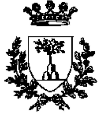

UNIVERSITÀ DEGLI STUDI DI FERRARA

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica e

dell'Automazione

Sviluppo di un visualizzatore grafico per il linguaggio SCIFF

Tesi di Laurea di:

Andrea Peano

Relatore:

Ing. Marco Gavanelli

Anno Accademico: 2007/2008

Indice

1 Introduzione **5**

2 Background **7**

2.1 Il linguaggio SCIFF 7

2.2 Analisi dei requisiti 10

2.3 Eclipse 15

3 Sviluppo di un visualizzatore grafico per SCIFF **19**

3.1 DOT e il pacchetto Graphviz 19

3.2 Il parser 25

3.3 Ricerca delle relazioni tra atomi 28

3.4 Creazione del file DOT 30

3.5 Un esempio 34

4 Integrazione del visualizzatore nell'ambiente Eclipse **37**

4.1 Prelievo degli output dalla console di Eclipse 37

4.2 Creazione degli strumenti grafici necessari per l'interazione con l'utente 40

5 Installazione ed utilizzo di SCIFFgraph **47**

5.1 Installazione dei pacchetti necessari per il corretto funzionamento 47
5.2 Come utilizzare SCIFFgraph 49

6 Conclusioni **53**

Bibliografia **55**

Capitolo 1

Introduzione

La programmazione logica è un paradigma di programmazione dichiarativo, in cui non è necessario esplicitare un algoritmo, ma ci si focalizza sulle relazioni logiche che definiscono un problema. Il linguaggio SCIFF [1] fa parte dei linguaggi di programmazione logica abduttiva [2] ed è stato utilizzato per diverse applicazioni, fra cui la verifica a run-time del comportamento di agenti. Il comportamento degli agenti è descritto dagli *atomi* abdotti [3], che possono ad esempio rappresentare eventi che in un determinato istante di tempo ci si aspetta o non ci si aspetta che accadano. Tali atomi possono contenere dalle *variabili*, che a loro volta possono essere soggette a *vincoli* [4].

Al linguaggio è associata una procedura di dimostrazione automatica, chiamata anch'essa SCIFF. La procedura è implementata in SICStus Prolog [5] e visualizza l'output in formato testuale sulla console. In particolare il risultato dell'esecuzione dei programmi SCIFF che SICStus stampa è di difficile lettura, in quanto contiene molti atomi e relazioni che è difficile interpretare in una visione globale.

Visto che spesso i vincoli vengono rappresentati tramite grafi, risulta

possibile visualizzare gli output della SCIFF in grafi facilmente leggibili.

Il linguaggio SCIFF e l'applicazione SICStus Prolog sono stati integrati nell'ambiente di Eclipse [6], attraverso un plug-in, in una tesi precedente [7]. Risulta quindi naturale pensare di estendere l'embiente integrato con le funzionalità del visualizzatore, in modo da fornire al programmatore SCIFF un ambiente efficace per sviluppare ed eseguire nuove applicazioni.

Capitolo 2

Background

2.1 Il linguaggio SCIFF

Il linguaggio SCIFF [1] è un linguaggio che appartiene al paradigma della Programmazione Logica e che include alcune delle più importanti estensioni che nel corso degli anni sono state proposte: la programmazione logica abduttiva [2] e la programmazione logica a vincoli [8]. Il linguaggio SCIFF è associato ad una procedura di dimostrazione automatica, chiamata anch'essa SCIFF, che è un'estensione della procedura abduttiva IFF [8].

SCIFF è stato sviluppato inizialmente per la specifica e verifica di protocolli di interazione fra agenti. Il linguaggio SCIFF permette infatti di definire un protocollo di interazione tramite regole della logica, in forma di implicazioni; successivamente le regole definite per il protocollo possono essere usate dalla procedura di dimostrazione SCIFF per verificare il comportamento degli agenti a run-time, ovvero durante l'interazione.

Ad esempio, un semplice protocollo potrebbe essere il seguente:

$$\begin{aligned}&h(tell(Sender, Receiver, request(X)), T)\\ &\rightarrow e(tell(Receiver, Sender, accept(X)), T_a) \wedge T_a > T\\ &\vee e(tell(Receiver, Sender, refuse(X)), T_r) \wedge T_r > T.\end{aligned} \tag{2.1}$$

in cui si stabilisce che se un agente $Sender$ manda un messaggio di richiesta ad un agente $Receiver$ per un certo servizio X, allora *ci si aspetta* che l'agente $Receiver$ risponda o accettando la richiesta o rifiutando. Nel linguaggio SCIFF, gli eventi che accadono (in Inglese *happen*) sono identificati da:

$$h(Descrizione, Tempo)$$

e le aspettative (*expectation*) sono identificate da

$$e(Descrizione, Tempo).$$

È possibile imporre dei vincoli fra le variabili logiche che compaiono in eventi ed aspettative: nell'esempio, il tempo T_a in cui ci si aspetta arrivi la risposta deve essere successivo al tempo T in cui è accaduto l'evento di richiesta.

Il linguaggio SCIFF contiene molti altri costrutti: le variabili possono essere quantificate esistenzialmente ($\exists$) o universalmente ($\forall$); le aspettative possono essere positive (ci si aspetta che qualcosa accada) identificate come nell'esempio precedente dal funtore e, oppure negative (ci si aspetta che un certo evento non accada), con funtore en. I dettagli sul linguaggio SCIFF sono descritti nell'articolo [1] e non verranno riportati qui perché esulano dalle finalità di questa tesi.

La procedura di dimostrazione SCIFF è stata implementata nel linguaggio SICStus Prolog [5] ed è stata utilizzata per numerose applicazioni. Purtroppo l'output risultante è spesso difficile da leggere e poco intuitivo: contiene

in genere vari *atomi* che sono stati abdotti (cioè ipotizzati) dalla procedura. Questi possono essere aspettative o altri tipi di atomi abducibili. Oltre agli eventi ed alle aspettative, nell'output ci sono spesso numerosi vincoli, che collegano le variabili. Ad esempio, se si esegue la SCIFF con la regola mostrata nell'equazione 2.1 e si fornisce la sequenza di eventi (detta anche *storia* o *history*)

$$h(tell(alice, bob, request(aiuto), 1)$$

(ovvero l'agente *alice* chiede *aiuto* all'agente *bob* all'istante di tempo 1), una possibile risposta (semplificata) è la seguente:

```
exists(A)
e(tell(bob,alice,accept(aiuto),A)
A > 1
```

La risposta significa che esiste un tempo $A > 1$ in cui *bob* dovrebbe rispondere ad *alice*.

Si può facilmente immaginare che in protocolli più complessi ci siano molti più atomi, vincoli e quantificatori che in questo semplice esempio. Per questo, sarebbe utile un visualizzatore che mostri in maniera intuitiva l'output della SCIFF, essenzialmente i vincoli e gli atomi.

Un modo classico per rappresentare i vincoli è per mezzo di *grafi*. Spesso le variabili vengono rappresentate come *nodi* di un grafo e sono collegate tramite *vincoli*, che costituiscono gli *archi* del grafo. Una visualizzazione adatta dell'output della procedura SCIFF potrebbe rappresentare gli atomi come nodi ed i vincoli come archi che collegano i nodi. Scopo di questa tesi sarà quindi sviluppare un software, integrato nel pacchetto SCIFF, che permetta di visualizzare semplicemente l'output secondo questa modalità.

2.2 Analisi dei requisiti

Con l'ausilio di un frammento di output di SCIFF, verranno ora discussi i requisiti fondamentali che un'applicazione dovrebbe implementare per poter essere considerata un buon visualizzatore SCIFF.

La seguente successione di stringhe è composta da atomi e vincoli che possono rappresentare un tipico esempio di output di un programma SCIFF lanciato su SICStus Prolog:

```
exists(A),
exists(B),
h(tell(alice,bob,request(aiuto)),1),
h(tell(alice,dan,request(aiuto)),2),
e(tell(bob,alice,accept(aiuto)),A),
h(tell(bob,dan,request(help(alice))),4),
e(tell(dan,alice,accept(aiuto)),B),
en(tell(dan,alice,accept(aiuto)),C),
e(tell(dan,bob,accept(help(alice))),D),
en(tell(bob,dan,accept(help(alice))),D),
A > 1,
B > 1,
C > A,
D > 4
```

Negli output sopra elencati viene mostrato come *bob* e *dan* reagiscono ad una richiesta di aiuto di *alice*, e sarebbe interessante evidenziare graficamente le relazioni che sono presenti tra gli atomi *h*, *e* ed *en*.

Come già affermato, un modo per tradurre gli output di SCIFF in grafi sarebbe quello di rappresentarne gli atomi come nodi. Se si volessero inoltre evidenziare le relazioni tra i vari atomi, si potrebbero unire con un arco le coppie di nodi contenenti atomi che condividono una variabile, riportando accanto all'arco il nome della variabile in questione. Nella figura 2.1 sono raffigurati quindi tutti gli atomi presenti negli output sopra elencati e le loro relazioni dirette.

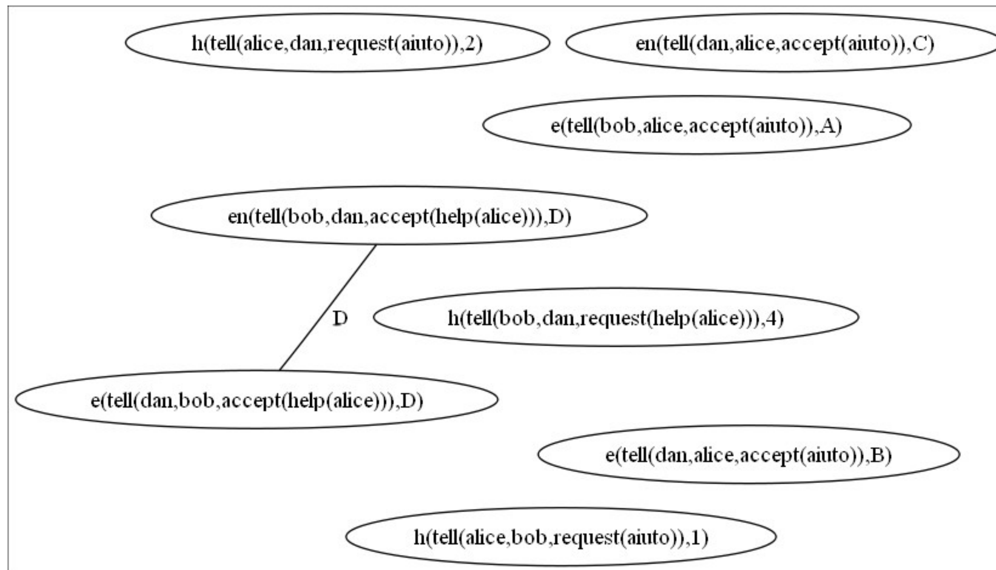

Figura 2.1: Rappresentazione grafica degli atomi e delle relazioni dirette

I vincoli sono un altro tipo di relazione che può collegare due atomi. Perciò sarebbe opportuno che il visualizzatore riconoscesse e raffigurasse tali relazioni come archi, analogamente alla figura precedente. Negli output mostrati in apertura di questo paragrafo, si può notare che gli atomi:

```
e(tell(bob,alice,accept(aiuto)),A)
```

ed

```
en(tell(dan,alice,accept(aiuto)),C)
```

sono relazionati dal vincolo $C > A$.

Disegnando anche questo tipo di relazione nella figura 2.1 si otterrebbe il grafo in figura 2.2.

I funtori indicano quale significato si deve attribuire agli atomi (paragrafo 2.1); ciò significa che a funtori diversi corrispondono significati differenti. I grafi perciò dovrebbero evidenziare tali diversità attraverso la morfologia e il posizionamento dei nodi. Di conseguenza il visualizzatore dovrebbe colorare

gli atomi in modo diverso a seconda del loro funtore, raggruppandoli inoltre all'interno di un sottografo bordato; nella figura 2.3 si può vedere un esempio conforme alle specifiche appena introdotte.

Il parametro "Tempo" presente all'interno degli atomi dà informazioni cronologiche riguardo l'evento che l'atomo rappresenta. Tale parametro può essere costante o variabile, come si può anche vedere nell'esempio di output di SCIFF presente all'inizio di questo paragrafo. È possibile quindi effettuare un ordinamento degli atomi, in senso crescente, secondo il parametro temporale. Graficamente si dispongono gli atomi appartenenti allo stesso sottografo in modo sequenziale, visualizzandoli tanto più in basso quanto è grande il parametro temporale che contengono. Anche gli atomi con tempo variabile debbono essere disposti l'uno sotto l'altro, nonostante non si possa definire per loro un esatto ordinamento cronologico. Il visualizzatore perciò deve essere in grado di generare grafi di tipo gerarchico, cioè grafi che abbiano i nodi di uno stesso sottografo disposti su livelli sempre più bassi, man mano che cresce il loro valore temporale.

Un programma SCIFF di elevata complessità può dar luogo, in termini di lunghezza della stringa, ad atomi molto estesi. Inserire parole lunghe all'interno dei nodi di un grafo che utilizza forme rotonde od ovali può compromettere la leggibilità del grafo stesso. È necessario quindi che i nodi dei grafi raffiguranti gli output di SCIFF abbiano una forma rettangolare, che permettano quindi un'estensione orizzontale senza comportare perdite dal punto di vista della comprensibilità.

Nel linguaggio SCIFF gli atomi con funtore h hanno una rilevanza particolare, infatti è attraverso le osservazioni su questi che scaturiscono tutti gli altri tipi di atomi. Il visualizzatore grafico quindi dovrebbe assegnare una forma diversa a quei nodi del grafo che raffigurano atomi con funtore h.

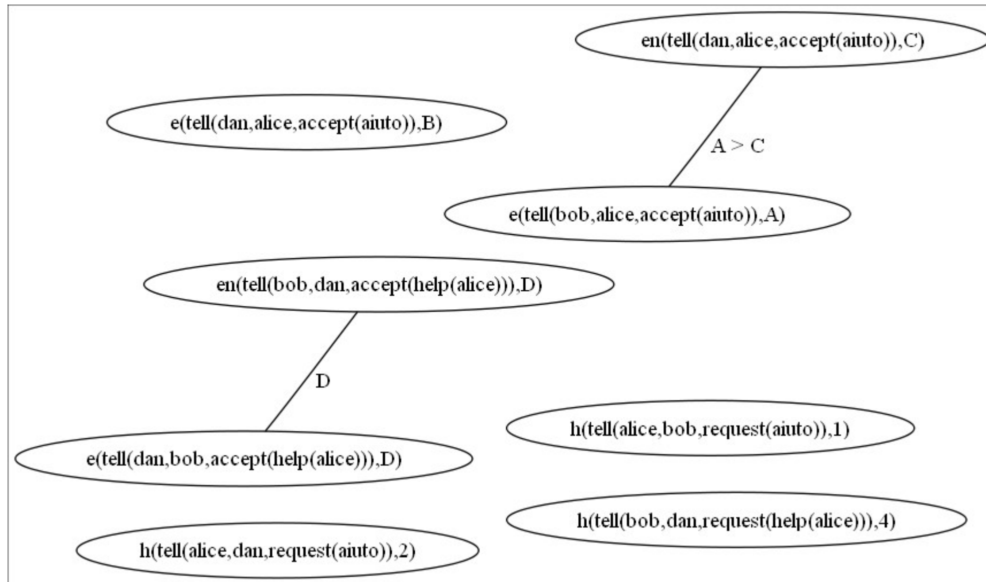

Figura 2.2: Rappresentazione grafica delle varie relazioni tra atomi

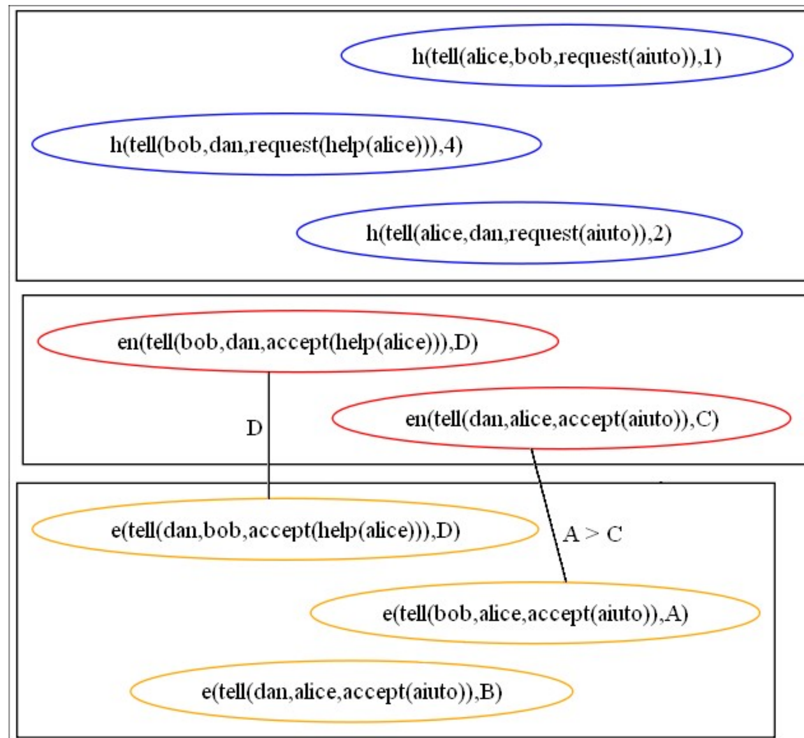

Figura 2.3: Nodi colorati e raggruppati in base al funtore degli atomi

A partire dagli output SCIFF presentati in apertura di questo paragrafo, ed implementando le specifiche fin qui discusse, il visualizzatore avrebbe il compito di creare un grafo con le stesse caratteristiche di quello mostrato nella figura sottostante.

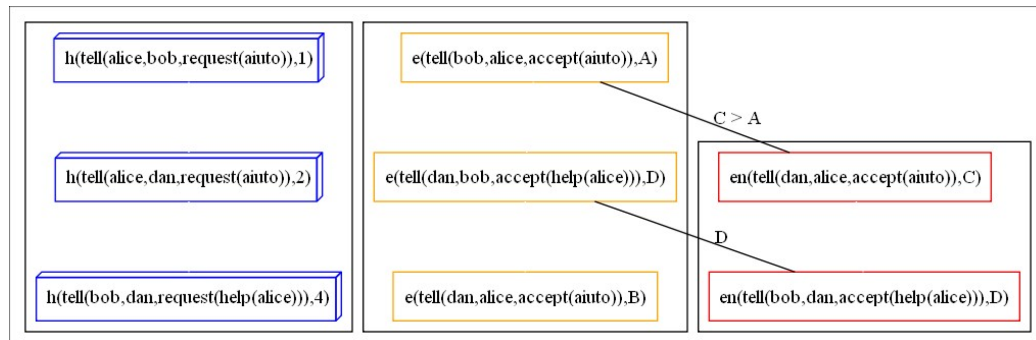

Figura 2.4: Grafo SCIFF gerarchico

Si noti che i nodi del sottografo contenente gli atomi con funtore h sono disposti in ordine cronologico.

2.3 Eclipse

Attualmente il linguaggio SCIFF è integrabile nella piattaforma di Eclipse [6], che è un ambiente di sviluppo integrato (IDE), attraverso un apposito Plug-in, chiamato appunto ‘‘SCIFF’’ [7].

Comunemente per Eclipse si intende l'Eclipse Software Development Kit (SDK), il quale non è altro che un insieme di tool aggiuntivi che estendono le funzionalità di una piattaforma sviluppata appositamente per essere ampliata e che prende il nome di Eclipse Platform. Proprio alla funzione estensiva di questi tool e alla loro dipendenza dalla piattaforma è dovuto il termine tecnico con il quale vengono chiamati: Plug-in. In figura 2.9 viene mostrato lo schema della struttura di Eclipse.

La necessità di avere la massima compatibilità tra la platform e i diversi plug-in ha prodotto una notevole standardizzazione degli interfacciamenti. In particolare ogni plug-in si presenta ad Eclipse attraverso un file manifesto che contiene tutte le infomazioni riguardo, principalmente, le parti che si interessa di estendere; tale interfacciamento è chiamato ‘‘Extension”. Un'altra importante dichiarazione che può essere presente nel file manifesto è ‘‘Extension Point”, la quale ha lo scopo di rendere noto un punto su cui moduli terzi possono interfacciarsi al plug-in, per estenderne a loro volta le funzionalità.

Una volta avviata la piattaforma, essa ricerca, all'interno della cartella che per default ospita i plug-in, tutti i loro file manifest, creando così un registro di quelli disponibili, i quali vengono avviati solamente in caso di necessità, per esplicita richiesta dell'utente o di un altro plug-in. Tale modo di operare permette di non appesantire l'applicazione con l'avvio di moduli inutili.

La Eclipse Platform contiene anche un particolare modulo chiamato ‘‘Workspace” che rappresenta lo spazio di lavoro di Eclipse; all'interno di que-

sto vengono memorizzati i vari progetti, ognuno dentro una diversa cartella, che vengono differenziati da un nome univoco e attraverso una particolare caratteristica intrinseca, la “nature”. Tale attributo permette di differenziare i diversi progetti secondo la loro natura, la loro appartenenza. Per esempio la nature dei progetti SCIFF su Eclipse è “SCIFF.SCIFFNature”.

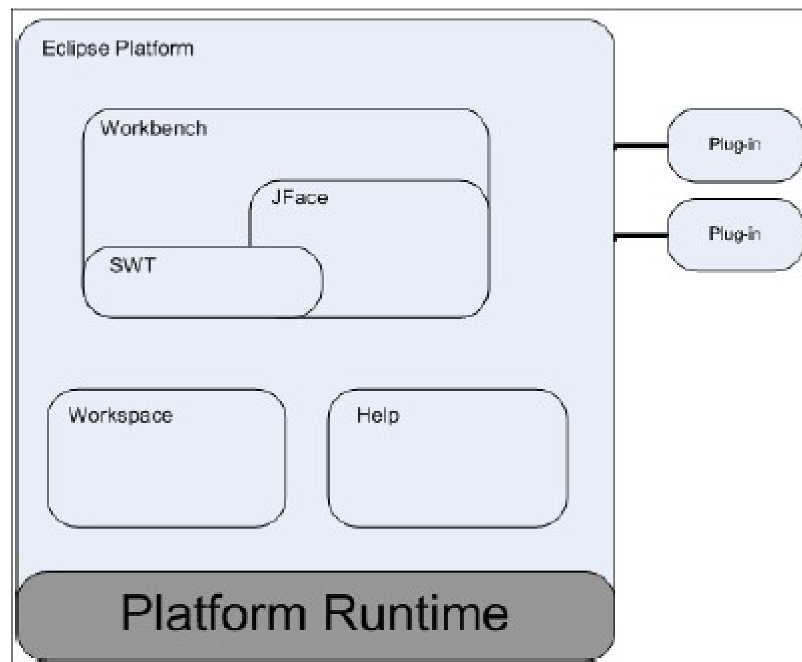

Figura 2.5: Struttura di Eclipse

Il modulo SWT (Standard Widget Toolkit) definisce delle interfacce indipendenti dal sistema operativo e serve per creare elementi grafici come bottoni, checkbox, ecc.

Il modulo JFace permette invece di aggiungere elementi all'interfaccia grafica di Eclipse; uno di questi è di rilevante importanza per l'interagibilità con l'utente: l'“Action”, cioè un comando che può essere richiamato per svolgere una determinata azione. Tali action vengono dichiarate all'interno del file manifesto del plug-in che le implementa, così da permettere ad Eclipse

di visualizzare i relativi elementi grafici, come icone e menu, senza dover istanziare l'intero plug-in, che viene attivato solo quando l'utente interagisce con le sue action.

Fondamentale è un ulteriore modulo, il "Workbench", per inserire il plug-in all'interno del contesto grafico di Eclipse. Il plug-in SCIFF è caratterizzato, in questo senso, da un Editor ed una View, dove il primo permette di stilare un documento in liguaggio SCIFF, e con il secondo si può navigare all'interno delle proprietà dell'oggetto in uso. Anche per Editor e View vale un concetto simile a quello di "nature", in quanto essi possono essere funzionali solamente ad un determinato tipo di contesto definito da un particolare plug-in; e grazie a tale appartenza si può unire Editor e View in una stessa "Perspective", letteralmente "prospettiva", che serve per fare della interfaccia di Eclipse l'ambiente specializzato per una data funzionalità, definita dal plug-in che si vuole utilizzare.

Nel caso del plug-in SCIFF è la "SCIFF perspective" che raggruppa Editor e View contestuali all'ambiente di programmazione di SCIFF. L'Editor in particolare può visualizzare con diversi colori le parole chiave del linguaggio. Attraverso la View ci si può spostare facilmente da un progetto all'altro, visualizzando se necessario le proprietà ed i file dei progetti SCIFF.

Per eseguire un progetto SCIFF è necessario lanciare per prima cosa il programma SICStus Prolog attraverso l'apposita icona "Start SCIFF", di conseguenza all'interno di una scheda di Eclipse si apre e viene visualizzata la console con tutti gli output relativi all'esecuzione di SICStus, solo allora si può compilare il progetto desiderato con l'ausilio del tasto "compile". Infine, se la compilazione è stata terminata con successo, si può lanciare il progetto utilizzando il tasto "run", sulla console di SCIFF verranno stampati gli output dovuti all'esecuzione dello stesso sotto SICStus Prolog.

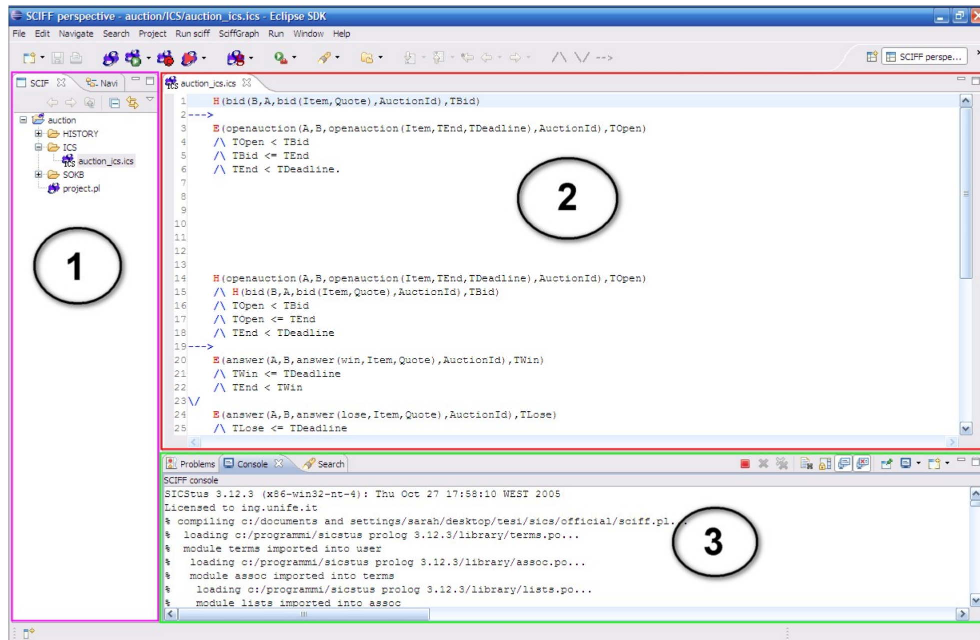

Figura 2.6: SCIFF perspective

Nella figura precedente i numeri *1*, *2* e *3* individuano rispettivamente la finestra "*SCIFF Navigator*", l'editor e la console dove vengono stampati gli stream relativi all'esecuzione di SICStus.

Attraverso il Plug-in Development Environment (PDE) è possibile sviluppare tool personalizzati, utilizzando le apposite librerie Java messe a disposizione del programmatore. Nel capitolo 3, a tal proposito, verranno illustrati i passaggi necessari per l'integrazione del visualizzatore degli output di SCIFF all'interno dell'ambiente di Eclipse, sotto forma di plug-in; e per permettere una totale compatibilità tra il visualizzatore e la piattaforma si è scelto di implementare le specifiche utlizzando il linguaggio Java.

Capitolo 3

Sviluppo di un visualizzatore grafico per SCIFF

3.1 DOT e il pacchetto Graphviz

Per disegnare i grafi, si è scelto di utilizzare il linguaggio DOT [9], che permette di descrivere testualmente gli elementi costitutivi di un grafo, le relazioni tra di essi e le loro proprietà grafiche.

Nel linguaggio DOT le entità fondamentali sono due: i nodi e le relazioni, rappresentate graficamente da un arco. Le configurazioni basilari sono invece il nome del nodo ed il tipo di arco, diretto o non diretto, che può collegare due nodi diversi oppure un nodo con se stesso. È possibile altresì che un nodo non sia legato ad altri nodi. Numerosissime altre opzioni sono a disposizione del programmatore per diversificare le forme e i contenuti dei grafi.

In primis è necessario definire all'interno del file DOT di nostro interesse — che è un semplice file di testo e per convenzione ha estensione *.dot* — se il grafo è di tipo diretto oppure no ed il suo nome, in questo modo:

```
(graph\digraph) nome {codice...}
```

Detto questo è molto semplice ed intuitivo descrivere un primo grafo non diretto, attraverso le seguenti dichiarazioni:

```
graph grafo{
  a -- b;
  c;
  d -- d;
}
```

Il seguente è un grafo diretto:

```
digraph grafo{
  a -> b;
  c;
  d -> d;
}
```

Tali documenti in linguaggio DOT descrivono i seguenti grafi:

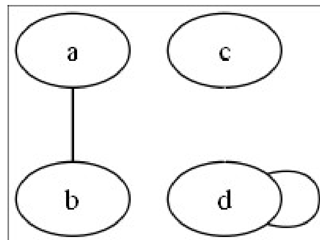

Figura 3.1: Grafo non direzionale

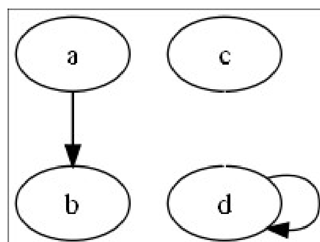

Figura 3.2: Grafo direzionale

Se può anche etichettare l'arco che unisce i due nodi a e b con il nome che rappresenta la relazione che intercorre tra loro, tramite la seguente dichiarazione messa a disposizione dal linguaggio DOT:

```
a -- b [label=''etichetta''];
```

Applicando tale costrutto agli atomi ed ai vincoli di SCIFF (come introdotto nel paragrafo 1.2), si può definire il primo grafo in cui gli atomi vengono mostrati come nodi ed i vincoli come relazioni, come segue:

```
graph grafoSCIFF{
"h(tell(alice,bob,request(aiuto)),A)" --
  "e(tell(bob,alice,accept(aiuto)),B)" [label="B>A"];
}
```

Ottenendo graficamente il seguente risultato:

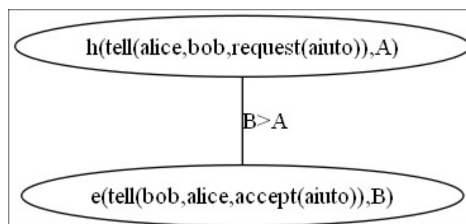

Figura 3.3: Esempio di grafo SCIFF

Volendo si possono utilizzare varie opzioni del linguaggio DOT per adattare i grafi alle proprie necessità. Ad esempio, si può modificare la forma o il colore dei nodi con la sintassi: `node[shape=forma, color=colore]`, dove *shape* definisce la forma mentre *color* il colore. Per nodi che contengono nomi lunghi, come tipico degli atomi di SCIFF, è utile impostare la forma con l'attributo *box* oppure *box3d* dove il primo è un rettangolo semplice ed il secondo un parallelepipedo rettangolo. Ogni volta che si inserisce una dichiarazione del tipo *node* essa si propaga "a cascata", cioè vale per tutti i nodi definiti successivamente fino ad una eventuale altra dichiarazione.

Nel paragrafo 1.2 è stato affrontato il problema dell'ordinamento cronologico degli atomi di un output di SCIFF, in figura 1.5 in particolare è presente il grafo, definito gerarchico, che dispone in modo ordinato gli atomi all'interno di sottografi con cornice. Il linguaggio DOT, a tal proposito, dà la possibilità di raggruppare un insieme di nodi all'interno di sottografi incorniciati da un bordo. La successione di atomi:

```
h(tell(alice, bob, request(aiuto)),1)
h(tell(alice, bob, request(aiuto)),6)
e(tell(bob,alice,accept(aiuto)),A)
e(tell(bob,alice,accept(aiuto)),B)
```

si può scrivere in DOT nel modo seguente:

```
graph grafoSCIFF{
  node[shape=box3d, color=blue];
  subgraph cluster_h{
     "h(tell(alice,bob,request(aiuto)),1)";
     "h(tell(alice,bob,request(aiuto)),6)";
  }
  node[shape=box, color=orange];
  subgraph cluster_e{
     "e(tell(bob,alice,accept(aiuto)),A)";
     "e(tell(bob,alice,accept(aiuto)),B)";
  }
}
```

dove `[subgraph cluster_`*`nomeSottografo`*`]` dichiara che tutto ciò che compare all'interno delle graffe successive viene racchiuso all'interno di sottografo bordato, come mostrato in figura 3.4.

Verrà trattato ora come il linguaggio DOT permetta di creare grafi gerarchici. La dichiarazione `rankdir=TB|RL|BT`, da inserire all'inizio del documento, in quanto descrive un'impostazione globale, determina la direzione verso cui il grafo si evolve; dove *TB* sta per TOP-to-BOTTOM, *RL* per RIGHT-to-LEFT e *BT* per BOTTOM-to-TOP.

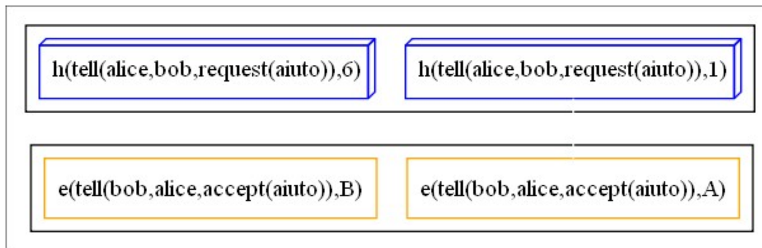

Figura 3.4: Esempio di sottografi

Nel caso degli output di SCIFF, si è scelto di ordinare i nodi dall'alto verso il basso, quindi all'inizio del documento DOT si inserisce la dichiarazione `rankdir=TB`. In realtà questo non è sufficiente per creare di fatto un grafo gerarchico; infatti, affinché i nodi vengano disposti su livelli man mano più bassi, è necessario che essi siano tutti uniti in successione da archi. L'utente SCIFF non ha alcun interesse di vedere gli archi funzionali al corretto posizionamento verticale dei nodi, inoltre questi potrebbero impedire la giusta interpretazione del grafo. Colorando gli archi di bianco, con l'opzione `[color=white]` da aggiungere come impostazione delle relazioni funzionali alla dispozione per livelli, si evita quindi che tali archi vengano visualizzati. Il documento DOT perciò diventa:

```
graph grafoSCIFF{
  rankdir=TB;
  node[shape=box3d, color=blue];
  subgraph cluster_h{
     "h(tell(alice,bob,request(aiuto)),1)" --
     "h(tell(alice,bob,request(aiuto)),6)" [color=white];
  }
  node[shape=box, color=orange];
  subgraph cluster_e{
     "e(tell(bob,alice,accept(aiuto)),A)" --
     "e(tell(bob,alice,accept(aiuto)),B)" [color=white];
  }
}
```

E tradotto graficamente appare così:

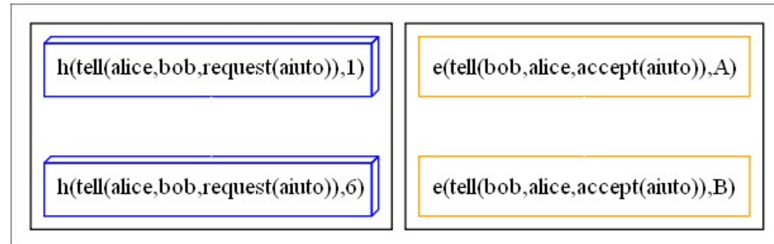

Figura 3.5: Esempio di sottografi

Per migliorare ulteriormente la leggibilità di un grafo si possono aggiungere all'inizio del documento DOT le seguenti opzioni: la distanza tra i diversi livelli del grafo (che è utile aumentare di poche unità nel caso in cui vi siano numerosi archi) attraverso l'opzione `ranksep=0..n`; e l'impostazione necessaria per far in modo che gli archi non oltrepassino i confini dei nodi e che questi ultimi non si sovrappongano: `graph[splines=true, overlap=false];`.

Il pacchetto Graphviz [10] –acronimo di Graphic Visualization Software– permette di generare grafi a partire da documenti DOT, ed è il software che si è deciso di utilizzare in questa sede per sviluppare grafi a partire dagli output dei progetti SCIFF lanciati su SICStus Prolog. Graphviz è composto da cinque applicazioni che producono diversi tipi di grafi a seconda delle esigenze dell'utente:

- **dot** – genera grafi gerarchici (a livelli) seguendo la direzione prestabilita;
- **neato e fdp** – generano grafi che si evolvono in tutto lo spazio senza una determinata direzione;
- **twopi** – genera grafi che si evolvono radialmente posizionando i nodi su cerchi concentrici, con distanze dal centro progressivamente maggiori;

- **circo** – adatto per generare grafici contenenti strutture cicliche con alte complessità come i sistemi di telecomunicazione.

Dopo aver installato correttamente il pacchetto Graphviz lo si può utilizzare da console attraverso il seguente comando:

`(dot\neato\fdp\circle\twopi) -Testensione -onome_immagine.estensione file_sorgente_DOT`

dove per estensione si intende uno dei formati grafici supportati da Graphviz, tra i quali alcuni di nostro interesse come ps, png, svg e gif.

Il visualizzatore grafico degli ouput di SCIFF, che come già affermato dovrà rappresentare gli atomi all'interno di grafo, avrà perciò bisogno di compiere una system call utilizzando il suddetto comando, specializzato per l'applicazione *dot*. Infatti, tra tutte, *dot* è la sola che può generare grafi di tipo gerarchico, ed è con questa che sono state create le immagini in figura 3.1, 3.2, 3.3, 3.4 e 3.5, nonché quelle del paragrafo 2.2.

3.2 Il parser

Il primo passo fondamentale è quello di implementare un parser, che avrebbe il compito di riconoscere e scindere gli atomi dai vincoli, cercare poi al loro interno le variabili, ed infine memorizzare tali informazioni su apposite strutture. Oltre agli atomi abducibili attualmente definiti in SCIFF ed ai vincoli predefiniti di SICStus, può essere utile prevedere estensioni del linguaggio. Inoltre, per l'utente SCIFF è utile poter decidere quali atomi e quali vincoli visualizzare nel grafo (alcuni atomi e vincoli potrebbero essere irrilevanti); per fare ciò abbiamo previsto due file di configurazione dove si dichiarano in uno i funtori degli atomi e nell'altro quelli dei vincoli che si vuole tenere in considerazione. Il parser ha quindi anche il compito di confrontare ogni funtore letto dagli output, che sia esso di un atomo o di un vincolo, con

quelli presenti nei file di configurazione, in modo da memorizzare solamente le strutture per cui esiste una corrispondenza.

Con l'usilio del linguaggio Java si può ora introdurre l'unità costituiva del visualizzatore: la classe *Item*, che rappresenta un atomo oppure un vincolo. Essa è definita come segue:

```
public class Item{
  public String stringa;
  public String funtore;
  public boolean tipo;
  public String[] variabili;
  public String ultima_posizione;
    public Item(String stringa, String[] ListaAtomi, String[] ListaVincoli){
              /*codice costruttore*/
    }
}
```

Al costruttore della classe *Item* viene passata una stringa appartenente agli output di SCIFF, la quale viene subito memorizzata nel campo *funtore*; vengono inoltre copiati gli array di stringhe contenenti i funtori degli atomi e dei vincoli presenti nei file di configuarazione ed ottenuti in seguito ad una apposita lettura. Il costruttore della classe in questione implementa le funzionalità del parser appena descritto, provvederà quindi a riconoscere se la stringa passata è un atomo o un vincolo, impostando nel campo *tipo* il valore *true* nel caso l'oggetto rappresenti un atomo, oppure *false* se l'oggetto è un vincolo.

Per quanto riguarda il riconoscimento dei vincoli va specificato che all'interno degli output SCIFF si possono riscontrare due tipologie di vincolo che hanno diversa struttura sintattica. Un vincolo può avere un operatore infisso, secondo la seguente struttura "*VAR1 operatore VAR2*" ed il suo riconoscimento è cablato con procedure definite all'interno del costruttore. Invece vincoli che presentano sintassi prefissa, del tipo "*funtore(parametri)*",

devono essere dichiarati all'interno del file di configurazione, che si è scelto di chiamare "constraintsConfig".

Il risultato dell'esecuzione del costruttore della classe *Item* può dar luogo ai seguenti risultati:

- nessuna corrispondenza: l'istanza della classe non viene memorizzata.
- Corrispondenza con un atomo, si memorizza il nome del funtore ed il tipo di elemento nelle apposite stringhe della classe *Item*; in particolare il parametro *tipo* in questo caso ha valore "*true*".
- Corrispondenza con un vincolo, si imposta semplicemente *tipo* a "*false*".

Il passo successivo che compie il costruttore (il parser) è quello di cercare all'interno della stringa le variabili, e memorizzarle (senza ripetizioni) all'interno dell'array *variabili*. Le possibilità sono perciò due: *variabili* è vuoto, oppure contiene almeno una variabile. Nel caso l'oggetto rappresenti un atomo, il costruttore continuerà la sua esecuzione andando a memorizzare nella stringa *ultima_posizione* il valore dell'ultimo parametro, in modo da permettere ad appositi metodi di ordinare le istanze della classe *Item* secondo l'andamento cronologico.

A questo punto, avendo ripetuto l'operazione per tutte le stringhe degli output, e memorizzato in un array di *Item* le istanze che rispondono ai requisiti descritti in precedenza, si possono avere una o più delle seguenti possibilità:

- l'array è vuoto: nessuna stringa è stata riconosciuta come atomo o vincolo;
- sono presenti degli atomi con tempo costante;

- sono presenti degli atomi con tempo variabile;
- sono presenti dei vincoli.

3.3 Ricerca delle relazioni tra atomi

Il secondo passo è quello di cercare le diverse relazioni che possono intercorrere tra diversi atomi. Due atomi sono collegati da un vincolo se condividono una variabile, oppure se esiste un vincolo che collega una variabile del primo atomo con una del secondo. Chiameremo il primo caso *relazione diretta* ed il secondo *relazione indiretta*. Si può notare che due atomi possono essere contemporaneamente in relazione diretta e indiretta.

Scopo di questo paragrafo è illustrare i metodi che si devono implementare per riconoscere se due atomi siano in relazione tra loro.

L'array di Item viene inizialmente diviso in due array, uno di vincoli ed uno di atomi, in modo da avere due strutture specializzate e pronte per essere passate ai metodi incaricati di trovare al loro interno le varie relazioni.

Le relazioni vengono rappresentate dalla seguente classe:

```
public class Relation{
   Item atomo1;
   Item atomo2;
   String[] variabiliCondivise;
   Item[] vincoli;

     public Relation(Item atomo1, Item atomo2, String[] variabiliCondivise){
           /*codice costruttore*/
     }
     public Relation(Item atomo1, Item atomo2, Item[] vincoli){
           /*codice costruttore*/
     }
}
```

Per le relazioni dirette è sufficiente creare delle istanze della suddetta classe dove si memorizzano le variabili condivise nell'array di stringhe *variabiliCondivise*, utilizzando il primo costruttore; invece per le relazioni indirette si memorizzano i vincoli per i quali *atomo1* e *atomo2* sono in relazione indiretta, attraverso il secondo costruttore.

Intuitivamente la ricerca delle relazioni dirette è molto semplice: è necessario solamente comparare le variabili di tutte le istanze di *Item* dell'array specializzato per gli atomi, prese a due a due, ed escludendo le permutazioni che prendono in considerazione due elementi identici. Il risultato sarà un array di *Relation* che può essere vuoto (non vi sono atomi in relazione diretta) o non vuoto.

Cercare le relazioni indirette risulta invece più laborioso. Si verifica innanzitutto che l'array *Item* dei vincoli presenti almeno un elemento e che quello degli atomi ne presenti almeno due, in caso contrario non vi sono sicuramente relazioni di tipo indiretto. Si può procedere verificando per ogni atomo e per ogni vincolo che vi siano variabili in comune, memorizzando le corrispondenze trovate nelle istanze della classe definita come segue:

```
public class SingleRelation{
  public Item atomo;
  public Item vincolo;
  public String[] variabili_condivise;
     public Relation(Item atomo, Item vincolo, String[] variabili_condivise){
            /*codice costruttore*/
     }
}
```

Il metodo che ha il compito di costruire l'array di *SingleRelation* prende perciò i due array di atomi e vincoli, li scandisce comparando le variabili di ogni *Item* atomo con quelle di ogni *Item* vincolo, e memorizza tutte le corrispondenze. L'array di *SingleRelation* che tale metodo costruisce può:

- essere *null*, cioè non vi è alcuna relazione indiretta;
- contenere una sola istanza di *SingleRelation*, anche in questo caso non vi sono sicuramente relazioni indirette;
- contenere almeno due *SingleRelation*.

In generale si può affermare che nell'array di *SingleRelation* vi sono due istanze contenenti atomi relazionati in modo indiretto se e solo se almeno due di queste condividono lo stesso vincolo. Successivamente perciò il programma verifica, attraverso un apposito metodo, che gli elementi dell'array presi a due a due in tutte le diverse permutazioni, abbiano in comune lo stesso vincolo, memorizzando le corrispondenze dentro istanze di *Relation*, intese come istanze che rappresentano relazioni indirette. Il metodo restituisce un array che può essere null (se non vi sono atomi relazionati in modo indiretto), oppure può contenere uno o più elementi.

Per evitare che sul file DOT vengano stampate più volte le stesse relazioni, dirette e indirette, si può ricorrere ad un metodo che elimina le rindondanze all'interno dei due array *Relation*.

3.4 Creazione del file DOT

Una volta terminata la fase di ricerca delle relazioni non rimane che salvarne i risultati sul file DOT. Il file DOT si costituirà fondamentalmente di quattro parti:

- dichiarazioni delle impostazioni grafiche;
- sottografi contenenti atomi (nodi) ordinati cronologicamente su più livelli;

- relazioni dirette (se esistono);
- relazioni indirette (se esistono).

Seguendo le istruzioni del paragrafo 3.1 si può stilare un documento DOT che funge da contenitore ottimale per gli output di SCIFF, in modo che essi possano essere visualizzati attraverso un grafo facilmente leggibile:

```
graph contenitore{
  graph [splines=true overlap=false];
  rankdir=TB;
  ranksep=0..n;
  node [fontsize=size];

    subgraph cluster_funtoreAtomo1{
    node[shape=formaAtomo1, color=coloreAtomo1];
        ...
    }
        ...

    subgraph cluster_funtoreAtomoN{
    node[shape=formaAtomoN, color=coloreAtomoN];
        ...
    }

    /*RELAZIONI DIRETTE*/
        ...

    /*RELAZIONI INDIRETTE*/
        ...
}
```

I parametri: `ranksep`, `fontsize`, `formaAtomo` e `coloreAtomo` vengono decisi direttamente dall'utente tramite un file di configurazione, chiamato "DOTConfig". In tale file, si aggiungono i comandi:

SHAPE_*funtore*=*forma*;

COLOR_*funtore*=*colore*;

che dichiarano forma e colore dei nodi associati ai vari atomi.

Come accennato nel paragrafo 1.2, per migliorare la leggibilità del grafo si ordinano gli atomi in senso crescente e li si riunisce in sottografi. È stata quindi definita una classe che ha al proprio interno atomi con lo stesso funtore, divisi a seconda che abbiano tempo costante oppure variabile, ed in particolare quelli appartenenti alla prima categoria, nelle istanze di tale classe, sono ordinati in senso cronologico. La classe in questione è definita come segue:

```
public class SortAtoms {
  public Item[] atomiVariabili;
  public Item[] atomiCostanti;
  public String funtore;

     public SortAtoms(Item[] atomo){
      /*codice costruttore*/
     }
}
```

Dove in *atomiVariabili* sono riuniti tutti gli atomi che hanno funtore uguale e che sono di tipo variabile; ed in *atomiCostanti*, invece, sono contenuti quelli con tempo costante, ordinati in senso cronologico. Sarà il costruttore di *SortAtoms*, al quale viene passato come parametro un array di atomi con lo stesso funtore, che si occuperà di dividere tale array secondo le caratteristiche degli atomi, ed in particolare di ordinare in senso crescente l'array relativo agli atomi con tempo costante.

Effettuata questa operazione per tutti gli atomi dell'array *Item*, si avrà perciò un nuovo array composto da istanze di *SortAtoms*, pronto per essere letto agevolmente dal metodo che stamperà il suo contenuto sul file DOT.

Una volta a disposizione l'array di *SortAtoms* e quelli di *Relation*, non serve altro che compiere in primis un ciclo di lettura su *SortAtoms*, stampando via via ogni elemento (separato da una relazione "bianca" da quello

successivo come indicato nel paragrafo 3.1) all'interno del rispettivo sottografo. In seguito si stamperanno tutte le relazioni identificate dagli elementi degli array di *Relation* (in caso ve ne siano), procedendo alla stampa prima dell'array delle relazioni dirette, poi a quello delle relazioni indirette.

E' interessante mostrare che il linguaggio DOT permette di inserire all'interno delle etichette dei vincoli la combinazione di caratteri "\n", e di avere quindi etichette distribuite su piu' righe. Sfruttando questa funzionalità si può quindi inserire nelle etichette degli archi un "\n" tra i diversi vincoli di una stessa *Relation* di tipo indiretto, e tra le diverse variabili di una *Relation* di tipo diretto, nel caso che queste contengano rispettivamente più di un vincolo o più di una variabile.

Con l'ausilio della classe *BufferedWriter* si può dunque simulare la stampa di una singola *Relation* che può contenere più vincoli:

```
...
 Relation rel=new DoubleRelation(...);
 BufferedWriter file=new BufferedWriter(...);
   ...
   file.write(rel.atomo1+" -- "+rel.atomo1+" [label=");
    for(int i=0; i<rel.vincoli.lenght; i++)
     (i==(rel.vincoli.lenght-1))?file.write(rel.vincoli[i]) : file.write(DR.vincoli[i]+"\n");
   file.write("];");
...
```

Inoltre si può implementare un eventuale altro metodo che stampa gli atomi e le relazioni su grafi semplici, non gerarchici, senza utilizzare perciò il modello a sottografi e l'ordinamento cronologico.

Non resta perciò che creare il grafo a *runtime* attraverso una system call, utilizzando il comando che serve per generare i grafici partendo da un file sorgente di tipo DOT, come descritto nel paragrafo 3.1. Java mette a disposizione il metodo:

```
Runtime.getRuntime().exec(comando)
```

con il quale si incarica il sistema operativo di eseguire il comando inserito all'interno del metodo *exec*. Nel paragrafo 3.2 verrà illustrato l'iter necessario per passare i corretti parametri al metodo precedente, per la generazione del grafo attraverso l'applicazione *dot*.

3.5 Un esempio

La seguente successione di stringhe:

```
exists(A),
exists(B),
exists(C),
h(tell(alice,bob,request(aiuto)), 1),
h(tell(alice,eva,request(aiuto)), 6),
e(tell(bob,alice,accept(aiuto)),A),
e(tell(eva,alice,accept(aiuto)),B),
en(tell(eva,alice,accept(aiuto)),C),
A > 1,
B > 6,
C < B
```

può rappresentare un output "tipo" di SCIFF. Con l'ausilio di un'applicazione che implementa le specifiche descritte nei paragrafi 3.2 e 3.3 si ottiene un file DOT così composto:

```
graph esempio{
  graph [splines=true overlap=false];
  rankdir=TB;
  ranksep=0;
  node [fontsize=10];

    subgraph cluster_h{
    node[shape=box3d, color=blue];
        "h(tell(alice,bob,request(aiuto)), 1)" --
        "h(tell(alice,eva,request(aiuto)), 6)" [color=white];
    }
    subgraph cluster_e{
```

```
    node[shape=box, color=orange];
        "e(tell(bob,alice,accept(aiuto)),A)" --
        "e(tell(eva,alice,accept(aiuto)),B)" [color=white];

    }

    subgraph cluster_en{
    node[shape=box, color=red];
        "en(tell(bob,alice,accept(aiuto)),C)";

    }

    /*RELAZIONI DIRETTE*/

    /*RELAZIONI INDIRETTE*/
        "e(tell(eva,alice,accept(aiuto)),B)" --
        "en(tell(bob,alice,accept(aiuto)),C)" [label="C < B"];
}
```

E con l'applicazione *dot* si ottiene l'immagine in figura 3.6.

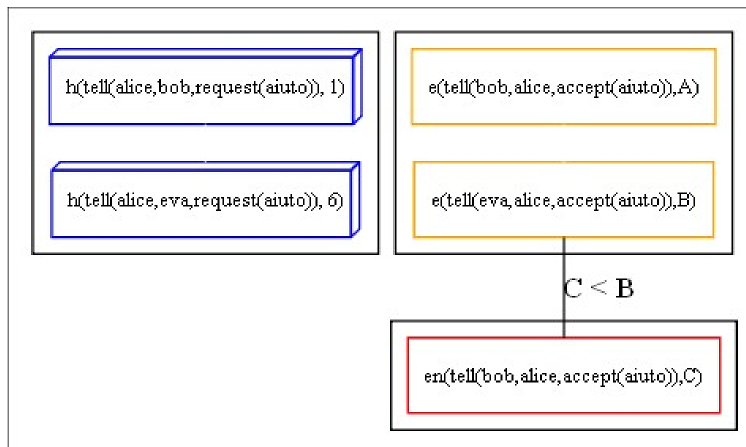

Figura 3.6: Esempio di grafo SCIFF

Si noti come la history sia in un riquadro a parte, diverso da quelli delle aspettative. Inoltre, gli eventi sono visualizzati in ordine cronologico

Capitolo 4

Integrazione del visualizzatore nell'ambiente Eclipse

4.1 Prelievo degli output dalla console di Eclipse

Nel paragrafo 2.2 è stata descritta la struttura di Eclipse, mettendo l'accento sul plug-in *SCIFF* implementato appositamente per poter utilizzare il linguaggio SCIFF nell'ambiente integrato di Eclipse, così da avere le funzionalità computazionali e grafiche consone per una comoda programmazione.

Volendo estendere il plug-in *SCIFF* con le funzionalità del visualizzatore grafico, si è utilizzato il PDE per creare da esso un progetto, attraverso l'"Import Wizard" di Eclipse. All'interno del project *SCIFF* così ottenuto sono stati aggiunti i package relativi al visualizzatore costruito seguendo le specifiche del capitolo 3. In questo capitolo invece verranno discusse le modalità per integrare il visualizzatore all'interno del plug-in *SCIFF*, mostrando cioè come dovrebbero essere implementati quei moduli necessari per connet-

tere i package del visualizzatore alla piattaforma di Eclipse ed alla console di *SCIFF*.

Il visualizzatore grafico, che d'ora in poi chiameremo col nome "*SCIFFgraph*", prende in ingresso un array di stringhe, una per ogni riga presente negli output di SISCtus, il parser perciò analizza ognuna di queste stringhe e ne memorizza le informazioni. Indi si vorrebbe prelevare tali output SCIFF dalla console di Eclipse e disporli all'interno dell'array da mettere a disposizione del visualizzatore.

Il package "`org.eclipse.ui`" [11], dove "*ui*" sta per *user interface*, mette a dispozione numerosi altri package contenenti classi ed interfacce utili per interagire con tutti gli elementi della finestra di Eclipse. In particolare all'interno di `org.eclipse.ui.console` vi sono le classi necessarie per operare sulle varie console aperte nella finestra, per aprirne di nuove e per modificarne le impostazioni. Una di queste classi è *ConsolePlugin* che permette, attraverso appositi metodi, di ottenere un array di istanze di una classe che implementa l'interfaccia *IConsole*, e che mette a disposizione i metodi necessari per interagire con la struttura delle console aperte sulla finestra di Eclipse. Le console di Eclipse sono etichettate con un nome assegnato loro dal processo che le ha istanziate.

Il plug-in *SCIFF*, in seguito all'avvio di SICStus Prolog, apre due diverse console: la "*SCIFF console*", dove vengono reindirizzati gli stream generati dall'esecuzione di SICStus, e la "*SCIFF errors*" dove vengono riportati i messaggi di errore riscontrati all'interno della console di *SCIFF*. È necessario cercare tra tutte le console aperte, ottenute dal metodo:
`ConsolePlugin.getDefault().getConsoleManager().getConsoles()`
quella che ha come etichetta la stringa "*SCIFF console*", ottenuta dal metodo *IConsole.getName()*, e visualizzando un una finestra di errore che informa

l'utente nel caso non sia stata trovata alcuna console corrispondente. Tale eventualità è dovuta al fatto che non è stato avviato SICStus Prolog, oppure che la console di SCIFF è stata intenzionalmente chiusa dall'utente.

Le interfacce *IConsole* possono essere associate attraverso un cast alla classe *TextConsole*, che rappresenta una console di tipo testuale. Attraverso il metodo ***getDocument()***, *TextConsole* restituisce un'istanza di una classe che implementa l'interfaccia *IDocument*, la quale funge da contenitore di documenti testuali e da cui si può ottenere, con l'apposito metodo ***get()***, una stringa contenente il testo di tutto il documento. Per quanto riguarda la stringa ottenuta dalla *TextConsole* è necessario tenere conto del fatto che, mentre nella console le righe vengono riportate una sotto l'altra, all'interno della stringa restituita da *IDocument* tali righe sono identificate da sottostringhe separate tra loro da un carattere speciale (che indica la fine della riga della console) e da "\n". Per ottenere un array di stringhe contenente tutte le voci presenti all'interno della console, è necessario quindi separare tutte le suddette sottostringhe.

Di seguito viene mostrato il corretto utilizzo delle classi e delle interfacce sopra indicate:

```
import org.eclipse.ui.console.IConsole;
import org.eclipse.ui.console.ConsolePlugin;
import org.eclipse.ui.console.TextConsole;
...
String output;
IConsole[] consoles=ConsolePlugin.getDefault().getConsoleManager().getConsoles();
//ricerca di "SCIFF console"
...
TextConsole tc=(TextConsole)consoles[i];
output=tc.getDocument().get();
...
```

Ciò che serve al visualizzatore SCIFFgraph non è tutto il documento della console, bensì solo la parte di output inerente i risultati del lancio di

un progetto SCIFF, effettuato con il comando inviato dal tasto della toolbar di Eclipse chiamato "run".

A questo punto sarebbe molto importante riuscire a riconoscere ed estrapolare da tutta la console solo gli output dell'ultimo progetto SCIFF eseguito, così da passare al visualizzatore le stringhe di nostro interesse. Non avendo di default alcuna indicazione utile che permettesse di distuinguere a runtime gli output dei progetti SCIFF tra tutti quelli generati da SICStus, si è chiesto agli sviluppatori del linguaggio SCIFF di far in modo che i risultati dei progetti venissero sempre preceduti da una speciale successione di caratteri, individuata nella concatenazione di più asterischi (ad es. "*****"). Grazie a questo delimitatore si può perciò estrapolare dall'array di output, ottenuti precedentemente dalla console, l'ultimo blocco di output inerenti il lancio del progetto SCIFF, isolato attraverso la ricerca dell'ultima ricorrenza della successione di asterischi all'interno dell'array. Va da sè che, in caso non sia presente almeno un delimitatore, si deve informare l'utente che all'interno della console non sono stati trovati output validi per l'esecuzione del visualizzatore. Si dovrebbe procedere in questo caso col compilare e lanciare un progetto SCIFF.

A questo punto si ha a disposizione un array di stringhe contenenti le dichiarazioni delle variabili, gli atomi ed i vincoli risultanti dall'analisi dell'ultimo progetto lanciato su SICStus.

4.2 Creazione degli strumenti grafici necessari per l'interazione con l'utente

Per fornire all'utente SCIFF gli strumenti grafici necessari per l'utilizzo di SCIFFgraph è necessario che il plug-in dichiari innanzitutto all'interno del

file *plugin.xml* gli *extension* che si vogliono implementare. Volendo estendere le funzionalità del plug-in *SCIFF* con l'aggiunta di *SCIFFgraph*, si andrà perciò ad aggiungere tali dichiarazioni nel file *plugin.xml* all'interno del project *SCIFF*.

Si vogliono mettere a disposizione due menu, uno sulla toolbar ed uno sulla barra dei menu di Eclipse, entrambi contenenti la stessa collezione di comandi: uno per l'avvio del visualizzatore attraverso una finestra di dialogo, uno per l'apertura del file di configurazione dei documenti DOT ed uno per l'apertura del file di configurazione dove aggiungere i funtori dei vincoli.

Come affermato nel paragrafo 2.3, Eclipse permette di dichiarare nel file *plugin.xml* le action necessarie per rendere visibile all'utente il plug-in, attraverso icone e menu, nonostante esso non sia stato ancora avviato. Le action vanno inserite sotto forma di extension (quindi dentro i marcatori "<extension></extension>") come segue:

```
<extension>
...
<action
   class="sciffgraph.OpenConfigurationFile"
   disabledIcon="icons/conf_file.png"
   hoverIcon="icons/conf_file.png"
   icon="icons/conf_file.png"
   id="SCIFFgraph.CreateGraph_PULLDOWN.OpenConfigurationFile"
   label="Open DOTConfiguration File"
   menubarPath="SCIFFgraph.Graph/SCIFFgraph.GraphMarker"
   style="push"
   tooltip="Open Configuration File">
</action>
...
</extension>
```

Nel campo *class* si inserisce la classe che deve essere chiamata per compiere l'azione rappresentata dalla relativa icona o dalla voce del menu. Tali classi devono implementare delle specifiche interfacce; per una sempli-

ce action l'interfaccia è *IWorkbenchWindowActionDelegate*, appartenente al package `org.eclipse.ui`, dove il metodo più importante da definire è `public void run(IAction)` all'interno del quale devono essere inserite le istruzioni relative alla action in questione. Se invece la action è "polivalente", come nel caso delle icone della toolbar che possono lanciare una normale action ed allo stesso tempo aprire un sottomenu, l'interfaccia da implementare è *IWorkbenchWindowPulldownDelegate*. In questo caso si ridefinisce il metodo *run()* come per l'action semplice, che deve eseguire l'azione collegata all'icona principale del menu, ed il metodo `public Menu getMenu(Control)` per generare il menu con tutti gli elementi di cui si deve disporre. All'interno di quest'ultimo, per ogni elemento che si vuole inserire, è necessario istanziare una classe *MenuItem* appartenente al package `org.eclipse.swt.widgets`; ad ognuno viene assegnato un nome interno, che vale come identificativo, e che il gestore degli eventi usa per determinare quale elemento è stato selezionato, eseguendo di conseguenza le relative azioni.

Il codice che segue mostra come viene istanziata la classe dell'elemento di un menu, e come deve essere gestito l'evento lanciato dalla selezione di tale elemento da parte dell'utente.

```
elemento1=new MenuItem(istanzaMenu,SWT.PUSH);
  elemento1.setText("elemento1");
    elemento1.addSelectionListener(new SelectionAdapter(){
      public void widgetSelected(SelectionEvent e)
      {
        if (e.getSource().equals("elemento1"))
        {
              /*codice*/
        }
  }
});
```

Il metodo *addSelectionListener()* aggiunge alla classe il gestore dell'evento

inserito come parametro. In questo caso la classe che gestisce l'evento è *SelectionAdapter* che implementa l'interfaccia ***SelectionListener***, entrambe definite nel package `org.eclipse.swt.events`; al metodo ***widgetSelected()***, infine, viene passata l'istanza della classe che rappresenta l'evento, in questo caso *SelectionEvent*.

Una volta create tutte le icone ed i menu con relativi elementi (figura 4.1 e 4.2) non resta che costruire la finestra di dialogo necessaria per l'esecuzione di SCIFFgraph. In questa finestra l'utente dovrà scegliere il project SCIFF di cui si vogliono visualizzare gli output sotto forma di grafi, scelti tra quelli presenti nell'Eclipse Navigator; la cartella dove si vogliono realizzare i documenti DOT ed i grafi, ed il formato dei file immagine attraverso cui visualizzare tali grafi. Inoltre l'utente può scegliere tra i due tipi di visualizzazione grafica (paragrafo 3.2) messi a disposizione.

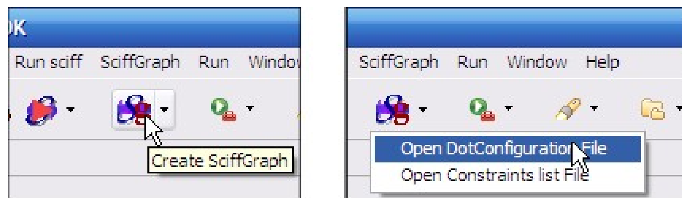

Figura 4.1: SCIFFgraph nella toolbar

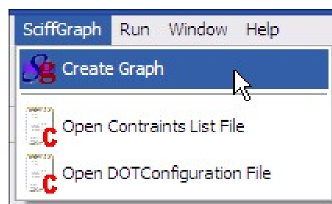

Figura 4.2: SCIFFgraph nella barra dei menu

La classe con la quale viene costruita la finestra di dialogo deve estendere la classe *Dialog*, contenuta nel package `org.eclipse.jface.dialogs`.

All'interno di questa classe, denominata *CreateGraph_Dialog*, si inseriranno gli elementi grafici individuati dalle classi Label, Text e Button contenuti nel package `org.eclipse.swt.widgets`. Per "*widgets*" si intende l'insieme degli oggetti grafici pronti per essere inseriti all'interno delle finestre, definiti dallo Standard Widget Toolkit. In ordine verranno inseriti:

- un menu costituito da tutti i project che hanno nature "SCIFF project", ottenuti attraverso la classe `ResourcePlugin` del package `org.eclipse.core.resources`, utilizzando il metodo:
  `ResourcesPlugin.getWorkspace().getRoot().getProjects()`
  che permette di navigare all'interno del Workspace, selezionarne la radice (cioè la directory dove risiedono i project aperti) ed avere infine un array di interfacce del tipo *`IProject`*, definita anch'essa nel suddetto package. Confrontando le nature con quella dei project di tipo SCIFF si forma perciò il menu di elementi di nostro interesse inserendo quelli per cui vi è una corrispondenza di nature.

- Una barra di testo, data dalla classe *Text*, dove inserire il percorso della cartella che ospiterà i file in uscita.

- Un bottone (*Button*), con etichetta "*browse..*", che apre una finestra di dialogo standard per la selezione delle cartelle all'interno del file system e che aggiorna di conseguenza il campo *Text*. Di default questo elemento ed il precedente debbono restare disattivati finché non viene selezionato un progetto dall'apposito menu; a seguito della selezione di un project il campo *Text* viene aggiornato con il path assoluto della cartella che lo contiene.

- Un bottone di tipo radio per ogni formato immagine: ps, svg, png, gif. Tale specializzazione di *Button* avviene all'interno del suo costruttore:

`new Button(composite, SWT.RADIO)`; dove `SWT.RADIO` è una costante definita dal package SWT che assegna all'istanza della classe *Button* le funzionalità tipiche del radio.

- I bottoni di tipo radio che permettono di scegliere il tipo di grafo: semplice, ordinato, oppure entrambi i tipi.
- I bottoni "Start" e "Cancel".

La finestra di dialogo deve essere implementata in modo che il bottone *Start* non sia attivo finché non viene selezionato un progetto dall'apposito menu. Con la pressione del tasto *Start* si accede al metodo ***`protected void okPressed()`*** messo a disposizione dalla classe *Dialog*, che deve essere ridefinito in modo tale da passare alla classe principale del visualizzatore tutti i parametri necessari per la corretta computazione:

- l'array di stringhe ottenute dalla lettura della *SCIFF console*;
- il percorso di output inserito nel campo *Text* (che può essere stato definito dall'utente oppure inserito automaticamente in base alla selezione del progetto);
- un array di stringhe contenenti i formati grafici selezionati;
- un intero che rappresenta il tipo di grafo selezionato (o entrambi).

La finestra di dialogo che risponde a tali requisiti è mostrata in figura 4.3.

Compilato il nuovo plug-in *SCIFF* con all'interno le funzionalità di SCIFFgraph, se ne ha perciò una nuova versione, denominata 1.2.

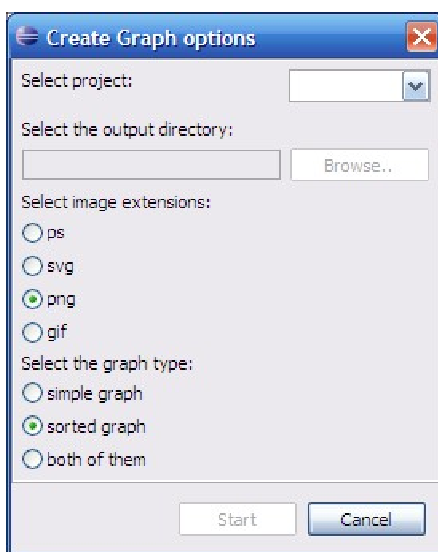

Figura 4.3: Finestra di dialogo per la gestione di SCIFFgraph

Capitolo 5

Installazione ed utilizzo di SCIFFgraph

5.1 Installazione dei pacchetti necessari per il corretto funzionamento

La procedura SCIFF è implementata in Sicstus Prolog, reperibile all'indirizzo `http://www.sics.se/isl/sicstuswww/site/index.html`, dove può trovare le istruzioni per il download e l'installazione del suddetto. Il passo successivo consiste nell'effettuare il download dei file del linguaggio SCIFF, da far eseguire a SICStus Prolog, reperibili all'indirizzo: `http://lia.deis.unibo.it/research/sciff/`. Si prosegue con l'installazione dell'ambiente Eclipse, scaricabile gratuitamente dal sito `http://www.eclipse.org/downloads/`.

Una volta che si hanno a dispozione le applicazioni sopra indicate, non resta che integrare su Eclipse il plug-in *SCIFF* di nuova versione; lo si trova alla seguente pagina web: `http://lia.deis.unibo.it/research/sciff/`. Si copia la directory così ottenuta, denominata "*SCIFF 1.2*", all'interno della

sottodirectory di Eclipse ‘‘plugins’’; nel caso si disponesse già della versione precedente di *SCIFF* è sufficiente la sostituzione delle due directory.

È necessario ora inserire nel pannello delle preferenze di Eclipse, più precisamente nella scheda relativa al plug-in ‘‘SCIFF”, le impostazioni necessarie al corretto funzionamento del plug-in, aggiungendo cioè i path delle applicazioni SICStus Prolog, SCIFF e SOCS–SI [12].

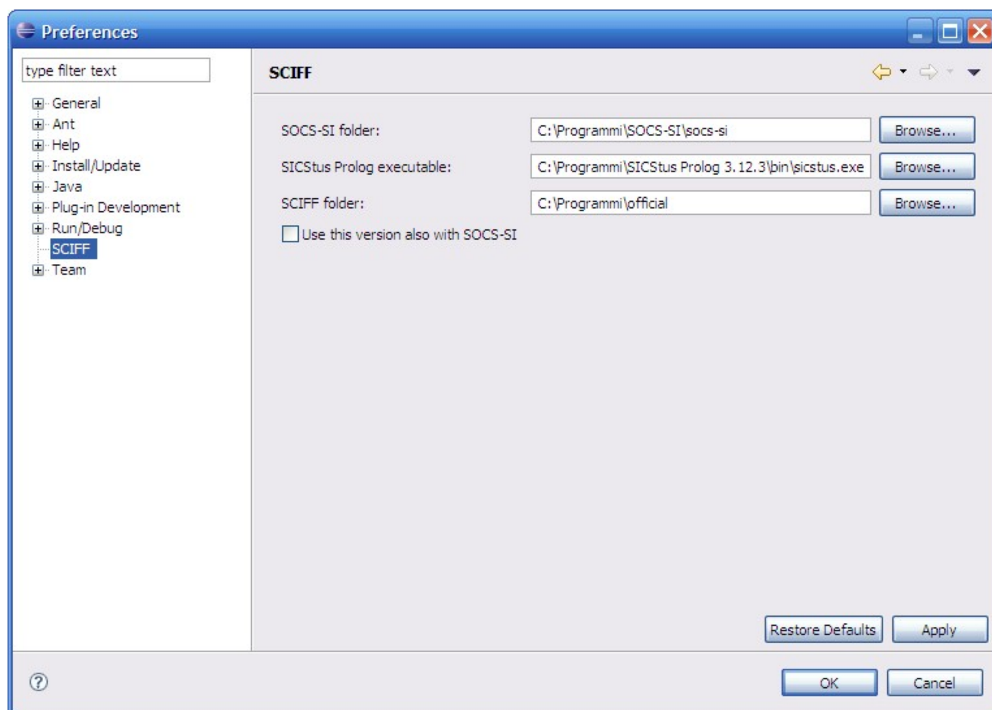

Figura 5.1: Finestra delle preferenze di *SCIFF*

Per ultimo, alla pagina `http://graphviz.org/Download.php`, si può ottenere l’ultima versione del pacchetto Graphviz da installare sul sistema, in modo da poter generare grafi a partire dai documenti DOT. Affinché il plug-in SCIFF funzioni correttamente è necessario che la cartella contenente le applicazioni del pacchetto di Graphviz sia stata dichiarata nel PATH.

5.2 Come utilizzare SCIFFgraph

Una volta effettuate le operazioni indicate nel paragrafo precedente, nella finestra di Eclipse si hanno le icone ed i menu relativi all'utilizzo del linguaggio SCIFF, come il lancio di SICStus Prolog, la compilazione e l'esecuzione dei project SCIFF.

In primis è necessario avviare SICStus con il comando "Start SCIFF"; avviene di conseguenza l'apertura della *SCIFF console*, sulla quale vengono stampati gli stream relativi all'esecuzione.

Con il wizard "*import*" si possono caricare sul Navigator i progetti di tipo SCIFF, scelti tra quelli presenti nel filesystem. Una volta che questi sono a disposizione, si può perciò compilare quello di nostro interesse con l'ausilio del comando "compile"; se la compilazione è andata a buon fine (informazione data dalla stampa su console della scritta "yes"), si può eseguire il project con il comando "run". Di conseguenza sulla console viene stampato il delimitatore composto da asterischi e, a seguire, i risultati del programma SCIFF. In figura 5.2 si può vedere la console di SCIFF in seguito al lancio di un progetto.

Si può ora accedere alla finestra di dialogo di SCIFFgraph attraverso il relativo comando del menu o bottone della toolbar *Create SCIFFgraph*, e scegliere in ordine:

- il progetto SCIFF che si vuole visualizzare attraverso grafi, tra tutti quelli presenti nel Navigator. La scelta del progetto, come anticipato nel paragrafo 4.2, si ripercuote sulla barra di testo sottostante, che viene aggiornata con il percorso assoluto della directory del progetto (sita all'interno del workspace), con l'aggiunta di una ulteriore nuova cartella, chiamata "*Graphs*", dentro la quale verranno riuniti tutti i file

```
e([openauction,4],openauction(_K,_L,openauction(_J,_M,_N),_I),_O),
e([bid,4],bid(_L,_K,bid(_J,_H),_I),_Q),
h([bid,4],bid(_L,_K,bid(_J,_H),_I),_Q),
e([openauction,4],openauction(_K,_L,openauction(_J,_M,_N),_I),_O),
h([openauction,4],openauction(_K,_L,openauction(_J,_M,_N),_I),_O),
e([answer,4],answer(_K,_L,answer(win,_J,_H),_I),_W1),
h([answer,4],answer(_K,_L,answer(win,_J,_H),_I),_W1),
en([answer,4],answer(_K,_L,answer(lose,_J,_H),_I),_X1),
clpfd:'x=<y IND'(_O,_M),
clpfd:'x=<y IND'(_W1,_N),
clpfd:'x=<y IND'(_Q,_M),
clpfd:'t=<u+c'(_M,_N,-1),
clpfd:'t=<u+c'(_O,_Q,-1),
clpfd:'t>=u+c'(_N,_M,1),
clpfd:'t>=u+c'(_W1,_M,1),
clpfd:'t>=u+c'(_Q,_O,1),
existsf(_W1,[st(_X1#>_W1)]),
forallf(_X1,[st(_X1#>_W1)]),
forallf(_Y1,[st(_Y1#>_W1)]),
_M in inf..8,
_N in inf..sup,
_O in inf..7,
_W1 in inf..9,
_Q in inf..8
```

Figura 5.2: Output di un progetto SCIFF sulla console di Eclipse

generati dall'esecuzione di SCIFFgraph (DOT ed immagini). Inoltre il nome del progetto selezionato servirà per nominare i file in uscita in questo modo: "*nomeproject_tipografo*".

- La directory che conterrà i file di output; essa può essere ulteriormente modificata agendo direttamente sulla barra di testo o accedendo alla finestra di dialogo, attraverso il tasto *browse*, che permette di navigare all'interno del filesystem e di selezionare la directory che si vuole utilizzare per il salvataggio dei file.

- Uno dei formati immagine disponibili (ps, svg, png, gif).

- Il tipo di grafo che si vuole realizzare: semplice (con *neato*), ordinato (con *dot*), oppure entrambi. Nel primo caso i file in uscita saranno denominati "*nomeproject_simpleGraph*", nel secondo invece "*nomeproject_sortedGraph*".

Nelle figure 5.3 e 5.4 si può vedere un esempio dell'esecuzione di SCIFFgraph su un progetto SCIFF, e le ricadute sulla cartella inserita nella finestra di dialogo.

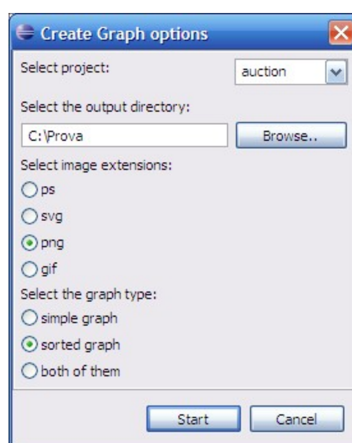

Figura 5.3: Esempio di impostazioni per l'esecuzione di SCIFFgraph

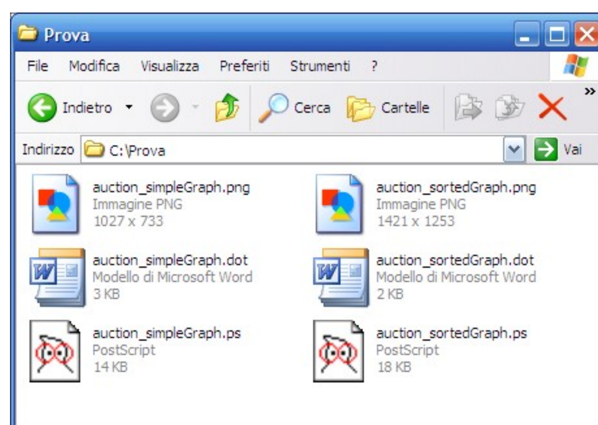

Figura 5.4: File creati da SCIFFgraph all'interno della cartella di output

Tra i comandi inseriti nella toolbar e nel menu di SCIFFgraph ne sono presenti due, chiamati "*Open DOTConfiguration File*" ed "*Open Constraints List File*" che aprono rispettivamente, con l'editor di Eclipse, il file per la configuarazione degli atomi e le relative impostazioni DOT, ed il file per la

dichiarazione dei funtori dei vincoli. In figura 4.1 e 4.2 vengono mostrati le icone ed i comandi appena descritti.

In particolare se si vuole dichiarare un nuovo atomo è necessario aggiungere nel file *DOTConfig* (come affermato nel paragrafo 3.4) una nuova dupla:

SHAPE_*funtore*=*forma*;

COLOR_*funtore*=*colore*;

Ad esempio se si vuole aggiungere un atomo con funtore *new*, con forma e colore standard nel file si scriverà:

SHAPE_new=*box*;

COLOR_new=*crimson*;

Invece se si vuole dichiarare un nuovo vincolo nel file *contraintsConfig* con sintassi prefissa è sufficiente aggiungere in una nuova riga il suo funtore.

Nelle figura 5.5 e 5.6 sono presenti rispettivamente uno spezzone del file *DOTConfig* ed uno del file *contraintsConfig*.

```
SHAPE_note=box;
COLOR_note=crimson;

SHAPE_noten=box;
COLOR_noten=crimson;

SHAPE_abd=box;
COLOR_abd=crimson;
```

Figura 5.5: DOTConfiguration File

```
reif_unify
reified_unif:disunif_list_constr
forallf
clpfd
existsf
```

Figura 5.6: DOTConfiguration File

Capitolo 6

Conclusioni

Il software sviluppato in questa tesi, chiamato SCIFFgraph, è volto a visualizzare in maniera semplice ed intuitiva i risultati prodotti dalla procedura di dimostrazione automatica SCIFF [1]. Infatti, l'output della procedura può risultare complesso da interpretare per un utente che non sia esperto e, spesso, anche per l'utente più accorto è difficile cogliere tutte le relazioni fra gli oggetti che vi compaiono. Il visualizzatore mostra l'output in formato di un grafo, in cui gli atomi abducibili sono mostrati come nodi ed i vincoli sono rappresentati come archi. Il software visualizza i grafi con diverse modalità: gli atomi possono essere raggruppati per tipologia (funtore), oppure ordinati in ordine temporale. Inoltre, visto che la SCIFF era stata integrata nella IDE Eclipse [7], il visualizzatore SCIFFgraph è stato aggiunto come componente nella medesima IDE. In questo modo, l'uso della SCIFF potrebbe risultare più intuitivo, permettendo il suo utilizzo in diversi ambiti anche da non esperti.

Ulteriori sviluppi di questo software potrebbero favorire una migliore integrazione in Eclipse, che eviti all'utente di re-inserire più volte le stesse informazioni. Inoltre, sarebbe auspicabile un metodo più sofisticato per or-

dinare gli atomi, considerando non solo quelli con tempi ground, ma anche quelli con tempi variabili. Infine, si potrebbe modificare la visualizzazione per mostrare sullo stesso livello gli eventi che accadono contemporaneamente.

Bibliografia

[1] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logics*, 9(4), 2008.

[2] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. The role of abduction in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.

[3] J. W. Lloyd. *Foundations of Logic Programming*. 2nd extended edition, 1987.

[4] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[5] SICStus prolog user manual, release 3.12.7, October 2006. http://www.sics.se/isl/sicstus/.

[6] Eclipse documentation, 2008. http://www.eclipse.org.

[7] Christian Cantelmo. Progettazione e sviluppo di un ambiente integrato di programmazione per il linguaggio SCIFF, 2006.

[8] Tze H. Fung and Robert A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, November 1997.

[9] The DOT language, 2008. http://graphviz.org/Documentation.php.

[10] Graphviz - Graphic Visualization Software, 2008. http://graphviz.org/About.php.

[11] Eclipse user interface guidelines, version 2.1, 2008. http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html.

[12] SOCS-SI, 2008. http://www.lia.deis.unibo.it/research/socs_si/-socs_si.shtml.