentenza visiva, da [6] . . . . .	9
4.1	Diagramma ER . . . . .	14
5.1	Diagramma del funzionamento di TiveJS . . . . .	15
5.2	Schermata principale di TiveJS . . . . .	18
5.3	Menu dei comandi per TiVE . . . . .	19
5.4	TiveJS con la traduzione semantica di un diagramma entità-relazione mostrata in una console (sulla destra) . . . . .	27
5.5	TiveJS con gli errori mostrati in una console (sulla destra) . . . . .	27

# Elenco delle tabelle

4.1	Specifica di un linguaggio, nel particolare di un Albero . . . . .	11
4.2	Specifica LCSD di un diagramma ER, costruita sulla specifica sintattica.	13
5.1	Specifica di un diagramma ER . . . . .	19
5.2	Specifica di un Albero . . . . .	20
5.3	Visit Table per il linguaggio Flowchart . . . . .	24
5.4	Path Table per il linguaggio Tree . . . . .	24



# Capitolo 1

## Introduzione

La comunicazione fra individui avviene in svariati modi, ad esempio attraverso il linguaggio verbale ed il linguaggio visivo (o linguaggio visuale).

Un linguaggio visuale non è altro che una forma di comunicazione, detta comunicazione visuale, che fa uso di simboli grafici o immagini. Simboli grafici, immagini e mappe sono esempi di elementi utilizzati all'interno della comunicazione visiva (o comunicazione visuale) che necessitano di un contesto per essere descritte in modo naturale. Spesso quest'ultima risulta essere molto più immediata e di facile comprensione rispetto alla tradizionale comunicazione verbale composta di lettere e parole.

In questa tesi presento **TiveJS**, un'estensione della piattaforma draw.io, che sfrutta simboli e definizioni semantiche per il riconoscimento dei linguaggi diagrammatici e la traduzione di questi in altri linguaggi.

### 1.1 Motivazioni

La piattaforma già esistente, LoCoMoTiVE, si basa su un meccanismo client-server. Il client è formato dalla piattaforma draw.io, opportunamente modificata, per la creazione di sentenze visuali. Il server è stato implementato in Java utilizzando i servlet per il riconoscimento e la traduzione delle sentenze visuali. Il funzionamento è molto semplice: il client esegue una chiamata HTTP di tipo POST contenute al suo interno un grafo o un diagramma, creato attraverso l'utilizzo di simboli ad hoc, in formato XML. Una volta ricevuta la sentenza visuale, essa è interpretata dal server che applica le definizioni per poi restituire la traduzione semantica oppure dei messaggi di errore. Le motivazioni che hanno portato alla creazione di un nuovo tool sono varie: rendere l'applicazione più scalabile e più veloce limitando l'interazione con il server a semplici

accessi a pagine statiche; l'aggiornamento di TiVe all'ultima versione di draw.io; essendo il core di TiveJS scritto completamente in JavaScript, ora si integra perfettamente con la piattaforma estesa e con la manipolazione del grafo. Le definizioni dei linguaggi ora sono in formato JSON rendendo ancora più alta l'interoperabilità dei sistemi.

## 1.2 Organizzazione della Tesi

Nel capitolo 2 illustrerò i lavori correlati al mio progetto di tesi. Nel capitolo 3 tratterò dei linguaggi visuali e dei loro componenti fondamentali. Nel capitolo 4 parlerò del Local Context e delle corrispondenti specifiche sintattiche e semantiche. Nel capitolo 5 introdurrò il risultato del mio lavoro di tesi, TiveJS e le sue funzioni, e illustrerò i dettagli dell'implementazione e le tecnologie usate. Nel capitolo 6 illustrerò vari casi d'utilizzo da me studiati. Nel capitolo 7 presenterò possibili sviluppi futuri dell'applicazione e le conclusioni.

# Capitolo 2

## Lavori Correlati

Il mio lavoro di tesi, essendo principalmente un porting ed una rivisitazione, si basa su strumenti precedentemente sviluppati. Gli strumenti in questione sono descritti nel dettaglio nei seguenti sotto-paragrafi.

### 2.1 Draw.io e mxGraph

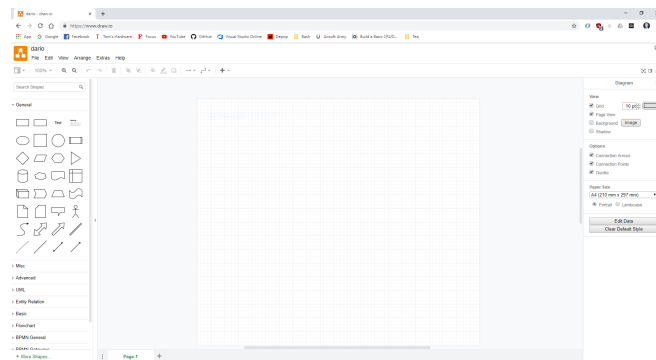


Figura 2.1 Schermata di draw.io

TiveJS, è basato su draw.io, un'applicazione web gratis che permette agli utenti di creare diagrammi e grafi direttamente dal proprio browser web, mostrato in figura 2.1. Ha un'integrazione con Google Drive e Dropbox per il salvataggio di dati che può avvenire anche con l'ausilio del **localStorage** del browser o attraverso il salvataggio di file sulla macchina. Draw.io è basato sulla libreria mxGraph. Il software è stato sviluppato nel 2005 dalla JGraph Ltd.

## 2.2 LoCoMoTiVE

All'attuale stato dell'arte vi è l'ecosistema LoCoMoTiVE, ovvero un'unione di due software, LoCoModeler e TiVE. Presentato in [7] e [8], questo tool permette l'analisi semantica basata sul contesto locale, illustrata nel dettaglio nel capitolo 4. Nei prossimi due paragrafi andrò ad illustrare singolarmente i due componenti di cui è composto.

### 2.2.1 LoCoModeler

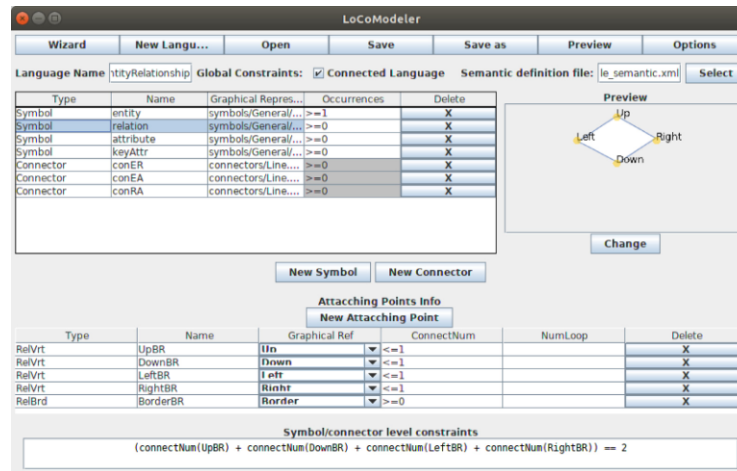


Figura 2.2 Schermata di LoCoModeler

Come descritto in [7], il modulo LoCoModeler consente ai designer la creazione e la modifica del linguaggio visivo in base al contesto locale, in maniera rapida e facile. Il suo output è la definizione in formato XML del linguaggio che verrà utilizzato durante il riconoscimento dei diagrammi. Una volta che il progettista ha completato la specifica del linguaggio, può compilarla in un ambiente web (il modulo TiVE) per consentire agli utenti di disegnare frasi e verificarne la correttezza. Durante la definizione del linguaggio, questa funzione consente al progettista di controllare la correttezza delle specifiche.

Una schermata principale dell'interfaccia grafica del tool è mostrata nella Figura 2.2. Le sue componenti principali sono:

- Una casella di testo contenente il nome del linguaggio e una checkbox che sta ad indicare se il diagramma o grafo deve essere o non essere necessariamente connesso<sup>1</sup>.

<sup>1</sup>Un grafo è detto connesso se, per ogni coppia di vertici  $(u, v) \in V$ , esiste un cammino che collega  $u$  a  $v$ .

- Una tabella riportante le informazioni principali dei simboli e dei connettori inclusi nel linguaggio. E' possibile modificare o eliminare un elemento interagendo con la riga di questo. L'utente può aggiungere nuovi simboli o connettori usando i bottoni sottostanti la tabella.
- Un pannello (sulla destra) mostra un'anteprima grafica del simbolo o connettore selezionato nella tabella. E' possibile cambiare la rappresentazione grafica dell'elemento utilizzando il bottone *Change*.
- Una tabella (al centro) mostra le informazioni relative al simbolo o connettore selezionato. Ogni riga mostra un punto d'attacco e i relativi vincoli. E' possibile aggiungere nuove righe utilizzando i bottoni sovrastanti la tabella.
- Un'area di testo dove è possibile specificare i vincoli per il simbolo o il connettore attraverso espressioni simili al  $C^2$ .

La definizione di un nuovo linguaggio può avvenire grazie all'ausilio di un *Wizard* diviso in tre fasi.

### 2.2.2 TiVE

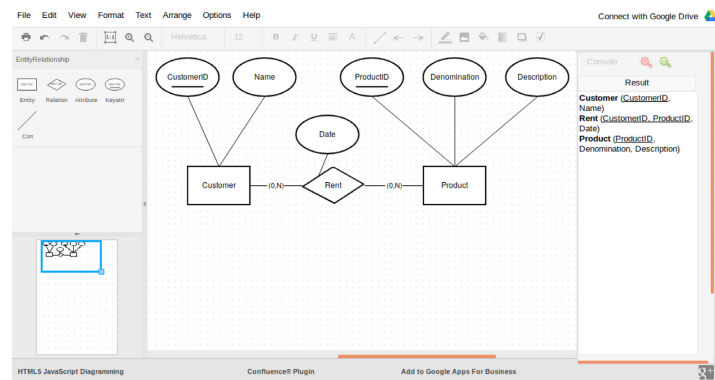


Figura 2.3 Schermata di TiVE

Una volta definito il linguaggio, i diagrammi possono essere composti utilizzando i simboli e i connettori definiti nella sua specifica. Questo può essere fatto attraverso un editor grafico TiVE, che è un'applicazione web che consente la composizione di diagrammi direttamente dal browser web.

Come mostrato nella Figura 2.3, l'applicazione è costituita da tre sezioni principali:

---

<sup>2</sup>Linguaggio di programmazione.

- Nella toolbar a sinistra troviamo la palette dei simboli e connettori utilizzabili per la creazione dei diagrammi.
- Nella zona centrale troviamo l'area di lavoro dove è possibile comporre i diagrammi trascinando gli elementi contenuti nella toolbar di sinistra.
- A destra troviamo la Console dove verrà mostrata la traduzione semantica o la lista di errori nel caso in cui si verificassero.

## 2.3 DrawSE

DrawSE è un'estensione di draw.io che permette la creazione di diagrammi con simboli altamente personalizzati. Le principali caratteristiche di drawSE sono le due modalità di editing: una per la creazione di simboli (*Shape Mode*) e una per la definizione dei punti d'attacco<sup>3</sup> dei simboli (*AP Mode*). Le due modalità sono attivabili attraverso il selettore mostrato in figura 2.4

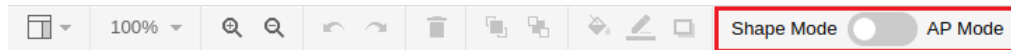


Figura 2.4 Switch per la selezione della modalità in drawSE

La *Shape Mode* permette la creazione di nuovi simboli e di personalizzarli in base al colore, lo spessore delle linee e così via. Un simbolo può essere formato anche dall'unione di più simboli semplici. Nella *AP Mode*, drawSE fornisce una palette con sette strumenti utili alla definizione degli *attaching point* del simbolo. Il punto d'attacco può essere composto da sette diverse forme geometriche:

- un punto
- una linea retta
- una linea curva
- un'area rettangolare
- un'area ellittica
- un contorno rettangolare

---

<sup>3</sup>Dove gli archi andranno ad attaccarsi sulla figura.

- un contorno ellittico

Altra funzionalità di drawSE è la creazione di set di simboli personalizzati o *custom palette*, ovvero un insieme dei simboli creati grazie alle due modalità specificate prima. Lo strumento è spiegato nel dettaglio in [5]. TiveJS fa uso delle palette generate in drawSE.

# Capitolo 3

## Linguaggi visuali

Nei precedenti capitoli ho parlato spesso di linguaggi verbali e linguaggi visuali e di quanto la comunicazione visuale può essere più efficiente della comunicazione verbale. In questo capitolo entrerò nel dettaglio senza dilungarmi, andando ad illustrare quali sono le principali differenze tra un linguaggio visuale ed uno verbale, i componenti che lo compongono e in quali casi o contesti un linguaggio visivo è più efficace rispetto ad uno verbale.

### 3.1 Linguaggio verbale

Il linguaggio verbale è un gruppo di elementi, come suoni e parole, che messe insieme formano frasi e infine permettono la comunicazione fra individui. Da questo deriva la comunicazione verbale che è quindi costituita dalle parole usate quando parliamo o scriviamo.

### 3.2 Componenti

Ogni linguaggio è formato da un proprio insieme di componenti. Un linguaggio visuale si distingue principalmente dal linguaggio verbale per i componenti da cui è formato. Il linguaggio visivo si basa su simboli grafici o immagini, elementi che il cervello umano interpreta e trasforma in concetti, linguaggio verbale ed emozioni. Quindi se il linguaggio visuale è costituito da testo e parole per la formazione di frasi, il linguaggio visivo è formato da simboli e disegni per formare sentenze visive. Una componente fondamentale di un linguaggio visuale è il contesto che viene dato ad ogni simbolo appartenente ad una frase visiva.



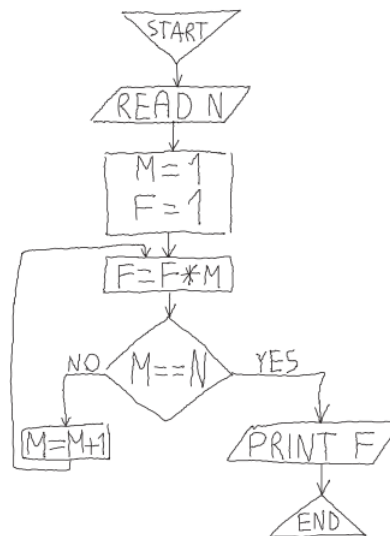


Figura 3.1 Esempio di sentenza visiva, da [6]

Ad esempio, come possiamo notare avvalendoci del disegno in figura 3.1, senza un contesto sono semplici simboli connessi fra loro da delle frecce. Invece, dando una definizione ai simboli del diagramma può essere interpretato come un diagramma di flusso o *Flowchart*<sup>1</sup>.

### 3.3 Vantaggi

I vantaggi possono essere molteplici, innanzitutto un linguaggio visuale può essere molto più efficace e di facile comprensione rispetto al linguaggio verbale per via della sua semplicità e naturalità. Non ha lingue o convenzioni in quanto un disegno o un'immagine non dipende da lingue o standard.

---

<sup>1</sup>Il diagramma di flusso (o *Flowchart*), in informatica, è una rappresentazione grafica delle operazioni da eseguire per l'esecuzione di un algoritmo.

# Capitolo 4

## Local Context

Il riconoscimento dei diagrammi viene effettuato in base al *Local Context*, presentato nei paper [6], [7] e [8].

Nel paper *Local context-based recognition of sketched diagrams* [6], il Local Context viene presentato come una nuova metodologia mirata alla creazione e all'implementazione di un framework<sup>1</sup> per il riconoscimento e l'interpretazione dei diagrammi. Nello specifico, i diagrammi possono contenere differenti elementi grafici quali simboli, connettori e testo. Una volta che i simboli sono stati identificati, il riconoscimento procede identificando il contesto locale di ogni simbolo. Il Local Context ha due diverse specifiche: sintattiche e semantiche.

### 4.1 Sintassi

Sempre in [6], viene definita la specifica sintattica di un linguaggio visuale. In particolare, usando il contesto locale, la specificazione della sintassi di un linguaggio visivo consiste di svariati elementi:

- Definizione dei simboli (token) che compongono il linguaggio visuale:
  - Definizione dell'apparenza "fisica" dei simboli (ad esempio forma, colore, ecc.);
  - Definizione degli attributi del simbolo della loro forma e del loro aspetto (ad esempio punti/area d'attaccamento, ecc);

---

<sup>1</sup>*'Un framework, in generale, include software di supporto, librerie, un linguaggio per gli script e altri software che possono aiutare a mettere insieme le varie componenti di un progetto.'*, [2]

- Definizione dei vincoli locali al simbolo riguardanti gli attributi (ad esempio il numero di connessioni permesse ad un punto d'attaccamento, ecc.).
- Definizione delle relazioni/connettori e i loro vincoli locali;
- Dichiarazione di vincoli al livello del diagramma (ad esempio numero di occorrenze ammissibili di un simbolo, se i simboli e i connettori devono formare un grafo connesso);
- Una grammatica per definire ulteriori vincoli di sintassi del linguaggio (quando necessario).



Token	Graphics	Token occurrences	name	Attachment points type	constraints
ROOT		1	<i>IN</i>	enter	<i>connectNum</i> = 0
			<i>OUT</i>	exit	<i>connectNum</i> ≤ 2
NODE		≥ 0	<i>IN</i>	enter	<i>connectNum</i> = 1
			<i>OUT</i>	exit	<i>connectNum</i> ≤ 2
	Connector		name	Attachment points type	constraints
	POLYLINE		<i>P0</i>	exit	<i>connectNum</i> = 1
			<i>P1</i>	enter	<i>connectNum</i> = 1
Further constraint					
the spatial-relationship graph must be connected					

Tabella 4.1 Specifica di un linguaggio, nel particolare di un Albero

Facendo riferimento alla tabella 4.1, ovvero la specifica di un linguaggio che identifica un Albero, possiamo notare che ogni riga identifica un simbolo (o token) e che ogni riga è composta da sei colonne, le prime tre sono indirizzate al livello del simbolo e le restanti tre colonne indicano i vincoli per gli attachment points:

- **Token:** indica il nome dell'elemento;
- **Graphics:** rappresentazione grafica dell'elemento e rappresentazione dei punti d'attaccamento evidenziati in giallo;
- **Token occurrences:** numero di occorrenze ammissibile per quel simbolo;
- **name:** può essere composta da più righe ed indica i nomi dei punti d'attaccamento;

- **type**: la tipologia di AP<sup>2</sup>, un tipo di punto di attaccamento può avere più nomi;
- **constraints**: i vincoli al livello dei punti d'attaccamento, *connectNum* indica il numero di archi incidenti a quel determinato AP.

I vincoli al livello della sentenza sono riportati nell'ultima riga. Una parte della formalizzazione della tabella 4.1 in formato XML è mostrata nello snippet di codice 4.1, l'intera specifica è consultabile nell'appendice A.1.

**Listing 4.1 Frammento della definizione in formato XML di un linguaggio, nel particolare la specifica per il simbolo *root* (o radice).**

```

1  <token name="root" ref="circle.svg" occurrences=="1">
2      <ap type="enter" name="in" ref="hiSC" connectNum=="0"/>
3      <ap type="exit" name="out" ref="lowSC" connectNum=="2"/>
4  </token>
5

```

Nel paper [7], viene esteso il concetto esposto in [6] aggiungendo nuove caratteristiche alla specifica del contesto locale per permettere la specifica di linguaggi visuali più complessi quali diagrammi entità-relazione, use case diagrams e class diagram. In più viene presentato un tool (LoCoMoTiVE, specificato nel paragrafo 2.2) che implementa il framework *Local Context*.

Le principali caratteristiche aggiunte sono tre:

- La definizione di vincoli al livello del simbolo può coinvolgere più di un area di attaccamento di un simbolo/connettore al contrario di vincolare aree d'attaccamento individuali;
- La definizione di un vincolo per limitare auto-cicli di un connettore;
- Assegnazione di più tipi alle aree di attaccamento.

## 4.2 Semantica

In [8] viene esteso il framework definendo una nuova tecnica per una traduzione semantica del linguaggio visuale basata sul contesto locale. Questa tecnica usa delle espressioni simili a quelli di XPath, chiamate SGPath, illustrate nel dettaglio nel capitolo

---

<sup>2</sup>Punti d'attaccamento.

5.

Per consentire la traduzione semantica di un linguaggio visuale è stata descritta una definizione semantica basata sul contesto locale o *LCSD*<sup>3</sup> e il suo algoritmo per valutarla. L'*LCSD* consiste di una sequenza di regole semantiche, una per ogni elemento del linguaggio. Ogni regola calcola una proprietà attraverso delle procedure che fanno uso degli *SGPath* o esegue un'azione. Ogni proprietà può avere una post-condizione. Attraverso le post-condizioni, una *LCSD* può dare una definizione migliore della struttura sintattica; attraverso le azioni restituisce una traduzione delle sentenze. Un'azione può dipendere dalle proprietà e dagli attributi dell'elemento.

ENTITY				
Property	Procedure	Params		Post-condition
\$Key : list<string>	add	CON_EA/KEY_ATTR	@KeyName	size(\$Key) > 0
\$Attributes : list<string>	add	CON_EA/ATTRIBUTE	@AttrName	
	addAll	CON_ER[@Cardin='([01],1)']/RELATION \$Attributes		
print(@EntName + " (<u>" + explode(" ", \$Key) + "</u>"); if(size(\$Attributes) > 0) print(" " + explode(" ", \$Attributes)); print("\n");				

RELATION				
Property	Procedure	Params		Post-condition
\$Key : list<string>	addAll	CON_ER[@Cardin='([01],N)']/ENTITY	\$Key	
\$Attributes : list<string>	add	CON_RA/ATTRIBUTE	@AttrName	
\$Entities : list<string>	add	CON_ER[@Cardin='([01],N)']/ENTITY	@EntName	
\$KeyAttrs : list<string>	add	CON_RA/KEY_ATTR	@KeyName	size(\$KeyAttrs) = 0
if(size(\$Entities) == 2) { print(@RelName + " (<u>" + explode(" ", \$Key) + "</u>"); if(size(\$Attributes) > 0) print(" " + explode(" ", \$Attributes)); print("\n"); }				

Tabella 4.2 Specifica LCSD di un diagramma ER, costruita sulla specifica sintattica.

La tabella 4.2 mostra una specifica semantica di un diagramma ER<sup>4</sup>. Ogni tabella fornisce le regole semantiche di un elemento, in questo caso per i simboli ENTITY e RELATION, rispettivamente. Ogni riga della specifica è composta dai seguenti elementi:

- **Property:** indica il nome della proprietà (preceduta dal carattere \$) e il tipo di variabile contenuta.
- **Procedure:** indica il nome della procedura da utilizzare per assegnare/modificare il valore di una proprietà, ogni proprietà può avere più procedure.
- **Params:** i parametri da utilizzare nella procedura. Il primo parametro è un *SGPath*, il secondo parametro è il nome della proprietà dove leggere le informazioni del/i simbolo/i raggiungibile/i con quel percorso.

<sup>3</sup>Local Context-based Semantic Definition<sup>4</sup>Entità Relazione.

- **Post-condition:** la post-condizione che deve essere rispettata al termine dell'esecuzione della procedura.

L'ultima colonna indica l'azione da eseguire. A differenza dell'attuale implementazione, TiVeJS può eseguire anche azioni parziali, ovvero posizionata tra le proprietà e non dopo le proprietà.

In questo caso, applicando le regole semantiche definite nella tabella 4.2 al diagramma

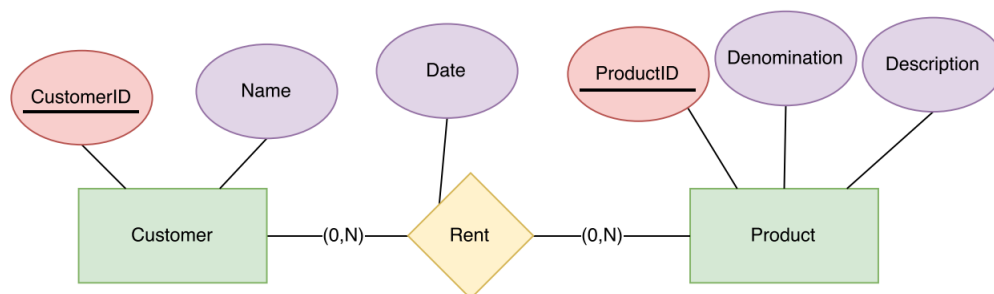


Figura 4.1 Diagramma ER

4.1 ottengo la seguente traduzione semantica:

Customer (CustomerID, Name)

Product (ProductID, Denomination, Description)

Rent (CustomerID, ProductID, Date)

Tutti i dettagli implementativi saranno illustrati nel prossimo capitolo.

# Capitolo 5

## TiveJS

Come introdotto, in questo lavoro presento TiveJS, che è un porting ed un'evoluzione di TiVE e come esso è in grado di: permettere la progettazione di sentenze visive grazie all'ausilio di palette di simboli personalizzate, il riconoscimento di linguaggio diagrammatici e la traduzione di quest'ultimi.

Il paper da cui deriva questo lavoro è *Using the local context for the definition and implementation of visual languages* [8] in cui si parla di un metodo per la traduzione di diagrammi o in generale di sentenze visuali in sentenze semantiche.

### 5.1 Funzionamento e Implementazione

Il funzionamento si suddivide principalmente in tre fasi, mostrate schematicamente nella figura 5.1: la progettazione del diagramma, il riconoscimento di quest'ultimo e l'applicazione delle definizioni sintattiche e semantiche. Durante tutte le procedure, la navigazione all'interno del grafo viene effettuata attraverso delle espressioni chiamate SGPath.

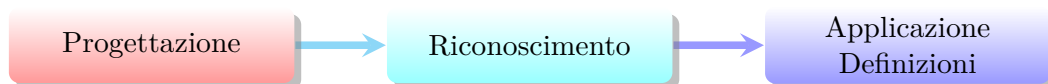


Figura 5.1 Diagramma del funzionamento di TiveJS

Come vedremo, ogni fase può suddividersi in più sotto-fasi illustrate nel dettaglio nelle prossime sezioni.

### 5.1.1 SGPath

Questa sezione descrive una specifica simile a XPath<sup>1</sup> per la navigazione all'interno del grafo chiamata SGPath. Mentre XPath definisce delle espressioni per la navigazione all'interno dei file XML, SGPath definisce delle espressioni per navigare in una struttura a grafo.

Un SGPath consiste di passaggi (o *step*). Ogni passo è valutato su un insieme di nodi del grafo. Il risultato di ogni passo è un insieme di nodi del grafo. SGPath può far uso delle proprietà dei simboli quali nome, id, ecc.

La sintassi di un SGPath è strutturata in questo modo:

```
sgpath_step_1/sgpath_step_2/ ... /sgpath_step_n
```

ogni `sgpath_step` è formato in questo modo:

```
axis(edge-filter)::node-test[predicate]
```

dove:

- **axis(edge-filter)::** è opzionale e indica come continuare la navigazione partendo dal nodo corrente. A differenza di XPath, concede di spostarsi solo tra nodi adiacenti.
- **node-test** indica il nodo da raggiungere. Può essere il nome di un elemento o \* per indicare qualsiasi nodo.
- **[predicate]** è opzionale e permette di selezionare i nodi con attributi specifici.

Ecco alcuni esempi di SGPath:

- `CON_E_A/KEY_ATTR`
- `CON_E_R[@Cardin='([01],N)']/ENTITY`
- `(#attName = 'Up')::ARROW/PRED`
- `(#attName = 'Up')::ARROW/*`

---

<sup>1</sup> "In informatica XPath è un linguaggio, parte della famiglia XML, che permette di individuare i nodi all'interno di un documento XML. Le espressioni XPath, a differenza delle espressioni XML, non servono a identificare la struttura di un documento, bensì a localizzarne con precisione i nodi." [4]



In TiveJS l'algoritmo che si occupa della risoluzione degli SGPath è visibile all'interno dello snippet 5.1. La funzione `resolvePath` divide la stringa al carattere `'/'` così da ottenere ogni step del percorso. A questo punto ogni step viene scorporato di tutti i suoi campi dalla funzione `scorporatePath` che restituisce un oggetto con all'interno tutte le informazioni sul singolo step.

#### Listing 5.1 Funzione che si occupa della risoluzione delle path

```
1      /**
2      * Resolve path
3      */
4      CheckUtil.prototype.resolvePath = function (node, path) {
5          console.log('RESOLVING A PATH FOR: ' + node.id);
6          console.info('node, path', node, path);
7          let nodes = [];
8          nodes = nodes.concat(node);
9          let splittedPath = path.split('/');
10         for (let elem in splittedPath) {
11             let pathStep = splittedPath[elem];
12             nodes = this.resolvePathStep(nodes, pathStep);
13         }
14         console.info('returned nodes', nodes);
15         return nodes;
16     }
```

### 5.1.2 Progettazione di sentenze visive

La progettazione delle sentenze visive avviene attraverso una GUI<sup>2</sup> molto semplice ed intuitiva, simile a quella nella figura 2.1.

E' composta di quattro sezioni principali (figura 5.2):

- Barra laterale sinistra (evidenziata **rosso**)
- Barra laterale destra (evidenziata **blu**)
- Area di lavoro (evidenziata **giallo**)
- Barra dei menu (evidenziata **viola**)

---

<sup>2</sup>Interfaccia Grafica.

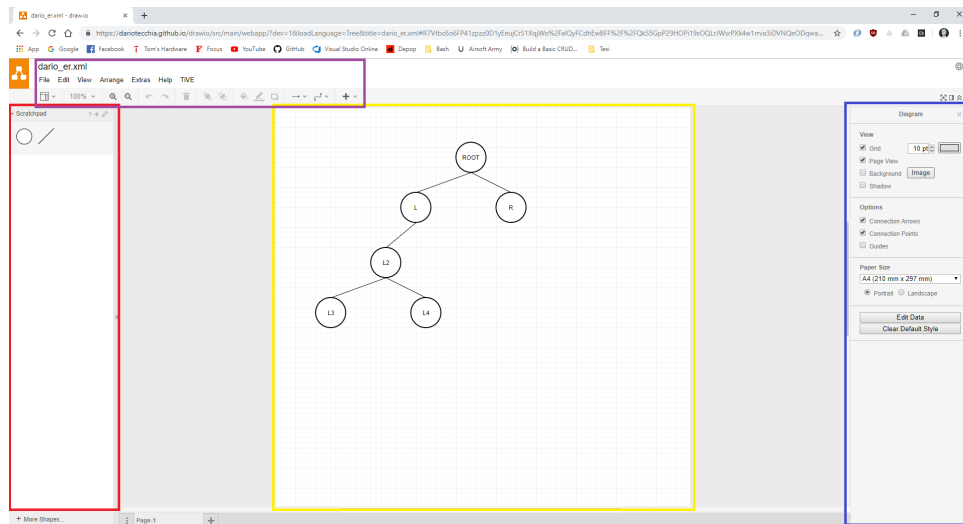


Figura 5.2 Schermata principale di TiveJS

All'interno della barra laterale sinistra c'è la palette dei simboli importata precedentemente creata con l'ausilio di *drawSE*. Per importare una nuova libreria (o palette) bisogna andare nel menu *File -> Open Library From* e selezionare il metodo di importazione.

La barra laterale destra serve a personalizzare ulteriormente il diagramma o il singolo simbolo. Mette a disposizione vari menù ed opzioni quali la dimensione della pagina su cui disegniamo il diagramma, il colore di un simbolo, le proprietà del testo contenuto all'interno di un simbolo, ecc.

L'area di lavoro è dove viene composto il diagramma trascinando i simboli dalla barra laterale. All'interno di quest'area possiamo selezionare, modificare e spostare il diagramma e i relativi simboli a nostro piacimento.

La barra dei menu è composta da tanti sotto menu ognuno dei quali ha una funzione specifica. Oltre ai menu di *draw.io*, ho aggiunto un nuovo menu "TiVE" (figura 5.3) per il caricamento delle definizioni e per la verifica del grafo. Nello specifico, "*Load Rules...*" serve per il caricamento delle definizioni sintattiche; "*Load Semantic Rules...*" serve per il caricamento delle definizioni semantiche e "*Apply Rules*" serve ad eseguire l'applicazione di quest'ultime e ad eseguire la traduzione semantica del diagramma.

### 5.1.3 Riconoscimento grafo

All'interno del diagramma, più simboli con scopi diversi possono condividere lo stesso elemento grafico creando delle ambiguità. Una delle prime fasi del processo di traduzione

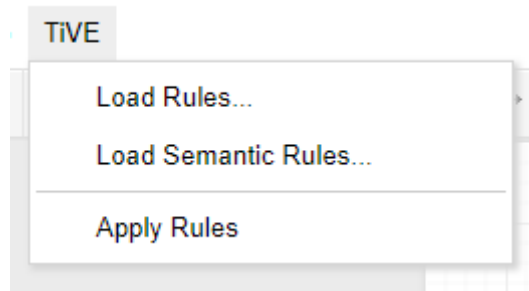


Figura 5.3 Menu dei comandi per TiVE

è il riconoscimento degli elementi e quindi la risoluzione di queste ambiguità. Come possiamo notare nella tabella 5.1, in particolare nella seconda sotto-tabella, i connettori pur avendo uno scopo diverso condividono lo stesso elemento grafico, una linea.

Ancora, nella tabella 5.2 a condividere lo stesso elemento grafico è un simbolo: ROOT e NODE sono entrambi rappresentati da un cerchio.

Symbol name	Graphics	Symbol occurrence	name	Attaching areas	constraints
ENTITY		$\geq 1$	<i>BORDER</i>	EntBrd	$connectNum \geq 0$
			<i>CNT</i>	<i>text</i> :EntName	<i>string</i>
RELATION		$\geq 0$	<i>UP</i>	RelVrt	$\sum (connectNum) = 2$
			<i>LEFT</i>	RelVrt	
			<i>RIGHT</i>	RelVrt	
			<i>DOWN</i>	RelVrt	
			<i>BORDER</i>	RelBrd	$connectNum \geq 0$
			<i>CNT</i>	<i>text</i> :RelName	<i>string</i>
ATTRIBUTE		$\geq 0$	<i>BORDER</i>	AttBrd	$connectNum = 1$
			<i>CNT</i>	<i>text</i> :AttrName	<i>string</i>
KEY_ATTR		$\geq 0$	<i>BORDER</i>	AttBrd	$connectNum = 1$
			<i>CNT</i>	<i>text</i> :KeyName	<i>string</i>

Connector name	Graphics	Connector occurrence	name	Attaching areas	constraints
CON_E.R		$\geq 0$	<i>P1</i>	EntBrd	$connectNum = 1$
			<i>P2</i>	RelVrt	$connectNum = 1$
			<i>CNT</i>	<i>text</i> :Cardin	$([0], [1N])$
CON_E.A		$\geq 0$	<i>P1</i>	EntBrd	$connectNum = 1$
			<i>P2</i>	AttBrd	$connectNum = 1$
CON_R.A		$\geq 0$	<i>P1</i>	RelBrd	$connectNum = 1$
			<i>P2</i>	AttBrd	$connectNum = 1$

Sentence level constraint					
<i>the spatial-relationship graph must be connected</i>					

Tabella 5.1 Specifica di un diagramma ER

A caratterizzare un elemento sono le proprietà della definizione sintattica correlata e la disambiguazione viene effettuata tenendo in considerazione questi elementi.

L'algoritmo risolutivo innanzitutto rileva se si sta analizzando un vertice o un connettore e nel caso in cui venissero rivelate delle ambiguità procede come segue: Se




Symbol name	Graphics	Symbol occurrence:	Attaching areas		
			name	type	constraints
ROOT		1	<i>UP</i>	enter	<i>connectNum</i> = 0
			<i>DOWN</i>	exit	<i>connectNum</i> ≥ 0
			<i>CNT</i>	<i>text:Desc</i>	<i>string</i>
NODE		≥ 0	<i>UP</i>	enter	<i>connectNum</i> = 1
			<i>DOWN</i>	exit	<i>connectNum</i> ≥ 0
			<i>CNT</i>	<i>text:Desc</i>	<i>string</i>
Connector name	Graphics	Connector occurrence:	Attaching areas		
			name	type	constraints
EDGE		≥ 0	<i>P1</i>	exit	<i>connectNum</i> = 1
			<i>P2</i>	enter	<i>connectNum</i> = 1
Sentence level constraint					
the spatial-relationship graph must be connected					

Tabella 5.2 Specifica di un Albero

l'elemento analizzato è un connettore prende in considerazione il campo *Type* delle due aree d'attacco. All'interno di questa proprietà vengono indicati i tipi di area d'attacco in cui il connettore è collegato alle due estremità. Ad esempio, il connettore *CON\_E\_A* (Connettore Entità-Attributo) alle due estremità si connetterà ad una *EntBrd* e ad una *AttBrd* in posizioni arbitrali; Nel caso in cui l'elemento analizzato è un vertice allora il controllo viene effettuato sui vincoli definiti per ogni simbolo. Il vertice sarà associato al simbolo per cui verranno rispettati tutti i suoi vincoli. Ad esempio, il simbolo ROOT dovrà avere un numero di connessioni (*connectNum*) uguali a zero sull'attaching area UP di tipo enter e un numero maggiore o uguale a zero di connessioni sull'attaching area DOWN di tipo exit.

#### 5.1.4 Applicazione definizioni

Eseguito il riconoscimento di ogni simbolo appartenente al grafo, l'algoritmo procede con l'applicazione delle definizioni. Come ho già accennato, la specifica delle definizioni si divide in due parti: sintattica e semantica.

##### Definizioni Sintattiche

Ogni linguaggio ha delle regole sintattiche da rispettare. Un esempio di definizione sintattica è raffigurata nella tabella 5.1. Nella colonna *Symbol occurrences* vi è un vincolo al livello del simbolo, indica quante occorrenze del simbolo devono essere

presenti all'interno del diagramma. Nell'ultima colonna, *constraints*, vi troviamo i vincoli al livello degli Attaching Point e il formato che deve avere il campo di testo dell'elemento se specificato. Nell'ultima riga, *Sentence level constraint*, troviamo i vincoli che il diagramma deve rispettare.

Come ho già accennato, in TiVE le definizioni erano scritte in formato XML (vedi Appendice A.1). Successivamente le ho implementate in formato JSON<sup>3</sup> per poter essere interpretate in maniera nativa dal linguaggio di programmazione da me usato. All'interno dello snippet di codice 5.2 vi è una parte della definizione nel nuovo formato. Il valore di *"ap"* è un array contenente i vincoli per i punti d'attacco; *"text"* contiene le informazioni riguardanti le aree di testo; *"\_name"* è il nome del simbolo; *"\_ref"* indica a quale figura grafica si fa riferimento; *"occurrences"* è il vincolo sulle occorrenze del simbolo.

L'intera definizione è consultabile all'Appendice A.2.

**Listing 5.2 Frammento della definizione sintattica in formato JSON di un linguaggio, nel particolare la specifica per il simbolo *ROOT* (o Radice).**

```

1      {
2          "ap": [
3              {
4                  "_type": "exit",
5                  "_name": "OUT",
6                  "_ref": "Down",
7                  "_connectNum": ">=0"
8              },
9              {
10                 "_type": "enter",
11                 "_name": "IN",
12                 "_ref": "Up",
13                 "_connectNum": "==0"
14             }
15         ],
16         "_name": "ROOT",
17         "text": [
18             {
19                 "_graphicRef": "Center",
20                 "_name": "Desc",
21                 "_type": "string"
22             }
23         ],
24         "_ref": "Circle",
25         "_occurrences": "==1"
26     },

```

## Definizioni Semantiche

Le definizioni semantiche sono necessarie alla traduzione del diagramma. Un esempio di definizione semantica (o definizione LCSD) è raffigurata nella tabella 4.2. Come

<sup>3</sup>JavaScript Object Notation

già detto nella sezione 4.2, la specifica semantica è composta da quattro elementi fondamentali:

- **Property** indica il nome della proprietà semantica da attribuire all'elemento della specifica;
- **Procedure** è il nome della procedura da utilizzare per assegnare o modificare la proprietà;
- **Params** sono i parametri da passare alla procedura;
- **Post-condition** è la condizione che deve essere rispettata al termine dell'esecuzione della procedura.

Anche in questo caso ho trascritto le definizioni sintattiche dal formato XML al formato JSON. All'interno dello snippet di codice 5.3 vi è una parte della definizione nel nuovo formato.

- **property** contiene tutte le proprietà da computare. Ogni proprietà è composta da:
  - **\_\_name** è il nome della proprietà;
  - **\_\_type** è il formato della proprietà;
  - **procedure** contiene le procedure da eseguire. Ogni procedura è composta da:
    - \* **\_\_postCondition** è la condizione da rispettare al termine della procedura;
    - \* **name** è il nome della procedura;
    - \* **\_\_path** insieme a **\_\_param** sono i parametri da passare alla procedura.
- **visit** verrà utilizzato dall'algoritmo di ordinamento del grafo.

L'intera definizione è consultabile all'Appendice A.4.

**Listing 5.3 Frammento della definizione semantica in formato JSON di un linguaggio, nel particolare la specifica per il simbolo STAT**

```

1 | {
2 |   "property": [
3 |     {
4 |       "_name": "$Count",
5 |       "_type": "number",
6 |       "procedure": [
7 |         {
8 |           "_postCondition": "",
9 |           "_name": "size",
10 |          "_path": "(#attName = 'Up')::*",
11 |          "_param": ""
12 |        }
13 |      ]
14 |    },
15 |    {
16 |      "_name": "$PrePred",
17 |      "_type": "boolean",
18 |      "procedure": [
19 |        {
20 |          "_postCondition": "",
21 |          "_name": "exist",
22 |          "_path": "(#attName = 'Up')::ARROW/PRED",
23 |          "_param": ""
24 |        }
25 |      ]
26 |    },
27 |    {
28 |      "action": "if($Count > 1 || $PrePred) { print(#id +
29 |        ' : ') } print(@Code + ';\n')",
30 |    }
31 | ],
32 | "visit": {
33 |   "path": [
34 |     {
35 |       "_value": "(#attType='exit')::ARROW",
36 |       "_flag": "D"
37 |     }
38 |   ],
39 |   "_priority": "1",
40 |   "_order": "3"
41 | },
42 | "_ref": "STAT"

```

### 5.1.5 Applicazione Regole Semantiche

Se l'applicazione delle definizioni è andata a buon fine, l'algoritmo procede con il calcolo delle regole semantiche. La procedura si suddivide in due fasi: ordinamento del grafo e calcolo delle definizioni semantiche.

### Ordinamento del grafo

In precedenza ho parlato del campo *visit* presente all'interno della definizione semantica di un linguaggio. Questo campo serve a costruire una *Visit Table* necessaria a definire l'ordine in cui gli elementi del diagramma verranno visitati dall'algoritmo.

Visit table			
Order	Priority	Paths	
BEGIN	-1	D	ARROW
ARROW	1	D	(#attName='HEAD')::*
STAT	1	D	(#attType='exit')::ARROW
IO	1	D	(#attType='exit')::ARROW
PRED	1	D	(#attType='exit')::ARROW[@Rel='true']
		D	(#attType='exit')::ARROW[@Rel='false']
END	2		

Tabella 5.3 Visit Table per il linguaggio Flowchart

Ogni simbolo ha una priorità, come mostrato nella tabella 5.3. Questa tabella specifica la priorità per ogni elemento del linguaggio flowchart, in particolare -1 per il simbolo BEGIN, 2 per il simbolo END e 1 per tutti gli altri simboli. In questo modo il simbolo BEGIN verrà posizionato all'inizio, END alla fine e gli altri simboli in una posizione arbitraria.

La colonna *Paths* della tabella indica il prossimo nodo da esplorare, se presente. Questa è composta da due sezioni, una è un flag che indica se l'esplorazione viene effettuata in ampiezza (B) o un profondità (D) e l'altra è il percorso da seguire.

Altro modo di definire un ordine è con la *Path Table*, mostrata nella tabella 5.4.

Path table		
Order	Paths	
ROOT	D	(#attType='exit')::EDGE/NODE
NODE	D	(#attType='exit')::EDGE/NODE

Tabella 5.4 Path Table per il linguaggio Tree

La tabella di visita viene applicata al diagramma dalla funzione `ApplyVisitTable` (snippet 5.4) che fa uso di un'altra funzione fondamentale, `FollowPath` (snippet 5.5).



## Listing 5.4 La funzione applyVisitTable

```

1      // Apply the visit table to the graph
2      CheckUtil.prototype.applyVisitTable = function (graph, visitTable) {
3          let N = Object.values(graph).sort((a, b) => {
4              return visitTable[a.name].order - visitTable[b.name].order;
5          });
6          let L = [];
7          while (N.length !== 0) {
8              let rem = N.shift();
9              L.push(rem);
10             let nodes = [];
11             nodes = this.followPath(rem, graph, N, visitTable, nodes);
12             L = L.concat(nodes);
13
14             N = N.filter((i) => {
15                 return nodes.indexOf(i) < 0;
16             });
17         }
18
19         L = this.stableSort(L, (a, b) => {
20             if (visitTable[a.name].priority > visitTable[b.name].priority) {
21                 return 1;
22             }
23             if (visitTable[a.name].priority < visitTable[b.name].priority) {
24                 return -1;
25             }
26         });
27         return L;
28     }

```

## Listing 5.5 La funzione followPath

```

1      // Follow the path from a node
2      CheckUtil.prototype.followPath = function (node, G, N, PTPATHS, nodes) {
3          let nname = node.name;
4          let npaths = PTPATHS[nname].paths;
5          for (let elem in npaths) {
6              let npath = npaths[elem];
7              let nds = this.resolvePath(node, npath.__value);
8              nds = nds.filter((i) => {
9                  return nodes.indexOf(i) < 0;
10             });
11             nds = nds.filter((i) => {
12                 return N.indexOf(i) > -1;
13             });
14             if (npath.__flag === 'B') {
15                 nodes = nodes.concat(nds);
16             }
17             for (let elem in nds) {
18                 let n = nds[elem];
19                 if (npath.__flag === 'D') {
20                     if (nodes.indexOf(n) < 0) {
21                         nodes.push(n);
22                     }
23                 }
24                 nodes = this.followPath(n, G, N, PTPATHS, nodes);
25             }
26         }
27         return nodes;
28     }

```

Ordinato il grafo, l'algoritmo procede con l'applicazione delle definizioni semantiche.

### Calcolo definizioni semantiche

Altro ruolo importante è quello della funzione `ApplySemanticRules` (snippet 5.6).

Listing 5.6 La funzione che computa tutte le proprietà semantiche

```

1      // Apply the semantic rules to the graph
2      CheckUtil.prototype.applySemanticRules = function (graph, semanticRules) {
3
4          let completedNodes = [];
5
6          // DELETING ALL THE NODES WHICH AREN'T SEMANTIC RULES
7          for (let elem in graph) {
8              let graphElem = graph[elem];
9              let graphElemSemRule = semanticRules[graphElem.name];
10             if (!graphElemSemRule) {
11                 delete graph[elem];
12             }
13         }
14
15         graph = graph.filter((el) => {
16             return el != null;
17         });
18
19         let loop = 0;
20
21         while (!graph.length == 0) {
22
23             if (++loop == 100) {
24                 console.log('loop');
25                 break;
26             }
27
28             for (let elem in graph) {
29
30                 let x = graph[elem];
31                 let xRules = semanticRules[x.name];
32                 let propertyToCalculate = xRules.property.length;
33                 let calculatedProperties = this.calculateProperties(x, xRules);
34                 console.log("CALCULATED: " + calculatedProperties);
35                 console.log("TO CALCULATE: " + propertyToCalculate);
36                 if (propertyToCalculate == calculatedProperties) {
37                     x.status = "COMPLETE";
38                     completedNodes.push(x);
39                 } else {
40                     x.status = "INCOMPLETE";
41                 }
42             }
43             console.info('COMPLETED NODES: ', completedNodes);
44             graph = graph.filter((i) => {
45                 return completedNodes.indexOf(i) < 0;
46             });
47             console.info('INCOMPLETED NODES: ', graph);
48         }
49         return true;
50     }

```

Come si può notare dal codice, l'algoritmo prende in input il grafo resituito dalla funzione `applyVisitTable` (5.4) e le regole semantiche, ad esempio A.3 o A.4. L'algoritmo analizza ogni simbolo e gli assegna lo status "COMPLETE" solo se sono state

calcolate tutte le proprietà compresa l'azione. Lo stesso simbolo può essere analizzato più volte fin quando non vengono calcolate tutte le proprietà definite all'interno delle regole semantiche. Per evitare la creazione di deadlock<sup>4</sup> è stata inserita una variabile "loop" che farà arrestare l'algoritmo se viene raggiunto un numero alto di iterazioni senza terminare il calcolo di tutte le proprietà.

Una volta applicate le definizioni, TiveJS mostrerà all'interno di una Console la traduzione semantica del diagramma (figura 5.4) se non ci sono errori, altrimenti mostrerà all'interno della console gli errori che si sono verificati. Inoltre gli elementi coinvolti in quell'errore saranno evidenziati in rosso, figura 5.5.

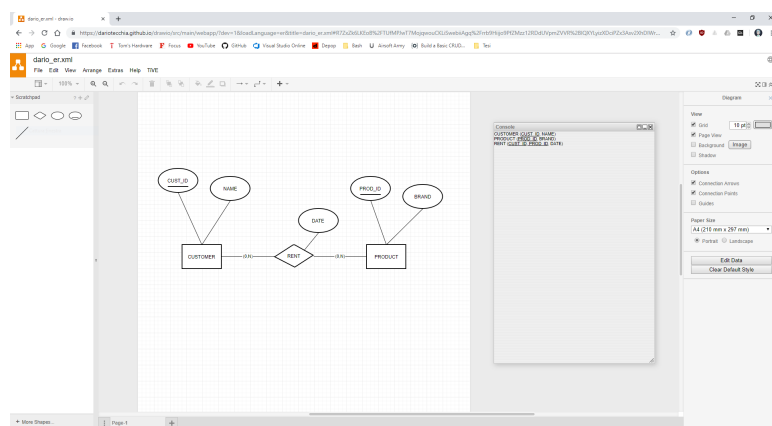


Figura 5.4 TiveJS con la traduzione semantica di un diagramma entità-relazione mostrata in una console (sulla destra)

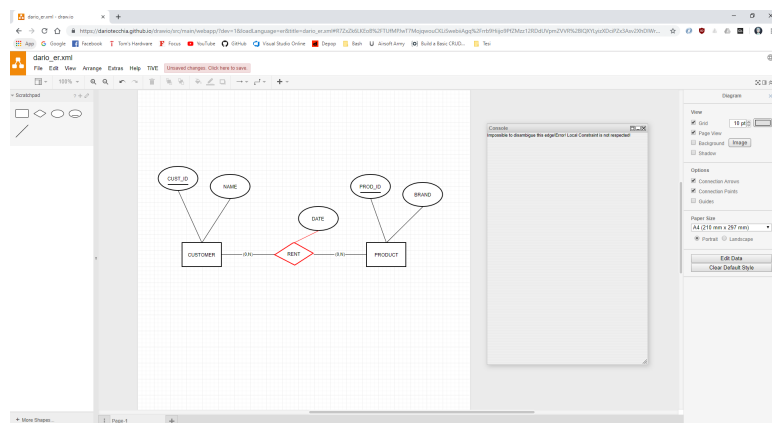


Figura 5.5 TiveJS con gli errori mostrati in una console (sulla destra)

<sup>4</sup>In questo caso si verifica una deadlock se la computazione delle proprietà di uno o più simboli creano uno stallo

## 5.2 Tecnologie Utilizzate: JavaScript

TiveJS è stata implementata completamente in JavaScript per far sì che potesse girare su ogni browser web. JavaScript, a volte abbreviato con JS, è un linguaggio di programmazione interpretato ad alto livello. Standardizzato per la prima volta nel 1997 con il nome di ECMAScript (attualmente l'ultima release è ECMAScript 2018 [1]), insieme all'HTML e il CSS è una delle tecnologie alla base del World Wide Web. In questa sezione esamino alcuni dei punti di forza di JavaScript, così da capire quali sono le sue particolarità.

### 5.2.1 Punti di Forza di JavaScript

#### Ricco di librerie

JavaScript, oltre le sue librerie standard, conta un numero impressionante di librerie scritte da una community molto attiva. Dispone di API per lavorare con testo, matrici, date, espressioni regolari e DOM<sup>5</sup>.

#### Supporto universale

Tutti i moderni browser web supportano nativamente JavaScript con gli interpreti implementati al loro interno.

#### Multi-paradigma

JavaScript è un linguaggio multi-paradigma, che supporta la programmazione basata sugli eventi, la programmazione funzionale e quella imperativa (inclusa la programmazione orientata agli oggetti).

#### Portabile

JavaScript è un linguaggio portabile. E' possibile usarlo su diverse piattaforme come: *Linux, Windows, OSX, iOS e Android*. Questo è possibile perchè non dipende dalla macchina dove gira essendo interpretato e non compilato. E' necessario però un interprete JavaScript.

---

<sup>5</sup>Document Object Model

### Semplice

Molto semplice da imparare e perfetto da usare come linguaggio accademico essendo un linguaggio ad alto livello.

### Gratis

JavaScript è totalmente gratis ed è possibile utilizzarlo e distribuirlo senza restrizioni di copyright. Nonostante questo, alle spalle ha una community attivissima.

## 5.2.2 Principali Applicazioni

Pur essendo un linguaggio rilasciato per i browser web, Netscape, nel 1995 introdusse una nuova implementazione del linguaggio per lo scripting server-side. Una delle implementazioni server-side più famosa è Node.js.

## 5.3 Librerie utilizzate nel Progetto

### mxGraph

Come accennato, una delle librerie utilizzate all'interno di TiveJS è mxGraph. Serve per lo sviluppo di diagrammi e permette la creazione di applicazioni interattive per la creazione di grafi e diagrammi. Oltre che in JavaScript, la libreria è scritta anche in linguaggi server side quali PHP, .NET e Java [9].

### jQuery

jQuery risulta essere la libreria JavaScript più utilizzata su Internet per via della facilità di installazione e utilizzo.

Lo slogan di jQuery non a caso è «*write less, do more*» poichè nasce con l'intenzione di semplificare la selezione, la manipolazione, la gestione degli eventi e l'animazione di elementi DOM in pagine HTML, nonchè implementare funzionalità AJAX [3].

Fornisce agli sviluppatori un'interfaccia semplice, accessibile attraverso il caratteristico simbolo \$, astruendo comandi molto più complessi offerti da JavaScript. E' un software libero.

# Capitolo 6

## Contesti d'utilizzo

6.1 Entitiy Relationship

6.2 Flowchart

6.3 Tree

# Capitolo 7

## Conclusioni e Sviluppi Futuri

Riassumendo in questo lavoro ho presentato un sistema scritto interamente in JavaScript per il riconoscimento e la verifica di linguaggi diagrammatici. E' in grado di permettere lo sviluppo di una sentenza visiva e di ottenere, applicando delle definizioni, una traduzione semantica di quest'ultima.

In tutta la tesi ho analizzato vari casi d'utilizzo, uno in cui l'applicazione delle definizioni va a buon fine ed uno in cui non va a buon fine. Ho presentato i dettagli implementativi di TiveJS illustrando le parti di codice di maggior interesse.

In futuro l'applicazione verrà integrata con l'ecosistema LoCoMoTiVE permettendo di creare un'unica applicazione dove il progettista può creare delle definizioni e testarle direttamente all'interno di quest'ultima.

# Appendice A

## Codici

Listing A.1 Specifica di un linguaggio in formato XML, in questo caso quello per Albero Binario

```
1 <language name="binaryTree">
2
3   <token name="root" ref="circle.svg" occurrences=="1">
4     <ap type="enter" name="in" ref="hiSC" connectNum=="0"/>
5     <ap type="exit" name="out" ref="lowSC" connectNum=="2"/>
6   </token>
7
8   <token name="node" ref="circle.svg" occurrences=">0">
9     <ap type="enter" name="in" ref="hiSC" connectNum=="1"/>
10    <ap type="exit" name="out" ref="lowSC" connectNum=="2"/>
11  </token>
12
13  <connector ref="polyline">
14    <cap type="exit" ref="p0" connectNum=="1" />
15    <cap type="enter" ref="p1" connectNum=="1" />
16  </connector>
17
18  <constraint>connected</constraint>
19 </language>
```



## Listing A.2 Specifica sintattica di un Albero

```

1 {
2   "language": {
3     "token": {
4       {
5         "ap": [
6           {
7             "_type": "exit",
8             "_name": "OUT",
9             "_ref": "Down",
10            "_connectNum": ">=0"
11          },
12          {
13            "_type": "enter",
14            "_name": "IN",
15            "_ref": "Up",
16            "_connectNum": "==0"
17          }
18        ],
19        "_name": "ROOT",
20        "text": {
21          {
22            "_graphicRef": "Center",
23            "_name": "Desc",
24            "_type": "string"
25          }
26        },
27        "_ref": "Circle",
28        "_occurrences": "==1"
29      },
30    },
31    "ap": [
32      {
33        "_type": "enter",
34        "_name": "IN",
35        "_ref": "Up",
36        "_connectNum": "==1"
37      },
38      {
39        "_type": "exit",
40        "_name": "OUT",
41        "_ref": "Down",
42        "_connectNum": ">=0"
43      }
44    ],
45    "_name": "NODE",
46    "text": {
47      {
48        "_graphicRef": "Center",
49        "_name": "Desc",
50        "_type": "string"
51      }
52    },
53    "_ref": "Circle",
54    "_occurrences": ">=0"
55  },
56  "connector": [
57    {
58      "ap": [
59        {
60          "_type": "exit",
61          "_name": "OUT",
62          "_ref": "P2:P1",
63          "_connectNum": "==1"
64        },
65        {
66          "_type": "enter",
67          "_name": "IN",
68          "_ref": "P1:P2",
69          "_connectNum": "==1"
70        }
71      ],
72      "_name": "EDGE",
73      "_ref": "connectors/Line.svg"
74    }
75  ],
76  "constraint": "Connected",
77  "_name": "tree"
78 }
79 }
80 }

```

**Listing A.3** Specifica semantica di un linguaggio in formato JSON, in questo caso quello per Albero

```

1 | {
2 |   "language": {
3 |     "semantic": [
4 |       {
5 |         "property": [
6 |           {
7 |             "action": "print(@Desc + '\\n')"
8 |           }
9 |         ],
10 |        "visit": {
11 |          "path": [
12 |            {
13 |              "_value": "(#attType='exit')::EDGE/NODE",
14 |              "_flag": "D"
15 |            }
16 |          ],
17 |          "_priority": "0",
18 |          "_order": "1"
19 |        },
20 |        "_ref": "ROOT"
21 |      },
22 |      {
23 |        "property": [
24 |          {
25 |            "action": "print(@Desc + '\\n')"
26 |          }
27 |        ],
28 |        "visit": {
29 |          "path": [
30 |            {
31 |              "_value": "(#attType='exit')::EDGE/NODE",
32 |              "_flag": "D"
33 |            }
34 |          ],
35 |          "_priority": "0",
36 |          "_order": "2"
37 |        },
38 |        "_ref": "NODE"
39 |      }
40 |    ],
41 |    "_name": "tree"
42 |  }
43 | }

```

# Listing A.4 Specifica semantica di un linguaggio in formato JSON, in questo caso quello per FlowChart

```

1 {
2   "language": {
3     "semantic": [
4       {
5         "property": [
6           {
7             "action": "print('#include <stdio.h>\n\n'); print('int main(void) {\n\n')"

```

```

    ],
    {
      "_name": "$Printed",
      "_type": "int",
      "procedure": [
        {
          "_postCondition": "",
          "_name": "assign",
          "_path": "(#attName='HEAD')::*",
          "_param": "status"
        }
      ]
    },
    {
      "_name": "$PrePred",
      "_type": "boolean",
      "procedure": [
        {
          "_postCondition": "",
          "_name": "exist",
          "_path": "(#attName='TAIL')::PRED",
          "_param": ""
        }
      ]
    },
    {
      "_name": "$End",
      "_type": "boolean",
      "procedure": [
        {
          "_postCondition": "",
          "_name": "exist",
          "_path": "(#attName='HEAD')::END",
          "_param": ""
        }
      ]
    },
    {
      "action": "if(!$PrePred && ($Printed == 'COMPLETE' || $End)) { print('goto ' + $Goto + ';\n') }"
    },
    "visit": {
      "path": [
        {
          "_value": "(#attName='HEAD')::*",
          "_flag": "D"
        }
      ],
      "_priority": "1",
      "_order": "2"
    },
    "_ref": "ARROW"
  },
  {
    "property": [
      {
        "_name": "$Count",
        "_type": "number",
        "procedure": [
          {
            "_postCondition": "",
            "_name": "size",
            "_path": "(#attName = 'Up')::*",
            "_param": ""
          }
        ]
      },
      {
        "_name": "$PrePred",
        "_type": "boolean",
        "procedure": [
          {
            "_postCondition": "",
            "_name": "exist",
            "_path": "(#attName = 'Up')::ARROW/PRED",
            "_param": ""
          }
        ]
      }
    ],
    {
      "action": "if($Count > 1 || $PrePred) { print('#id + ': ') } print(@Code + ';\n')"
    },
    "visit": {
      "path": [
        {
          "_value": "(#attType='exit')::ARROW",
          "_flag": "D"
        }
      ],
      "_priority": "1",
      "_order": "4"
    },
    "_ref": "IO"
  },
  {
    "property": [
      {
        "_name": "$Count",
        "_type": "number",
        "procedure": [
          {
            "_postCondition": "",
            "_name": "size",
            "_path": "(#attName = 'Up')::*",
            "_param": ""
          }
        ]
      }
    ]
  }
}

```

```

213     },
214     {
215         "_name": "$PrePred",
216         "_type": "boolean",
217         "procedure": [
218             {
219                 "_postCondition": "",
220                 "_name": "exist",
221                 "_path": "(#attName = 'Up')::ARROW/PRED",
222                 "_param": ""
223             }
224         ]
225     },
226     {
227         "action": "if($Count > 1 || $PrePred) { print(#id + ': ') } print(@Code + ';\n')",
228     },
229 ],
230 "visit": {
231     "path": [
232         {
233             "_value": "(#attType='exit')::ARROW",
234             "_flag": "D"
235         }
236     ],
237     "_priority": "1",
238     "_order": "3"
239 },
240 "_ref": "STAT"
241 },
242 {
243     "property": [
244         {
245             "action": "print(#id + ': ;\n')",
246         }
247     ],
248     "visit": {
249         "path": [],
250         "_priority": "2",
251         "_order": "6"
252     },
253     "_ref": "END"
254 }
255 ],
256 "_name": "FlowChartP"
257 }
258 }

```

# Bibliografia

- [1] URL <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.
- [2] URL <http://www.pc-facile.com/glossario/framework/>.
- [3] URL <https://it.wikipedia.org/wiki/JQuery>.
- [4] URL <https://it.wikipedia.org/wiki/XPath>.
- [5] cluelab. About - drawse. URL <https://cluelab.github.io/drawSE/>.
- [6] Vittorio Fuccella Gennaro Costagliola, Mattia De Rosa. Local context-based recognition of sketched diagrams. *Journal of Visual Languages and Computing*, pages 955–962, 10 2014. doi: 10.1016/j.jvlc.2014.10.021.
- [7] Vittorio Fuccella Gennaro Costagliola, Mattia De Rosa. Extending local context-based specifications of visual languages. *Journal of Visual Languages and Computing*, pages 184–195, 12 2015. doi: 10.1016/j.jvlc.2015.10.013.
- [8] Vittorio Fuccella Gennaro Costagliola, Mattia De Rosa. Using the local context for the definition and implementation of visual languages. *Computer Languages, Systems and Structures*, pages 20–38, 12 2018. doi: 10.1016/j.cl.2018.04.002.
- [9] JGraph Ltd. mxgraph. URL <https://jgraph.github.io/mxgraph/>.