Consiglio Nazionale delle Ricerche

RsdEditor: un'interfaccia visuale per la specifica delle risorse in un metacomputer

Architettura, specifiche di progetto e aspetti implementativi

Ranieri Baraglia, Domenico Laforenza
Mauro Michelotti , Simone Nannetti

Rapporto
CNUCE-B4-1999-015

Pisa

RsdEditor : un'interfaccia visuale per la specifica delle risorse in un metacomputer

Architettura, specifiche di progetto e aspetti implementativi

R. Baraglia, D. Laforenza, M. Michelotti, S. Nannetti
CNUCE - Institute of the Italian National Research Council
Via S. Maria, 36 - I-56100 Pisa (Italy)
Tel. +39-050-593111 - Fax +39-050-904052
e-mail: (R.Baraglia, D.Laforenza)@cnuce.cnr.it

Abstract

In questo rapporto viene descritta l'architettura dell'interfaccia visuale *RsdEditor*, analizzando in modo dettagliato alcune scelte di progetto e le caratteristiche dei principali moduli che la compongono.

RsdEditor è un'interfaccia grafica che permette di rappresentare le risorse di calcolo di un metacomputer, definendo anche una loro strutturazione a livelli, senza la necessità di seguire un particolare ordine nelle operazioni. È possibile salvare il lavoro svolto su file e riutilizzare file contenenti specifiche definite in precedenza; inoltre, l'interfaccia consente di editare tutti gli elementi disegnati per modificarne le caratteristiche.

RsdEditor è stata progettata per facilitare il lavoro sia dell'amministratore del sistema che dell'utente del metacomputer.

L'idea di realizzare un'interfaccia così fatta nasce da una collaborazione dell'Istituto CNUCE con il $(PC)^2$ (Paderborn Center for Parallel Computing) dell'Università di Paderborn (Germania) nell'ambito del progetto MOL.

RsdEditor è stata implementata utilizzando il linguaggio $Java^{TM}$ di *Sun Microsystem* [11], in modo da garantire la sua portabilità sul maggior numero possibile di macchine.

1 Introduzione

L'uso di un insieme eterogeneo di risorse computazionali interconnesse per mezzo di una rete e viste come un singolo calcolatore parallelo a memoria distribuita è oggi una via praticata per la soluzione di alcune classi di applicazioni complesse, anche dette *meta-applicazioni* [3, 2, 1].

Per definire questo approccio, in letteratura vengono anche utilizzati termini quali *Distributed Heterogeneous Computing* e *Metacomputing.*

Questo approccio offre notevoli vantaggi in termini di: *potenza di calcolo* (disponibilità di potenza "aggregata" derivante dall'uso congiunto di più risorse computazionali), *espandibilità della configurazione* (il numero ed il tipo delle macchine può variare secondo le specifiche necessità delle applicazioni) e *specializzazione dei moduli* (ciascun modulo di un'applicazione si può avvalere di macchine dedicate).

Un importante obiettivo dei sistemi software usati per lo sfruttamento di questo paradigma di calcolo è quello di offrire all'utente funzionalità elaborative che gli permettano di interagire facilmente con il sistema di metacalcolo "mascherandogli" l'eterogeneità presente a vari livelli. Ogni funzione dell'ambiente di metacalcolo quale, ad esempio, l'allocazione dei moduli paralleli sui processori, la gestione della sicurezza, dovrebbe nascondere all'utente la complessità della struttura che sta utilizzando. In tal senso, un metacalcolatore deve poter offrire, oltre ad una significativa potenza computazionale, anche la facilità d'uso.

L'interfaccia *RsdEditor* è una componente di un sistema di Resource Management per metacomputer. L'idea di realizzare un'interfaccia così fatta nasce da una collaborazione dell'Istituto CNUCE con il $(PC)^2$ (Paderborn Center for Parallel Computing) dell'Università di Paderborn (Germania) nell'ambito del progetto MOL [4, 5], con l'intento di fornire agli utenti del metacomputer uno strumento visuale con il quale specificare le loro richieste di risorse.

Il progetto MOL utilizza il sistema CCS (Computing Center Software) [6] per la gestione delle risorse. In tale sistema viene utilizzato il linguaggio di specifica RSD (*Resource and Service Description*) [10] per la descrizione delle risorse appartenenti al metacomputer gestito da CCS. Sia CCS che RSD sono stati sviluppati presso il $(PC)^2$.

RsdEditor è stata implementata utilizzando il linguaggio $Java^{TM}$ di *Sun Microsystem* [11], in modo da garantire la sua portabilità sul maggior numero possibile di macchine.

Utilizzando *RsdEditor* è possibile rappresentare le risorse di calcolo, definendo anche una loro strutturazione a livelli, senza la necessità di seguire un particolare ordine nelle operazioni. È possibile salvare il lavoro svolto su file e riutilizzare file contenenti specifiche definite in precedenza; inoltre, l'interfaccia

consente di editare tutti gli elementi disegnati per modificarne le caratteristiche.

RsdEditor è stata progettata per facilitare il lavoro sia dell'amministratore del sistema che dell'utente del metacomputer.

2 Il linguaggio di programmazione

Il linguaggio di programmazione usato per implementare l'interfaccia grafica *RsdEditor* è **Java** [13, 11, 12] di *Sun Microsystem*, in modo da soddisfare uno degli obiettivi primari di progetto, la *portabilità* dell'applicazione realizzata. Infatti un'applicazione Java è *indipendente dalla piattaforma hardware*. In pratica, è sufficiente scrivere il codice dell'applicazione e compilarlo su una macchina sulla quale è installato un compilatore Java; esso produce in output alcune classi (o *bytecode*[1]) che è possibile eseguire su tutte le macchine su cui è installata una *JVM* (*Java Virtual Machine*), l'interprete runtime delle classi Java.

Java è un linguaggio orientato agli oggetti. Questo tipo di programmazione organizza un programma come una serie di componenti chiamati `oggetti`, che sono indipendenti l'uno dall'altro ma seguono delle regole per comunicare tra loro. Seguendo questo criterio i programmi sono più adattabili per l'uso in progetti diversi, più facili da capire e meno inclini agli errori.

Gli strumenti di sviluppo che abbiamo utilizzato sono: la versione 1.1.6 del *Java Development Kit (JDK)* e la versione 1.0.2 delle librerie *Swing*, entrambe sviluppate da Sun Microsystem. JDK è stato scelto perché all'inizio dell'implementazione dell'interfaccia era l'ambiente più stabile tra quelli analizzati e che metteva a disposizione le JVM per tutte le piattaforme su cui l'applicazione *RsdEditor* doveva essere eseguita. Le librerie Swing forniscono una serie di oggetti grafici già implementati che semplificano il lavoro del programmatore.

Prima del completamento dell'implementazione di *RsdEditor* *Sun Microsystems* ha rilasciato anche la versione 1.2 del JDK, ma è stato deciso di non adottarla perché non erano ancora disponibili tutte le versioni per le piattaforme su cui doveva essere testato il software sviluppato. Il vantaggio che introduce JKD 1.2 è l'inclusione del pacchetto Swing come libreria grafica ed il perfezionamento di alcune classi già esistenti. In ogni caso, nel momento in cui saranno disponibili tutte le versioni, per adeguare il codice dell'applicazione alla nuova

[1] I *bytecode* sono una serie di istruzioni simili a quelle in codice macchina create dalla compilazione di un programma Java; la differenza è che il codice macchina deve essere eseguito sul computer per il quale è stato compilato, mentre i bytecode possono essere eseguiti su un qualsiasi sistema sul quale sia installata una JVM (Java Virtual Machine).

versione del JDK basterà semplicemente sostituire i riferimenti (istruzioni `import`) alle classi con quelle del nuovo JDK e ricompilare l'applicazione.

L'interfaccia *RsdEditor* è stata testata con successo su piattaforma Microsoft Windows 95, Microsoft Windows NT, RedHat Linux e Sun Solaris.

3 Il linguaggio GUI

RsdEditor produce in output due file: il primo contiene la specifica della richiesta di risorse in sintassi RSD prodotta dall'utente, mentre, il secondo contiene le caratteristiche grafiche di tutti gli oggetti disegnati. La scelta di generare due file separati è stata dettata dall'esigenza di produrre un file in sintassi RSD, rigoroso, cioè non appesantito da informazioni non pertinenti con la specifica di risorse.

La sintassi del linguaggio RSD è stata definita presso il *Paderborn Center for Parallel Computing* $(PC)^2$. Per rappresentare le caratteristiche grafiche degli oggetti è stato definito un semplice linguaggio testuale denominato **linguaggio GUI**.

3.1 Definizione del linguaggio GUI

Gli oggetti grafici da memorizzare sono di tre tipi: nodi, edge fisici ed edge virtuali. Il formato del linguaggio GUI riflette la struttura logica del formato del linguaggio RSD e quindi vi ritroviamo sia la stessa gerarchia fra gli oggetti che lo stesso ordine di specifica.

Ogni oggetto è identificato da una parola chiave a cui è associato un insieme fissato di righe testuali che hanno il seguente formato:

caratteristica=valore

dove *caratteristica* è la chiave utilizzata per la memorizzazione delle proprietà grafiche degli oggetti. Ciò permette di recuperare, nel momento della rilettura del file in formato GUI, le informazioni in modo più veloce. Quindi, ciascuna di queste coppie descrive una particolare informazione grafica (nome, colore, coordinate, ecc.).

La Figura 1 mostra il formato generale di specifica di un oggetto.

Per rappresentare tre tipi di oggetti, sono state definite le tre parole chiave:

- `NODE`, per i nodi, gli ipernodi e per le topologie;
- `EDGE`, per gli edge fisici (orizzontali);
- `ASSIGN`, per gli edge virtuali (verticali).

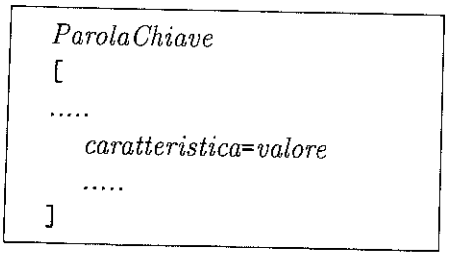

Figure 1: Formato generale di specifica di un oggetto

```
NODE
[
        Topo=Node
        Name=...
        Point=...,...
        Fill=...
        Shape=...
        Size=...
        Color=...,...,...
        Font=...,...
]
```

Figure 2: Specifica di un nodo secondo il formato GUI

NODE. La parola chive `NODE` è utilizzata per indicare tre oggetti grafici diversi,che sono trattati allo stesso modo grazie alle loro affinità dal punto di vista grafico. Tali oggetti sono i **nodi** veri e propri (cioè gli elementi che rappresentano risorse fisiche), gli **ipernodi** (cioè gli elementi che contengono altri oggetti) e le **topologie** (cioè le interconnessioni tra macchine omogenee). Le topologie eterogenee non hanno un corrispettivo nel linguaggio GUI anche se, utilizzando l'interfaccia *RsdEditor*, è possibile introdurle creando singolarmente ogni nodo che compone la topologia eterogenea.

Un **nodo** generico viene rappresentato graficamente basandosi sul valore di otto parametri: tipo, nome, posizione, riempimento, forma, grandezza, colore e font per il nome. Esso viene rappresentato con la porzione di testo mostrata in Figura 2.

Il nome del nodo (`Name`) viene specificato mediante una stringa. L'elemento `Point` è rappresentato da una coppia di interi che indicano le coordinate x ed y del nodo sul workspace. `Fill` può assumere solo due valori, `Fill` od `Empty`, per indicare se il nodo è rappresentato da una figura geometrica piena o no. Anche `Shape` ha due valori possibili, `Round` o `Square`, per indicare la forma assegnata al nodo. Il valore di `Size` è un intero che può variare da 30 a 100 ed esprime la

dimensione in punti del nodo. Per `Color` devono essere indicati tre valori interi che specificano la terna RGB[2] del colore applicato. Infine l'elemento `Font` ha una prima componente che indica il nome del font con cui scrivere il nome del nodo e una seconda componente che indica la grandezza del carattere.

Gli **ipernodi** indicano gli oggetti grafici che contengono, al loro interno, altri oggetti grafici. La specifica in formato GUI degli ipernodi è composta da due sezioni. La prima sezione è identica a quella che esprime i nodi, mentre la seconda elenca, seguendo l'ordine `NODE`, `EDGE`, `ASSIGN`, le varie sezioni che descrivono gli oggetti costituenti l'ipernodo.

La generica struttura sintattica dell'ipernodo è illustrata in Figura 3.

Ciò che diversifica le topologie è la prima riga ed in particolare il valore attribuito alla caratteristica `Topo` che può assumere i valori:

GRID	`Topo=GRID,...,...`
RING	`Topo=RING,...`
STAR	`Topo=STAR,...`
TORUS	`Topo=TORUS,...,...`

dove ogni componente "..." sarà sostituita da un intero che indica il numero di nodi che compongono la topologia. Per `GRID` e `TORUS` le componenti numeriche saranno due in quanto deve essere definita una matrice di nodi.

Le **topologie** rappresentano un insieme di macchine omogenee interconnesse tra di loro da reti dello stesso tipo. L'interfaccia *RsdEditor* permette di specificare quattro tipi di topologie di ampio utilizzo: `GRID`, `RING`, `STAR` e `TORUS`.

Le quattro topologie sono rappresentate da uno statement composto da tre righe (vedi Figura 4 (a)).

EDGE. La stringa `EDGE` identifica l'oggetto grafico che rappresenta uno o più edge fisici od orizzontali. Infatti, un insieme di edge fisici che collegano gli stessi due nodi viene rappresentato da un unico oggetto grafico, la stessa cosa vale per gli edge virtuali. Questo tipo di oggetto viene rappresentato attraverso quattro parametri: nome, nome nodo sorgente, nome nodo destinazione e colore. Il relativo statement viene mostrata in Figura 4 (b) in cui i valori degli elementi `Source` e `Destination` sono stringhe ed indicano, rispettivamente, il nome del nodo sorgente ed il nome del nodo destinazione che l'oggetto grafico `EDGE` collega.

ASSIGN. Con la stringa `ASSIGN` (vedi Figura 4 (c)) si indica l'oggetto grafico che rappresenta uno o più edge virtuali o verticali. In questo caso non è necessario introdurre l'elemento `Destination`. Infatti, il nodo destinazione

[2]Il termine RGB è la sigla di **R**ed**G**reen**B**lue, esso viene utilizzato dalla maggior parte dei software di grafica per definire un colore attraverso tre valori numerici. Tali valori possono variare da 0 a 255 e si basano su tre colori fondamentali: rosso, verde e blu.

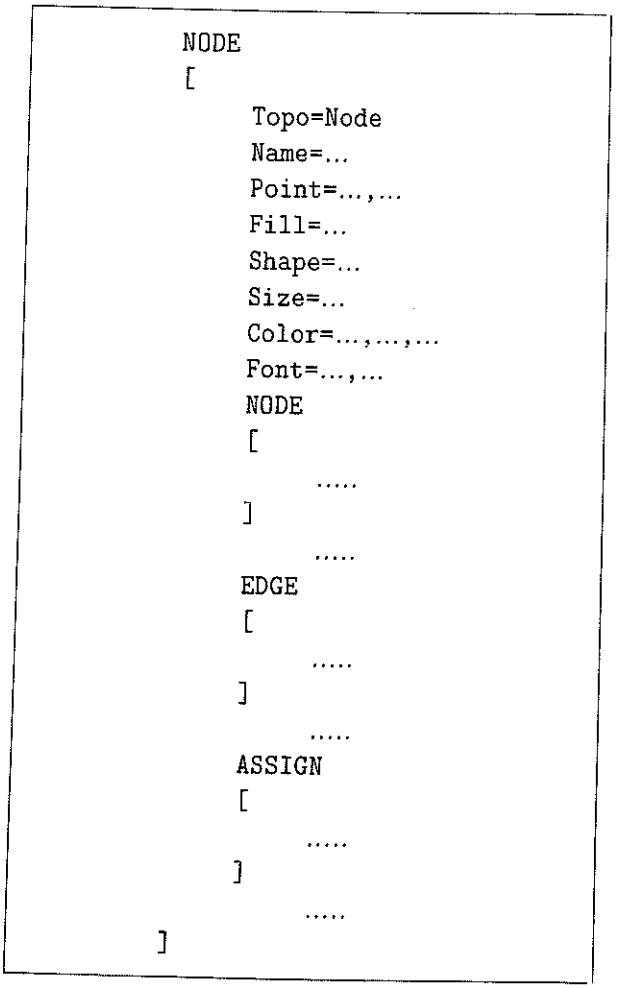

Figure 3: Specifica di un generico ipernodo secondo il formato GUI

dello edge è il suo ipernodo, cioè il nodo padre che contiene tale edge, e questa informazione può essere recuperata immediatamente.

Di seguito è mostrato un esempio di file in formato GUI in cui si possono ritrovare, in maniera dettagliata, tutti gli statement sopra elencati.

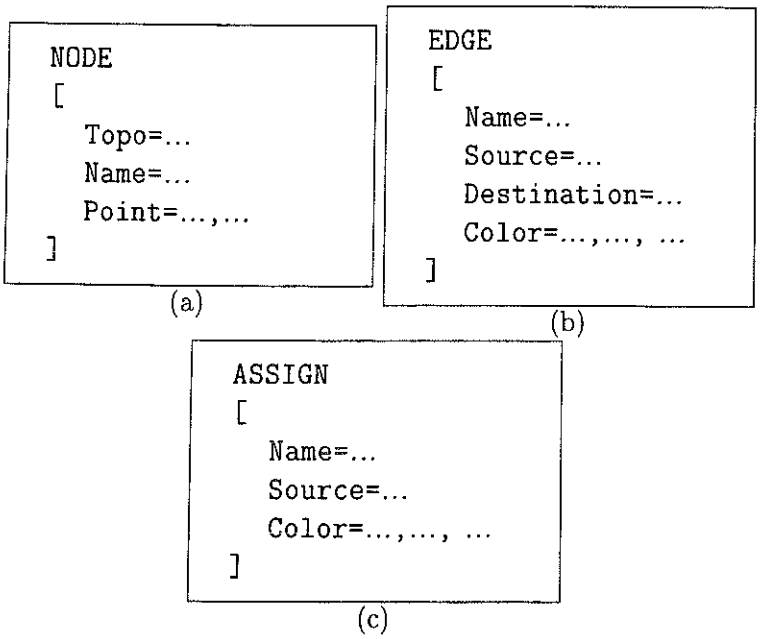

Figure 4: Specifica di una topologia (a), di uno edge fisico (b) e di uno edge virtuale (c) secondo il formato GUI

```
<
# Nome File: Esempio.gui
#
# File generato automaticamente
# quindi non modificare
>
{

NODE
[
   Topo=Node
   Name=CNUCE
   Point=269,204
   Fill=Fill
   Shape=Round
   Size=100
   Color=128,128,128
   Font=Monospaced,24
   NODE
   [
      Topo=Node
      Name=Others
      Point=460,163
      Fill=Fill
      Shape=Round
      Size=100
      Color=255,97,0
      Font=DialogInput,16
      NODE
      [
         Topo=Node
         Name=Brunello
         Point=166,125
         Fill=Fill
         Shape=Round
         Size=80
         Color=255,200,0
         Font=TimesRoman,16
      ]
      NODE
      [
         Topo=Node
         Name=Prosecco
         Point=458,120
         Fill=Fill
         Shape=Round
         Size=80
         Color=255,200,0
         Font=TimesRoman,16
      ]
      NODE
      [
         Topo=Node
         Name=Ethernet2
         Point=322,361
         Fill=Fill
         Shape=Round
         Size=50
         Color=255,0,0
         Font=TimesRoman,16
      ]
      EDGE
      [
         Name=GraphicEdge3
         Source=Prosecco
         Destination=Ethernet2
         Color=217,142,106
      ]
      EDGE
      [
         Name=GraphicEdge4
         Source=Brunello
         Destination=Ethernet2
         Color=217,142,106
      ]
      ASSIGN
      [
         Name=GraphicVirtual1
         Source=Brunello
         Color=99,113,142
      ]
]
NODE
[
   Topo=Node
   Name=Topology
   Point=309,315
   Fill=Fill
   Shape=Round
   Size=100
   Color=255,86,0
   Font=SansSerif,16
   NODE
   [
      Topo=Node
      Name=Controller
      Point=275,89
      Fill=Fill
      Shape=Round
      Size=80
      Color=0,0,255
      Font=Monospaced,18
   ]
   NODE
   [
      Topo=RING,10
      Name=Anello
      Point=100,256
   ]
   NODE
   [
      Topo=STAR,10
      Name=Stella
      Point=484,232
   ]
   NODE
   [
      Topo=GRID,16,8
      Name=Matrice
      Point=283,345
   ]
   EDGE
   [
      Name=GraphicEdge6
      Source=Stella
```

```
            Destination=Controller
            Color=217,142,106
        ]
        EDGE
        [
            Name=GraphicEdge7
            Source=Anello
            Destination=Controller
            Color=217,142,106
        ]
        EDGE
        [
            Name=GraphicEdge0
            Source=Matrice
            Destination=Controller
            Color=217,142,106
        ]
        ASSIGN
        [
            Name=GraphicVirtual2
            Source=Controller
            Color=99,113,142
        ]
    ]
    EDGE
    [
        Name=GraphicEdge8
        Source=Others
        Destination=Topology
        Color=217,142,106
    ]
]
}
```

3.2 Grammatica

Il linguaggio GUI riflette le modalità di rappresentazione e di introduzione degli oggetti grafici sul foglio di disegno dell'interfaccia *RsdEditor*. Per la generazione del linguaggio abbiamo innanzitutto definito una *grammatica non contestuale*[3] [14] come riportato in seguito:

- Assioma : **S**
- Simboli non terminali : **N**, **E**, **A**
- Simboli terminali : {, }, [,], NODE, EDGE, ASSIGN, ATTR-N, ATTR-E
- Produzioni :
  S $\longrightarrow$ { **N** }
  N $\longrightarrow$ NODE [ATTR-N **N**] **N**
  N $\longrightarrow$ **E**
  E $\longrightarrow$ EDGE [ATTR-E] **E**
  E $\longrightarrow$ **A**
  A $\longrightarrow$ ASSIGN **A**
  A $\longrightarrow$ ε

Data la grammatica, per vedere se una stringa appartiene al linguaggio GUI, il modo più semplice è quello di fare un'analisi TOP-DOWN. In pratica, supponiamo di avere un puntatore alla stringa in ingresso e, basandoci su un certo numero di simboli in lettura, decidiamo, fra le varie alternative, la produzione da applicare. Partendo dall'assioma, l'obiettivo è quello di costruire un albero la cui frontiera sia la stringa data in ingresso.

Un grammatica che è analizzabile in modo TOP-DOWN si dice **grammatica LL** (dove LL significa "Left to right" "Left most derivation" cioè si leggono le stringhe da sinistra verso destra e si effettua una derivazione dal lato più a sinistra). Abbiamo costruito la grammatica precedente in modo che risulti di tipo **LL(1)**, cioè, nella fase di analisi è sufficiente analizzare il primo simbolo in lettura per sapere quale produzione applicare. Nel seguito verifichiamo che la grammatica proposta presenti tali caratteristiche.

Data una grammatica G, con x $\in$ V $\cup$ T e A $\in$ V(con V simboli non terminali e T simboli terminali), definiamo gli **INIZI** ed i **SEGUITI**.

Gli INIZI di un simbolo x vengono definiti nel seguente modo:

1. x $\in$ T allora INIZI(x) = {x};

[3]Un grammatica non contestuale è una quadrupla G=(S,T,N,P) con S assiomi, T simboli terminali, N simboli non terminali e P insieme di regole della forma $\alpha \Longrightarrow \beta$ (α,β stringhe di terminali e non terminali).

2. x $\in$ V e x $\rightarrow$ aα è una produzione allora a $\in$ INIZI(x)
   (se x $\rightarrow \varepsilon$ allora $\varepsilon \in$ INIZI(x));

3. x $\rightarrow y_1 y_2 ... y_k$ allora per ogni i tale che $y_1 ... y_{i-1} \in$ V e $\varepsilon \in$ INIZI(y_j) per j=1,2,...,i-1 INIZI(y_i)-{ε} $\subseteq$ INIZI(X)
   (se ε *in* INIZI(y_j) per j=1,2,...,k allora $\varepsilon \in$ INIZI(X)).

I SEGUITI vengono definiti secondo i seguenti criteri:

1. $\varepsilon \in$ SEGUITI(S) dove S simbolo iniziale;

2. se A $\rightarrow \alpha$Bβ INIZI(β)-{ε} $\subseteq$ SEGUITI(B);

3. se A $\rightarrow \alpha$B oppure A $\rightarrow \alpha$Bβ con $\varepsilon \in$ INIZI(β) allora
   SEGUITI(A) $\subseteq$ SEGUITI(B).

Data una grammatica G ed una sua produzione p:$A \longrightarrow d$ la **GUIDA** di p è definita come:

`GUIDA(p) = { a | a ∈ INIZI(d) ∨ (d ⟹* ε ∧ a ∈ SEGUITI(A))}`.

Date le precedenti definizioni possiamo dire che una grammatica è LL(1) se e solo se per ogni coppia di produzioni le guide sono disgiunte; in maniera formale:

`G è LL(1)` $\iff \forall$ `coppia di produzioni` p_i`:A⟶`d_i, p_j`:A⟶`d_j`:`
`GUIDA(`p_i`) ∩ GUIDA(`p_j`) = ∅`.

A questo punto possiamo passare al calcolo vero e proprio degli INIZI e dei SEGUITI per la nostra grammatica che verranno utilizzati per confrontare le GUIDE.

3.2.1 Inizi

- INIZI (**A**) = { ASSIGN , ε }
- INIZI (**E**) = { EDGE } $\cup$ INIZI(**A**) = { EDGE , ASSIGN , ε }
- INIZI (**N**) = { NODE } $\cup$ INIZI(**E**) = { NODE EDGE , ASSIGN , ε }
- INIZI (**S**) = { '{' }

3.2.2 Seguiti

- SEGUITI (**S**) = { ε }
- SEGUITI (**N**) = { '}' , ']' }
- SEGUITI (**E**) = SEGUITI(**N**) = { '}' , ']' }
- SEGUITI (**A**) = SEGUITI(**E**) = SEGUITI(**N**) = { '}' , ']' }

3.2.3 Guide

1. • GUIDE (**N** $\longrightarrow$ NODE [ATTR-N **N**] **N**) =
   = INIZI (NODE [ATTR-N **N**] **N**) = { NODE }
   • GUIDE (**N** $\longrightarrow$ **E**) = INIZI (**E**) $\cup$ SEGUITI (**N**) =
   = { EDGE , ASSIGN , ε , '}' , ']' }
2. • GUIDE (**E** $\longrightarrow$ EDGE [ATTR-E] **E**) =
   = INIZI (EDGE [ATTR-E] **E**) = { EDGE }
   • GUIDE (**E** $\longrightarrow$ **A**) = INIZI (**A**) $\cup$ SEGUITI(**E**) =
   = { ASSIGN , ε , '}' , ']' }
3. • GUIDE (**A** $\longrightarrow$ ASSIGN **A**) =
   = INIZI (ASSIGN **A**)= { ASSIGN }
   • GUIDE (**A** $\longrightarrow \varepsilon$) = SEGUITI (**A**) =
   = { '}' , ']' }

Poiché le Guide risultano essere disgiunte per ogni coppia di produzioni, è chiaro che la grammatica proposta è LL(1); può quindi essere analizzata mediante un*analizzatore a discesa ricorsiva.* In pratica, per la realizzazione di un tale analizzatore, viene creata una procedura per ogni simbolo non terminale della grammatica. Questa procedura, costruita in base alla definizione della grammatica, ha multiple attivazioni che vengono chiuse in ordine inverso rispetto a come vengono aperte.

3.2.4 Tabella per analisi LL(1)

Per una maggiore completezza dell'analisi, si mostra anche la tabella LL(1) necessaria per l'*analisi predittiva.* La tabella viene costruita inserendo una riga per ogni simbolo non terminale e una colonna per ogni simbolo terminale (ε viene sostituito da \$). Inoltre, per ogni produzione A$\rightarrow\alpha$, devono essere osservati i seguenti criteri:

1. a $\in$ INIZI(α) allora aggiungi la produzione A$\rightarrow\alpha$ a T[A,a];
2. $\varepsilon \in$ INIZI(α) allora aggiungi la produzione A$\rightarrow\varepsilon$ a T[A,b] per ogni b $\in$ SEGUITI(A);
3. $\varepsilon \in$ INIZI(α)e $\varepsilon \in$ SEGUITI(A) allora aggiungi la produzione A$\rightarrow\varepsilon$ a T[A,\$].

Da essa risulta che non esistono, in nessun caso, conflitti fra le produzioni da applicare perché in ogni cella della tabella c'è al massimo una produzione. Questo equivale a dire che per ogni simbolo in lettura sappiamo quale produzione applicare.

Tabella LL(1)				
	$	{	}	]
S		S ⟶ { N }		
N			N ⟶ ε	N ⟶ ε
E			E ⟶ ε	E ⟶ ε
A			A ⟶ ε	A ⟶ ε

Tabella LL(1)			
	NODE	EDGE	ASSIGN
S			
N	N ⟶ NODE [ATTR N] N	N ⟶ E	N ⟶ E
E		E ⟶ EDGE [ATTR] E	E ⟶ A
A			A ⟶ ASSIGN A

4 Internazionalizzazione

L'*internazionalizzazione* di un'applicazione consiste nella possibilità di mostrare menu, pulsanti, etichette, messaggi di errore e altri elementi testuali (ed eventualmente grafici) in varie lingue.

Per l'implementazione di questo caratteristica Java introduce il concetto di "locale" espresso mediante la classe **java.util.Locale**. Tale classe rappresenta una zona dal punto di vista sia linguistico che geografico. Una zona viene infatti individuata sia per lingua che per area geografica. Più precisamente, per indicare le lingue si utilizza lo standard *ISO Language Code* [4], mentre per indicare le aree geografiche si utilizza lo standard *ISO Country Code* [5]. Il primo identifica una lingua con due lettere minuscole mentre il secondo utilizza due lettere maiuscole per identificare la regione. Per ottenere l'istanza desiderata è necessario passare, come parametri, al costruttore della classe Locale le due stringhe Language e Country code. Per esempio per l'Italia, si può creare il locale con **new Locale("it", "IT")**.

Per riuscire a internazionalizzare facilmente un'interfaccia grafica, occorre concentrare tutte le informazioni che possono variare da una lingua all'altra in file con un formato facilmente modificabile. La traduzione, infatti, spesso viene fatta da personale che non è necessariamente un programmatore in grado di modificare il codice dell'applicazione.

Ovviamente occorre scrivere l'applicazione in modo che faccia uso esplicito di questa tecnica di progettazione. La classe principale da utilizzare è **java.util.ResourceBundle**. Questa classe è astratta e non può essere istanzi-

[4] Una lista completa di questi codici si può trovare all'indirizzo
http:/www.ics.uci.edu/pub/ieft/http/related/iso639.txt

[5] Una lista completa di questi codici si può trovare all'indirizzo
http:/www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html

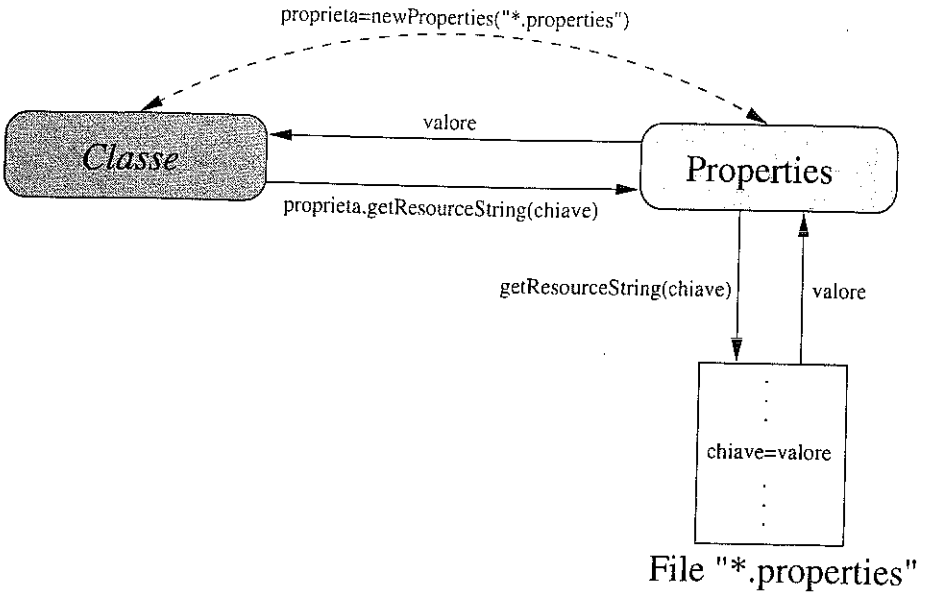

Figure 5: Interazione con la classe Properties per la richiesta di una stringa

ata. Si utilizzano invece le classi instanziabili **java.util.ListResourceBundle** (che definisce le risorse utilizzando delle costanti espresse in codice Java) o **java.util.PropertyResourceBundle** (che consente di specificare le risorse utilizzando dei file di proprietà).

L'interfaccia *RsdEditor* applica il secondo metodo e quindi utilizza dei file di proprietà. Questi file sono contenuti nella directory **language** del pacchetto completo dell'interfaccia ed il loro nome ha il seguente formato:

NomeFile_CodiceLingua.`properties`.

L'interfaccia *RsdEditor*, nel momento in cui viene attivata, mette a disposizione una prima finestra per scegliere la lingua da utilizzare. Nella finestra sono elencate le lingue selezionabili (attualmente sono previste l'inglese e l'italiano) ed in base alla scelta effettuata, ad esempio italiano, viene impostata una variabile globale con l'istruzione:

```
Locale.setDefault(new Locale(''it'',''IT''));
```

A questo punto viene visualizzata la finestra principale dell'interfaccia e tutti gli elementi che la compongono e che si diversificano in base alla lingua, utilizzano il locale come parametro per individuare la giusta traduzione.

La Figura 5 mostra l'interazione tra la generica classe *Classe* e la classe **RsdEditor.common.Properties** per la richiesta di una stringa.

In pratica, *Classe* effettua un'istanziazione della classe *Properties*, denom-

inata *proprieta*, passando come parametro il file "*.properties" in cui è memorizzato l'elemento da ricercare. Attraverso questa istanza, *Classe* richiama il metodo **getResourceString** della classe *Properties* passandogli come parametro la stringa **chiave**. Tale metodo ricerca all'interno del file la stringa **chiave** e restituisce la stringa **valore** ad essa associata. La stringa **valore** può identificare un'etichetta, un messaggio, un path del file system, e viene utilizzata per inserire l'elemento identificato in una sezione dell'interfaccia *RsdEditor*.

Come detto precedentemente, l'interfaccia *RsdEditor* offre la possibilità di utilizzare due lingue, l'italiano e l'inglese. Per utilizzarla con un'altra lingua è sufficiente fare una copia di tutti i file del tipo "*nomeFile*_en.properties", contenuti nella directory **language**, e sostituire la parte destra di ogni riga con la traduzione nella nuova lingua.

L'unico file che deve esistere in unica copia è il file "*Language*_en.properties" (vedi Figura 6) perchè contiene le informazioni da inserire nella finestra iniziale dell'interfaccia, visualizzata in ogni caso con i suoi componenti tradotti in inglese.

```
###################################
#        STARTUP CONFIGURATION        #
###################################

# Languages
languages=english italian
# add a string for a new language
# other languages are:german, french, spanish

# Codes
englishCode=en EN
italianCode=it IT
# germanCode=de DE
# frenchCode=fr FR
# spanishCode=es ES
# You can find a full list of first code at:
# http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt
# You can find a full list of second code at:
# http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html

# Images
englishImage=images/flag_en.gif
italianImage=images/flag_it.gif
# germanImage=images/flag_de.gif
# frenchImage=images/flag_fr.gif
# spanishImage=images/flag_es.gif
```

Figure 6: File Language_en.properties

Il file mostrato in Figura 6 è già predisposto per altre tre lingue: tedesco, francese e spagnolo. Per tali lingue basta aggiungere la stringa che rappresenta la lingua (`german`, `french`, `spanish`) alla fine della riga `languages=...` e,

togliere il carattere di commento (#) dalla relativa voce nelle sezioni `Codes` e `Images`. Per le lingue non previste è necessario inserire un nuovo elemento in ogni sezione del file.

Concludendo, possiamo dire che questa scelta di progetto permette una "traduzione" dell'interfaccia in un specifica lingua in modo semplice ma soprattutto senza la necessità di modificare il codice dell'applicazione.

5 Descrizione delle classi

La struttura dei moduli che compongono l'applicazione *RsdEditor* sfrutta uno degli strumenti forniti da Java: la suddivisione in *package*. Essi costituiscono un valido meccanismo per gestire gruppi composti da molte classi. L'utilizzo dei package permette di:

- raggruppare interfacce e classi correlate;
- definire identificatori disponibili solo all'interno di un package che siano utilizzabili dal package stesso ma inaccessibili dal di fuori, evitando così possibili conflitti fra i nomi.

In pratica all'inizio di ogni file sorgente deve essere inserita l'istruzione:

`package` *Identificatore*;

che dichiara il package a cui appartengono le classi Java scritte nel file. Per poter utilizzare classi che non fanno parte del package in cui si sta lavorando è necessario utilizzare l'istruzione:

`import` *Identificatore*;

dove *Identificatore* è il nome della classe esterna al package.

La nostra applicazione è costituita da un insieme di package organizzati secondo la struttura mostrata in Figura 7. Come criterio di raggruppamento delle classi, è stato definito un package per ogni oggetto grafico che può essere creato con l'interfacccia (**node**, **edge**, **topology** e **virtual**) associandogli proprio il nome dell'oggetto. Quindi, le classi che si riferiscono direttamente ad un oggetto grafico e che sono utilizzate solamente in quel contesto sono state incluse nel relativo package.

La Figura 8 mostra la composizione dei quattro package associati agli oggetti grafici. Il package **window** include le classi che sono collegate direttamente alla finestra principale dello *RsdEditor* e, per maggior chiarezza, è stato suddiviso in due ulteriori package **gui** e **option**. Il primo racchiude le classi che

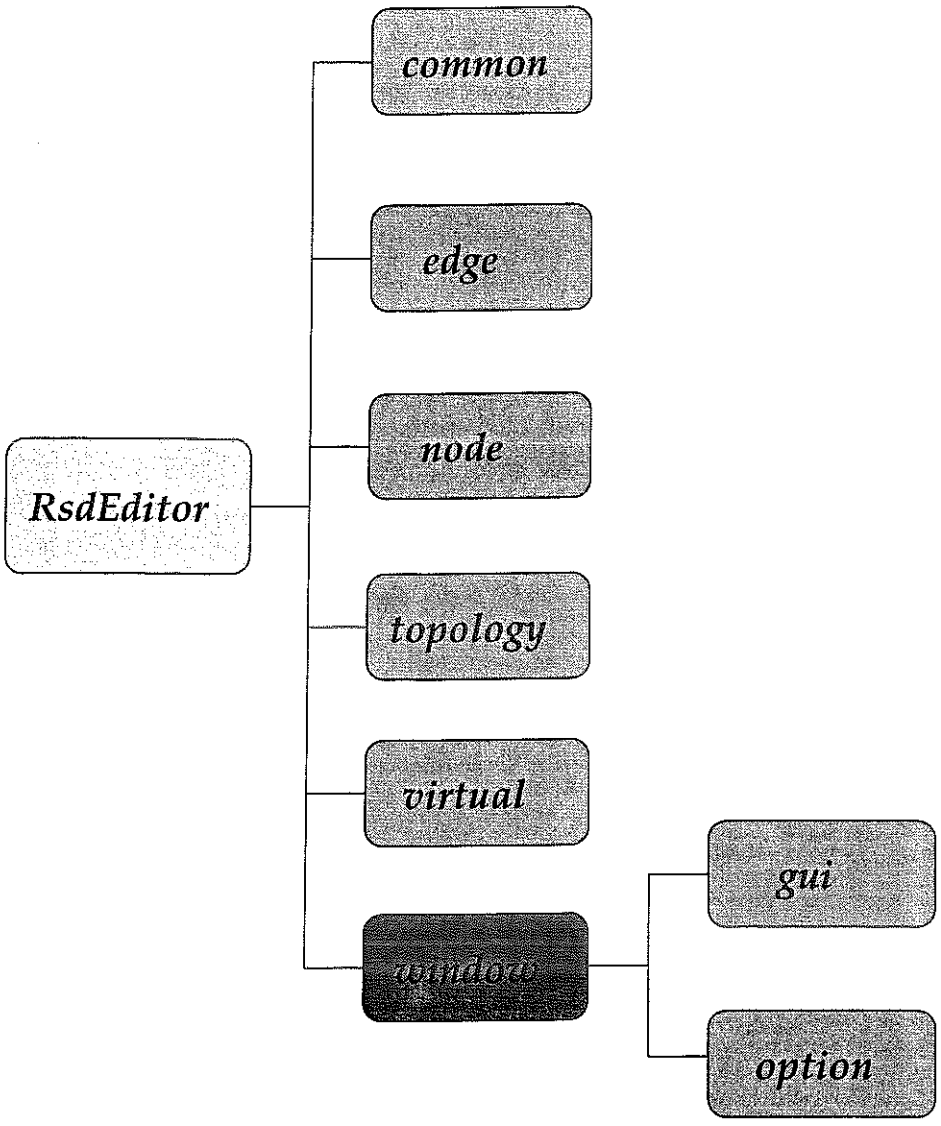

Figure 7: Struttura dei package costituenti l'applicazione

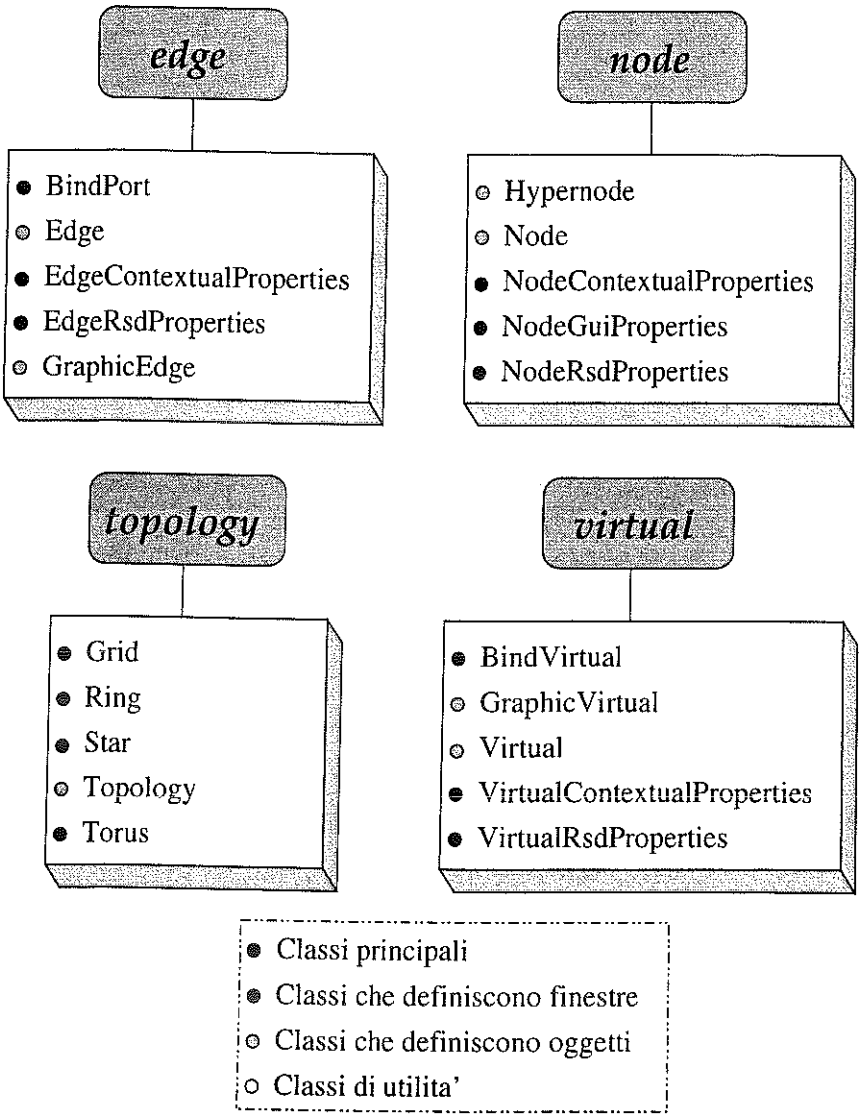

Figure 8: Elenco delle classi incluse nei package *edge*, *node*, *topology* e *virtual*

definiscono gli elementi che costituiscono la finestra principale (menu, barra di stato, barre degli strumenti, foglio di disegno e albero associato al grafo disegnato).

Il secondo package, invece, è composto da tutte le classi che definiscono le finestre secondarie visualizzate solo in determinate operazioni (file manager, scelta del colore, ecc.) o che definiscono azioni sul foglio di disegno (ad esempio la classe `RsdEditor.window.option.GraphicAction`). In quest'ultimo package sono comprese anche alcune classi di utilità necessarie alla definizione delle finestre (ad esempio i filtri del file manager) od associate agli oggetti grafici (ad esempio il tooltip composto da più righe).

Infine è stato creato il package **common** per raggruppare le classi comuni a più oggetti grafici e che non sono quindi associate ad un singolo oggetto. La Figura 9 mostra la composizione di quest'ultimi tre package.

5.1 Classi principali

In questo paragrafo viene data una breve descrizione di ogni classe, indicandone i principali metodi ed il loro utilizzo. Inoltre vengono evidenziate le interazioni tra le classi utilizzate per l'implementazione di alcune funzionalità dell'interfaccia *RsdEditor*.

5.1.1 RsdEditor.RsdEditor

La classe `RsdEditor` rappresenta la classe principale dell'applicazione. Infatti, essa definisce il metodo `main` al quale fa riferimento l'interprete Java per attivare l'esecuzione del programma.

Con la classe `RsdEditor` viene creata la finestra iniziale attraverso la quale viene scelta la lingua e viene impostato il tipo di utente (amministratore di sistema o utente). Alla fine di queste impostazioni viene visualizzata la finestra principale dell'interfaccia creata mediante l'istanziazione delle classi che definiscono i suoi componenti (`Menu`, `TopBar`, `FunctionBar`, `GraphTree`, `StateBar`, `WorkCanvas` e `WorkSpace`). In Figura 10 viene evidenziata la corrispondenza fra queste classi ed i componenti dell'interfaccia.

I metodi definiti nella classe `RsdEditor` riguardano la visualizzazione dell'albero, il controllo sulla validità dei nomi degli attributi secondo le regole del linguaggio RSD e la modalità che descrive l'operazione che si sta svolgendo con l'interfaccia. La sezione finale del codice di questa classe riguarda la gestione degli eventi generati dai componenti della finestra iniziale.

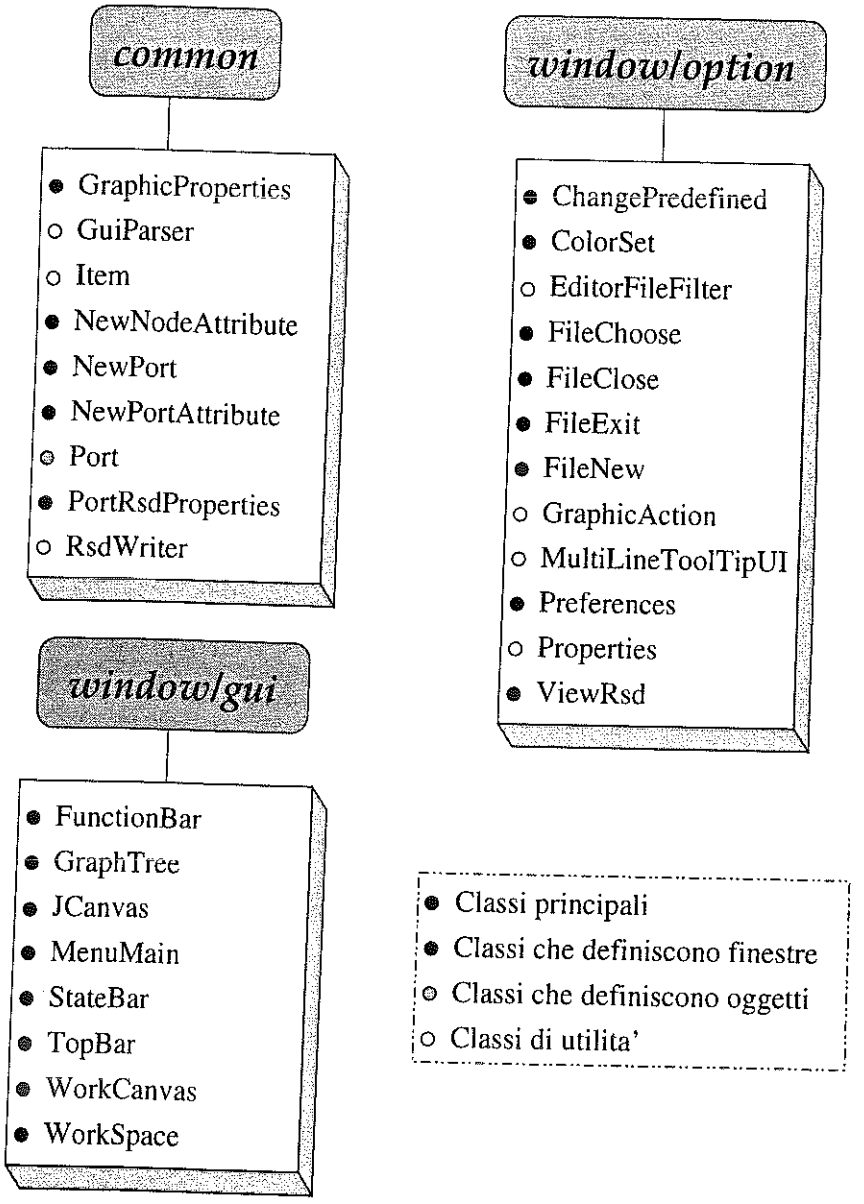

Figure 9: Elenco delle classi incluse nei package *common*, *window/gui* e *window/option*

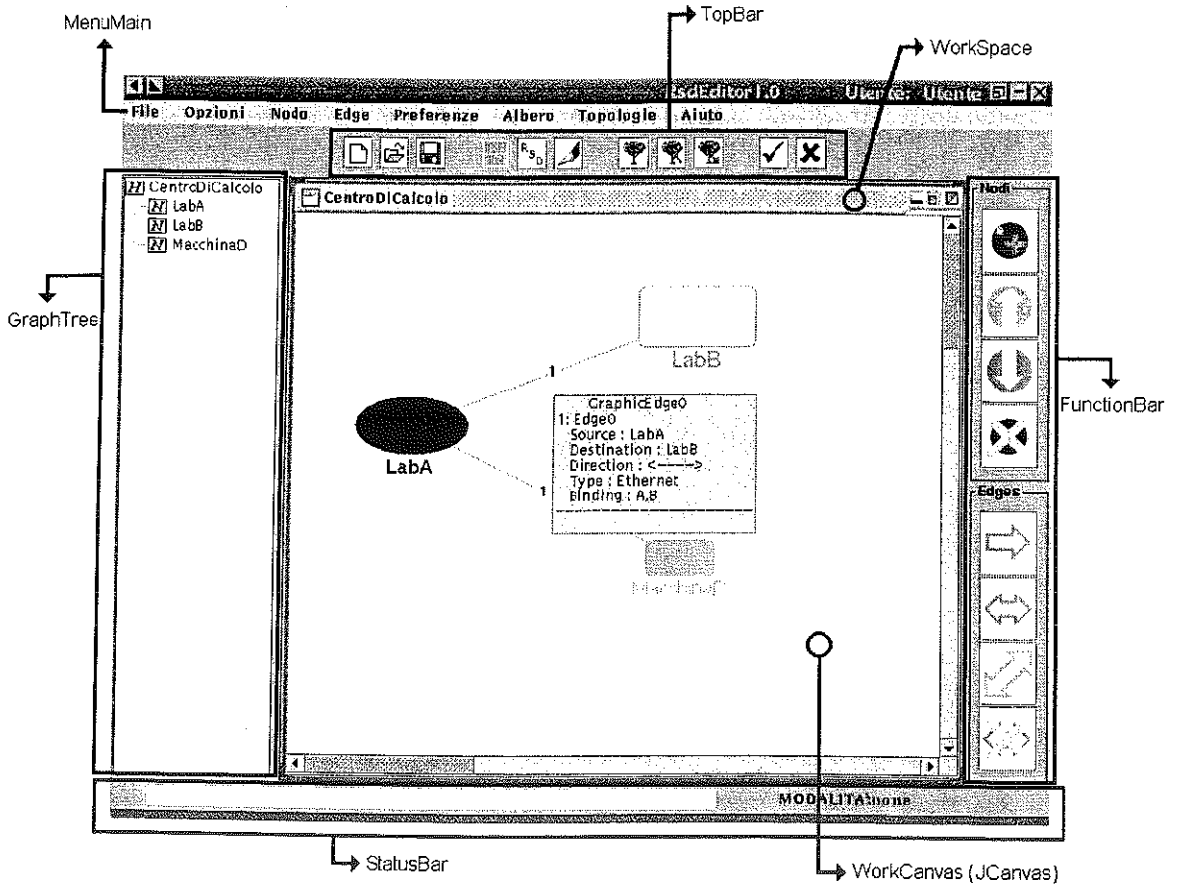

Figure 10: Dislocazione delle classi che definiscono i componenti dell'interfaccia *RsdEditor*

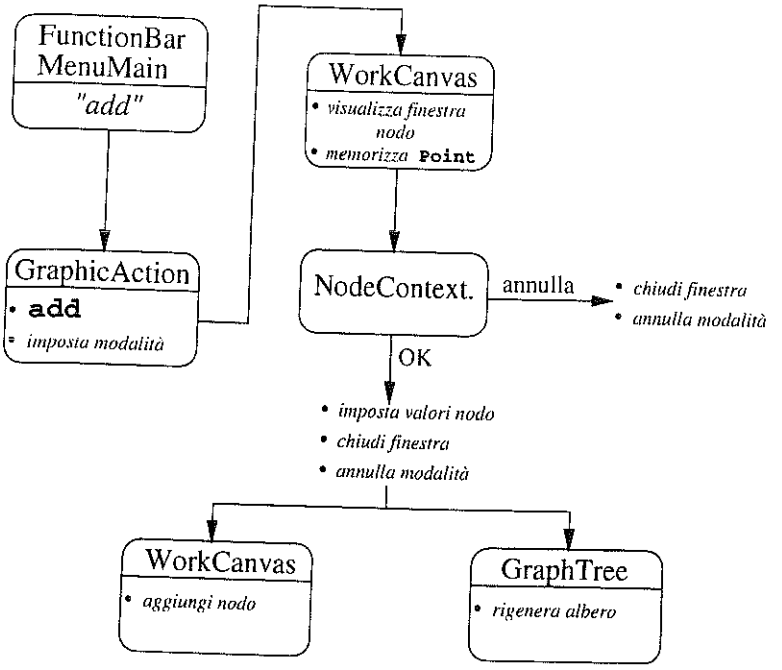

Figure 11: Interazione fra le classi per l'aggiunta di un nodo

5.1.2 RsdEditor.window.gui.FunctionBar

La classe `FunctionBar` rappresenta il toolbar dedicata alle principali funzioni grafiche che possono essere eseguite. Infatti, essa contiene due gruppi di pulsanti che replicano alcune delle funzioni presenti nel menu principale.

Fra tali funzioni troviamo la creazione di un nodo, di un edge fisico e di un edge virtuale, la rimozione di un nodo e di un edge, la creazione di un sottolivello e di un ipernodo.

Nella sezione di codice che effettua la gestione degli eventi, per ogni evento generato da un pulsante viene semplicemente richiamato il relativo metodo della classe `GraphicAction` in cui avviene la vera e propria gestione dell'operazione scelta. Ad esempio, per l'aggiunta di un nodo si utilizza la seguente riga di codice:

```
GraphicAction.add(editor);
```

Il parametro `editor` indica l'istanza della classe principale `RsdEditor` e viene utilizzato dalla classe `FunctionBar` per poter interagire con gli altri componenti dell'interfaccia.

Nella Figura 11 viene mostrata l'interazione fra le classi che contribuiscono alla realizzazione dell'aggiunta di un nodo. L'operazione può essere attivata sia da un pulsante della `FunctionBar` che dalla corrispondente voce

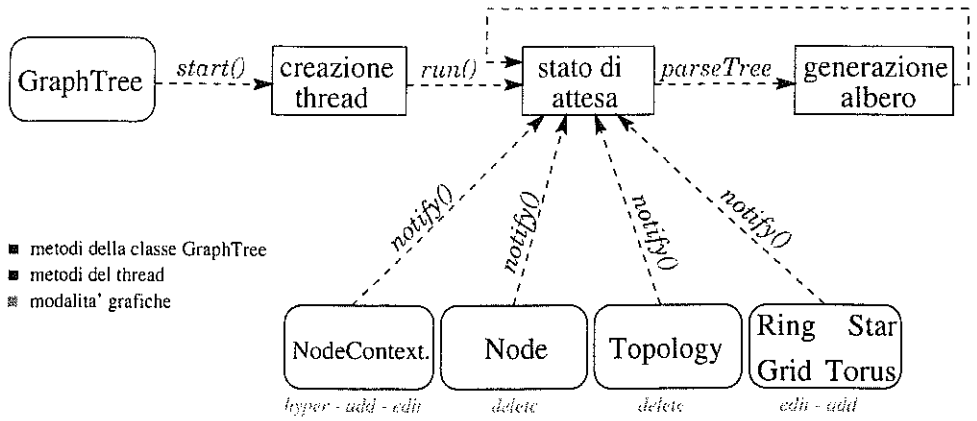

Figure 12: Interazione delle classi con il thread che genera l'albero dei nodi

di menu. Viene richiamato il metodo add della classe `GraphicAction` in cui viene impostata la modalità "add"; la successiva fase viene gestita dalla classe `WorkCanvas` che visualizza la finestra delle proprietà del nodo. A questo punto se l'operazione viene annullata, viene chiusa la finestra e annullata la modalità; se invece viene premuto il tasto di conferma, oltre a memorizzare tutte le informazioni associate all'oggetto `Node` ed aggiungere il nodo al foglio di disegno, viene riattivato il thread creato nella classe `GraphTree` che rigenera l'albero dei nodi, completando così l'operazione.

5.1.3 RsdEditor.window.gui.GraphTree

La classe `GraphTree` permette la creazione e la gestione dell'albero posizionato nella finestra principale dell'interfaccia. Praticamente l'albero rappresenta, in modo più intuitivo, la struttura gerarchica completa dei nodi che compongono il grafo correntemente disegnato.

L'albero viene modificato automaticamente e dinamicamente dopo ogni operazione grafica, per questo è stato introdotto un thread, la cui interazione con le altre classi è mostrata in Figura 12.

Il thread, una volta inizializzato con il metodo `start` ed attivato utilizzando il metodo `run`, viene posto in stato di attesa. In corrispondenza di un'operazione grafica sui nodi ed utilizzando il metodo `notify`, il thread viene riattivato allo scopo di ridisegnare completamente l'albero. La riattivazione avviene quindi nel caso di un'operazione di rimozione o aggiunta di un nodo, nel caso di modifica del nome di un nodo oppure dopo la creazione di un ipernodo. Per la rigenerazione dell'albero viene utilizzato il metodo `parseTree` del quale viene mostrato lo pseudo-codice in Figura 13.

Il metodo `parseTree` è una procedura ricorsiva. La sua prima attivazione

```
public DefaultMutableTreeNode parseTree(Hypernode root)
{
    //aggiungo l'ipernodo alla tabella dei nomi unici
    editor.node.put(nome di root,root);
    DefaultMutableTreeNode father =
          new DefaultMutableTreeNode(nome di root);
    //eseguo la scansione dei figli
    Vector level = elementi contenuti in root;
    for (int to numero_elementi_level)
    {
        if (oggetto(i) e' ipernodo)
        {
            //ho un'ipernodo, creo ed aggiungo il suo sottoalbero
            DefaultMutableTreeNode sub = parseTree(oggetto(i));
            father.add(sub);
            aggiorna albero;
        }
        else if (oggetto(i) e' di tipo Classe)
        {
            DefaultMutableTreeNode node =
                new DefaultMutableTreeNode(nome di oggetto(i));
            father.add(node);
            //aggiungo il nome alla tabella dei nomi unici
            editor.node.put(nome di oggetto(i)),oggetto(i));
            aggiorna albero;
        }
    }
    return father;
}
```

Figure 13: Pseudo-codice del metodo `parseTree` della classe `GraphTree`

riceve come parametro la radice dell'albero mentre nelle successive chiamate riceve un oggetto di tipo `Hypernode`. All'inizio aggiunge il nome dell'ipernodo nella tabella hash che garantisce l'unicità dei nomi dei nodi; inoltre crea un nodo dell'albero (definito come istanza della classe Java `com.sun.java.swing.tree.DefaultMutableTreeNode`) con il nome dell'ipernodo. Quindi controlla il tipo degli elementi che sono contenuti nel vettore dell'ipernodo: se l'elemento corrente è un ipernodo richiama, in modo ricorsivo, il metodo `parseTree` ed aggiunge la struttura dei nodi che verrà generata al nodo creato precedentemente. Se l'elemento è invece di tipo *Classe* (in pratica nel codice Java della classe è presente un controllo per ognuna delle classi `Node`, `Grid`, `Ring`, `Star` e `Torus`) crea un nuovo nodo con il nome dell'oggetto di tipo *Classe*, lo aggiunge al nodo padre, aggiunge il nome alla tabella per i nomi unici e aggiorna l'albero. Alla fine del metodo viene restituito il nodo creato all'inizio del metodo stesso.

Chiaramente quando tutte le chiamate ricorsive hanno concluso la loro sezione di codice è stata generata una completa gerarchia di nodi.

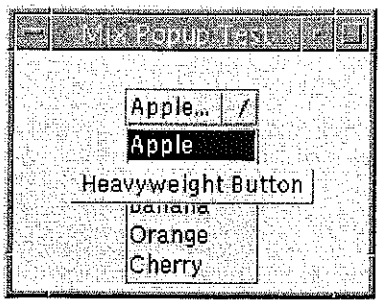

Figure 14: Sovrapposizione sugli elementi swing di tipo "popup".

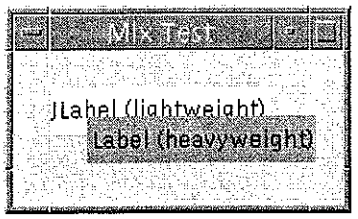

Figure 15: Sovrapposizione di etichette swing ed awt.

Nella classe `GraphTree` sono inclusi anche due metodi che pemettono l'espansione completa dell'albero o la sua chiusura (in questo secondo caso viene visualizzata la sola radice).

Per quanto riguarda la gestione degli eventi l'unico generatore possibile è il tasto sinistro del mouse, mediante il quale si può selezionare un nodo dell'albero. La selezione di un nodo permette la visualizzazione, all'interno del foglio di disegno dell'interfaccia, del livello completo a cui appartiene il nodo. In particolare se viene selezionato un nodo vengono visualizzati anche tutti i suoi fratelli cioè i nodi che sono sullo stesso livello, se, invece, viene selezionato un ipernodo vengono visualizzati tutti i suoi figli cioè i nodi che sono nel livello sottostante.

5.1.4 RsdEditor.window.gui.JCanvas

La classe `JCanvas` rappresenta il foglio di disegno dell'interfaccia che viene incluso in una finestra definita dalla classe `WorkSpace`. Su di esso è possibile introdurre dei componenti grafici complessi definiti dalle classi `Node`, `GraphicEdge`, `GraphicVirtual` e `Topology`; ognuno di essi si basa su primitive grafiche (linee, quadrati, ecc..).

Nella versione 1.1.6 del JDK è già presente una classe, il cui nome completo è `java.awt.Canvas`, simile alla classe `JCanvas` e che permette di introdurre primitive grafiche. Il suo utilizzo insieme ai componenti *Swing* crea però dei problemi di visualizzazione e di incompatibilità fra gli elementi. Ciò è dovuto al fatto che i componenti Swing hanno un "peso" minore rispetto ai componenti del package `java.awt` di Java. In pratica l'utilizzo contemporaneo di elementi Swing ed awt causa la sovrapposizione di tali elementi lasciando in primo piano gli elementi awt. La Figura 14 mostrato l'effetto visivo ottenuto con un elemento swing di tipo "popup" che, nel momento della sua visualizzazione, rimane coperto dal pulsante awt.

Nella Figura 15 viene, invece, mostrato l'effetto che si ottiene posizionando

un'etichetta awt sopra un'etichetta swing.

```
public class JCanvas extends JComponent
{
    // Ipernodo associato al foglio di disegno
    public static Hypernode hyperNode;
    // Costruttore: imposta il layout e la grandezza del foglio di disegno
    public JCanvas()
    {
        setMinimumSize(new Dimension(100,100));
        setMaximumSize(new Dimension(2000,2000));
        setPreferredSize(new Dimension(2000,2000));
        setBackground(Color.white);
        setOpaque(true);
        setLayout(null);
        hyperNode = new Hypernode();
    }
    // Ridisegna tutti gli oggetti associati a quel particolare
    // oggetto grafico.
    public void paintChildren(Graphics g)
    {
        super.paintChildren(g);
    }
    // Restituisce l'oggetto ipernodo associato al foglio di disegno.
    public static Hypernode getHypernode()
    {
        return hyperNode;
    }
    // Consente di associare un ipernodo al foglio di disegno
    // in modo da individuare gli elementi da disegnare.
    public static void setHypernode(Hypernode hyper)
    {
        hyperNode = hyper;
    }
}
```

Figure 16: Codice sorgente della classe JCanvas

In Figura 16 viene mostrato il codice sorgente della classe JCanvas. Essa dichiara un'oggetto Hypernode che realizza la corrispondenza fra il foglio di disegno e gli elementi correntemente disegnati su di esso. Infatti, tali elementi vengono individuati leggendo il contenuto del vettore associato all'ipernodo.

5.1.5 RsdEditor.window.gui.MenuMain

Con la classe MenuMain viene costruito il menu principale dell'interfaccia. Alcune funzionalità associate alle relative voci di menu vengono replicate nella barra in alto e nella barra di destra dell'interfaccia.

Nella classe vengono utilizzate, come strutture dati, tre tabelle hash per individuare direttamente la voce di menu che viene selezionata:

`vocipop`: memorizza le voci dei popup menu visualizzabili attraverso la voce "Predefiniti";

`elementipop`: memorizza le voci degli elementi dei popup menu precedenti;

`items`: memorizza tutte le altre voci di menu selezionabili.

Oltre ai metodi per la costruzione del menu, questa classe include due metodi `add` e `remove` che rispettivamente aggiungono ed eliminano una voce nel menu dei nodi predefiniti. Tali metodi vengono richiamati dalla classe `ChangePredefined` in cui avviene la scelta delle etichette per il menu dei nodi predefiniti.

Nella gestione degli eventi si richiama semplicemente la classe che gestisce l'operazione associata alla voce di menu selezionata.

5.1.6 RsdEditor.window.gui.StateBar

La classe `StateBar` definisce la barra di stato posizionata in basso nell'interfaccia. Essa si compone di due elementi testuali: nel primo vengono visualizzati dei messaggi per indicare all'utente le azioni da svolgere in corrispondenza all'operazione attivata oppure il verificarsi di un'azione scorretta nella visualizzazione dell'albero dei nodi; il secondo invece evidenzia l'operazione grafica corrente dell'interfaccia, detta *modalità*.

In Figura 17 viene mostrato il codice sorgente dell'unico metodo definito in questa classe. Esso può essere richiamato da tutte le altre classi che intendono modificare il testo nella parte di visualizzazione dei messaggi.

```
public void updateCommento(String testo)
{
        commento.setText(testo);
}
```

Figure 17: Codice sorgente del metodo `updateCommento` della classe `StateBar`

5.1.7 RsdEditor.window.gui.TopBar

La classe `TopBar` costruisce la toolbar posizionata sotto il menu. In pratica essa replica, attraverso un insieme di pulsanti, le funzionalità più importanti e di frequente utilizzo del menu (ad esempio l'apertura di un file, l'espansione dell'albero, la visualizzazione del formato RSD del grafo corrente).

Come per il menu, nella parte di gestione degli eventi viene richiamata la classe che gestisce l'operazione scelta. Per individuare l'azione associata a ciascun pulsante viene utilizzata una tabella hash, denominata `commands`, che memorizza la coppia (pulsante,comando).

5.1.8 RsdEditor.window.gui.WorkCanvas

La classe `WorkCanvas` estende la classe `JCanvas` ed è rivolta alla gestione delle operazioni effettuate sul foglio di disegno.

Per diminuire il tempo di esecuzione delle singole operazioni grafiche, in questa classe vengono fatte le istanze delle classi che definiscono le finestre per tutti gli oggetti grafici. Infatti, invece di creare una nuova finestra ogni volta che dobbiamo fare un'operazione grafica, la finestra viene creata una volta per tutte e, al momento dell'operazione essa viene visualizzata e riempita con i valori di default dell'oggetto, oppure, nel caso di una modifica, con i valori precedentemente scelti per tale oggetto.

Le istanze delle classi che definiscono queste finestre vengono fatte al momento dell'attivazione dell'interfaccia utilizzando un apposito thread. Naturalmente il tempo di questa fase aumenta, però, diminuisce notevolmente il tempo richiesto per ogni operazione grafica.

Il metodo più importante della classe è il `drawLevel` la cui esecuzione permette di ridisegnare tutti gli oggetti appartenenti ad un livello del grafo corrente. Esso viene richiamato dopo ogni aggiunta di un oggetto grafico e quando viene selezionato un nodo dell'albero definito dalla classe `GraphTree`. La Figura 18 mostra lo pseudo-codice del metodo `drawLevel`.

```
public void drawLevel()
{
        rimuovi tutti gli elementi dal foglio di disegno;
        ridisegna il foglio;
        Vettore iper = elementi che costituiscono l'ipernodo
                                associato al foglio di disegno;
        for (int i=0 to numero_elementi_iper)
        {
                elemento = iper(i);
                if (elemento è un oggetto di tipo X)
                {
                        aggiungi elemento di tipo X al foglio;
                        ridisegna l'elemento di tipo X;
                }
        }
        ridisegna il foglio;
}
```

Figure 18: Pseudo-codice del metodo drawLevel

Il foglio di disegno è sempre associato ad un oggetto `Hypernode` cioè ad un ipernodo; in questo modo sul foglio di disegno sono visualizzati tutti gli oggetti grafici che costituiscono l'ipernodo ad esso associato. Quindi il metodo `drawLevel` dopo aver eliminato tutti gli oggetti presenti sul foglio di disegno, vi aggiunge tutti gli elementi memorizzati nel vettore `iper`. Tale vettore viene definito nella classe `Hypernode` ed è associato all'ipernodo corrente.

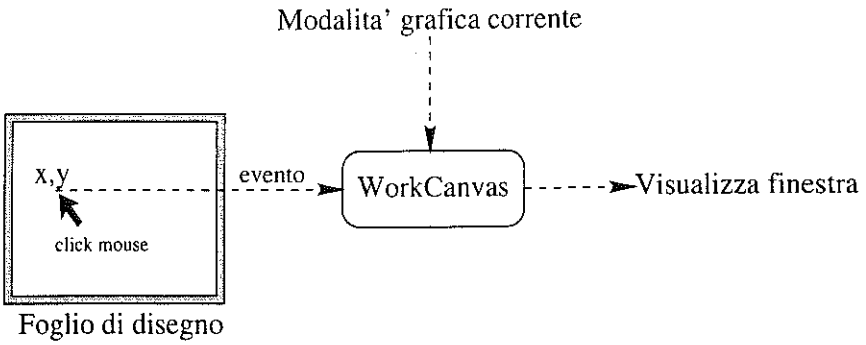

Figure 19: Gestione degli eventi nella classe `WorkCanvas`

La gestione degli eventi della classe `WorkCanvas` è stata schematizzata in Figura 19.

L'evento viene generato dal click del tasto sinistro del mouse: il punto del foglio di disegno in cui avviene il click viene memorizzato in quanto rappresenta il punto in cui verrà aggiunto l'oggetto grafico. Al momento della generazione dell'evento, la classe `WorkCanvas` utilizza la modalità grafica corrente per capire quale finestra deve essere visualizzata.

5.2 Classi che definiscono oggetti

Le classi che vengono adesso analizzate definiscono gli elementi caratteristici del linguaggio RSD. Infatti, ciascuna classe realizza l'oggetto che rappresenta il relativo costrutto sintattico dello RSD. Il nome di ciascuna classe individua l'oggetto che la classe realizza.

5.2.1 RsdEditor.node.Node

La classe `Node` relizza il costrutto sintattico

```
NODE <Node_Identifier> ...
```

del linguaggio RSD.

Questa classe estende la classe `com.sun.java.swing.JComponent`, e ne ridefinisce il metodo `paint()`. Adottando questa soluzione, utilizzando una sola classe si può mantenere sia le informazioni per la generazione del costrutto sintattico precedente che le informazioni ed i metodi per disegnare l'oggetto nodo sul foglio di disegno. Infatti, semplicemente aggiungendo un nodo al foglio di disegno, il metodo `paint()` invocato automaticamente permette di ridisegnare l'oggetto con i parametri impostati.

Per memorizzare tutte le informazioni utili a definire l'oggetto nodo vengono utilizzate alcune strutture dati:

- `nodeGUI`: vettore che memorizza le caratteristiche grafiche del nodo. Questo vettore viene utilizzato dal metodo `paint()` per disegnare il nodo in base alle caratteristiche associate, e le informazioni in esso memorizzate vengono scritte in un file in formato "gui" (vedi paragrafo 3). Tale file verrà utilizzato per rigenerare il grafo precedentemente salvato.

- `nodeRSD`: vettore che memorizza gli attributi RSD specificati per un nodo. Questo vettore sarà utilizzato per generare il file con estensione "rsd" che contiene il formato RSD di tutto il grafo disegnato.

- `port`: vettore che memorizza le porte definite all'interno del nodo;

La classe `Node` include una serie di metodi che accedono a questi tre vettori e permettono sia la memorizzazione che la rilettura delle informazioni. Ad esempio, il metodo `setGUIColor` memorizza il colore scelto per il nodo, mentre il metodo `getRSDAttribute` restituisce il valore di un attributo RSD.

La gestione degli eventi nella classe `Node` è di rilevante importanza essendo questa la classe fondamentale fra quelle che definiscono gli oggetti grafici dell'interfaccia *RsdEditor*. Le operazioni di modifica del nodo sono associate al tasto sinistro del mouse (gestite con il metodo `MousePressed`) che individua l'operazione da fare in base alla modalità corrente. Lo pseudo-codice di questo metodo viene mostrato in Figura 20.

All'inizio del metodo viene memorizzato il punto in cui è avvenuto il click del mouse. Questo punto viene utilizzato dal metodo `MouseDragged` come vedremo successivamente. Viene poi individuata l'istanza della classe `RsdEditor` (`editor`). Quest'ultima è utilizzata per conoscere il valore della variabile globale `mode`, la quale, dichiarata in tale classe, indica la modalità grafica corrente.

A questo punto viene controllato il tasto del mouse che ha effettuato il click: se è il tasto destro viene visualizzato il menu a comparsa associato al nodo, se invece è il tasto sinistro viene gestita l'operazione grafica corrente.

Le operazioni grafiche gestite dalla classe `Node` sono: creazione di un sottolivello, rimozione di un nodo, creazione di un edge fisico, creazione di un edge virtuale, modifica di un nodo, aggiunta di un nodo tra i nodi predefiniti ed esportazione di un nodo. Vediamo lo pseudo-codice usato per implementare ciascuna operazione:

1. **sub**: *creazione di un sottolivello* (vedi Figura 21). Tale operazione trasforma un nodo in ipernodo per poter introdurre elementi di più basso livello logico all'interno del nodo stesso.

```
public void mousePressed(MouseEvent e)
{
    Point lockPoint = punto del foglio di disegno
           su cui e' avvenuto il click del mouse;
    WorkCanvas workCanvas = foglio di disegno;
    RsdEditor editor = workCanvas.editor;
    if (SwingUtilities.isLeftMouseButton(e))
    {
        // controlla l'azione da fare in base al valore
        // della variabile 'MODE'
        if (editor.isMode("sub")) { vedi Figura 21 }
        else if (editor.isMode("remNode")) { vedi Figura 22 }
        else if ((editor.isMode("edge1"))
                || (editor.isMode("edge2"))) { vedi Figura 23 }
        else if (editor.isMode("virt")) { vedi Figura 24 }
        else if (editor.isMode("edit"))
                visualizza finestra associata al nodo;
        else if (editor.isMode("predef"))
                visualizza finestra per l'introduzione
                di nodi predefiniti;
        else if (editor.isMode("export"))
                visualizza file manager per la scelta
                del file in cui esportare il nodo;
    }
    if (SwingUtilities.isRightMouseButton(e))
    {
        visualizza menu a comparsa associato al nodo;
    }
}
```

Figure 20: Pseudo-codice del metodo `MousePressed` della classe `Node`

Però, non è sufficiente assegnare al nuovo ipernodo tutte le informazioni che erano associate al nodo. Infatti, è necessario individuare tutti gli edge (sia fisici che virtuali) che hanno il nodo come estremo e rimpiazzare questo valore con il nuovo ipernodo.

Infine, prima di ridisegnare completamente il foglio di disegno, viene tolto il nodo dall'ipernodo a cui apparteneva e vi viene aggiunto il nuovo ipernodo.

2. **remNode**: *rimozione di un nodo* (vedi Figura 22). In seguito a quest'operazione, oltre alla cancellazione del nodo e di tutta la struttura ad esso associata, vanno cancellati anche tutti gli edge che hanno il nodo come estremo. Per far questo viene considerato il vettore che memorizza le porte definite nel nodo e, per ogni porta, si controlla quali edge la utilizzano.

   Dopo aver tolto il nodo dal suo ipernodo bisogna controllare se quest'ultimo non contiene altri elementi e se non rappresenta la radice dell'albero. Se sono verificate queste condizioni bisogna trasformare l'ipernodo in nodo e

```
// devo trasformare questo nodo in ipernodo
Hypernode newHyper = new Hypernode(); // nuovo ipernodo
assegna i vettori di questo nodo a newHyper;
Hypernode father = ipernodo che contiene il nodo;
Vettore iper = elementi contenuti in father;
for i = 0 to numero_elementi_iper)
    if ((iper(i) e' "GraphicEdge")
                && (questo nodo e' suo estremo))
    {
        sostituisci estremo 'nodo' con estremo 'ipernodo';
    }
    else if ((iper(i) e' "GraphicVirtual")
                && (questo nodo e' suo sorgente))
    {
        sostituisci sorgente 'nodo' con sorgente 'ipernodo';
    }
togli questo nodo da father;
aggiungi newHyper a father;
ridisegna completamente questo livello;
```

Figure 21: Pseudo-codice della sezione "sub" del metodo `MousePressed` della classe `Node`

riportare tali modifiche anche all'interno degli edge che hanno l'ipernodo come estremo.

La rimozione del nodo deve attivare il thread che rigenera l'albero dei nodi visualizzato nella parte sinistra della finestra principale, in quanto è stata modificata la struttura dei nodi.

3. **edge1/edge2**: *creazione di un edge fisico* (vedi Figura 23).

   La prima cosa da verificare per l'introduzione di un edge fisico è l'esistenza di porte utilizzabili per creare il corrispondente arco (vedi pseudo-codice mostrato in Figura 23). Le porte utilizzabili sono sia le porte definite all'interno dei nodi estremi allo edge, che sono memorizzate nel vettore associato alle porte del nodo, sia le porte "esportate". Quest'ultime esistono solamente se il nodo corrente è un ipernodo, infatti, in questo caso è possibile che il nodo contenga edge virtuali che hanno l'attributo RSD "binding" del tipo:

   <*porta*,**null**>

   La componente **null** indica che la porta esportata non viene rinominata.

   Se il numero delle porte utilizzabili è diverso da zero si controlla se il nodo corrente è stato selezionato come primo estremo dell'edge (nodo

```
// sto cancellando un nodo e quindi tutte le sue porte
Hypernode father = ipernodo che contiene il nodo;
Vector porte = porte definite nel nodo;
cancella tutti gli elementi del vettore porte e per ogni
       elemento cancella tutti gli edge che lo utilizzano;
togli questo nodo da father;
if ((father non ha elementi)
           && (father non e' la radice dell'albero))
{
    Node node = new Node(); // nuovo nodo
    assegna i vettori di father a node;
    per tutti gli edge che hanno father come estremo
        sostituisci l'estremo 'ipernodo' con l'estremo 'nodo';
    elimina father dal suo ipernodo;
    aggiungi node all'ipernodo di father;
}
ridisegna completamente questo livello;
attiva il thread che genera l'albero dei nodi;
```

Figure 22: Pseudo-codice della sezione "remNode" del metodo `MousePressed` della classe `Node`

sorgente). Se così non è, significa che è il nodo destinazione dell'arco e quindi si può passare alla creazione vera e propria dello edge.

L'ultima cosa da controllare è l'esistenza di un oggetto `GraphicEdge` tra i due nodi estremi dello edge; se tale oggetto non esiste andrà creato ed aggiunto al foglio di disegno.

Al completamento di queste operazioni viene visualizzata la finestra degli attributi RSD associati allo edge fisico.

4. **virt**: *creazione di un edge virtuale* (vedi Figura 24).

   Dallo pseudo-codice di Figura 24 si può vedere che la prima sezione è simile a quella dello pseudo-codice di Figura 23.

   La particolarità della creazione di un edge virtuale, nel caso che esista già un `GraphicVirtual` sul nodo corrente, è subordinata alla possibilità di utilizzare almeno una porta fra quelle disponibili. Infatti, se gli oggetti `Virtual` (edge virtuali) creati precedentemente su questo nodo utilizzano tutte le porte disponibili non è possibile creare un ulteriore edge virtuale.

   Se, invece, tutti i controlli vanno a buon fine viene visualizzata la finestra degli attributi RSD associati al nuovo edge virtuale.

5. **edit**: *modifica di un nodo* (vedi Figura 20).

6. **predef**: *aggiunta di un nodo tra i nodi predefiniti* (vedi Figura 20).

```
// devo controllare se il nodo eredita delle porte
// dal livello sottostante
int expoPort = 0; // numero porte esportate
if (questo nodo e' ipernodo)
{
    Vettore iper = elementi contenuti in questo nodo;
    for (int i = 0 to numero_elementi_iper)
        if (enum(i) e' un "GraphicVirtual")
            //controllo ogni edge virtuale incluso nell'arco
            for (ogni Virtual incluso nel "GraphicVirtual)
                if (Virtual esporta una porta)
                    incrementa di uno il valore di expoPort;
}
Vettore p = porte definite in questo nodo;
if ((numero_elementi_p) && (expoPort = 0))
    ERRORE: non esistono porte su questo nodo,
            non e' possibile creare l'arco;
else if (questo nodo e' il primo scelto)
        // questo nodo e' la sorgente
        sorgente_edge = questo nodo;
    else if (sorgente_edge e' diverso da questo nodo)
        // questo nodo e' la destinazione e quindi genero l'arco
        {
            Edge edge = new Edge(); // creo edge
            imposta valori di default per edge;
            if (non esiste "GraphicEdge" tra i due nodi scelti)
            {
                GraphicEdge graphicEdge = new GraphicEdge();
                imposta valori di default per graphicEdge;
                aggiungi graphicEdge all'ipernodo corrente;
                aggiungi graphicEdge al foglio di disegno;
            }
            imposta graphicEdge per edge;
            visualizza finestra associata al nuovo edge;
        }
}
```

Figure 23: Pseudo-codice della sezione "edge" del metodo `MousePressed` della classe `Node`

7. **export**: *esportazione di un nodo* (vedi Figura 20).

Il metodo `MouseEntered` invece costruisce il tooltip del nodo e lo visualizza nel momento del passaggio del puntatore del mouse sul nodo. Il tooltip visualizza tutti gli attributi RSD del nodo e per questo la sua generazione usa i dati memorizzati nel vettore `nodeRSD`.

Con il metodo `MouseDragged` viene invece gestito lo spostamento del nodo in una nuova posizione sul foglio di disegno. In Figura 25 si può vedere come, con poche righe di codice, viene gestito questo trascinamento del nodo.

In pratica, nel metodo `MousePressed` viene fissato il punto (`lockPoint`) in cui avviene il click del mouse che, naturalmente, è all'interno dell'oggetto grafico che rappresenta il nodo. In fase di trascinamento viene calcolato dinamicamente il discostamento relativo da `lockPoint` e, in base alla posizione

```
// controllo se il nodo eredita delle porte
// dal livello sottostante
int expoPort = 0; // numero porte esportate
if (questo nodo e' ipernodo)
{
    Vettore iper = elementi contenuti in questo nodo;
    for (int i = 0 to numero_elementi_iper)
        if (enum(i) e' un "GraphicVirtual")
            //controllo ogni edge virtuale incluso nell'arco
            for (Virtual incluso nel "GraphicVirtual)
                if (Virtual esporta una porta)
                    incrementa di uno il valore di expoPort;
}
Vettore p = porte definite in questo nodo;
if ((numero_elementi_p) && (expoPort = 0))
    ERRORE: non esistono porte su questo nodo,
            non e' possibile creare l'arco;
else
{
    Virtual virtual = new Virtual(); // creo virtual
    imposta valori di default per virtual;
    if (esiste gia' un "GraphicVirtual" su questo nodo)
        if ((numero_elementi_p + expoPort) <=
                numero_virtual_definiti_in_GraphicVirtual)
            ERRORE: non ci sono porte utilizzabili,
                    non e' possibile creare l'arco;
    else
    {
        GraphicVirtual graphicVirtual = new GraphicVirtual();
        imposta valori di default per graphicVirtual;
        aggiungi graphicVirtual all'ipernodo corrente;
        aggiungi graphicVirtual al foglio di disegno;
    }
    imposta graphicVirtual per virtual;
    visualizza finestra associata al nuovo virtual;
}
```

Figure 24: Pseudo-codice della sezione "virtual" del metodo `MousePressed` della classe `Node`

iniziale del componente nodo (`location`), viene fissata la nuova posizione del nodo stesso. Al momento del rilascio del mouse viene memorizzato, nel vettore `nodeGUI` la nuova posizione del nodo.

Un ultimo metodo gestisce il menu a comparsa associato al nodo e visualizzabile attraverso il click del tasto destro del mouse sul nodo. In tale menu vengono replicate tre funzionalità del menu *Nodo*: modifica, rimozione ed esportazione di un nodo.

5.2.2 RsdEditor.node.Hypernode

La classe `Hypernode` realizza l'oggetto ipernodo definito dal costrutto sintattico

```
<Node_Definition> ::= NODE <Node_Ident> {
```

```
public void mouseDragged(MouseEvent e)
{
        Point point = e.getPoint();
        if (SwingUtilities.isLeftMouseButton(e))
        {
                int dx = point.x - lockPoint.x;
                int dy = point.y - lockPoint.y;
                Point location = getLocation();
                setLocation(location.x + dx, location.y + dy);
                setGUIPoint(new Point(location.x + dx,                  location.y + dy));
        }
}
```

Figure 25: Codice sorgente del metodo MouseDragged della classe Node

```
<Node_Decl_Command>+};

<Node_Decl_Command> ::= <Node_Definition>
```

del linguaggio RSD.

Essa viene definita come estensione della classe Node e, per questo, gran parte dei metodi definiti per l'oggetto nodo vengono ereditati direttamente dall'oggetto ipernodo.

Come struttura dati principale, la classe Hypernode utilizza un vettore (hyperNode) per tenere traccia di tutti gli elementi che costituiscono l'ipernodo da essa dichiarato. Definisce poi un contatore (son) per avere a disposizione il numero di questi elementi.

Il metodo che permette di memorizzare gli oggetti nel vettore è il metodo addObject, esso aggiunge un'oggetto all'ipernodo discriminando sulla classe di cui l'oggetto è istanza. Viene fatto un controllo in base al nome dell'oggetto per verificare che non sia già presente nel vettore.

L'oggetto ipernodo viene anche utilizzato per effettuare l'associazione del foglio di disegno con il livello gerarchico del grafo da visualizzare. Infatti, attraverso il metodo setWorkCanvas viene memorizzato l'oggetto istanza della classe WorkCanvas nel vettore associato all'ipernodo.

L'ipernodo viene trattato graficamente come un nodo in quanto l'oggetto Hypernode non rappresenta di per se un oggetto grafico. Per disegnare l'ipernodo si utilizza il metodo paint della classe Node con la seguente istruzione Java:

```
super.paint(g);
```

la parola super permette infatti di richiamare un metodo della *superclasse* che è la classe originaria su cui è basata l'estensione.

Per quanto riguarda la gestione degli eventi, l'unica differenza rispetto alla classe Node, riguarda la creazione di un sottolivello. Se si effettua questa

operazione su un ipernodo viene semplicemente ridisegnato il sottolivello, cioè vengono visualizzati tutti i componenti dell'ipernodo stesso. Per gli altri eventi vengono utilizzati i metodi definiti nella superclasse cioè i metodi ereditati.

Prima di passare alla descrizione delle classi che definiscono i collegamenti tra i nodi dobbiamo parlare della classe che permette di realizzare questi collegamenti.

5.2.3 RsdEditor.common.Port

La classe `Port` definisce l'oggetto porta, cioè realizza il seguente costrutto sintattico del linguaggio RSD:

```
PORT <Port_Identifier> ...
```

Una porta può essere definita solo all'interno dell'oggetto nodo e rappresenta un'interfaccia di comunicazione di tale nodo verso l'esterno. Le varie porte definite per un nodo vengono memorizzate in un vettore associato al nodo. Ciascun oggetto porta utilizza poi un proprio vettore, denominato `portRSD`, per memorizzare gli attributi RSD che vengono associati alla porta. I metodi definiti in questa classe riguardano esclusivamente la gestione di questi attributi ed in particolare sono presenti dei metodi per l'aggiunta, la rimozione e la lettura di queste informazioni dal vettore `portRSD`.

Va evidenziato che l'oggetto porta non ha un elemento grafico associato che la rappresenta ma bensì può essere aggiunta utilizzando la finestra dell'oggetto grafico nodo. Le porte hanno comunque un notevole importanza perché impongono un vincolo di creazione degli oggetti: *un edge che colleghi due nodi può essere aggiunto solo se entrambi i nodi contengono una porta utilizzabile (definita od importata dal livello sottostante).*

5.2.4 RsdEditor.edge.Edge

La classe `Edge` realizza l'oggetto "edge fisico" (arco del grafo) che consente il collegamento fra due nodi del metacomputer. Questo elemento realizza il seguente costrutto sintattico RSD :

```
EDGE <Edge_Identifier> {<Edge_Specification>}
<Edge_Specification> ::= NODE <Node_Ident_1> PORT <Port_Ident_1>
                                    ( '=>' | '<=>' | '<=' )
                         NODE <Node_Ident_2> PORT <Port_Ident_2>;
```

La differenza rispetto alla struttura della classe `Node` è che questa classe gestisce

solo la parte della sintassi RSD. L'elemento grafico che rappresenta gli edge fisici viene definito nella classe `GraphicEdge`; questa scelta di progetto permette di definire più di un edge fra due nodi che per semplicità abbiamo deciso di rappresentare con un unico oggetto grafico.

Alla luce di quanto detto la classe `Edge` necessita di un'unico vettore, lo `edgeRSD`, per la memorizzazione degli attributi RSD associati all'oggetto edge. Oltre ai normali attributi RSD che possono essere introdotti per specificare determinate caratteristiche di un edge fisico (ad esempio l'ampiezza di banda), è presente un insieme fissato di attributi, che ritroviamo nella definizione di ogni edge. Quest'ultimi attributi, che permettono di definire completamente il costrutto RSD precedente, sono:

`GraphicEdge`: oggetto grafico che rappresenta lo edge,

`Name`: nome dello edge,

`Source`: nome del nodo sorgente dello edge,

`Destination`: nome del nodo destinazione dello edge,

`Direction`: direzionalità dello edge (unidirezionele o bidirezionale),

`Binding`: porte definite nei due nodi ed utilizzate dallo edge.

Per ciascuno di questi attributi viene definito un metodo sia per la loro memorizzazione nel vettore `edgeRSD` sia per loro lettura da tale vettore. Ciascun metodo è implementato analogamente al metodo `setGraphicEdge` il cui codice sorgente è mostrato in Figura 26. Quest'ultimo costituisce il metodo con cui viene memorizzato l'oggetto grafico che rappresenta lo edge. Questo metodo riceve in input l'oggetto g ed effettua un ciclo che controlla se l'attributo `"GraphicEdge"` è già presente nel vettore `edgeRSD`. Se l'attributo esiste viene aggiornato con il nuovo valore g, se invece non esiste, esso viene creato ed un nuovo elemento definito dalla coppia `<"GraphicEdge",g>` è memorizzato nel vettore `edgeRSD`.

Naturalmente, questa classe non deve gestire nessun evento in quanto, come detto precedentemente, essa non definisce nessun elemento grafico.

5.2.5 RsdEditor.edge.GraphicEdge

La classe `GraphicEdge` realizza l'oggetto che rappresenta graficamente un edge fisico, definito dalla classe `Edge`. Tale oggetto avrà associato un numero che indica il numero di collegamenti presenti fra due nodi del metacomputer.

Nella classe `GraphicEdge` vengono utilizzate varie strutture dati :

```
public void setGraphicEdge(GraphicEdge g)
{
        int i = 0;
        while ((i < edgeRSD.size()) &&
                !(((Item)edgeRSD.elementAt(i))
                .getAttribute().equals("GraphicEdge")))
        {
                i++;
        }
        if (i < edgeRSD.size())
        {
                // l'attributo GraphicEdge c'e', modifico quello esistente
                ((Item)edgeRSD.elementAt(i)).setValue(g);
        }
        else
        {
                // l'attributo GraphicEdge non c'e', lo aggiungo
                item = new Item("GraphicEdge", g);
                edgeRSD.addElement(item);
        }
}
```

Figure 26: Codice sorgente del metodo `setGraphicEdge` della classe `Edge`

`edgeList`: tabella hash che memorizza, mediante la coppia <`nome`,`Edge`>, gli edge rappresentati da questo oggetto grafico;

`edgeGUI`: vettore che memorizza le caratteristiche grafiche specificate per l'oggetto grafico;

`vociPopup`: tabella hash che memorizza gli elementi del menu a comparsa associato all'oggetto grafico (per ogni edge viene inserita una voce nel menu).

Tale classe implementa un insieme di metodi per la gestione di queste strutture dati, come ad esempio, il metodo `addEdgeList` che, in corrispondenza all'aggiunta di un edge fisico, aggiunge un elemento sia in `edgeList` che in `vociPopup`. Un altro esempio è costituito dal metodo `setSource` che memorizza in `edgeGUI` il nodo sorgente dell'oggetto grafico, che è il corrispondente nodo sorgente degli edge fisici rappresentati da tale oggetto.

Un rilievo particolare va dato al metodo `paint` che, oltre a disegnare un arco tra due nodi, gestisce l'eventuale spostamento di uno dei due nodi. La Figura 27 mostra lo pseudo-codice di questo metodo. Nella prima parte del metodo viene individuato quale nodo estremo di un arco si sta muovendo.

Da considerare attentamente sono i casi in cui un arco assume un allineamento verticale od orizzontale rispetto al foglio di disegno. Infatti, nel punto di mezzo dell'arco viene disegnato anche un piccolo quadrato, denominato `control`, con il numero degli edge fisici rappresentati dall'arco. Come mostrato

in Figura 28 bisogna evitare di nascondere il quadrato quando l'arco si avvicina agli assi cartesiani "virtuali" , che hanno come origine il centro del nodo `fixed`. L'intervallo critico è di 22 punti sia intorno all'asse orizzontale che all'asse verticale, questo perché `control` ha i lati di 11 punti.

Nello pseudo-codice mostrato in Figura 27 (*(*)*) possiamo vedere che, prima di gestire il posizionamento del nodo `moving` in uno dei quattro quadranti del piano cartesiano, vengono gestiti i casi in cui il nodo sia posizionato in prossimità degli assi (parte tratteggiata in Figura 28). In ciascun caso deve essere fissata la posizione dell'arco, con l'elemento `control` posizionato a metà di esso e, alla fine del metodo, viene ridisegnato l'arco.

Analizziamo adesso la gestione degli eventi della classe `GraphicEdge`. In questa classe viene definito un menu a comparsa visualizzabile con il click del tasto sinistro del mouse sull'oggetto grafico che rappresenta gli edge fisici. Attraverso questo menu è possibile eliminare gli edge rappresentati da tale oggetto grafico oppure visualizzare la finestra per modificare i valori impostati per ciascun edge. Invece, con il click del tasto destro del mouse viene visualizzata la finestra che contiene le informazioni grafiche dell'oggetto `GraphicEdge`. Il click del mouse viene gestito dal metodo `mousePressed`.

Il metodo `mouseEntered`, invece, genera il tooltip che elenca, per ogni edge fisico, gli attributi RSD e viene visualizzato con il passaggio del puntatore del mouse sull'oggetto grafico.

5.2.6 RsdEditor.virtual.Virtual

La classe `Virtual` realizza l'oggetto "edge virtuale" (verticale). Lo edge virtuale rappresenta uno degli elementi principali del linguaggio RSD ed è rappresentato con il costrutto sintattico :

```
<Assign_Definition> ::=
  ASSIGN NODE <Node_Ident> PORT <Port_Ident>;
| ASSIGN NODE <Node_Ident> PORT <Port_Ident> '<=>' <Port_Ident>;
```

la sua funzione è quella di "esportare" le porte definite nel nodo sorgente dell'arco considerato al livello superiore nel grafo dei nodi. Così facendo le porte definite in un livello possono essere visibili e quindi utilizzabili anche in altri livelli, ciò permette di collegare due nodi che sono su due livelli diversi.

Per un edge virtuale vale quanto detto per un edge fisico, cioè la classe `Virtual` gestisce la sola parte degli attributi RSD associati ad un edge viruale. L'oggetto grafico che rappresenta un edge virtuale è definito, invece, dalla successiva classe `GraphicVirtual`.

La classe utilizza il vettore `virtualRSD` per memorizzare tutti gli attributi RSD necessari alla definizione di un edge e perciò i metodi contenuti in questa

classe riguardano esclusivamente la gestione di queste informazioni. A differenza di un edge fisico, un edge virtuale non prevede gli attributi `Destination` e `Direction`. Il primo, perché il nodo destinazione dell'arco è l'ipernodo del nodo sorgente e quindi, tale informazione può essere recuperata direttamente dall'attributo `Hypernode` del nodo sorgente. Il secondo, invece, non è presente perché un edge verticale è in ogni caso di tipo bidirezionale. Chiaramente un edge verticale ha l'attributo `GraphicVirtual` e non l'attributo `GraphicEdge`.

5.2.7 RsdEditor.virtual.GraphicVirtual

La classe `GraphicVirtual` gestisce l'oggetto grafico che rappresenta un edge verticale.

Come per l'oggetto grafico che definisce gli edge fisici, l'utilizzo di questa classe permette l'introduzione di vari edge virtuali per uno stesso nodo sorgente. La creazione di un oggetto grafico che rappresenta edge virtuali avviene al momento della creazione del primo edge virtuale e la sua rimozione avviene contemporaneamente alla rimozione dell'ultimo edge virtuale definito su quel nodo.

Le strutture dati definite in questa classe sono un vettore (`virtualGUI`) e due tabelle hash (`virtualList` e `vociPopup`) che vengono utilizzate in maniera analoga alle corrispondenti strutture dati utilizzate nella classe `GraphicEdge`.

Il metodo `paint` permette di disegnare l'oggetto grafico `GraphicVirtual`; in questo caso deve essere gestito il solo trascinamento del nodo sorgente nell'arco.

Gli eventi che possono verificarsi su questo oggetto grafico sono, come nel caso del `GraphicEdge`, la visualizzazione del tooltip, della finestra che visualizza le proprietà grafiche dell'arco e del menu a comparsa.

In Figura 29 viene mostrato il codice sorgente per la generazione del tooltip associato ad un arco.

Il tooltip viene generato utilizzando un buffer di memoria centrale (`tooltip`). Viene scandita la tabella hash `virtualList` che contiene l'elenco degli edge virtuali rappresentati da questo oggetto grafico e, per ogni edge, vengono aggiunti al buffer gli attributi RSD: nome edge, nome nodo sorgente e binding.

Alla fine della generazione il buffer viene trasformato in stringa e viene associato all'oggetto grafico (`setToolTipText(tooltip.toString())`), quindi viene ridisegnato il tooltip eseguendo il metodo `repaint`.

5.2.8 RsdEditor.topology.Topology

La classe `Topology` permette la definizione delle topologie: `Grid`, `Ring`, `Star` e `Torus` supportate dall'interfaccia. Questa classe definisce gli elementi comuni

a tutte le topologie mentre esiste un'altra classe per ogni tipo di topologia che definisce e gestisce solo gli elementi caratteristici della topologia stessa.

La finestra associata a ciascuna topologia è caratterizzata da quattro sezioni ognuna delle quali permette di specificare le caratteristiche dei vari elementi che compongono la topologia. Le quattro sezioni sono:

1. *attributi GUI della topologia*: è la sezione che contraddistingue una topologia dall'altra. Quindi, questa parte è differente per ciascuna topologia;

2. *attributi RSD dei nodi interni*: permette di modificare le caratteristiche dei nodi che compongono la topologia mediante l'aggiunta di nuovi attributi;

3. *attributi RSD degli edge interni*: permette di modificare le caratteristiche degli edge che interconnettono i nodi nella topologia;

4. *attributi RSD delle porte interne e della porta esterna*: è dedicata alla specifica degli attributi RSD associati alle porte utilizzate per la connessione di due nodi. Inoltre, è possibile associare attributi RSD alla porta utilizzata per collegare la topologia con altri oggetti grafici.

Le ultime tre sezioni della finestra vengono realizzate con metodi definiti nella classe `Topology`: `nodeWindow`, `edgeWindow` e `portWindow`. La creazione vera e propria della finestra avviene nella classe che definisce una specifica topologia eseguendo sia il metodo `makeWindow` che i tre metodi precedenti.

Per disegnare l'oggetto grafico che rappresenta una topologia, viene richiamato il metodo `paint` che introduce l'immagine del tipo di topologia scelto sul foglio di disegno.

Per la memorizzazzione di tutte le informazioni che descrivono una topologia, la classe `Topology` utilizza i vettori: `topoGUI`, `nodeRSD`, `edgeRSD`, `internRSD` e `externRSD`. Questi vengono creati all'interno della classe che definisce una specifica topologia ed i metodi per accedere a questi vettori sono simili a quelli usati nelle classi che definiscono gli altri oggetti grafici.

La gestione degli eventi è gestita, in gran parte, nella classe `Topology` lasciando alla classe che definisce ogni specifica topologia la gestione dei pulsanti inseriti nella sezione "*attributi GUI della topologia*" della finestra.

La topologia viene trattata come l'oggetto `Node` e quindi le operazioni che possono essere fatte sulla topologia sono le stesse permesse per i nodi. Unica eccezione è che sulla topologia non può essere fatta l'operazione di creazione di un sottolivello e quindi essa non può diventare ipernodo.

Naturalmente devono essere gestiti tutti gli eventi che possono essere generati dagli elementi presenti nelle ultime tre sezioni della finestra (pulsanti, combobox, ecc.).

Come detto precedentemente la classe `RsdWriter` non effettua la registrazione degli oggetti topologie all'interno dei file `*.gui` e `*.rsd`, bensì richiede l'esecuzione dei metodi di scrittura implementati nella classe che definisce ciascuna specifica topologia. Infatti, le topologie si diversificano per le loro caratteristiche grafiche e per il formato RSD che descrive ciascuna di esse.

5.3 Classi di utilità

5.3.1 RsdEditor.common.Item

La classe `Item` realizza il formato dell'elemento che viene memorizzato sia nei vettori in cui sono contenuti gli attributi RSD sia in quelli che contengono le caratteristiche grafiche degli oggetti; tale formato è utilizzato dalla maggior parte delle classi che compongono l'applicazione.

Il formato è composto dalla coppia `<nome,valore>` dove `nome` è una stringa di caratteri mentre `valore` è un oggetto. In Figura 30 viene mostrato il codice sorgente della classe `Item` dove le due variabili `attribute` e `value` rappresentano, rispettivamente, gli elementi della coppia precedente.

I metodi usati in questa classe permettono di memorizzare nuovi elementi nelle strutture di tipo vettore e di leggere tali elementi.

5.3.2 RsdEditor.common.GuiParser

La classe `GuiParser` realizza la scansione dei file `*.gui` e `*.rsd`, che vengono utilizzati per la memorizzazione di tutte le informazioni associate al grafo costruito utilizzando l'interfaccia *RsdEditor* .

Tale classe è realizzata come estensione della classe `Thread`, questo significa che essa costituisce una sequenza di esecuzione distinta, in modo che le altre operazioni non debbano attendere il completamento della sua esecuzione.

La ricostruzione di un grafo precedentemente salvato si basa sulla struttura del file in formato GUI. Per ogni oggetto grafico che viene ricostruito leggendo le sue caratteristiche da questo file, viene effettuata la relativa lettura, dal file con estensione `rsd`, degli attributi RSD associati ai suoi componenti.

La classe `GuiParser` implementa un analizzatore a discesa ricorsiva, in quanto adatto per analizzare il linguaggio GUI che viene generato da una grammatica di tipo **LL**. Questa classe definisce un metodo per ogni tipo di produzione della grammatica descritta in paragrafo 3.2.

Come possiamo vedere dallo pseudo-codice di Figura 31, il metodo `run` effettua l'apertura di entrambi i file, copia il contenuto del file `*.gui` in un buffer della memoria centrale e chiude il file.

Il contenuto del buffer viene suddiviso in stringhe per poter individuare le parole chiave che contraddistinguono i vari oggetti nel formato GUI (`NODE`,

EDGE ed ASSIGN).

La classe GuiParser viene utilizzata per l'importazione e l'inserimento nel grafo di, rispettivamente, un nodo precedentemente salvato o predefinito. Generalmente queste operazioni vengono effettuate quando la costruzione di un grafo è già iniziata, quindi, viene passato, come parametro, al costruttore della classe, l'ipernodo (padre) in cui deve essere aggiunto l'elemento. Se, invece, si sta effettuando l'apertura di un file *.gui, precedentemente salvato, e quindi si sta creando un grafo completo, il precedente parametro non è significativo perché, in tal caso, viene creato un nuovo ipernodo che rappresenta la radice del grafo.

A questo punto, richiamando il metodo S, inizia l'esecuzione ricorsiva dei vari metodi della classe. Una volta terminata la ricorsione viene chiuso il file *.rsd e viene riattivato il thread che ricostruisce l'albero dei nodi.

Analizziamo adesso i metodi che realizzano tutte le produzioni della grammatica. In Figura 32 viene mostrato lo pseudo-codice del metodo S che realizza la produzione "*S -> { N }*"; questa produzione rappresenta il passo iniziale dell'analisi sintattica del contenuto del file *.gui.

La prima stringa in lettura, dopo la parte di commento del file *.gui, deve contenere il solo carattere {; se ciò non si verifica viene generato un errore di sintassi. Successivamente viene eseguito il metodo N che ricostruisce i nodi del grafo; infine, viene controllato se la stringa successiva contiene il solo carattere } di fine sequenza.

Lo pseudo-codice di Figura 33 è relativo al metodo N che realizza due produzioni della grammatica: "*N -> NODE[attrn N]N*" e "*N -> E*".

La seconda produzione viene realizzata se la stringa letta non è uguale a NODE ed equivale ad eseguire il metodo E. Se, invece, è letta la stringa NODE, significa che un oggetto ti tipo nodo deve essere ricostruito e quindi, la stringa attesa successivamente è costituita dal carattere [.

La parte *attrn* della prima produzione comporta l'individuazione di tutte le caratteristiche grafiche dell'oggetto nodo e di tutti gli attributi RSD ad esso associati. Dopo che l'oggetto nodo è stato ricostruito ed aggiunto all'ipernodo corrente, è necessario verificare se questo nodo è un ipernodo. Ciò è fatto dichiarando un nuovo ipernodo (father) ed eseguendo il metodo N (vedi *(1)* in Figura 33) per la costruzione di eventuali figli di questo ipernodo, cioè nodi che sono nel livello sottostante a quello del nodo corrente. Se all'uscita di questo metodo, l'oggetto costruito (father) non contiene, al suo interno, altri elementi significa che esso non è ipernodo.

Il passo successivo consiste nel controllare se l'ultima stringa letta è costituita dal carattere]. Se ciò è verificato viene eseguito il metodo N (vedi *(2)* in Figura 33) per la costruzione di eventuali nodi fratelli, cioè nodi che sono sullo stesso livello del nodo trattato.

Il metodo `E` realizza le produzioni "*E -> EDGE[attre]E* " e "*E -> A*" ed il suo pseudo-codice è mostrato in Figura 34.

Il metodo `E` inizialmente controlla se la stringa in lettura è uguale ad `EDGE` e se la successiva è costituita dal carattere `[`. In tal caso l'oggetto grafico `GraphicEdge` deve essere ricostruito usando le informazioni memorizzate nel file `*.gui` e gli attributi RSD memorizzati nel file `*.rsd`.

Al completamento di queste operazioni l'oggetto grafico viene aggiunto all'ipernodo corrente e, se la successiva stringa è costituita dal carattere di fine sequenza `]`, viene eseguito ricorsivamente il metodo `E` per la ricostruzione di eventuali `GraphicEdge` posti sullo stesso livello dell'oggetto trattato.

Se la stringa in lettura non risulta uguale ad `EDGE`, viene eseguito il metodo `A` per controllare se l'ipernodo corrente contiene anche oggetti `GraphicVirtual` che rappresentano gli edge virtuali.

Lo pseudo-codice del metodo `A` è mostrato in Figura 35. Questo realizza le ultime due produzioni: "*A -> assign A*" e "*A -> ε*".

La sequenza corretta che ci si aspetta di trovare è data dalla stringa `ASSIGN` seguita dal carattere `[`. In questo caso deve essere ricostruito sia l'oggetto `GraphicVirtual` che gli edge virtuali in esso presenti.

L'oggetto `GraphicVirtual` viene aggiunto all'ipernodo corrente e, se la stringa successiva contiene il solo carattere di terminazione `]`, viene eseguito il metodo `A` per individuare altri eventuali oggetti `GraphicVirtual`.

Gli altri metodi che sono definiti nella classe `GuiParser` vengono utilizzati per leggere dal file `*.rsd` e dal file `*.gui`, rispettivamente , gli attributi RSD e le informazioni grafiche registrate per ciascun oggetto.

5.3.3 RsdEditor.common.RsdWriter

La classe `RsdWriter` è statica[6] e permette la registrazione dei due file:

1. *nome*`.gui` contiene le caratteristiche grafiche degli oggetti;

2. *nome*`.rsd` contiene gli attributi RSD associati a ciascun oggetto.

Questi sono utilizzati per ricreare grafi precedentemente realizzati e salvati in tali file.

Il metodo di questa classe, la cui esecuzione viene richiesta da altre classi per la creazione dei file prima detti, è `saveRsd`. Questo, dopo aver registrato nei due file la parte di commento, in base alla classe che richiede tale registrazione, esegue il metodo di scrittura necessario. Le operazioni effettuate con l'interfaccia ê che accedono ai metodi definiti nella classe `RsdWriter`, sono:

[6]Quando un classe viene dichiarata statica è possibile accedere alle variabili e ai metodi, in essa definiti, senza dover creare un'istanza di tale classe.

memorizzazione di un grafo, esportazione di un nodo, compresa l'eventuale struttura in esso contenuta, aggiunta di un nodo tra i nodi predefiniti e visualizzazione del grafo in formato RSD. L'interazione fra le classi per la realizzazione dell'ultima operazione viene mostrata in Figura 36.

Tale funzionalità può essere attivata sia agendo sulla barra degli strumenti definita dalla classe `TopBar` che dal menu principale. Il metodo `saveRsd` della classe `RsdWriter` è usato per memorizzare, nel file con estensione `rsd`, il formato RSD corrispondente al grafo disegnato. La finestra che visualizza tale formato viene definita dalla classe `ViewRsd` che utilizza il metodo `creaFormato` per la lettura del file `*.rsd` generato precedentemente.

La classe `RsdWriter` definisce i seguenti metodi:

- `writeRoot` per la memorizzazione di un grafo, l'elemento di partenza è la radice del grafo;
- `writeHyper` per la memorizzazione di una sottostruttura del grafo contenuta nell'ipernodo passato come parametro al metodo;
- `writeNode` per la memorizzazione di un nodo;
- `writeEdge` per la memorizzazione dell'oggetto grafico `GraphicEdge` che rappresenta gli edge fisici;
- `writeVirtual` per la memorizzazione dell'oggetto grafico `GraphicVirtual` che rappresenta gli edge virtuali (viene seguito lo stesso criterio applicato per l'oggetto grafico `GraphicEdge`);
- `writeTopo` per la memorizzazione di una topologia.

5.3.4 RsdEditor.window.option.EditorFileFilter

La classe `EditorFileFilter` viene istanziata dalla classe `FileChoose` che costituisce il file manager dell'interfaccia. Questa classe permette di aggiungere un particolare filtro al file manager mediante il quale vengono visualizzati solo i file che hanno l'estensione uguale al filtro specificato.

Per *RsdEditor* è stato aggiunto il filtro che permette di selezionare solo i file con l'estensione `gui`. Per visualizzare i nomi di tali file viene utilizzato il metodo `accept` che controlla tutti i file della directory corrente ed aggiunge a quest'ultimi solo quelli che hanno l'estensione `gui`.

Questa classe prevede anche un metodo per associare al filtro una breve descrizione che viene visualizzata insieme alla struttura del file manager.

5.3.5 RsdEditor.window.option.GraphicAction

La classe `GraphicAction` gestisce le operazioni effettuate all'interno del foglio di disegno dell'interfaccia. Le operazioni implementate riguardano gli oggetti nodo, edge e topologia. In corrispondenza di ogni operazione viene definito un metodo che "setta" la variabile globale `mode`, che è definita all'interno della classe `RsdEditor`, e rappresenta la modalità di disegno corrente (aggiunta di un nodo, rimozione di un edge, creazione di un sottolivello, ecc.).

Un metodo importante di questa classe è il metodo `hyper` il cui pseudo-codice è mostrato in Figura 37.

Questo metodo permette di costruire un nuovo ipernodo utilizzando gli oggetti (nodi e edge) di un ipernodo creato precedentemente. La classe `GraphicVirtual` effettua l'esportazione delle porte al livello superiore e, per conservare le scelte fatte precedentemente, mantiene inalterata la visibilità delle porte da parte degli oggetti che le utilizzano.

In Figura 38 viene mostrato un esempio di raggruppamento di oggetti in un nuovo ipernodo; in tale esempio viene evidenziata la gestione degli oggetti `GraphicVirtual`.

L'ipernodo corrente contiene due nodi, `N1` e `N2`, e due oggetti `GraphicVirtual`, `GV` e `GV1`. Quando viene analizzato il primo oggetto `GraphicVirtual` (`GV` o `GV1`) deve essere creato un nuovo `GraphicVirtual` (`newGV`) che rappresenterà tutti gli edge virtuali contenuti nell'ipernodo corrente. Per ciascuno di questi edge virtuali deve essere creato un nuovo edge virtuale per mantenere corretta l'esportazione delle porte.

Supponiamo che il primo `GraphicVirtual` considerato sia `GV` contenente lo edge virtuale denominato `oldV`. Viene creato un nuovo oggetto `Virtual` denominato `newV`; a tale oggetto vengono assegnati tutti i valori che sono associati ad `oldV`. Successivamente l'oggetto `GV` viene tolto dall'ipernodo corrente ed aggiunto al nuovo ipernodo `newHyper`, mentre l'oggetto `newGV` viene aggiunto all'ipernodo corrente. Dopo la creazione del nuovo ipernodo l'edge virtuale `oldV` è associato al nuovo oggetto `GraphicVirtual` `newGV` mentre il nuovo edge virtuale `newV` è associato al `GraphicVirtual` `GV`.

La coppia $\{p1,p2\}$ associata ad `oldV` rappresenta l'attributo RSD denominato *Binding*: *p1* è la porta definita nel nodo `N1`, mentre *p2* indica come viene rinominata tale porta nel livello superiore (ipernodo corrente). Al nuovo edge virtuale `newV` viene associato l'attributo *Binding* $\{p1,null\}$ perché la porta *p1* deve essere visibile, con lo stesso nome, anche all'oggetto `newHyper`; l'edge virtuale `oldV` rinomina la porta utilizzata, così come avrebbe fatto prima della creazione del nuovo ipernodo.

Le stesse operazioni devono essere fatte per tutti gli altri oggetti di tipo `GraphicVirtual` quale, ad esempio, `GV1` ma, indipendentemente dal loro nu-

mero, viene creato un solo `GraphicVirtual` (`newGV`) perchè è possibile aggiungere ad un nodo un solo oggetto `GraphicVirtual`.

Terminate queste operazioni, come mostrato nello pseudo-codice di Figura 37, rimane da aggiungere l'oggetto `newHyper` all'ipernodo corrente, togliere tutti gli oggetti dal foglio di disegno e ridisegnarlo. Il resto dell'operazione grafica viene gestito dalla classe `WorkCanvas` che, dopo aver intercettato il click del mouse sul foglio di disegno, visualizza la finestra relativa al nuovo ipernodo. Attraverso questa finestra è possibile modificare le caratteristiche di default nel nuovo ipernodo e l'operazione è considerata conclusa nel momento in cui questa finestra viene chiusa.

5.3.6 RsdEditor.window.option.MultiLineToolTipUI

La classe `MultiLineToolTipUI` permette di costruire i tooltip su più linee. Essa viene utilizzata per visualizzare l'elenco dei principali attributi RSD associati a ciascun oggetto.

Nel pacchetto *swing* di Sun Microsystems esiste la classe `JTooltip` che, però, prevede la visualizzazione dei dati su una sola linea. Pertanto, in *RsdEditor* abbiamo utilizzato la classe sviluppata da Albert Ting che è di pubblico dominio.

5.3.7 RsdEditor.window.option.Properties

La classe `Properties` realizza la gestione dei file con l'estensione *properties*. L'utilizzo di questi file permette di:

- internazionalizzare l'interfaccia (vedi ***S4***), basta fare delle copie di tali file riscrivendo nella lingua desiderata tutte le voci che potranno apparire nelle finestre dell'interfaccia stessa;
- costruire automaticamente alcuni componenti dell'interfaccia come, ad esempio, il menu o le barre degli strumenti.

Oltre ai metodi `getResourceString`, `getColor` e `tokenize` che permettono la lettura delle informazioni dai file **.properties* e la loro formattazione, questa classe definisce i metodi per la modifica della sezione del file *PredefNode_*.properties* dedicata alle voci del menu utilizzate per i nodi predefiniti. I metodi `aggiungi` e `togli` vengono richiamati, infatti, dopo aver introdotto le voci da associare ai nodi predefiniti nella relativa finestra.

Infine, è stato definito il metodo `hashTofile` che viene richiamato dalla classe `Preferences` quando viene premuto il tasto "Salva". Questa operazione ha l'effetto di memorizzare nel file *RsdEditor_*.properties* le caratteristiche

grafiche impostate per i nodi, gli edge e per la finestra principale come valori di default per le successive attivazioni dell'interfaccia.

Infatti, all'attivazione dell'interfaccia vengono letti i valori di default da questo file e memorizzati in una tabella hash denominata `valoriDefault`. Il contenuto di questa tabella può essere modificato selezionando il pulsante "Imposta" nella finestra "Preferenze" definita dalla classe `Preferences`. Quest'ultima funzionalità ha l'effetto di modificare i valori di default solo per la sessione corrente.

References

[1] R. R. Freund. *Optimal selection theory for superconcurrency.* In Proceedings of Supercomputing 89, p.699-703, 1989.

[2] A. S. Grimshaw, J. B. Weissman, E. A. West, E. C. Loyot. *Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems.* Journal of Parallel and Distributed Computing, 21:257-270, 1994.

[3] L. Smarr, C. E. Catlett. *Metacomputing.* Communications of the ACM, 35(6):45-52, June 1992.

[4] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, F. Ramme, T. Römke, J. Simon. *The MOL Project: An Open Extensible Metacomputer.* Proc. Heterogenous Computing Workshop HCW'97, IEEE Computer Society Press, 17-31.

[5] Metacomputer Online (progetto MOL).
http://www.uni-paderborn.de/pc2/projects/mol.

[6] CCS: Computing Center Software.
http://www.uni-paderborn.de/pc2/projects/ccs.

[7] A. Keller, A. Reinefeld. *CCS Resource Management in Networked HPC Systems.* Proc. Heterogeneous Computing Workshop HCW'98 at IPPS, Orlando, to appear 1998.

[8] M. Brune, J. Gehring, A. Keller, B. Monien, F. Ramme, A. Reinefeld. *Specifyng Resources ans Services in Metacomputing Environments.* Parallel Computing, to appear 1998.

[9] B. Bauer, F. Ramme. *A general purpose Resource Description Language.* TAT 91, Parallelle Datenverarbeitung mit dem Transputer, R. Grebe, M. Baumann, Springer Verlag, Reihe Informatik aktuell, 1991, 68-75.

[10] M. Brune, J. Gehring, A. Keller, A. Reinefeld. *RSD - Resource and Service Description.* Proc. of the Intern. Conf. on High-Performance Computing Systems HPCS 98, Edmonton, Canada, Springer, LNCS, May 1998.

[11] The $Java^{TM}$ Development Kit (JDK^{TM}).
http://java.sun.com/products/jdk/.

[12] Ken Arnold, James Gosling
Java - Didattica e programmazione.
Addison Wesley Italia, 1997.

[13] *Java 1.1 Tutto & Oltre.*
Casa Editrice Apogeo Informatica.

[14] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison Wesley, Reading, Mass., 1986.

```
public void paint(Graphics g)
{
   imposto colore dell'arco;
   // leggo l'attributo "Point" dal vettore GUI
   Point s = Point(nodo sorgente dell'arco);
   Point d = Point(nodo destinazione dell'arco);
   Point oldSource = Point memorizzato nel metodo "SetSource";
   Point moving;
   Point fixed;
   if (oldSource.equals(s)) {
      // si sta muovendo il nodo destinazione
      moving = d;
      fixed = s;
   }
   else {
      // si sta muovendo il nodo sorgente
      moving = s;
      fixed = d;
      oldSource = s;
   }
   // Disegno l'arco
   if (Math.abs(moving.y - fixed.y) < 11) (*)
      // casi di allineamento orizzontale
      if (moving.x > fixed.x)
         imposta l'arco al confine primo/secondo quadrante;
      else
         imposta l'arco al confine terzo/quarto quadrante;
   else if (Math.abs(moving.x - fixed.x) < 11) (*)
      // casi di allineamento verticale
      if (moving.y > fixed.y)
         imposta l'arco al confine secondo/terzo quadrante;
      else
         imposta l'arco al confine quarto/primo quadrante;
   else if ((moving.x > fixed.x) && (moving.y < fixed.y))
         imposta l'arco nel primo quadrante;
   else if ((moving.x > fixed.x) && (moving.y > fixed.y))
         imposta l'arco nel secondo quadrante;
   else if ((moving.x < fixed.x) && (moving.y > fixed.y))
         imposta l'arco nel terzo quadrante;
   else
         imposta l'arco nel quarto quadrante;
   ridisegna l'arco g;
}
```

Figure 27: Pseudo-codice del metodo `paint` della classe `GraphicEdge`

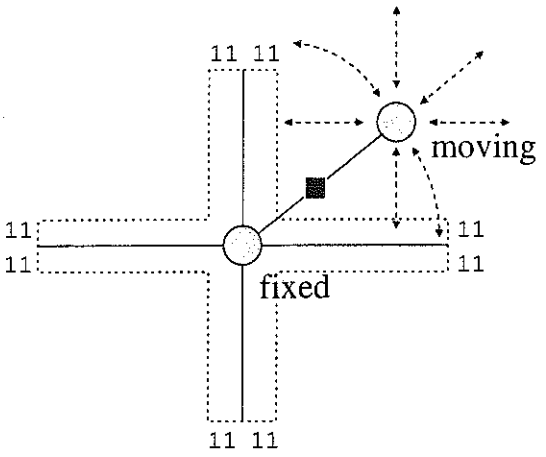

Figure 28: Casi critici da gestire nel metodo `paint` della classe `GraphicEdge`

```
public void mouseEntered (MouseEvent e)
{
    if (this.getToolTipText() == null)
    {
        //creo il tooltip dell'arco
        int num = 0;
        StringBuffer tooltip = new StringBuffer(" "
                    + getGUIAttribute("Name") + "\n");
        for (Enumeration s = virtualList.elements();
                    s.hasMoreElements(); )
        {
            Virtual virtual = (Virtual)s.nextElement();
            tooltip.append(++num + ":  " +
                    virtual.getRSDName() + "\n");
            tooltip.append(" Source :  " +
                    virtual.getRSDSource() + "\n");
            tooltip.append(" Binding :  " +
                    virtual.getBinding() + "\n");
            tooltip.append("--------------------\n");
        }
        setToolTipText(tooltip.toString());
        repaint();
    }
}
```

Figure 29: Codice sorgente per la generazione del tooltip visualizzabile sull'oggetto `GraphicVirtual`

```
public class Item
{
    protected String attribute; // nome dell'attributo
    protected Object value; // valore dell'attributo
    // crea un elemento attributo con valori indefiniti
    public Item()
    {
        attribute = new String();
        value = new Object();
    }
    // crea un elemento attributo con valori definiti
    // dai parametri passati al costruttore
    public Item(String attribute, Object value)
    {
        his.attribute = attribute;
        this.value = value;
    }
    // Imposta l'elemento attributo
    public void setItem(String attribute, Object value)
    {
        this.attribute = attribute;
        this.value = value;
    }
    // Imposta il nome dell'attributo
    public void setAttribute(String attribute)
    {       this.attribute = attribute;       }
    // Imposta il valore dell'attributo
    public void setValue(Object value)
    {       this.value = value;       }
    // Restituisce il nome dell'attributo
    public String getAttribute()
    {       return attribute;       }
    // Restituisce il valore dell'attributo
    public Object getValue()
    {       return value;       }
}
```

Figure 30: Codice sorgente della classe `Item`

```
public GuiParser(String file,Hypernode padre,...)
{ ..... }

public void run()
{
    Hypernode father = ipernodo dell'oggetto da ricostruire;
    fileGUI = file in formato GUI;
    fileRSD = file in formato RSD;
    apri fileGUI e fileRSD;
    buffer = fileGUI;
    chiudi fileGUI;
    if (padre ≠ null)
        // non devo creare un nuovo ipernodo nel caso di
        // un'importazione o di un'aggiunta di un nodo predefinito
        father = padre;
    else
        father = new Hypernode();
    ignora righe di commento di fileRSD;
    S();
    chiudi fileRSD;
    if (non e' operazione di importazione o di aggiunta
              di nodi predefiniti)
        imposta radice dell'albero;
    riattiva thread della generazione dell'albero;
    if (e' operazione di importazione o di aggiunta
              di nodi predefiniti)
        ridisegna sul foglio di disegno il livello corrente;
    annulla modalita';
}
```

Figure 31: Pseudo-codice del metodo `run` della classe `GuiParser`

```
// Metodo che realizza la produzione S -> {N}
// cioe' il passo iniziale dell'analisi sintattica
public void S()
{
    token = prossima stringa in lettura;
    ignora commento fileGUI;
    if (token = "{")
        token = prossima stringa in lettura;
        N()
        if (token = "}")
            lettura dei file completata con successo;
        else
            errore di sintassi in fileGUI;
    else
        errore di sintassi in fileGUI;
}
```

Figure 32: Pseudo-codice del metodo `S` della classe `GuiParser`

```
// Metodo che realizza le produzioni:
// N -> NODE[attrn N]N cioe' una sequenza di nodi
// N -> E cioe' l'inizio di una sequenza di edge
public void N()
{
    if (token = "NODE")
        token = prossima stringa in lettura;
        if (token = "[") {
            oggetto = oggetto grafico ricostruito con la lettura
                    delle caratteristiche grafiche e
                    degli attributi RSD;
            Node node = null;
            if (oggetto e' di tipo "Node")
                node = oggetto;
            aggiungi oggetto all'ipernodo corrente (father);
            token = prossima stringa in lettura;
            if (node ≠ null) {
                // controllo se node e' ipernodo
                Hypernode currFather = father;
                father = new Hypernode();
                N(); (1)
                if (father contiene degli elementi) {
                    copia i vettori di node in quelli di father;
                    aggiungi father a currFather;
                }
                father = currFather;
                currfather = null;
            }
            if (token = "]") {
                token = prossima stringa in lettura;
                N(); (2)
            }
            else
                errore di sintassi in fileGUI;
        }
        else
            errore di sintassi in fileGUI;
    else
        E();
}
```

Figure 33: Pseudo-codice del metodo `N` della classe `GuiParser`

```
// Metodo che realizza le produzioni:
// E -> EDGE[attre]E cioe' una sequenza di edge
// E -> A cioe' l'inizio di una sequenza di assign
public void E()
{
    if (token = "EDGE")
        token = prossima stringa in lettura;
        if (token = "[")
        {
            GraphicEdge gEdge = oggetto ricostruito con la
                lettura di tutte le sue caratteristiche grafiche
                e di tutti gli attributi RSD dei suoi componenti;
            aggiungi gEdge all'ipernodo corrente (father);
            token = prossima stringa in lettura;
            if (token = "]")
            {
                token = prossima stringa in lettura;
                E();
            }
            else
                errore di sintassi in fileGUI;
        }
        else
            errore di sintassi in fileGUI;
    else
        A();
}
```

Figure 34: Pseudo-codice del metodo `E` della classe `GuiParser`

```
// Metodo che realizza le produzioni:
// A -> assign A cioe' un sequenza di assign
// A -> ε cioe' il passo finale dell'analisi sintattica
public void A()
{
    if (token = "ASSIGN")
        token = prossima stringa in lettura;
        if (token = "[")
        {
            GraphicVirtual gVirt = oggetto ricostruito con la
                lettura delle sue caratteristiche grafiche
                e degli attributi RSD dei suoi componenti;
            aggiungi gVirt all'ipernodo corrente (father);
            token = prossima stringa in lettura;
            if (token = "]")
            {
                token = prossima stringa in lettura;
                A();
            }
            else
                errore di sintassi in fileGUI;
        }
        else
            errore di sintassi in fileGUI;
}
```

Figure 35: Pseudo-codice del metodo `A` della classe `GuiParser`

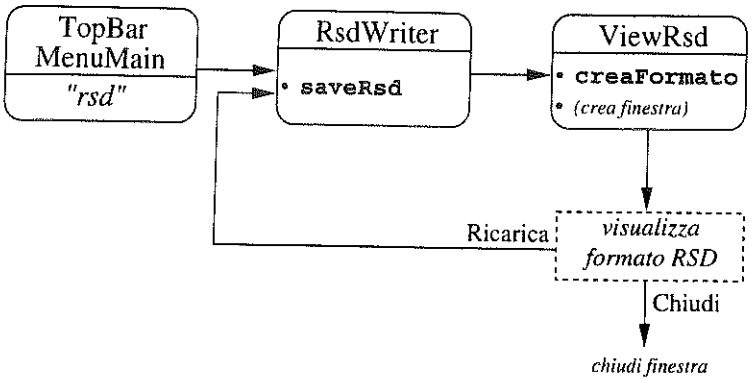

Figure 36: Interazione fra le classi per la visualizzazione del grafo in formato RSD

```
public static void hyper
{
    Hypernode newHyper = new Hypernode(); //nuovo ipernodo
    Vettore iper = elementi contenuti nell'ipernodo corrente;
    for (int i = 0 to numero_elementi_iper) {
        // cambio il padre ad ogni oggetto
        if (iper(i) non e' "GraphicVirtual") {
            imposta newHyper come ipernodo di iper(i);
            aggiungi iper(i) a newHyper;
            togli iper(i) dall'ipernodo corrente;
        }
        else {
            GraphicVirtual GV = oggetto corrente;
            if (GV e' il primo considerato) {
                // creo un nuovo GraphicVirtual al livello superiore
                GraphicVirtual newGV = new GraphicVirtual();
                imposta i valori di newGV;
            }
            for (ogni Virtual V incluso in GV) {
                // sposto al livello superiore tutti gli edge virtuali
                // e ne inserisco altrettanti nel livello corrente
                Virtual oldV = V;
                Virtual newV = new Virtual();
                imposta il nome di newV;
                copia gli altri valori di oldV in newV;
                togli oldV da GV;
                aggiungi newV a GV;
                aggiungi oldV a newGV;
                imposta i nuovi valori per oldV;
                imposta newHyper come ipernodo di GV;
            }
            aggiungi GV a newHyper;
            togli GV dall'ipernodo corrente;
        }
    }
    aggiungi l'eventuale newGV all'ipernodo corrente;
    aggiungi newHyper all'ipernodo corrente;
    togli tutti gli elementi dal foglio di disegno;
    ridisegna il foglio di disegno;
}
```

Figure 37: Pseudo-codice del metodo hyper della classe `GraphicAction`

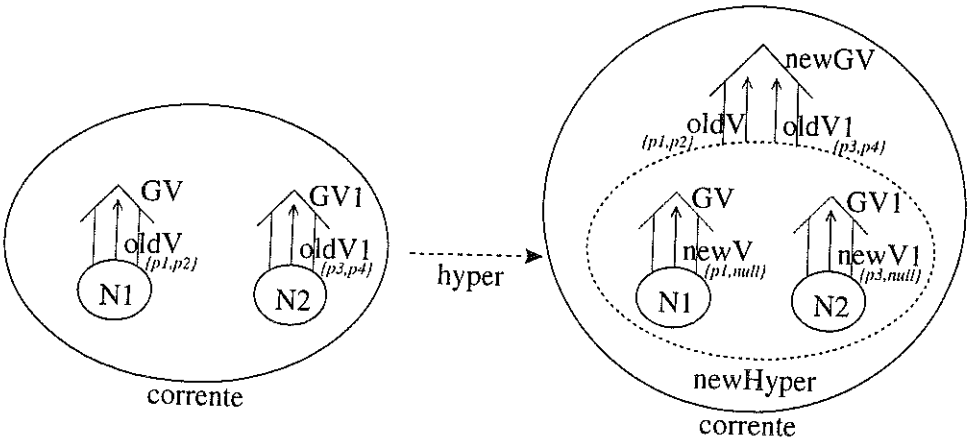

Figure 38: Esempio di raggruppamento di oggetti in un nuovo ipernodo, evidenziando la gestione degli oggetti di tipo "GraphicVirtual"