# Università degli studi di Roma
# “La Sapienza”

## Facoltà di Scienze Matematiche Fisiche e Naturali
## Corso di laurea in Informatica


**Un sistema per la verifica di software distribuito basato su riscrittura di ipergrafi**

**I**


Relatore: Prof. Pietro Cenciarelli

Candidato: Ivano Talamo



Anno Accademico 2003/2004

# Indice

**Indice** **1**

**1 Introduzione** **3**

**2 Riscrittura sincronizzata di ipergrafi** **10**

2.1 Ipergrafi Sincronizzati ($SG$) . . . . . . . . . . . . . . . . . . . . 11

2.1.1 Transizioni . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.1.2 Sincronizzazione di Transizioni . . . . . . . . . . . . . . . 15

2.1.3 Produzioni . . . . . . . . . . . . . . . . . . . . . . . . . 18

2.2 Esempi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

2.2.1 Comunicazione n-aria . . . . . . . . . . . . . . . . . . . . 20

2.2.2 Un contatore . . . . . . . . . . . . . . . . . . . . . . . . 23

**3 Synchronized Hypergraph Environment** **26**

3.1 Progettazione di un sistema in *SHE* . . . . . . . . . . . . . . . . 30

3.2 Generazione del grafo delle transizioni . . . . . . . . . . . . . . 33

3.3 Esplorazione del grafo delle transizioni . . . . . . . . . . . . . . 34

**4 Implementazione della riscrittura** **39**

4.1 Mur$\varphi$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

4.2 Riscrittura di ipergrafi . . . . . . . . . . . . . . . . . . . . . . . . 43
4.2.1 Rappresentazione dei grafi . . . . . . . . . . . . . . . . . . 44
4.2.2 Strategia di riscrittura . . . . . . . . . . . . . . . . . . . . 46
4.3 Stategie di riduzione degli stati . . . . . . . . . . . . . . . . . . . . 52
4.3.1 Il problema della duplicazione degli stati . . . . . . . . . 53
4.3.2 Stati validi . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.3.3 Modifiche al codice di Mur$\varphi$ . . . . . . . . . . . . . . . . . 59

**5 Un sistema per l'esplorazione di grafi 61**
5.1 La generazione di grafi con *GraVE* . . . . . . . . . . . . . . . . . 68
5.2 L'interfaccia GraphViewer . . . . . . . . . . . . . . . . . . . . . . . 74
5.3 L'interfaccia DataViewer . . . . . . . . . . . . . . . . . . . . . . . . 75
5.4 Selezione di nodi . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
5.5 File di configurazione . . . . . . . . . . . . . . . . . . . . . . . . . 82

**6 Casi studio 84**
6.1 Ordinamento di interi . . . . . . . . . . . . . . . . . . . . . . . . . 85
6.2 Modifica di un'area di memoria condivisa . . . . . . . . . . . . . 88
6.3 Verifica di una semplice proprietà di sicurezza . . . . . . . . . . 95

**7 Conclusioni 103**

**Bibliografia 108**

# Capitolo 1

# Introduzione

Nell'ultimo decennio si è assistito ad una notevole crescita di interesse nell'ambito dei sistemi distribuiti. Uno dei motivi di tale fenomeno è senza dubbio la rete Internet, che attualmente conta milioni di utenti in tutto il mondo, e che rappresenta *il* fenomeno che ha modificato e condizionato maggiormente l'evoluzione della comunicazione in questi ultimi anni. La presenza di Internet nella vita umana è ormai evidente in moltissime attività: dall'uso massiccio di posta elettronica, alla gestione del proprio conto bancario via *Web*, fino agli esperimenti di voto elettronico. I sistemi distribuiti intervengono inoltre anche nell'ambito delle reti a dimensione locale ($LAN$), presenti ormai in ogni realtà aziendale ed istituzionale, e nello studio dei sistemi operativi concorrenti.

I problemi coinvolti in tali situazioni sono molteplici e vanno dalla *sicurezza* delle comunicazioni all'*affidabilità* delle operazioni. Ad esempio, durante la consultazione dei dati della propria cartella clinica collocati su un sito Internet, sarebbe auspicabile essere gli unici ad usufruire di tali informazioni; durante l'utilizzo di un sistema di *web-banking*, oltre ad essere gli unici a

visualizzare i nostri dati, sarebbe desiderabile essere certi che, quando effettuiamo un trasferimento di una somma di denaro tra due conti, questo vada a buon termine, oppure fallisca, ma non rimanga in uno stato intermedio, in cui magari la somma trasferita non compare né sul conto di origine né su quello di destinazione. Per questi motivi risulta dunque importante disporre di modelli formali per lo studio di tali sistemi. Un tale approccio può consentire infatti di *dimostrare* la validità di certe proprietà del sistema, o di individuarne delle debolezze.

Gli aspetti che è necessario investigare nello studio di un sistema distribuito sono molteplici e legati tra loro. Un primo aspetto è la *località*: le varie entità che prendono parte al sistema possono trovarsi in luoghi geograficamente molto distanti, e il costo di una comunicazione può variare molto in base a questo fattore. Inoltre la configurazione spaziale di un sistema distribuito può essere dinamica, cioè subire dei cambiamenti nel corso del tempo. Nei modelli per lo studio di sistemi distribuiti è quindi importante la possibilità di studiare efficacemente questo aspetto.

Un altro concetto cruciale è quello del *fallimento*. Per progettare un sistema reale è necessario considerare gli eventuali guasti delle componenti del sistema. Il fallimento può essere inteso sia come l'avaria di un componente che smette, ad un certo punto della sua esistenza, di funzionare (guasto di tipo *crash-failure*), sia come un componente che si comporta in maniera anomala all'interno del sistema, ossia in maniera non prevista da chi lo aveva progettato. Questo tipo di fallimento può essere particolarmente utile per modellare comportamenti maligni, infatti un utente che tenta di violare un sistema può essere modellato come un componente guasto, che può compiere dunque operazioni arbitrarie.

In questa tesi viene presentato un ambiente di supporto allo studio di si-

stemi distribuiti, denominato *Synchrnonized Hyperedge Environment* (*SHE*). Il sistema è stato interamente progettato ed implementato dall'autore della tesi e consta di circa 8000 linee di codice Java, integrate, mediante opportuni shell script, con un sistema di verifica automatica (Mur$\varphi$) di cui parleremo più sotto.

Con *SHE* la modellazione di un sistema avviene attraverso la manipolazione di *grafi*. I grafi risultano una struttura molto efficace per rappresentare informazioni strutturate, e la loro visualizzazione grafica fornisce informazioni sulla topologia e su proprietà del grafo in maniera molto più immediata della rispettiva rappresentazione testuale. Con *SHE* è possibile *descrivere* l'interazione di processi distribuiti mediante la trasformazione di grafi, impostare una configurazione iniziale, ed esplorare visivamente le evoluzioni del sistema.

*SHE* è basato sui *Synchronizing Graphs* (*SG*) [CTT04, CT04], un modello formale di *riscrittura di ipergrafi*, alla cui realizzazione ha contribuito anche l'autore della tesi. L'idea alla base di *SG* è che gli iperarchi rappresentano processi e i nodi su cui un iperarco incide i canali di comunicazione a disposizione del processo. La condivisione di un nodo tra due iperarchi rappresenta quindi l'esistenza di un canale di comunicazione tra i rispettivi processi. L'evoluzione di un arco è descritta da un insieme di riscritture, e l'evoluzione di un grafo avviene applicando una serie di riscritture agli archi che lo compongono. In tal modo il *non determinismo* è modellato fornendo diverse riscritture per lo stesso arco. Durante le riscritture, gli archi possono comunicare emettendo delle *azioni* sui nodi, realizzando quindi un meccanismo di comunicazione tra processi. Inoltre, assieme alle azioni, possono essere comunicati nodi, ottenendo così un meccanismo per la *mobilità* dei processi.

L'utilizzo di *SHE* si divide in tre fasi principali. Nella prima fase l'utente interagisce con un modulo per la descrizione del sistema che desidera modellare. Ciò avviene in uno stile di programmazione dichiaratica, *disegnando* un insieme di riscritture ed un ipergrafo che rappresenta lo stato iniziale del sistema. Terminata la *descrizione* del sistema, l'utente chiede a *SHE* di *generare* il grafo degli stati del sistema, cioè un grafo diretto che rappresenta tutte le possibili computazioni del sistema descritto. I nodi di questo grafo rappresentano degli ipergrafi, ovvero stati del sistema, e un arco $A \to B$ una transizione di stato, implementata come una riscrittura dell'ipergrafo $A$ nell'ipergrafo $B$. Successivamente inizia per l'utente la fase di *esplorazione* e *verifica* delle computazioni calcolate dal sistema. In tale fase l'utente può visitare il grafo degli stati generato precedentemente, selezionando attraverso il cursore i nodi del grafo degli stati, e visualizzando l'ipergrafo corrispondente. L'utente può inoltre interrogare il sistema, chiedendo di mostrare gli stati che verificano determinate proprietà, ad esempio tutti gli stati finali oppure tutti quelli in cui compare un arco con una particolare etichetta.

Osserviamo che il modulo utilizzato nella fase di esplorazione, che abbiamo denominato *GraVE* (*Graph Viewer and Explorer*), può essere in realtà utilizzato in contesti assai diversi per analizzare generiche strutture basate su grafi e può essere pertanto presentato come un programma a sé stante. Nel capitolo 5 verranno mostrati alcuni esempi dell'uso di *GraVE*.

Per quanto riguarda il calcolo delle riscritture, e quindi la generazione del grafo degli stati, viene utilizzato il *model-checker* Mur$\varphi$ [MU, DDHY92, D. 96, PITZ02]. Mur$\varphi$ (da leggersi come in inglese "Murphy") consente di modellare un sistema descrivendone le componenti e le possibili evoluzioni, mediante un meccanismo ispirato al modello Unity [CM88]. A partire da tale specifica, Mur$\varphi$ si occupa di effettuare un'esplorazione esaustiva dello

spazio degli stati del sistema, generando di fatto il grafo degli stati. Dato quindi un insieme di riscritture *SG* e un grafo di partenza, *SHE* genera una specifica Mur$\varphi$ in cui lo stato iniziale rappresenta il grafo di partenza, e le possibili evoluzioni del sistema rappresentano le riscritture descritte dall'utente. Mur$\varphi$ consente inoltre di esprimere delle *invarianti* su un sistema, cioè delle proprietà che devono valere in ogni stato che Mur$\varphi$ esplora. Nel caso di violazione di un'invariante Mur$\varphi$ è in grado di ricostruire la computazione che ha condotto dallo stato iniziale a quello in cui l'invariante è stata violata. Sebbene tale funzionalità di Mur$\varphi$ potesse essere utilizzata all'interno di *SHE* per verificare proprietà sulle computazioni *SG*, abbiamo tuttavia scelto un approccio differente, implementando la funzionalità di verifica all'interno di *GraVE*. Tale scelta è derivata dal fatto che le invarianti di Mur$\varphi$ consentono di analizzare soltanto uno stato del sistema e non le relazioni tra gli stati. Ad esempio risulterebbe difficile esprimere un'invariante che controlla che in tutti gli stati finali di una computazione *SG* compaia sempre un particolare arco. Tale discorso verrà approfondito nel capitolo 4.

La scelta di Mur$\varphi$ per la simulazione del calcolo *SG* deriva dalla semplicità con cui tale strumento consente di modellare un sistema e dal gran numero di campi applicativi in cui è stato utilizzato (vedi [SD95, MMS97, Mit98, LGM$^+$95]), dimostrandosi uno strumento flessibile e potente. In Mur$\varphi$ l'evoluzione di un sistema viene infatti descritta mediante un insieme di *regole*, ognuna delle quali è composta da una condizione di applicabilità della regola stessa e da un'azione corrispondente alla sua applicazione. Sarà poi il programma generato da Mur$\varphi$ a partire dalla specifica dell'utente, ad esplorare in maniera esaustiva lo spazio degli stati, applicando ad ogni passo l'insieme di regole per le quali è soddisfatta la condizione di applicabilità.

L'intero progetto è sviluppato con l'intento di creare una serie di moduli che possano essere facilmente modificati o eventualmente sostituiti per aggiungere funzionalità al sistema. Per tale motivo i moduli che compongono *SHE* comunicano attraverso lo scambio di file in formato XML [BPSM97]. In particolare, nel caso del grafo degli stati, viene utilizzato il formato graphml [BEH+01], un XML specifico per la rappresentazione di grafi.

Ad eccezione della parte di generazione del grafo degli stati, che è affidata a Mur$\varphi$, il resto del progetto è sviluppato in Java. Tale scelta consente di utilizzare *SHE* su diverse piattaforme senza necessità di modifiche. Va notato tuttavia che questo obiettivo è raggiunto solo in parte a causa dei vincoli imposti da Mur$\varphi$ e dall'utilizzo di alcuni strumenti base dei sistemi operativi UNIX. Notiamo comunque che in ambiente Windows è possibile utilizzare Mur$\varphi$ mediante Cygwin[1], un programma Windows che emula un ambiente UNIX. Infine, *SHE* utilizza la libreria software yfiles[2], che fornisce algoritmi e componenti per l'analisi, la visualizzazione e il disegno di grafi. Tale libreria è utilizzata sia nella parte di composizione, in cui l'utente *disegna* le riscritture di un sistema *SG*, sia nella fase di esplorazione, per visualizzare il grafo degli stati e per visualizzare gli ipergrafi corrispondenti agli stati della computazione.

**Struttura della tesi.** Nel capitolo 2 introduciamo formalmente il modello *SG* alla base di *SHE* e forniamo degli esempi per illustrare il funzionamento del modello.

Nel capitolo 3 descriviamo con maggiore in dettaglio l'utilizzo di *SHE* e la sua architettura, mostrando le interazioni tra le varie parti ed evidenziando

[1] http://www.cygwin.com/
[2] http://www.yworks.com/products/yfiles

la modularità del sistema. Nella sezione 3.2 viene introdotta la generazione del grafo degli stati, che analizziamo con maggiore dettaglio nel capitolo 4. Inoltre *GraVE*, descritto come componente di *SHE* in 3.3, viene analizzato come programma autonomo nel capitolo 5.

Nel capitolo 4 introduciamo Mur$\varphi$ e mostriamo come è possibile implementare con tale strumento l'esplorazione delle riscritture in *SG*. Effettuiamo inoltre un'analisi dello spazio degli stati esplorati da Mur$\varphi$ per implementare le riscritture. Concludiamo il capitolo con una descrizione di alcune modifiche che abbiamo apportato al codice di Mur$\varphi$ per ottenere un output in formato graphml.

Nel capitolo 5 viene analizzato *GraVE*. In particolare viene analizzata l'architettura del sistema e discusso come questo possa essere utilizzato in contesti diversi da *SHE* per esplorare informazioni strutturate come un grafo.

Nel capitolo 6 mostriamo infine tre casi studio. Nel primo realizziamo in *SHE* un algoritmo di ordinamento, con uno stile di programmazione *dichiarativo*. Questo caso studio mostra inoltre la flessibilità di *GraVE* nella visualizzazione degli ipergrafi, dove vengono rappresentate in maniera testuale solo determinate caratteristiche di un ipergrafo, fornendo così, per il particolare problema trattato, una rappresentazione più chiara di quella standard fornita con *SHE*. Nel secondo caso studio risolviamo invece il problema dell'accesso coordinato ad una risorsa condivisa, mostrando come *SHE* supporti un approccio *incrementale* alla progettazione di un sistema. Infine, nel terzo caso studio, affrontiamo un semplice problema di sicurezza e mostriamo come *SHE* possa essere utilizzato come strumento di verifica di proprietà di un sistema.

La tesi si conclude con una discussione dei possibili sviluppi di *SHE* e degli ulteriori ambiti applicativi in cui è possibile utilizzare *GraVE*.

# Capitolo 2

# Riscrittura sincronizzata di ipergrafi

In questo capitolo presentiamo *Synchronizing Graphs (SG)* [CTT04, CT04], il modello di calcolo alla base di *SHE*. Tale modello è ispirato all'*SHR*, il sistema di *riscrittura sincronizzata di ipergrafi* proposto in [FMT01], cui apporta però modifiche tali, soprattutto in senso semplificativo, da potersi considerare un calcolo completamente nuovo, con proprietà metateoretiche diverse dall'originale. Nel contesto di questa tesi considereremo una versione di *SG* *senza* l'operatore di restrizione. Una versione dei *Synchronizing Graphs* che include anche la restrizione sui nodi è presentata in [Tib04, CT04]. Notiamo che l'implementazione in *SHE* di un potente sistema di riscrittura di grafi è risultata particolarmente agevole proprio per la semplicità del modello matematico dei *Synchronizing Graphs*.

*SG* è un modello di calcolo basato su *riscrittura* di *ipergrafi* in cui gli archi rappresentano processi, o agenti, e i nodi canali di comunicazione. Nel seguito useremo i termini *grafo* e *arco* al posto di *ipergrafo* e *iperarco* quando

ciò non generi ambiguità. Archi che condividono nodi rappresentano processi che condividono canali di comunicazione.

Un arco evolve riscrivendosi in un grafo. Un grafo evolve mediante la riscrittura di tutti gli archi che lo compongono. Durante la riscrittura un arco può associare a ciascun nodo su cui incide un elemento, detto *azione*, di un insieme predefinito. Come in molti calcoli di processi, la comunicazione tra agenti è implementata in $SG$ mediante la sincronizzazione di azioni e co-azioni (azioni "complementate"). Poiché, durante una riscrittura, ad ogni nodo può essere associata al più una coppia di azioni complementari, $SG$ realizza un modello di comunicazione *binaria*. Alle azioni possono inoltre essere associate liste di nodi, ottenendo quindi un meccanismo di passaggio di informazione attraverso il grafo.

Se quindi l'emissione di azioni può essere visto come un meccanismo di comunicazione tra processi, l'associazione di liste di nodi alle azioni può essere visto come un meccanismo di passaggio di parametri.

Nel resto del capitolo presentiamo formalmente il modello $SG$ e mostriamo alcuni esempi per illustrare il funzionamento del calcolo.

## 2.1 Ipergrafi Sincronizzati ($SG$)

Siano $\mathcal{N}$ e $\mathcal{L}$ due insiemi (che d'ora in poi considereremo fissati), denominati rispettivamente insieme dei *nodi* e insieme delle *etichette* e indichiamo con $\mathcal{N}^*$ l'insieme di tutte le sequenze finite di elementi di $\mathcal{N}$. Un grafo $G$ è una tripla $G = (E, \lambda, G)$ dove:

- $E$ è l'insieme di *iperarchi*

- $\lambda : E \rightarrow \mathcal{L}$ è una funzione totale, detta di *etichettatura*, che associa etichette a iperarchi
- $G : E \rightarrow \mathcal{N}^*$ è una funzione totale, detta di *incidenza*, che associa ad ogni iperarco la sequenza di nodi su cui esso incide.

Quando $G(e) = x_1 x_2 \ldots x_n$ diciamo che l'*arietà* di $e$ è $n$ e che l'$i$-esimo tentacolo di $e$ è attaccato ad $x_i$. L'insieme dei nodi del grafo è indicato con:

$$|G| = \{x \in \mathcal{N} \mid \exists\, e.G(e) = x_1 \ldots x_n \text{ e } x = x_i\}$$

Indichiamo con $L(x_1, \ldots, x_n)$ un arco $e$ tale che $G(e) = x_1 \ldots x_n$ e $\lambda(e) = L$. Nel seguito indicheremo la sequenza $x_1 x_2 \ldots x_n$ con la notazione $\vec{x}$. Data $\sigma : \mathcal{N} \rightarrow \mathcal{N}$, indichiamo con $\sigma(\vec{x})$ l'applicazione di $\sigma$ a tutte le componenti di $\vec{x}$, cioè $\sigma(x_1 \ldots x_n){=}\sigma(x_1) \ldots \sigma(x_n)$. Dato il grafo $G = (E, \lambda, G)$ e una funzione $\sigma : \mathcal{N} \rightarrow \mathcal{N}$, definiamo l'applicazione di $\sigma$ a $G$ il grafo $(E, \lambda, \sigma \circ G)$, dove $(\sigma \circ G)(e) = \sigma(G(e))$.

Date due funzioni $g : A \rightarrow B$ e $f : C \rightarrow B$, con $A \cap C = \emptyset$, definiamo la funzione $g + f : A \cup C \rightarrow B$ come:

$$(g + f)(x) \triangleq \begin{cases} g(x) & x \in A \\ f(x) & x \in C \end{cases}$$

Dati due grafi $G = (E_G, \lambda_G, G)$ e $F = (E_F, \lambda_F, F)$, possiamo comporli con l'operatore di composizione parallela $\cdot|\cdot$ a patto che $E_G \cap E_F = \emptyset$. Il risultato, che indichiamo con $G|F$, è il grafo:

$$(E_G \cap E_F, \lambda_G + \lambda_F, G + F)$$

Graficamente rappresenteremo il nodo $x$ con un pallino etichettato $x$, l'arco $E(x_1, \ldots, x_n)$ come un rettangolo etichettato con $E$ connesso alle rap-

presentazioni grafiche di $x_1$,...,$x_n$ e il grafo $G|F$ come l'unione delle rappresentazioni grafiche di $G$ e $F$.

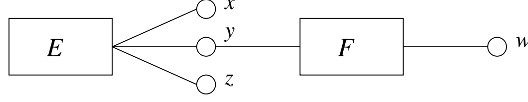



**Figura 2.1:** *Rappresentazione grafica del grafo* $E(x, y, z)|F(y, w)$

### 2.1.1 Transizioni

Sia $Act = \{a, b, \dots\} \cup \{\overline{a}, \overline{b}, \dots\}$ un insieme di *azioni* e *co-azioni*. L'azione complementare di $a$ viene indicata con $\overline{a}$ e vale la proprietà $\overline{\overline{a}} = a$. Indichiamo con $Act^+$ il prodotto cartesiano $Act \times \mathcal{N}^*$. Dato $(a, \vec{x}) \in Act^+$, chiamiamo le componenti di $\vec{x}$ gli *argomenti* di $a$. Indichiamo con $|\vec{x}|$ l'insieme degli elementi di $\vec{x}$, ossia $|x_1 \dots x_n| = \{x_1, \dots, x_n\}$ e con $||\vec{x}||$ la lunghezza di $\vec{x}$, cioè $||x_1 \dots x_n|| = n$.

Elementi di $Act^+$ vengono associati ai nodi di un grafo durante una riscrittura. Quando al nodo $x$ è associato $(a, \vec{y})$ diciamo che sul nodo $x$ viene *emessa* l'azione $a$ e vengono comunicati i nodi $\vec{y}$.

Consideriamo una relazione $\Lambda \subseteq \mathcal{N} \times Act^+$. Con $dom(\Lambda)$ denotiamo l'insieme degli elementi di $\mathcal{N}$ che compaiono come primo elemento in qualche coppia di $\Lambda$:

$$dom(\Lambda) = \{x \mid (x, (a, \vec{y})) \in \Lambda\}.$$

Con $\Lambda(x)$ indichiamo l'insieme degli elementi di $Act^+$ associati a $x$:

$$\Lambda(x) = \{(a, \vec{y}) \mid (x, (a, \vec{y})) \in \Lambda\}.$$

Infine indichiamo con $|\Lambda|$ l'insieme dei nodi che sono argomento di qualche azione in $\Lambda$:

$$|\Lambda| = \{y \mid (x, (a, y_1 \dots y_n)) \in \Lambda \text{ e } y = y_i\}.$$

Nel seguito $(x, (a, \vec{y}))$ verrà indicato con $(x, a, \vec{y})$ e, nel caso in cui la sequenza $\vec{y}$ fosse vuota, con $(x, a)$.

Definiamo *pre-transizione* la tripla $(G, \Lambda, H)$, dove $G$ e $H$ sono grafi, $\Lambda \subseteq \mathcal{N} \times Act^+$ e $dom(\Lambda) \subseteq |G|$. Solitamente indicheremo la pre-transizione $(G, \Lambda, H)$ con:

$$G \xrightarrow{\Lambda} H. \tag{2.1}$$

Il significato intuitivo di una pre-transizione è che $G$ si riscrive in $H$ emettendo sui nodi in $dom(\Lambda)$ le azioni definite in $\Lambda$ e comunicando le liste di nodi associate ad esse. Quando $\Lambda(x) = \{(a, \vec{y}), (\overline{a}, \vec{z})\}$ diciamo che su $x$ avviene una *sincronizzazione*. Quando $\Lambda(x) = \{(a, \vec{y})\}$ diciamo che su $x$ viene *osservata* l'azione $a$.

Diciamo che $\Lambda$ è *unificabile* se e solo se:

- $\forall x \in \mathcal{N}\ |\Lambda(x)| \leq 2$, e inoltre
- se $\Lambda(x) = \{(a, \vec{y}), (b, \vec{z})\}$ allora $b = \overline{a}$ e $||\vec{y}|| = ||\vec{z}||$.

Dato $\Lambda$ definiamo $\phi_\Lambda$ come la più piccola relazione di equivalenza su $\mathcal{N}$ tale che se $(x, a, y_1 \dots y_n), (x, \overline{a}, z_1 \dots z_n) \in \Lambda$ allora $y_i \phi_\Lambda z_i$ per $i = 1, \dots, n$. Diciamo che $\rho : \mathcal{N} \to \mathcal{N}$ è un *unificatore* di $\Lambda$ se e solo se:

- $\forall x \in \mathcal{N}\ x \phi_\Lambda \rho(x)$, e inoltre
- per ogni classe di equivalenza X indotta da $\phi_\Lambda$ vale $|\rho X| = 1$.

L'espressione $v(\Lambda)$ è definita se e solo se $\Lambda$ è unificabile e, in tal caso, denota *un* unificatore di $\Lambda$.

Una pre-transizione $G \xrightarrow{\Lambda} H$ è una *transizione* se e solo se:

- $\Lambda$ è unificabile
- $\upsilon(\Lambda)(H) = H$.

Le condizioni poste garantiscono quindi che vengano compiute solo azioni su nodi effettivamente presenti nel grafo ($dom(\Lambda) \subseteq |G|$) e che il grafo di arrivo rifletta le unificazioni indotte dagli argomenti associati alle azioni di sincronizzazione.

### 2.1.2 Sincronizzazione di Transizioni

La transizione di un grafo può avvenire mediante delle transizioni base, chiamate *assiomi*, e una regola che permette di combinare due transizioni, detta SYNC.

Date due transizioni $G \xrightarrow{\Lambda} H$ e $F \xrightarrow{\Theta} K$ la regola SYNC permette di combinarle per ottenere una transizione del grafo $G|F$, a condizione che valga una condizione di *non-interferenza* tra le premesse. La *non-interferenza* stabilisce che i nodi nuovi nella prima transizione (i nodi presenti in $\Lambda$ e $H$ ma non in $G$) non compaiano nella seconda transizione, e viceversa. Formalmente, la condizione di non interferenza tra due transizioni $G \xrightarrow{\Lambda} H$ e $F \xrightarrow{\Theta} K$ richiede che:

$$(|G| \cup |\Lambda| \cup |H|) \cap (|F| \cup |\Theta| \cup |K|) = |G| \cap |F|.$$

La regola per la sincronizzazione è la seguente:

$$[\text{SYNC}] \quad \frac{G \xrightarrow{\Lambda} H \qquad F \xrightarrow{\Theta} K}{G|F \xrightarrow{\Lambda \cup \Theta} \rho(H|K)} \quad (\star)$$

($\star$) se $\rho = \upsilon(\Lambda \cup \Theta)$ e le premesse non interferiscono.

Dato un insieme $\mathcal{T}$ di transizioni, dette *assiomi*, una $\mathcal{T}$*-computazione*, detta anche *computazione*, è una sequenza di transizioni $G_0 \xrightarrow{\Lambda_1} G_1 \xrightarrow{\Lambda_2} \ldots$ ciascuna delle quali è derivata dagli assiomi in $\mathcal{T}$ mediante zero o più applicazioni della regola SYNC. Per semplicità indicheremo $G_0 \xrightarrow{\Lambda_1} G_1 \xrightarrow{\Lambda_2} \ldots \xrightarrow{\Lambda_n} G_n$ con $G_0 \rightarrow^n G_n$, e $G \rightarrow^* H$ se esiste $n$ tale che $G \rightarrow^n H$ oppure $H = G$.

**Esempi.** Vediamo ora alcuni esempi di computazioni. Consideriamo i seguenti assiomi:

- $P(x, y) \xrightarrow{(y,c,v)} P(x, y, v)$
- $C(y, z) \xrightarrow{\substack{(y,\overline{c},s)\\(z,\overline{c},s)}} C(y, z)$
- $Q(w, z) \xrightarrow{(z,c,m)} Q(w, z, m)$.

Mediante un'applicazione della regola SYNC ai primi due assiomi è possibile ottenere una transizione del grafo $P(x, y)|C(y, z)$:

$$[\text{sync}] \quad \frac{P(x, y) \xrightarrow{(y,c,v)} P(x, y, v) \qquad C(y, z) \xrightarrow{\substack{(y,\overline{c},s)\\(z,\overline{c},s)}} C(y, z)}{P(x, y)|C(y, z) \xrightarrow{\substack{(y,c,v)\\(y,\overline{c},s)\\(z,\overline{c},s)}} P(x, y, v)|C(y, z)}$$

dove è stato scelto $v$ come l'unificatore che mappa $s$ in $v$.

Applicando nuovamente SYNC a quest'ultima transizione e al terzo assioma si ottiene la transizione:

$$P(x, y)|C(y, z)|Q(w, z) \xrightarrow{\substack{(y,c,v)\\(y,\overline{c},s)\\(z,\overline{c},s)\\(z,c,m)}} P(x, y, v)|C(y, z)|Q(w, z, v)$$

dove è stato scelto $v$ come l'unificatore che mappa sia $s$ che $m$ in $v$, poiché avvengono due sincronizzazioni, una sul nodo $y$ e una sul nodo $z$, che richiedono quindi l'unificazione di $\{v, s, m\}$. Questa transizione è rappresentata graficamente in figura 2.2.

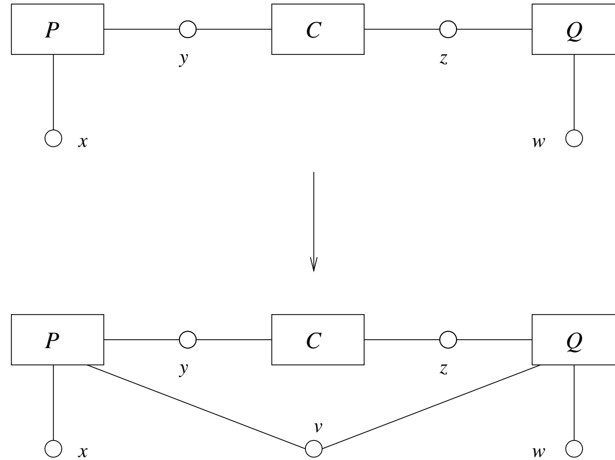



**Figura 2.2:** *Creazione di un canale di comunicazione tra 2 processi*

Notiamo come con la transizione appena vista abbiamo implementato un meccanismo di creazione di canali di comunicazione tra due processi: $P$ e $Q$ rappresentano due processi che non condividono canali di comunicazione, ma comunicano entrambi con un terzo processo, che ha il ruolo di intermediario. Attraverso quest'ultimo, e mediante l'unificazione dei nodi, è possibile creare un canale di comunicazione condiviso soltanto da $P$ e $Q$.

Un meccanismo simile può essere utilizzato per mettere in comunicazione processi anche molto distanti tra loro.

Consideriamo il seguente grafo:

$$P(x, y_1)|C(y_1, y_2)|C(y_2, y_3)|\ldots|C(y_n, y_{n+1})|Q(y_{n+1}, z)$$

e gli assiomi:

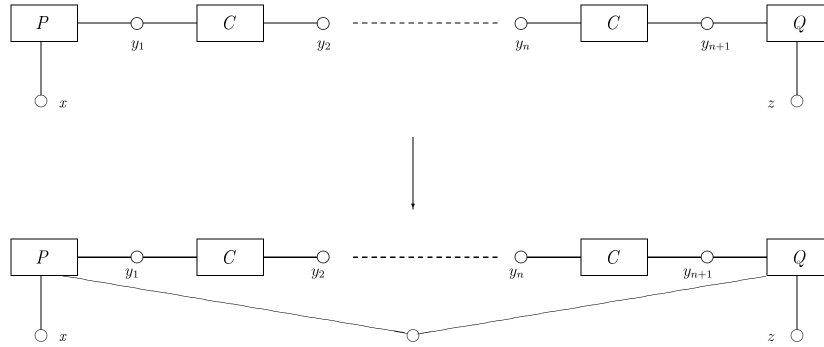



**Figura 2.3:** *Creazione di un canale di comunicazione tra 2 processi*

- $P(x, y_1) \xrightarrow{(y_1, c, n)} P(x, y_1, n)$
- $Q(y_{n+1}, z) \xrightarrow{(y_{n+1}, \overline{c}, m)} Q(y_{n+1}, z, m)$
- $C(y_i, y_{i+1}) \xrightarrow{\substack{(y_i, \overline{c}, s_i) \\ (y_{i+1}, c, s_i)}} C(y_i, y_{i+1})$ con $i = 1 \ldots, n$.

Allora, a partire da questi assiomi, e mediante $n+1$ applicazioni della regola SYNC è possibile ottenere la transizione di figura 2.3.

### 2.1.3 Produzioni

Come abbiamo visto, per ottenere la transizione di un grafo è necessario specificare un insieme di *assiomi*, e combinare questi assiomi con la regola SYNC. Osserviamo però che i nodi di un assioma non sono *variabili*. Quindi non è possibile applicare un assioma per l'arco $E(x)$ all'arco $E(y)$ perché $x$ e $y$ sono nodi diversi.

È spesso conveniente specificare un insieme di assiomi usando dei particolari *schemi* di transizione detti *produzioni*. Una *produzione* è simile ad un assioma, con la differenza che i nodi della transizione sono *metavariabili*

ed inoltre il lato sinistro è composto da un singolo arco. Data una produzione è possibile ottenere un assioma *istanziando* le metavariabili con nodi specifici. Una produzione va istanziata in maniera che metavariabili diverse che rappresentano nodi nuovi sono instanziate con nodi diversi. Nel seguito useremo per le metavariabili gli stessi nomi $x, y, z, \ldots$ usati anche per nodi. Il significato dei nomi sarà chiarito dal contesto in cui sono utilizzati.

Data la produzione:

$$L(x, y) \xrightarrow{(x,a,z)} L(z, w),$$

una sua possibile istanza sarà la transizione $L(v, v) \xrightarrow{(v,a,m)} L(m, n)$ mentre non lo sarà la transizione $L(v, v) \xrightarrow{(v,a,m)} L(m, m)$ poiché le due metavariabili $z$ e $w$ pur rappresentando due nodi nuovi, sono state istanziate con lo stesso nodo $(m)$.

Mediante l'uso di di produzioni è possibile semplificare notevolmente la scrittura di sistemi *SG*. Se infatti l'esempio mostrato in figura 2.3 richiede di specificare $n + 2$ assiomi, è sufficiente specificare tre produzioni che coprono l'esempio per ogni valore di $n$.

- $P(x, y) \xrightarrow{(y,c,n)} P(x, y, n)$
- $Q(y, z) \xrightarrow{(y,\bar{c},m)} Q(y, z, m)$
- $C(x, y) \xrightarrow{\substack{(x,\bar{c},s) \\ (y,c,s)}} C(x, y).$

Inoltre tali produzioni non si applicano soltanto al grafo in figura 2.3 ma ad un numero infinito di grafi, mentre gli assiomi specificati a pagina 17 possono essere applicati quando gli archi incidono proprio su *quei* nodi. Osserviamo infine che con un insieme finito di produzioni è possibile descrivere il comportamento di un grafo con un numero infinito di archi. Senza le produzioni

sarebbe necessario fornire un numero infinito di assiomi. Infatti, dato il grafo $E(x_1, x_2)|E(x_2, x_3)|\ldots$ e la produzione $E(x, y) \longrightarrow E(y, x)$ è possibile ottenere la transizione:

$$E(x_1, x_2)|E(x_2, x_3)|\ldots \longrightarrow E(x_2, x_1)|E(x_3, x_2)|\ldots$$

Senza produzioni avremmo avuto bisogno dell'insieme infinito di assiomi:

- $E(x_1, x_2) \longrightarrow E(x_2, x_1)$
- $E(x_2, x_3) \longrightarrow E(x_3, x_2)$
- $\ldots$

## 2.2 Esempi

Forniamo ora alcuni semplici esempi che aiutano a comprendere meglio il modello e le sue potenzialità.

### 2.2.1 Comunicazione n-aria

Mostriamo come, sebbene il meccanismo di comunicazione di *SG* sia binario, è possibile implementare una comunicazione *n-aria*. Nel seguito mostreremo due modi diversi per implementare tale comunicazione. Nella primo caso mostreremo una soluzione piuttosto intuitiva, che permette comunicazioni solo tra un numero finito e fissato di archi. Nel secondo caso la topologia della rete è più complessa ma è possibile comunicare un messaggio ad un numero arbitrario di archi.

Nella prima soluzione abbiamo un arco speciale, etichettato con *Broadcast*, che incide su $n$ nodi. Tale arco rappresenta un processo capace di fare

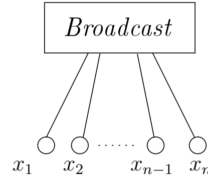



**Figura 2.4:** *L'arco che effettua il broadcast*

il *broadcast* di un messaggio. Ciascuno degli $n$ nodi rappresenta un canale di input/ouput. Intuitivamente, quando l'arco *Broadcast* riceve un messaggio (rappresentato da un'azione e da un nodo) su uno dei nodi, ritrasmette un messaggio su tutti gli altri nodi. In figura 2.4 vediamo un arco *Broadcast* capace di comunicare un messaggio a $n-1$ archi. Le sue produzioni sono:

$$Broadcast(x_1,\ldots,x_n) \xrightarrow{\substack{(x_i,\overline{a},y)\\ \bigcup_{j\neq i}(x_j,a,y)}} Broadcast(x_1,\ldots,x_n)\ i=1,\ldots,n$$

Gli archi che rappresentano i processi che desiderano ricevere o spedire messaggi di broadcast sono etichettati con $P$ e usano le seguenti produzioni:

- $P(x) \xrightarrow{(x,a,y)} P(x,y)$
- $P(x) \xrightarrow{(x,\overline{a},y)} P(x,y)$.

La prima produzione è utilizzata per inviare un messaggio di broadcast, la seconda per riceverne uno. In figura 2.5 vediamo una transizione che fa uso delle produzioni fornite. Sebbene l'esempio appena visto risolva il problema proposto, è tuttavia necessario conoscere in anticipo il numero massimo di archi che saranno connessi all'arco *Broadcast*.

É però possibile, utilizzando una diversa topologia, realizzare una comunicazione n-aria e, allo stesso tempo, gestire un numero arbitrario (eventualmente infinito) di processi che prendono parte alla comunicazione. L'idea è

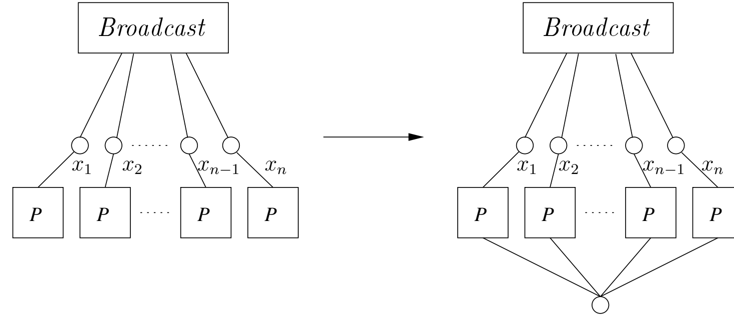



**Figura 2.5:** *Comunicazione n-aria*

quella di usare l'unificazione dei nodi comunicati durante una sincronizzazione, per *propagare* un nodo lungo una catena di archi, in maniera simile all'esempio visto in figura 2.3. Gli archi che costituiscono l'infrastruttura necessaria alla comunicazione sono etichettati con *Comm*. Gli archi che rappresentano i processi che prendono parte alla comunicazione *n-aria* sono etichettati con $P$. Un arco *Comm* può essere connesso ad altri due archi *Comm* e ad un arco $P$:

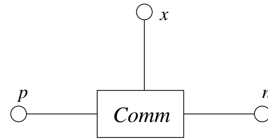



**Figura 2.6:** *Un arco* Comm

Un arco $P$ è connesso soltanto ad un arco *Comm*. Le produzioni del sistema sono quindi:

- $Comm(p, n, x) \xrightarrow[]{\substack{(p,a,y)\\(n,\overline{a},y)\\(x,\overline{b},y)}} Comm(p, n, x)$

- $P(x) \xrightarrow{(x,b,y)} P(x,y)$

Con queste produzioni è possibile ottenere ad esempio la transizione in figura 2.7, in cui sono state omesse per semplicità le azioni di sincronizzazione.

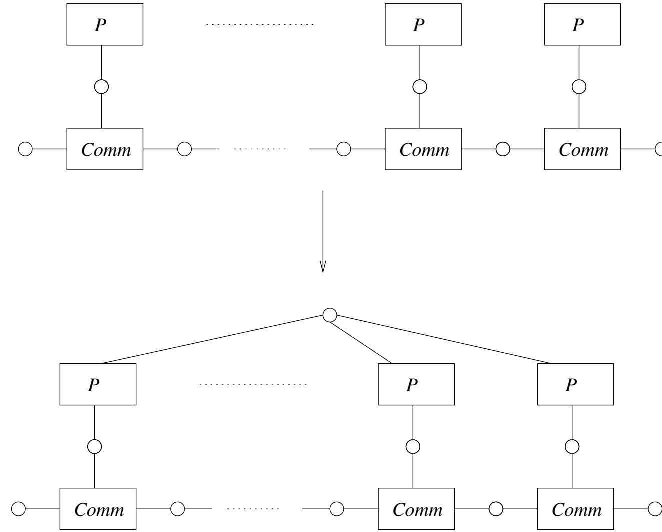



**Figura 2.7:** *Comunicazione* n-aria

### 2.2.2 Un contatore

Mostriamo ora come implementare in $SG$ un contatore, ossia un sistema che partendo da un valore iniziale, decrementa tale valore fino a giungere a zero. Tale contatore è stato realizzato con un insieme di archi che emettono su un certo nodo $x$, per un numero prescelto $n$ di riscritture, la stessa azione $a$. Dopo queste $n$ riscritture, il sistema non emetterà più azioni sul nodo $x$.

L'idea è quella di utilizzare una sequenza di archi lunga $n$ per contare fino ad $n$. Il nodo in testa alla sequenza è quello che emette l'azione prefissata, e dopo aver emesso l'azione, si riscrive nel grafo vuoto, comunicando al proprio

predecessore il nodo su cui compiere l'azione successiva. In tal modo, ad ogni riscrittura, la sequenza emette un'azione sempre sullo stesso nodo e *perde* un arco, fino ad esaurirsi del tutto. Un esempio della struttura della sequenza è mostrato in figura 2.8.

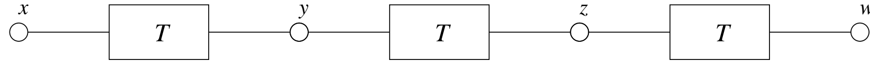



**Figura 2.8:** *Contatore fino a 3*

Le produzioni sono le seguenti:

- $T(x,y) \xrightarrow[]{(x,a)\,(y,\bar{b},x)} \emptyset$
- $T(x,y) \xrightarrow[]{(x,b,z)\,(y,c)} T(z,y)$
- $T(x,y) \xrightarrow[]{(x,\bar{c})\,(y,c)} T(x,y)$.

La prima produzione è utilizzata dall'arco in testa alla sequenza, e l'azione $a$ emessa sul nodo $x$ è quella che indica un passo nel conteggio. L'arco in testa alla sequenza compie inoltre, sul nodo che lo connette al resto della sequenza, un'azione con cui passa il nodo $x$. Il secondo arco della sequenza, sicronizzandosi sul primo nodo, riceve dal primo arco il nodo $z$ su cui andrà ad incidere nel risultato della riscrittura. La terza produzione sarà usata dai rimanenti archi della sequenza.

Tramite le suddette produzioni, e mediante l'applicazione della regola SYNC è possibile ottenere la seguente sequenza di transizioni (non sono mostrate le azioni associate ai nodi su cui avviene una sincronizzazione):

$$
\begin{aligned}
T(x_1,x_2)|T(x_2,x_3)|T(x_3,x_4)|T(x_4,x_5) &\xrightarrow[]{\substack{(x_1,a)\\(x_5,\overline{w})}} \\
T(x_1,x_3)|T(x_3,x_4)|T(x_4,x_5) &\xrightarrow[]{\substack{(x_1,a)\\(x_5,\overline{w})}} \\
T(x_1,x_4)|T(x_4,x_5) &\xrightarrow[]{\substack{(x_1,a)\\(x_5,\overline{w})}} \\
T(x_1,x_5) &\xrightarrow[]{\substack{(x_1,a)\\(x_5,\overline{p},x_1)}} \emptyset.
\end{aligned}
$$

Un arco che volesse *ascoltare* la sequenza di azioni dovrà emettere l'azione $\overline{a}$ su $x_1$. Notiamo che, in presenza di un tale arco, la sequenza di transizioni mostrata è l'unica possibile, poiché avviene una *sequenza* di sincronizzazioni lungo tutti i nodi che connettono gli archi $T$. Infatti se un arco $P$ emette $\overline{a}$ su $x_1$ allora $T(x_1,x_2)$ può utilizzare solo la produzione

$$T(x,y) \xrightarrow[]{\substack{(x,a)\\(y,\overline{b},x)}} \emptyset.$$

Questo *obbliga* $T(x_2,x_3)$ ad utilizzare

$$T(x,y) \xrightarrow[]{\substack{(x,b,z)\\(y,c)}} T(z,y).$$

Tutti gli altri archi nella sequenza *devono* quindi utilizzare la produzione:

$$T(x,y) \xrightarrow[]{\substack{(x,\overline{c})\\(y,c)}} T(x,y).$$

# Capitolo 3

# Synchronized Hypergraph Environment

*Syncronized Hypergraph Environment* (*SHE*) è un sistema di supporto alla programmazione distribuita basato sul modello *SG*. *SHE* fornisce ai propri utenti un ambiente in cui programmare graficamente un sistema distribuito specificando un insieme di produzioni *SG* e un ipergrafo di partenza. Da una specifica *SG* programmata dall'utente, *SHE* genera una specifica per il *model-checker* Mur$\varphi$ [D. 96], che calcola tutte le possibili riscritture e le organizza in un grafo degli stati, in cui un nodo rappresenta un ipergrafo e un arco tra due nodi una riscrittura dall'ipergrafo sorgente all'ipergrafo destinazione. Il sistema consente quindi di visualizzare il grafo degli stati esplorati, analizzarne il contenuto ed effettuare delle interrogazioni su di esso.

La programmazione con *SHE* è quasi esclusivamente visuale. Infatti l'utente *disegna* sullo schermo le produzioni e il grafo iniziale. Una volta generato il grafo di tutte le computazioni a partire dal dato grafo iniziale, l'utente, sempre attraverso l'uso di strumenti visuali, può esplorare il grafo degli stati

e analizzare il contenuto di ogni singolo nodo. La scelta di fare di *SHE* uno strumento di *programmazione visuale* deriva innanzi tutto da considerazioni sul calcolo *SG*. Infatti, poichè i termini di *SG* sono ipergrafi, la loro visualizzazione più immediata ci è parsa subito quella grafica, che consente di individuare immediatamente le caratteristiche topologiche di uno stato della computazione, come ad esempio il fatto che l'ipergrafo corrispondente sia sconnesso oppure che due particolari iperarchi condividano un nodo (ovvero che possano comunicare direttamente). Inoltre, un approccio visivo delle transizioni consente di analizzare la *struttura* della computazione, permettendo ad esempio di verificare velocemente la presenza di cicli, individuando quindi computazioni infinite.

In figura 3.1 è possibile vedere uno schema dell'architettura di *SHE*. Lo sviluppo di un'applicazione *SG* in *SHE* può essere suddiviso logicamente in tre fasi, che corrispondono ai moduli principali che compongono l'architettura del sistema:

- *composizione* di un insieme di produzioni e di un ipergrafo di partenza,
- *generazione* del grafo degli stati in base alle produzioni e all'ipergrafo specificati nella fase precedente,
- *esplorazione* del grafo degli stati.

Nella prima fase l'utente lavora attraverso un editor visuale, che produce dei file XML corrispondenti alle produzioni e al grafo iniziale descritti dall'utente.

La seconda fase è invece un programma UNIX da eseguire da linea di comando. Tale programma è costituito da una serie di moduli che lavorano

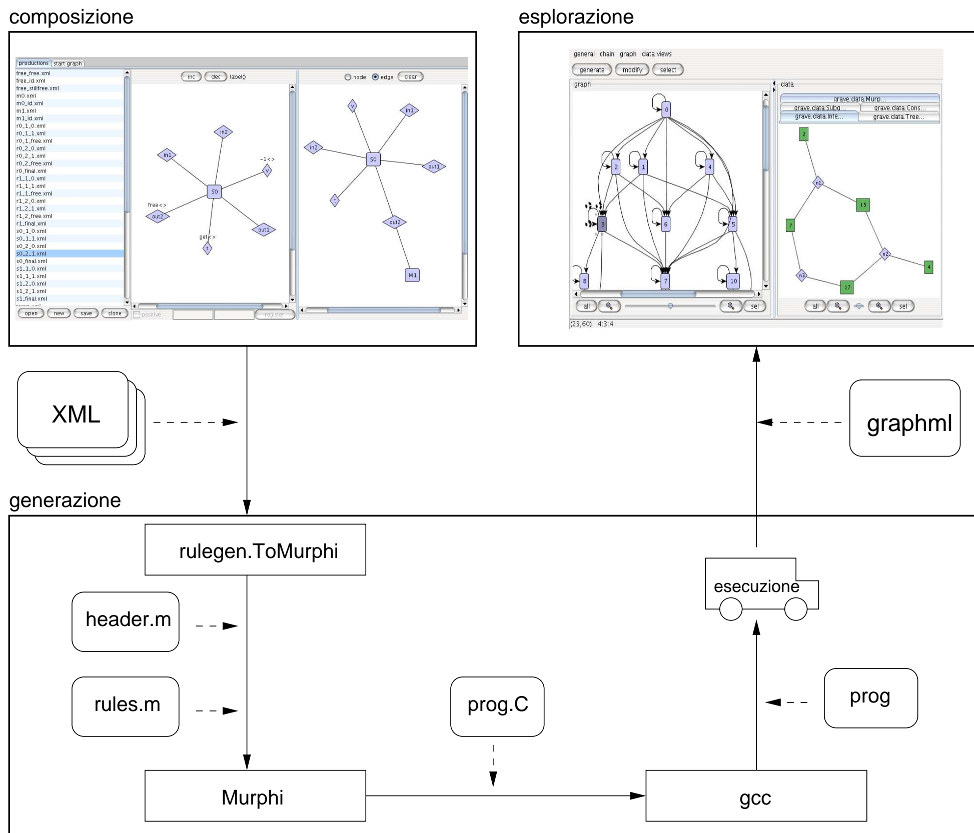



**Figura 3.1:** *Architettura di SHE*

scambiandosi file. Ogni modulo legge un file dal precedente e passa un file al successivo. Il primo modulo legge i file XML prodotti durante la fase di composizione. L'ultimo crea un file graphml che rappresenta il grafo degli stati calcolati.

Nella terza fase, il grafo prodotto nell'ultimo passo della fase di generazione viene visualizzato e l'utente può conoscere il contenuto dei singoli stati ed effettuare interrogazioni. Ad esempio può chiedere al sistema di visualizzare tutti gli stati finali della computazione, oppure effettuare la simulazione di una computazione.

Notiamo infine che l'architettura di *SHE* è stata progettata in maniera che sia possibile programmare in calcoli diversi da *SG* realizzando dei moduli appositi, capaci di tradurre altri calcoli in un insieme di produzioni e ipergrafi. Attualmente esistono già traduzioni in *SG* di alcuni calcoli noti: le traduzioni del calcolo dei *Mobile Ambients* [CG98] e del *DCCS* [RH01] sono fornite in [CTT04]; la traduzione del *Fusion Calculus* [PV98] è presente in [CT04]. Sarebbe quindi sufficiente realizzare un modulo che implementi queste traduzioni per ottenere con *SHE* un interprete per ciascuno dei suddetti calcoli.

Il resto del capitolo è strutturato nel seguente modo: nella sezione 3.1 illustriamo la composizione delle produzioni e dell'ipergrafo iniziale, nella sezione 3.2 mostriamo come viene generato il grafo degli stati e nella sezione 3.3 analizziamo come il programma *GraVE* viene utilizzato per effettuare l'analisi del grafo degli stati del sistema. *GraVE* viene analizzato come programma a sè stante nel capitolo 5.

## 3.1 Progettazione di un sistema in *SHE*

Ricordiamo brevemente che una produzione *SG* è una transizione in cui i nodi sono *metavariabili*, che vanno istanziate per ottenere una transizione, e in cui il lato sinistro è un arco. La composizione delle produzioni e del grafo iniziale avviene attraverso il programma Java rulegen.ProdBuilder. Le produzioni e l'ipergrafo prodotti in tale fase vengono memorizzati in dei file XML. Per avviare il programma è necessario eseguire il comando:

```
java rulegen.ProdBuilder <prod_1> ... <prod_n> <start-graph>
```

dove <prod_1> ... <prod_n> è una sequenza di file XML che rappresentano produzioni già esistenti e che l'utente desidera modificare. Tale sequenza può essere vuota. Il file <start-graph> è un file XML che rappresenta l'ipergrafo iniziale. Se <start-graph> non esiste il file verrà creato. Se <start-graph> è la stringa null l'editor non presenterà la parte relativa alla modifica dell'ipergrafo iniziale.

Descriveremo ora la composizione delle produzioni. Dato che una produzione include la specifica di due grafi (sorgente e destinazione), questa descrizione sussume quella della composizione del grafo iniziale, che verrà perciò tralasciata.

**Composizione delle produzioni.** L'interfaccia dell'editor delle produzioni è semplice e intuitiva. Uno schema dell'interfaccia è presente in figura 3.2. Si individuano tre parti:

- un elenco dei nomi dei file delle produzioni attualmente in uso (a sinistra),

**Figura 3.2:** *Schema dell'interfaccia per la composizione delle produzioni*

- un editor del lato sinistro della produzione attualmente selezionata (al centro),
- un editor del lato destro della produzione attualmente selezionata (a destra).

Ogni volta che l'utente seleziona una produzione dall'elenco, il suo lato destro e il suo lato sinistro vengono visualizzati negli appositi pannelli. É possibile aggiungere produzioni all'elenco in diversi modi:

- leggendole da file (bottone open),
- creandone una nuova (bottone new). In tal caso verrà chiesto il nome del file su cui andrà scritta la produzione,
- clonando la produzione visualizzata (bottone clone). In questo caso

verrà chiesto il nome del file su cui copiare la produzione correntemente visualizzata.

Durante l'editing delle produzioni e del grafo iniziale, i cambiamenti apportati alle produzioni dall'utente non vengono scritti immediatamente sui relativi file. Il salvataggio sui file avviene automaticamente alla chiusura del programma oppure quando l'utente preme il bottone save.

Nel pannello del lato sinistro della produzione è possibile modificare l'etichetta dell'arco, il numero e i nomi dei nodi sui quali incide. Selezionando un nodo è inoltre possibile specificare, nella sezione dell'interfaccia utente denominata azione, l'azione da emettere sul nodo selezionato e la lista di nodi da associare ad essa. L'azione e la lista di nodi associati ad un nodo vengono visualizzati in un'etichetta nella parte superiore del nodo. Quello che l'utente compone in questo pannello è dunque il lato sinistro della produzione e l'insieme $\Lambda$. In figura 3.3 è possibile vedere l'editor durante la composizione della produzione:

$$s(x, y, z, w) \xrightarrow[]{\substack{(x,\overline{b}) \\ (y,a,wx)}} s(x, y, z)|m(z)$$

Osserviamo che, in base a quanto visto fino ad ora l'utente potrebbe violare la condizione di *unificabilità* su $\Lambda$, non ottenendo quindi una produzione. Ad esempio potrebbe creare un arco $L(x, x)$ e associare l'azione $a$ ad entrambe le $x$, violando il secondo punto della condizione di unificabilità che richiede due azioni complementari. La violazione di tale condizione è tuttavia evitata dal sistema, che, ad ogni tentativo di modifica della produzione, controlla che la modifica *proposta* dall'utente sia consistente con la produzione. Se ciò è vero la modifica viene applicata alla produzione, altrimenti viene rifiutata.

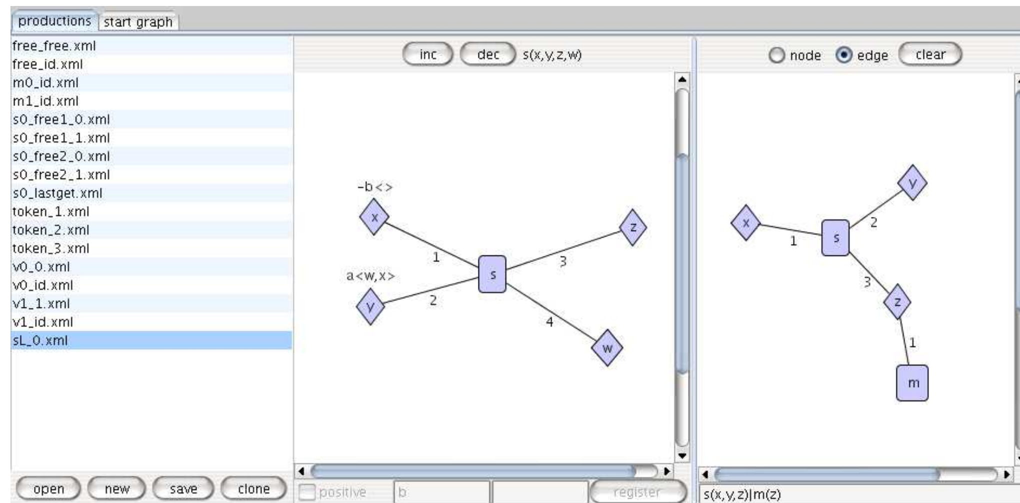



**Figura 3.3:** *Composizione di una produzione*

## 3.2 Generazione del grafo delle transizioni

In tale fase si genera, a partire dalle produzioni $\mathcal{P}$ e dall'ipergrafo $G$ prodotti dall'utente nella fase di composizione, il grafo di tutte le possibili transizioni a partire da $G$. Il programma destinato a tale compito è lo *shell script* denominato genall.sh. Tale programma prende come argomenti i file XML che rappresentano le produzioni e l'ipergrafo iniziale e, dopo una serie di passi intermedi, produce un file in formato graphml che rappresenta il grafo delle transizioni.

I passi intermedi compiuti da tale programma sono:

- creazione di una specifica Mur$\varphi$ a partire dai file XML delle produzioni e dell'ipergrafo iniziale. Tale specifica viene generata combinando l'ouput del programma rulegen.ToMurphi, che è in grado di tradurre le definizioni XML di $\mathcal{P}$ e $G$ in un insieme di dichiarazioni Mur$\varphi$, con il

file header.m, che contiene definizioni di costanti, funzioni e regole da inserire in ogni specifica Mur$\varphi$.

- Esecuzione del programma mu, che rappresenta il programma Mur$\varphi$ che genera un programma C++ a partire da una specifica Mur$\varphi$, con la specifica generata al passo precedente come argomento.
- Compilazione del programma C++ per la generazione del model-checker.
- Esecuzione del model-checker.

Utilizziamo una versione di Mur$\varphi$ modificata, la quale genera un model-checker che restituisce in output il grafo degli stati esplorati direttamente in formato graphml (per maggiori dettagli vedi 4.3.3). Quindi, nell'ultimo passo, genall.sh memorizza l'output del model-checker in un file, e comunica a schermo il nome di tale file.

## 3.3 Esplorazione del grafo delle transizioni

L'esplorazione del grafo degli stati prodotto nella fase di generazione avviene mediante il programma *GraVE*. Per avviarlo è necessario eseguire il comando shegrave.sh. *GraVE* è un programma sviluppato come componente di *SHE* e che qui descriveremo in tale contesto. Tuttavia *GraVE* è a tutti gli effetti un programma a sé con caratteristiche di espandibilità e generalità tali da meritare una trattazione più approfondita (capitolo 5). Qui ne considereremo soltanto alcune funzionalità specifiche utilizzate nel contesto di *SHE*.

In figura 3.4 è presente uno schema dell'interfaccia utente del programma. All'avvio il programma si presenta con la schermata in figura 3.5. Per scegliere il file degli stati da esplorare è necessario premere il bottone generate

**Figura 3.4:** *Schema dell'interfaccia*

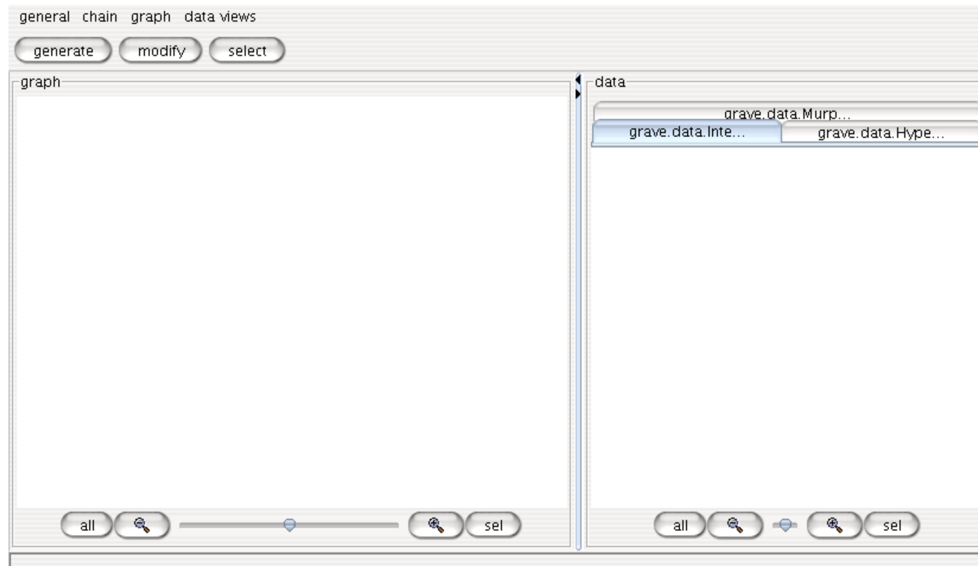



**Figura 3.5:** *Schermata principale di GraVE*

oppure selezionare nel menu la voce general→generate. Dopo l'apertura del file, il risultato sarà simile alla schermata in figura 3.6. Nella parte sinistra

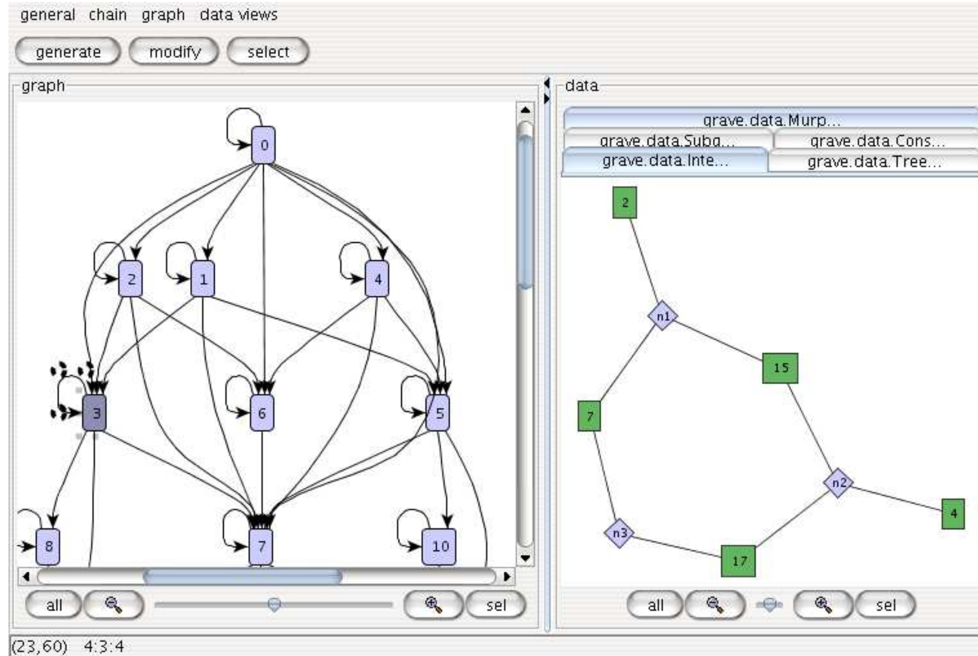



**Figura 3.6:** *GraVE durante l'esplorazione di uno stato della computazione*

di *GraVE* è possibile vedere il grafo degli stati. Un nodo rappresenta un ipergrafo e un arco tra due nodi l'esistenza di una riscrittura dall'ipergrafo rappresentato dal nodo di partenza a quello rappresentato dal nodo destinazione. Il nodo che rappresenta l'ipergrafo di partenza è etichettato con 0. Nella parte inferiore della finestra principale del programma è presente una *barra di stato* che indica con l'espressione (n,m):

- il numero n di nodi del grafo degli stati,
- il numero m di archi del grafo degli stati.

Nel caso in cui l'utente seleziona un nodo nel grafo degli stati, lo stato di Mur$\varphi$ corrispondente viene convertito in ipergrafo e visualizzato nel pannello di destra di *GraVE* e nella barra di stato viene aggiunta l'espressione in:lab:out dove:

- lab indica l'etichetta del nodo selezionato,
- in indica il numero di archi entranti nel nodo e
- out indica il numero di archi uscenti dal nodo.

In figura 3.6 è stato selezionato il nodo etichettato 3 nel grafo degli stati. Nella parte destra del programma viene quindi mostrato l'ipergrafo corrispondente a tale nodo, e nella barra di stato viene visualizzato (23,60), rispettivamente il numero di nodi e di archi del grafo degli stati, e 4:3:4, poichè il nodo etichettato con 3 ha quattro archi entranti e quattro archi uscenti.

Sebbene di default l'ipergrafo corrispondente ad uno stato della computazione venga visualizzato in modalità grafica, *SHE* fornisce altre due modalità di visualizzazione degli ipergrafi:

- una modalità in cui viene visualizzato lo stato *grezzo* di Mur$\varphi$ in modalità testuale e
- una modalità in cui l'ipergrafo viene visualizzato come documento graphml.

Queste modalità possono essere selezionate mediante una serie di *tabs* presenti nel pannello dedicato alla visualizzazione dell'ipergrafo, come è possibile vedere in figura 3.7.

**Figura 3.7:** *Le diverse modalità di visualizzazione di un ipergrafo*

Un aspetto interessante è che l'utente può programmare in Java altre modalità di visualizzazione, sia del grafo degli stati che degli ipergrafi, e caricarle all'interno di *GraVE*. Per maggiori dettagli riguardo a questo aspetto rimandiamo al capitolo 5 dedicato a *GraVE*.

Oltre a consentire di selezionare nodi del grafo degli stati attraverso l'uso del cursore, *GraVE* mette a disposizione dell'utente una funzionalità per la ricerca di nodi attraverso l'elaborazione di espressioni. Mediante tale funzionalità l'utente elabora delle espressioni booleane sui nodi e il sistema seleziona nel grafo degli stati tutti i nodi che soddisfano l'espressione data. La composizione di tali espressioni può avvenire sia con un linguaggio *testuale* che attraverso uno *grafico*. Per selezionare tale funzione è necessario selezionare il bottone select nella finestra principale di *GraVE*. Poichè tale funzionalità non è legata esclusivamente a *SHE*, la sua descrizione dettagliata è affrontata nel capitolo dedicato a *GraVE* (vedi 5.4).

# Capitolo 4

# Implementazione della riscrittura

In questo capitolo introduciamo Mur$\varphi$ [MU, DDHY92, D. 96, PITZ02], lo strumento utilizzato da *SHE* per implementare le riscritture di ipergrafi e descriviamo come è stata realizzata tale implementazione.

Mur$\varphi$ è uno strumento per il *model-checking* di sistemi a stati finiti. Per modellare un sistema all'interno di Mur$\varphi$ è necessario descriverlo attraverso un insieme di variabili, che ne rappresentano lo stato, e un insieme di *regole* che descrivono l'evoluzione del sistema. Una regola è composta da una condizione di applicabilità e da una sequenza di istruzioni che effettuano delle modifiche alle variabili che rappresentano lo stato. Data la descrizione di un sistema e di uno stato iniziale, Mur$\varphi$ è in grado di generare un programma che effettua l'esplorazione esaustiva di tutti gli stati che possono essere raggiunti mediante l'applicazione delle regole date a partire dallo stato iniziale. L'aspetto interessante di Mur$\varphi$ è che l'evoluzione di un sistema è descritta in maniera *dichiarativa* nella specifica. Infatti è necessario descrivere quando la regola può essere applicata e cosa deve effettuare la regola. Sarà poi Mur$\varphi$ a prendersi carico di effettuare i controlli e applicare le regole. Questa serie

di fattori ci ha fatto scegliere Mur$\varphi$ come strumento per modellare computazioni $SG$. Data un ipergrafo $G$ e un insieme di produzioni $\mathcal{P}$, l'idea è quella di fornire una specifica Mur$\varphi$ in cui lo stato iniziale è rappresentato dall'ipergrafo $G$ e in cui esiste una regola che rappresenta ciascuna produzione $P \in \mathcal{P}$. In queste regole la condizione di applicabilità deve verificare due condizioni:

- che l'arco che compare nel lato sinistro di $P$ possa essere istanziato per ottenere un arco del grafo,
- che l'applicazione della regola corrispondente alla produzione $P$ non violi la *condizione di unificabilità* su $\Lambda$ descritta a pagina 14.

L'evoluzione di un ipergrafo composto da $n$ iperarchi consiste nell'applicazione di $n$ regole, una per ogni arco presente nel grafo. Infine, quando tutti gli archi presenti nel grafo sono stati riscritti, viene applicata una regola speciale, denominata sync, che si occupa di unificare le liste di nodi comunicate durante le sincronizzazioni.

Osserviamo come, mediante il meccanismo descritto, una *singola* riscrittura di un ipergrafo viene realizzata in Mur$\varphi$ attraverso una serie di passi sequenziali. Ciò deriva da una differenza fondamentale tra $SG$ e Mur$\varphi$: il primo è un modello *concorrente* mentre Mur$\varphi$ applica le regole *sequenzialmente*. Come vedremo in 4.3.2, ciò causa in Mur$\varphi$ l'esplorazione di un numero molto elevato di stati *inutili* ai fini del calcolo delle transizioni di un sistema $SG$. Poiché dall'output di Mur$\varphi$ viene ricostruito il grafo degli stati di una computazione, l'elevato numero di stati inutili era un problema particolarmente oneroso, sia in termini di tempo che di spazio. Per tale problema sono state trovate soluzioni a diversi livelli: innanzi tutto è stato modificato il codice

sorgente di Mur$\varphi$, in maniera da generare un ouput che contenesse il contenuto completo soltanto degli stati validi ai fini di *SG*, ma informazioni su *tutte* le transizioni di stato effettuate. Praticamente viene costruito un grafo con informazioni su tutti gli archi e i nodi della computazione, ma viene fornito il contenuto di uno stato soltanto se questo è uno stato valido. É poi stato creato, all'interno di *GraVE*, un filtro che, a partire dal grafo generato con la versione modificata di Mur$\varphi$, è in grado di generare il grafo degli stati validi.

Notiamo che, sebbene l'implementazione di *SG* in Mur$\varphi$ si sia rivelata abbastanza semplice e intuitiva, tuttavia l'estremo grado di concorrenza di *SG* genera grafi degli stati di Mur$\varphi$ in cui solo una piccola porzione rappresenta effettivamente computazioni *SG*. A questo proposito si osservi che, data la struttura modulare di *SHE*, è possibile, nel caso in cui si trovasse un'implementazione di *SG* più efficiente di quella fornita (e che eventualmente utilizzi uno strumento diverso da Mur$\varphi$), integrare facilmente la nuova implementazione all'interno di *SHE*.

Nel resto del capitolo, dopo aver introdotto Mur$\varphi$, mostriamo come viene implementato in Mur$\varphi$ il modello *SG*, analizzando in dettaglio il problema degli stati validi ai fini di *SG* e concludiamo illustrando due versioni modificate di Mur$\varphi$, che abbiamo realizzato per generare un output in formato graphml, discutendone i rispettivi vantaggi e campi d'applicazione.

## 4.1 Mur$\varphi$

Mur$\varphi$ è uno strumento per il *model checking* di sistemi a stati finiti. É stato ampliamente utilizzato nella verifica di diversi protocolli industriali, soprattutto nell'ambito dei protocolli di *cache coherence*, di modelli per la memoria multiprocessore e di protocolli di sicurezza [SD95, LGM+95, MMS97, Mit98].

La tecnica utilizzata da Mur$\varphi$ per la verifica di un sistema è quella dell'esplorazione esaustiva di tutti i possibili stati raggiugibili da un dato stato iniziale.

Mur$\varphi$ è composto essenzialmente da due parti: un linguaggio e un compilatore. Il linguaggio (ispirato al modello Unity di [CM88]) consente di descrivere le componenti di un sistema e le sue possibili evoluzioni. La sintassi é simile a quella dei tradizionali linguaggi imperativi tipati, e include una serie di tipi base e un insieme di operatori per la creazione di tipi complessi.

Uno *stato* per Mur$\varphi$ è un assegnamento di valori alle variabili dichiarate all'inizio della specifica. Una variazione di stato corrisponde quindi ad un diverso assegnamento di valori. La descrizione delle variazioni di stato in Mur$\varphi$ avviene attraverso l'uso di regole chiamate *Rule*. Una regola è costituita da due parti: un'espressione booleana che rappresenta una condizione di applicabilità, e l'azione vera e propria associata ad essa, ossia del codice che effettua calcoli ed eventualmente modifica il valore delle variabili, cioé lo stato del sistema. L'applicazione di una regola va vista come un'azione atomica, cioè tale che lo stato del sistema sia osservabile soltanto *prima* o *dopo* tale applicazione ma non *durante*.

Il compilatore di Mur$\varphi$ è la componente che, prendendo in input un programma Mur$\varphi$, genera il *model checker*, ossia un programma C++ che realizza effettivamente l'esplorazione esaustiva degli stati del sistema, dando all'utente la possibilità di scegliere tra l'esplorazione in ampiezza e quella in profondità.

Essendo Mur$\varphi$ concepito principalmente per la verifica di sistemi, esso fornisce anche la possibilità di esprimere condizioni dette *invarianti*. Un'invariante è un'espressione booleana che deve essere vera in ogni stato del sistema. Nel caso in cui un'invariante fosse violata da uno stato $s$ Mur$\varphi$ lo

segnalerà, bloccando l'esplorazione dello spazio degli stati e mostrando una computazione che, dallo stato iniziale, conduca ad $s$. Enfatizziamo che le invarianti devono essere valide *prima* e *dopo* l'esecuzione di un'azione di una regola, ma non durante.

Poiché *SHE* è un sistema orientato alla verifica di sistemi avevamo inizialmente pensato di effettuare tali verifiche mediante l'uso delle invarianti messe a disposizione da Mur$\varphi$. Tale soluzione è stata però abbandonata per due motivi. Innanzitutto le invarianti consentono di effettuare verifiche solo sul contenuto di ciascuno stato e non sulle relazioni tra essi, rendendo quindi impossibile esprimere ad esempio la proprietà che in tutti gli stati finali della computazione ci sia sempre un arco con una certa etichetta. Inoltre, l'uso delle invarianti non è adatto ad una verifica interattiva delle proprietà di un sistema poiché, per ogni nuova invariante inserita nel sistema, sarebbe necessario generare da capo il model checker. L'utente dovrebbe quindi conoscere a priori tutte le proprietà che desidera verificare. Per tali motivi la verifica di proprietà della computazione avviene all'interno di *GraVE*, il programma per l'esplorazione degli stati di una computazione *SG*.

## 4.2 Riscrittura di ipergrafi

Mur$\varphi$ è utilizzato all'interno di *SHE* per simulare computazioni *SG*. In particolare, abbiamo scritto un programma Java, rulegen.ToMurphi, che, preso in input un grafo $G$ e un insieme di produzioni $\mathcal{P}$, genera un programma Mur$\varphi$ che ha come stato iniziale la rappresentazione di $G$ e un insieme di regole tali che il *model checker* esplori tutte le possibili computazioni di $G$.

Procederemo ora ad illustrare le strutture dati utilizzate all'interno di tali programmi Mur$\varphi$ e successivamente la strategia di esplorazione. La trattazio-

ne seguirà un'approccio top-down, mostrando il processo che ci ha condotti alle scelte effettuate.

Nel seguito ci troveremo a volte a dire, per semplicitá di discorso e nei casi in cui non si generi ambiguitá, *grafo, arco, nodo, azione, ...* invece di *la rappresentazione in Mur$\varphi$ di un grafo, di un arco, di un nodo, di un'azione, ....*

### 4.2.1 Rappresentazione dei grafi

Osserviamo preliminarmente che Mur$\varphi$ non permette la creazione di strutture dati dinamiche: non esistono puntatori nè array di dimensione variabile. Di conseguenza le dimensioni delle strutture dati saranno limitate da una serie di costanti specificate nel programma, ad esempio `NUM_OF_EDGES` rappresenta il numero massimo di archi presenti nel grafo. Dato un grafo $G$, assumiamo dunque di utilizzare per le nostre costanti un insieme di valori sufficiente a rappresentare $G$ e le sue possibili transizioni. Nel seguito assumiamo inoltre che il lettore abbia familiarità con i tipi di dato *array* e *record*, che sono state utilizzare per modellare $SG$ in Mur$\varphi$.

L'idea alla base della nostra struttura dati è di rappresentare un grafo come un array di archi, e un arco come una tripla composta da:

- un identificativo *unico* all'interno del grafo,
- un'etichetta e
- l'insieme di nodi su cui l'arco incide.

Il tipo di dato che descrive un *grafo* é `graph_t`. Esso é composto quindi da un array di archi e un contatore degli archi effettivamente presenti nel grafo:

```
graph_t:Record
edges:Array[1..NUM_OF_EDGES] of edge_t;
num_of_edges:0..NUM_OF_EDGES;
End;
```

Va notato che pur non essendo il contatore strattamente indispensabile, lo si é introdotto per ridurre il costo computazionale di dover scorrere la lista di archi ogni qualvolta fosse necessario conoscerne la lunghezza.

Un *arco* é descritto dal tipo `edge_t` che comprende l'identificatore dell'arco (unico all'interno del grafo), la sua etichetta e l'array di nodi su cui esso incide:

```
edge_t:Record
id:0..MAX_EDGE_ID;
label:0..MAX_EDGE_LABEL;
nodes:Array[1..MAX_NODES_PER_EDGE] of node_t;
End;
```

Poiché non possiamo descrivere strutture dinamiche in Mur$\varphi$, il campo `label` ha un duplice significato. Quando è uguale a zero esso indica l'assenza del corrispondente arco, mentre valori diversi da zero rappresentano effettivamente le etichette degli archi. Osserviamo inoltre che si é deciso di utilizzare un subrange del tipo intero per le etichette poiché il linguaggio di Mur$\varphi$ non fornisce il tipo di dato *stringa*.

Il tipo di un *nodo* é un intero:

```
node_t:-NUM_OF_NODES..NUM_OF_NODES;
```

Anche in questo caso lo zero indica l'assenza di un nodo.

Passiamo ora a descrivere la struttura dati per rappresentare le azioni sui nodi.

Il tipo corrispondente ad un'azione su un nodo é `act_t`:

```
act_t:Record
sign:-1..1;
a1:0..MAX_ACT;
a2:0..MAX_ACT;
args1:Array[1..ACT_ARGS_LEN] of node_t;
args2:Array[1..ACT_ARGS_LEN] of node_t;
arglen:0..ACT_ARGS_LEN;
End;
```

L'insieme delle azioni su tutti i nodi del grafo é un array di `act_t`:

```
acts_t:Array[1..NUM_OF_NODES] of act_t;
```

Per comprendere meglio le scelte riguardanti `act_t` é necessario conoscere la strategia utilizzata per rappresentare l'evoluzione dei termini di *SG*. Rimandiamo quindi la descrizione di `act_t` fino al momento in cui tale strategia verrá illustrata (vedi 4.2.2).

Ecco qualche esempio delle strutture dati appena descritte. La rappresentazione dell'arco $1(2,3)$ è la seguente, in cui è stato rimosso il campo `id` per semplicità:

| label: | 1 | | | | | |
|---|---|---|---|---|---|---|
| nodes: | 2 | 3 | 0 | 0 | ... | 0 |

La rappresentazione in Mur$\varphi$ del grafo $1(2,3)|4(3)|5(2,4,3)$, raffigurato in figura 4.1, è fornita in figura 4.2, in cui è stato rimosso il campo `id` degli archi per semplicità.

### 4.2.2 Strategia di riscrittura

Il problema principale dell'implementazione della riscrittura degli ipergrafi in Mur$\varphi$ riguarda una differenza fondamentale tra i *SG* e Mur$\varphi$: il primo é

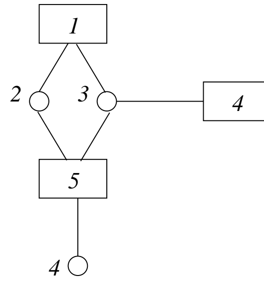



**Figura 4.1:** *Il grafo* $1(2,3)|4(3)|5(2,4,3)$

un modello *concorrente* mentre Mur$\varphi$ applica le regole *sequenzialmente*. L'idea alla base della nostra implementazione é quindi quella di rappresentare una produzione di $SG$ mediante una regola di Mur$\varphi$ e una transizione di $SG$ con una sequenza di applicazioni di regole. In particolare la riscrittura di un grafo con $n$ archi viene simulata in Mur$\varphi$ con l'applicazione delle $n$ regole corrispondenti alle $n$ produzioni utilizzate durante la riscrittura, più un'ultima regola, denominata sync, utilizzata per l'unificazione dei nodi comunicati durante le sincronizzazioni.

Nel dettaglio, un programma Mur$\varphi$ comprende due variabili di tipo graph_t, denominate g e gtemp. La variabile g rappresenta il grafo da riscrivere ed ogni regola corrispondente ad una produzione elimina il lato sinistro da g e inserisce il lato destro in gtemp. Quando il grafo g é vuoto, ossia g.num_of_edges=0, l'intero grafo é stato riscritto, e il risultato di tale riscrittura si trova in gtemp. A questo punto viene eseguita la sync, che provvede a copiare il contenuto di gtemp in g, e ad unificare i nodi in base alle azioni compiute nel corso della riscrittura. Quindi per riscrivere completamente un grafo con $n$ archi sono necessarie $n + 1$ applicazioni di regole, ossia $n + 1$ transizioni di stato di Mur$\varphi$.

| edges[0]: | label: 1<br>nodes: 2 3 0 0 ... 0 |
|---|---|
| edges[1]: | label: 4<br>nodes: 3 0 0 0 ... 0 |
| edges[2]: | label: 5<br>nodes: 2 4 3 0 ... 0 |
| edges[3]: | label: 0<br>nodes: 0 0 0 0 ... 0 |
| ⋮ | ⋮ |
| edges[NUM_OF_EDGES]: | label: 0<br>nodes: 0 0 0 0 ... 0 |
| num_of_edges: | 3 |

**Figura 4.2:** *Il grafo* $1(2,3)|4(3)|5(2,4,3)$ *in Mur*$\varphi$

In figura 4.3 è possibile vedere uno schema di come viene simulata in Mur$\varphi$ una semplice riscrittura in cui non vengono emesse azioni sui nodi. Le variabili che subiscono delle modifiche sono mostrate in grigio sia in questo esempio che nei seguenti.

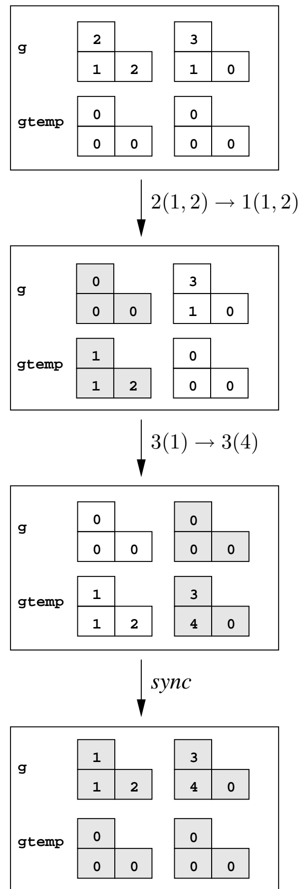



**Figura 4.3:** $2(1,2)|3(1) \rightarrow 1(1,2)|3(4)$

L'azione di ogni regola, oltre ad effettuare la riscrittura del grafo, memorizza le azioni che vengono compiute sui nodi e le liste di nodi che vengo-

no comunicate. Queste informazioni vengono memorizzate in una variabile denominata acts il cui tipo é:

```
acts_t:Array[1..NUM_OF_NODES] of act_t;
```

L'i-esimo elemento dell'array ci informa sulle azioni avvenute sul i-esimo nodo. L'insieme $Act = \{a, b, \ldots\} \cup \{\overline{a}, \overline{b}, \ldots\}$ delle azioni e co-azioni in $SG$ viene realizzato in Mur$\varphi$ rappresentando le azioni con interi positivi e le co-azioni con interi negativi, in cui la complementare dell'azione x é l'azione -x (é quindi mantenuta la proprietá che $\overline{\overline{a}} = a$).

Il campo sign di act_t puó assumere tre valori:

- -1: se sul nodo é stata compiuta un'azione *negativa*
- 1: se sul nodo é stata compiuta un'azione *positiva*
- 0: se sul nodo non é stata compiuta alcuna azione oppure sono state compiute sia un'azione *positiva* che una *negativa*

I campi a1 e a2 sono destinati entrambi a contenere il valore assoluto dell'azione compiuta sul nodo. Il primo viene modificato quando viene eseguita un'azione positiva, il secondo in caso di azione negativa. Il due array args1 e args2 contengono invece le liste di nodi comunicati durante un'azione. Il primo viene modificato nel caso di azione positiva, il secondo in caso di azione negativa. Infine arglen contiene la lunghezza di tali liste. Notiamo che, pur non essendo il campo arglen indispensabile, lo si é introdotto per ridurre il costo computazionale di dover scorrere una delle due liste (args1 e args2) ogni qualvolta fosse necessario conoscere la loro lunghezza.

La figura 4.4 mostra la simulazione in Mur$\varphi$ di una riscrittura in cui vengono emesse delle azioni sui nodi.

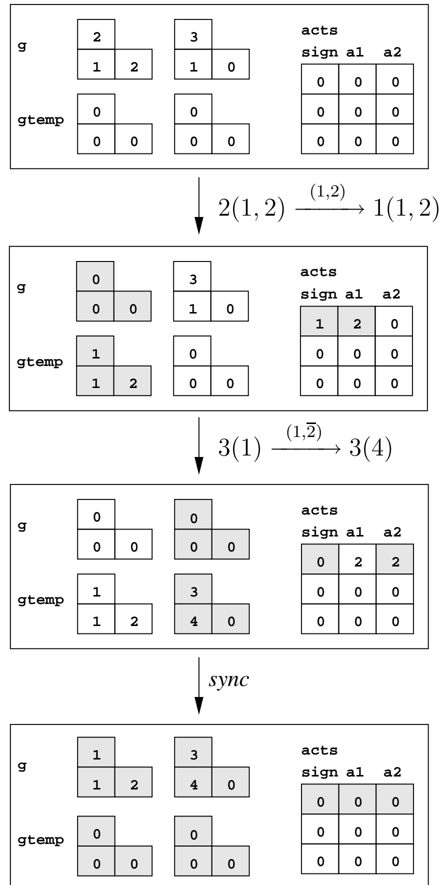



**Figura 4.4:** $2(1,2)|3(1) \xrightarrow{(1,2),(1,\overline{2})} 1(1,2)|3(4)$

I valori della variabile acts vengono *modificati* durante l'applicazione della regola corrispondente ad una produzione e vengono inoltre analizzati nelle condizioni di applicabilità di tali regole. Consideriamo ad esempio la produzione

$$1(x, y) \xrightarrow{(x,\overline{2},z)} 2(z)$$

Affinchè la regola corrispondente a questa produzione sia applicabile, è necessario che, sul primo nodo su cui incide l'arco 1 non sia stata compiuta nessuna azione, oppure sia stata compiuta l'azione 2 e che la lista di nodi comunicata in tal caso abbia lunghezza uguale a uno. Tali controlli vengono effettuati all'interno della condizione di applicabilità della regola corrispondente alla produzione, attraverso i valori memorizzati nell'array acts.

Quando il grafo g é stato interamente riscritto viene eseguita la regola sync. A tal punto g é vuoto e il risultato delle riscritture avvenute é contenuto in gtemp. In questa regola si copia da gtemp a g il risultato della riscrittura (con la funzione copygraph) e lo si elimina da gtemp. Successivamente avviene l'unificazione dei nodi comunicati durante le sincronizzazioni (mediante la funzione unifyGraph) e infine si provvede alla loro *semplificazione* (mediante la funzione nodeSimplify). Ció che la funzione nodeSimplify implementa é un algoritmo che *compatta* l'intervallo di nodi usati in g, ossia se sono utilizzati $n$ nodi, la funzione nodeSimplify garantisce che, dopo la sua chiamata, g utilizzerá i nodi da 1 a n.

## 4.3 Stategie di riduzione degli stati

In questa sezione mostriamo come l'implementazione di *SG* proposta presenti un esplosione nel numero di stati eplorati da Mur$\varphi$, in cui solo una piccola

parte è necessaria per ricostruire computazione $SG$. Un'analisi della strategia di esplorazione di Mur$\varphi$ e dell'implementazione di $SG$ in Mur$\varphi$ ha consentito di modificare parzialmente le regole e diminuire notevolmente il numero di stati esplorati senza perdere computazioni $SG$.

Inoltre, per evitare di dover ricostruire il grafo degli stati a partire dall'output di Mur$\varphi$, abbiamo effettuato delle modifiche direttamente al codice sorgente di Mur$\varphi$, realizzandone due diverse versioni che producono in output un grafo degli stati in formato graphml. Una versione è adatta a qualsiasi programma Mur$\varphi$, l'altra è specializzata per la rappresentazione di computazioni $SG$.

### 4.3.1 Il problema della duplicazione degli stati

Ricordiamo brevemente che uno stato in Mur$\varphi$ è un assegnamento di valori alle variabili dichiarate all'inizio della specifica, e quindi due stati sono diversi se differiscono i valori assegnati alle variabili di stato. Poichè la nostra rappresentazione di un grafo consiste semplicemente in un array di archi, ció significa che ordinando diversamente tale array rappresentiamo lo stesso grafo ma otteniamo uno stato *diverso* in Mur$\varphi$. In particolare, data la variabile g presente in ogni programma Mur$\varphi$ per la simulazione di $SG$, indichiamo con $n$ il valore del campo g.num_of_edges e con $m$ il valore della costante NUM_OF_EDGES. il numero di stati Mur$\varphi$ differenti, ma che rappresentano uno stesso ipergrafo è almeno:

$$\frac{m!}{(m-n)!} \tag{4.1}$$

Inoltre, poichè il nostro stato non è composto solo da una variabile di tipo graph_t, bensì da due di queste variabili (g e gtemp), piú la struttura

dati destinata alle informazioni sulle azioni, lo stesso ipergrafo di $SG$ sarà rappresentato da un numero di stati in Mur$\varphi$ ben superiore a (4.1).

Consideriamo ad esempio il grafo $G = 1(2,3)|2(1,3)$ e immaginiamo che nell'array che rappresenta $G$ l'arco etichettato con 1 compaia prima dell'arco etichettato con 2. Indichiamo sinteticamente tale array con (1(2,3),2(1,3)). Supponiamo di avere due produzioni identità per i due archi, ossia le produzioni $1(2,3) \longrightarrow 1(2,3)$ e $2(1,3) \longrightarrow 2(1,3)$, che danno quindi luogo ad altrettante regole Mur$\varphi$. Poichè Mur$\varphi$ effettua un'esplorazione esaustiva di *tutti* gli stati, dallo stato (1(2,3),1(2,3)), applicando prima la produzione identità dell'arco $1(2,3)$ e poi quella dell'arco $2(1,3)$ otterremo l'array (1(2,3),2(1,3)). Applicandole nell'ordine inverso otterremo invece l'array (2(1,3),1(2,3)), che rappresentano due stati *diversi* per Mur$\varphi$.

Una prima stategia di soluzione é stata quella di sfruttare quelli che in Mur$\varphi$ sono chiamati *scalarset* [CD93], che permettono di considerare uguali array che differiscano solo per l'ordinamento. Tale soluzione ha permesso di ridurre notevolmente il numero di stati esplorati da Mur$\varphi$ ma ci siamo accorti che tale numero era ancora troppo elevato e poteva essere migliorato.

Consideriamo ad esempio il seguente grafo di partenza:

$$2(1,2)|3(4) \tag{4.2}$$

e le seguenti produzioni:

$$2(1,2) \xrightarrow{(1,3,<>)} 1(1,2) \tag{4.3}$$

$$3(3) \xrightarrow{(3,5,<>)} 3(3) \tag{4.4}$$

Applicando le produzioni (4.3) e (4.4) al grafo (4.2) otterremmo il grafo $1(1,2)|3(3)$. Questa transizione verrebbe rappresentata in Mur$\varphi$, facendo

uso degli *scalarset*, dall'evoluzione in figura 4.5 (alcune variabili sono state omesse per semplicità).

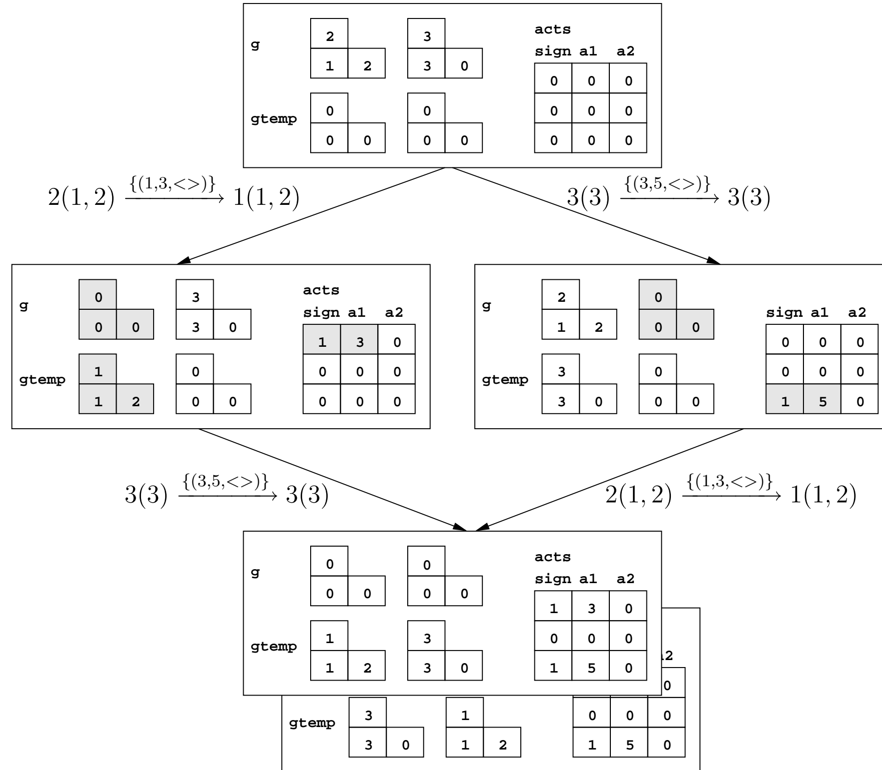



**Figura 4.5:** *Riscrittura con gli scalarset*

Dalla figura è dunque possibile notare che anche utilizzando gli *scalarset* (che permettono di equagliare i due stati finali delle due riscritture) si generano due stati intermedi diversi, corrispondenti al diverso ordine di applicazione della riscrittura. E se, nel caso di due soli archi, si generano solo due stati intermedi, in generale tale numero risulta molto elevato, a causa dei diversi ordini di applicazione di un insieme di regole.

Per risolvere questo problema si è pensato dunque di *guidare* la riscrittura,

facendo in modo che tra i vari possibili *percorsi* di riscrittura equivalenti ne venisse scelto sempre solo uno, riducendo ad $n$ il numero di stati intermedi incontrati nel corso di una riscrittura di un grafo con $n$ archi. Per fare ciò si è imposto a Mur$\varphi$ di effettuare sempre la riscrittura dell'arco presente nell'ultima posizione dell'array e di mantenere, ad ogni passo, l'array ordinato in maniera crescente in base al campo `id` degli archi.

Tuttavia tale metodo non può essere utilizzato con strutture dati di tipo *scalarset* poichè «symmetry-breaking constructs are outlawed for scalarset values» [CD93]. Si é quindi deciso di ordinare anche `gtemp` all'interno di ogni regola in maniera da non avere mai stati in cui le variabili di tipo `graph_t` differiscano solo per l'ordinamento.

Nonostante le diverse ottimizzazioni volte a ridurre lo stazio degli stati esplorati da Mur$\varphi$, anche per implementare una singola riscrittura il numero di stati esplorato è comunque elevato.

Consideriamo ad esempio il grafo $1(x)|2(x,y)|3(y)$ e le seguenti produzioni:

- $1(x) \xrightarrow{(x,n)} 1(x)$ con $n \in \{a,b,c\}$,
- $3(x) \xrightarrow{(x,\overline{n})} 3(x)$ con $n \in \{a,b,c\}$,
- $2(x,y) \xrightarrow{\substack{(x,\overline{a})\\(y,a)}} 2(x,y)$.

L'unica transizione possibile è:

$$1(x)|2(x,y)|3(y) \xrightarrow{\substack{(x,a)\\(x,\overline{a})\\(y,\overline{a})\\(y,a)}} 1(x)|2(x,y)|3(y).$$

Tuttavia, nel calcolo di tale transizione, Mur$\varphi$ esplorerà anche molti altri stati, inutili ai fini di *SG*. Supponendo che $1(x)$ sia il primo arco ad essere

riscritto, Mur$\varphi$ applica tre regole, una per ogni produzione di $1(x)$. In tutti e tre gli stati raggiunti dopo l'applicazione delle suddette regole, supponiamo che l'arco da riscrivere sia $3(y)$. A questo punto tutte e tre le regole corrispondenti alle produzioni di $3(y)$ possono essere applicate. Infatti $3(y)$ e $1(x)$ non hanno nodi in comune, quindi le reciproche produzioni non possono generare conflitti. Dall'applicazione delle tre regole per $3(y)$ otteniamo quindi 9 stati. Infine andrà riscritto l'arco $2(x, y)$. A questo punto la regola corrispondente all'unica produzione di $2(x, y)$ può essere applicata solo nello stato raggiunto dopo l'applicazione delle regole corrispondenti alle produzioni $1(x) \xrightarrow{(x,a)} 1(x)$ e $3(x) \xrightarrow{(x,\overline{a})} 3(x)$. Notiamo quindi che sono stati calcolati in totale 13 stati Mur$\varphi$ (3 dopo le regole di $1(x)$, 9 dopo le regole di $3(y)$ e uno solo dopo la regola di $2(x, y)$), di cui solo 3 utili per calcolare una transizione $SG$. Sarebbe quindi interessante riuscire ad implementare una strategia di esplorazione degli stati più efficace di quella proposta. Singole soluzioni possono essere trovate per i casi specifici. Ad esempio nel caso appena visto, se l'ordine di scelta degli archi da riscrivere *seguisse* l'ordine in cui gli archi sono connessi in sequenza si riuscirebbe a diminuire il numero di stati esplorati. In generale appare però difficile individuare una strategia che si possa applicare al caso generale, essendo dipendente sia dalla topologia del grafo che dalle produzioni fornite.

### 4.3.2 Stati validi

Poiché la natura concorrente di $SG$ viene implementata attraverso una serie di passi sequenziali in Mur$\varphi$, gli unici stati di Mur$\varphi$ che hanno un corrispondente in $SG$ sono lo stato iniziale e tutti gli stati ottenuti dall'applicazione della regola `sync`, e sono detti stati *validi*. Per tale motivo è stata aggiunta,

all'insieme delle variabili di stato, la variabile booleana valid. Tale variabile è posta a true dalla regola sync, a false dalle regole corrispondenti alle produzioni e vale true nello stato iniziale. In tal modo è possibile conoscere, osservando solo il contenuto di uno stato, se rappresenta uno stato *valido*, senza bisogno di conoscere le regole applicate per arrivare in esso. Tale informazione viene utilizzata all'interno di *SHE* per ricostruire il grafo degli stati *validi*. Tale grafo è un grafo composto soltanto di stati validi, in cui esiste un arco diretto dallo stato valido $V_1$ allo stato valido $V_2$ se e solo se esiste, nel grafo degli stati di Mur$\varphi$, un cammino di stati non validi da $V_1$ a $V_2$. Tale grafo rappresenta il grafo delle computazioni di un sistema *SG*. In figura 4.6 vediamo sulla sinistra un grafo degli stati di Mur$\varphi$, in cui gli stati validi sono indicati con un cerchio più grande, e sulla destra il corrispondente grafo degli stati validi.

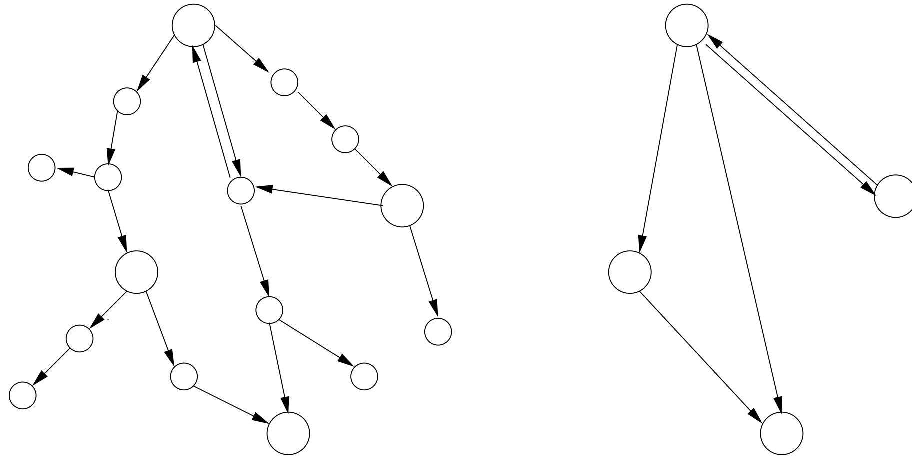


**Figura 4.6:** *Grafo degli stati di Mur$\varphi$ e relativo grafo degli stati validi*

### 4.3.3 Modifiche al codice di Mur$\varphi$

Mur$\varphi$ consente di ottenere informazioni sullo spazio degli stati del sistema a diversi livelli di dettaglio: è possibile conoscere solo il numero di stati esplorati (livello minimo), oppure ottenere il contenuto completo di tutti gli stati incontrati e informazioni sulle regole applicate per passare da uno stato ad un altro (livello massimo).

Chiedendo a Mur$\varphi$ il massimo livello di dettaglio, otteniamo un output che ci permette di ricostruire il grafo degli stati del sistema. A questo punto il problema è ottenere un file di tipo graphml [BEH+01] dall'output generato da Mur$\varphi$. Il problema principale è che l'output di Mur$\varphi$ è *ad albero*, corrispondente all'esplorazione in ampiezza effettuata dal programma sul grafo degli stati. Trattandosi dell'esplorazione di un grafo, in tale visita in ampiezza è possibile che un certo stato sia mostrato più volte. Purtroppo Mur$\varphi$ non ci avverte di questo, ma stampa indifferentemente il contenuto di tutti gli stati.

La prima soluzione implementata è un programma Java che legge l'output di Mur$\varphi$ e lo trasforma in un file graphml. Tale soluzione tuttavia è risultata poco efficiente, poichè era necessario memorizzare in una tabella *hash* gli stati incontrati leggendo l'output di Mur$\varphi$ (così da poter controllare se uno stato era nuovo, e quindi creare un nuovo nodo nel file graphml, oppure era già stato incontrato e poteva essere ignorato). A causa dell'uso intensivo di questa tabella hash anche con programmi di dimensione media, i tempi di attesa erano troppo elevati.

La soluzione successiva è stata quindi quella di modificare il codice Mur$\varphi$ (che è liberamente disponibile) per ottenere un output che fosse già in formato graphml. Tale metodo ha consentito di ridurre drasticamente i tempi di

attesa, ma in tale output è ancora presente il contenuto completo di tutti gli stati intermedi tra due stati *validi* (vedi 4.3.2), mentre ciò che a noi interessa è soltanto il contenuto di questi ultimi.

L'ovvia soluzione è stata quindi quella di modificare ulteriormente il codice di Mur$\varphi$ per far sì che venisse stampato solo il contenuto degli stati *validi*. Tale soluzione ha portato ad un altro drastico miglioramento dei tempi di generazione dei file graphml oltre ad una diminuzione della dimensione degli stessi file, come è possibile vedere nella seguente tabella:

| stati Mur$\varphi$/stati *validi* | 551/11 | | 8792/5 | |
|---|---|---|---|---|
| | tempo | dimensione | tempo | dimensione |
| Versione 1 | 170s | 1.9Mb | 2h24m | 122Mb |
| Versione 2 | 6s | 1.9Mb | 8m25s | 122Mb |
| Versione 3 | 0.6s | 145Kb | 33s | 1.7Mb |

Osserviamo che all'interno di *SHE* sono state mantenute sia la seconda che la terza versione. La seconda versione è particolarmente utile in fase di *debugging* di un sistema *SG*, in quanto permette di conoscere il dettaglio di tutte le fase intermedie della riscrittura. La seconda versione è invece più efficiente in termini di spazio e tempo, ma consente di conoscere solo il contenuto degli stati *validi*.

Inoltre la seconda versione può essere utilizzata per ottenere un grafo degli stati in formato graphml di un *qualsiasi* programma Mur$\varphi$, rendendo possibile utilizzare *SHE* per analizzare qualsiasi computazione Mur$\varphi$.

# Capitolo 5

# Un sistema per l'esplorazione di grafi

In questo capitolo analizziamo *GraVE* (**Gra***ph* **V***iewer and* **E***xplorer*), il programma usato da *SHE* per l'*esplorazione* del grafo degli stati.

Per tale componente era nostra intenzione fornire all'utente uno strumento che offrisse un approccio visuale all'esplorazione del grafo degli stati e che permettesse di selezionare i nodi di tale grafo in maniera intuitiva attraverso l'uso del cursore, e in maniera più complessa sottoponendo delle interrogazioni al sistema. Ci siamo resi conto che tali funzionalità potevano essere implementate progettando uno strumento più generale, capace di supportare l'esplorazione interattiva di un grafo orientato (che nel caso di *SHE* è un grafo degli stati) ai cui nodi fossero associate informazioni di varia natura (in *SHE* degli ipergrafi).

Abbiamo quindi deciso di creare un programma, *GraVE*, per l'esplorazione di grafi che non fosse legato a nessun particolare dominio applicativo, ma che potesse essere personalizzato in base alle singole esigenze. Oltre ad

essere utilizzato come esploratore delle computazioni *SG* all'interno di *SHE* (figura 5.1), nelle figure 5.2 e 5.3 possiamo vedere altre applicazioni realizzate con *GraVE*. Il programma in figura 5.2 permette all'utente di specificare un URL e visualizzare un grafo degli URL correlati ad esso. Per conoscere quali sono gli URL correlati all'URL data, vengono utilizzate le *Google Web APIs*[1], una serie di librerie che consentono di effettuare interrogazioni sul motore di ricerca Google[2]. Il programma 5.3 consente invece di visualizzare l'albero dei file radicati in una directory e di visualizzare il contenuto dei file che vengono selezionati.

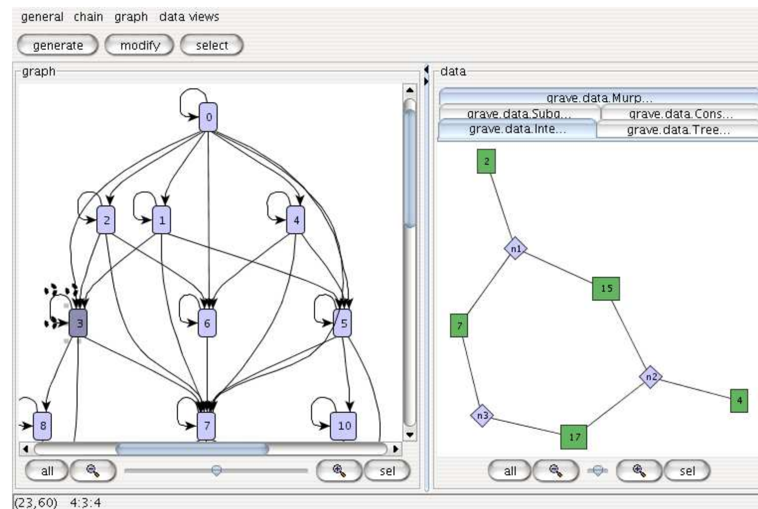



**Figura 5.1:** *GraVE all'interno di SHE*

I programmi che vogliamo realizzare con *GraVE* sono composti dai seguenti elementi:

- un componente per la generazione di un grafo $G$,

[1] http://www.google.com/apis/
[2] http://www.google.com/

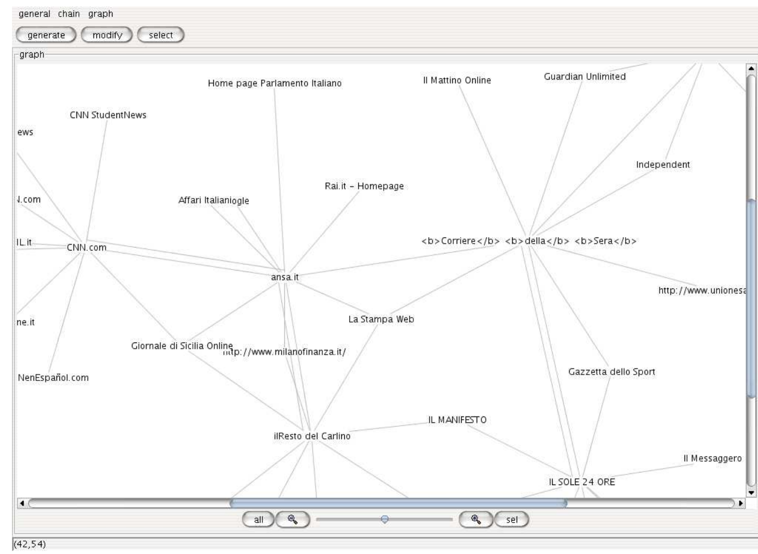



**Figura 5.2:** *GraVE che visualizza un grafo di URL correlate*

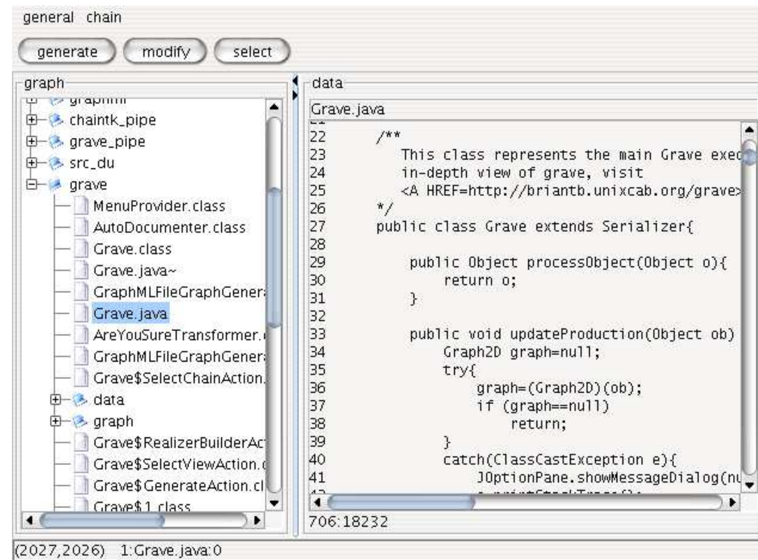



**Figura 5.3:** *GraVE come file browser*

- un componente per la visualizzazione del grafo $G$,
- un componente per la visualizzazione di informazioni sui nodi nel grafo $G$,
- un componente per la selezione di nodi del grafo $G$ mediante interrogazioni.

In figura 5.4 è possibile vedere la schermata principale di *GraVE*. Il compito

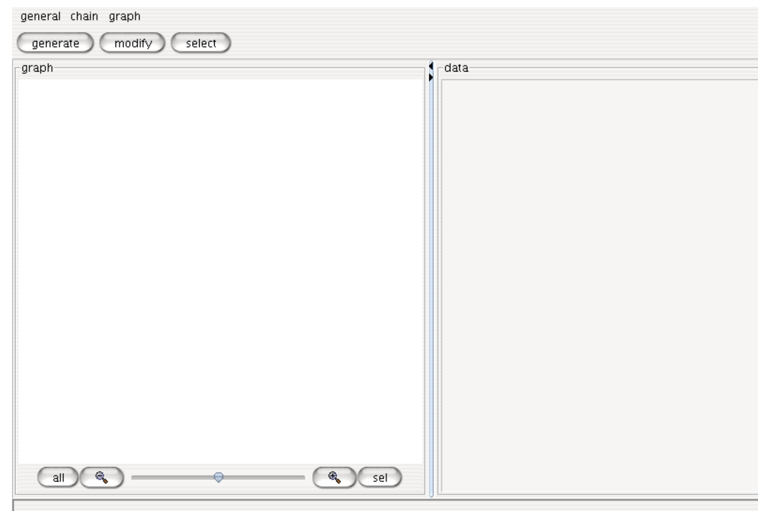



**Figura 5.4:** *Schermata principale di GraVE*

di *GraVE* è quello di ottenere un grafo $G$ dal componente dedicato alla generazione quando l'utente preme il bottone denominato generate e passare tale grafo al componente dedicato alla sua visualizzazione e a quello dedicato alla visualizzazione delle informazioni associate ai nodi. In figura 5.5 è presente il diagramma UML di collaborazione relativo alla pressione da parte dell'utente del bottone generate.

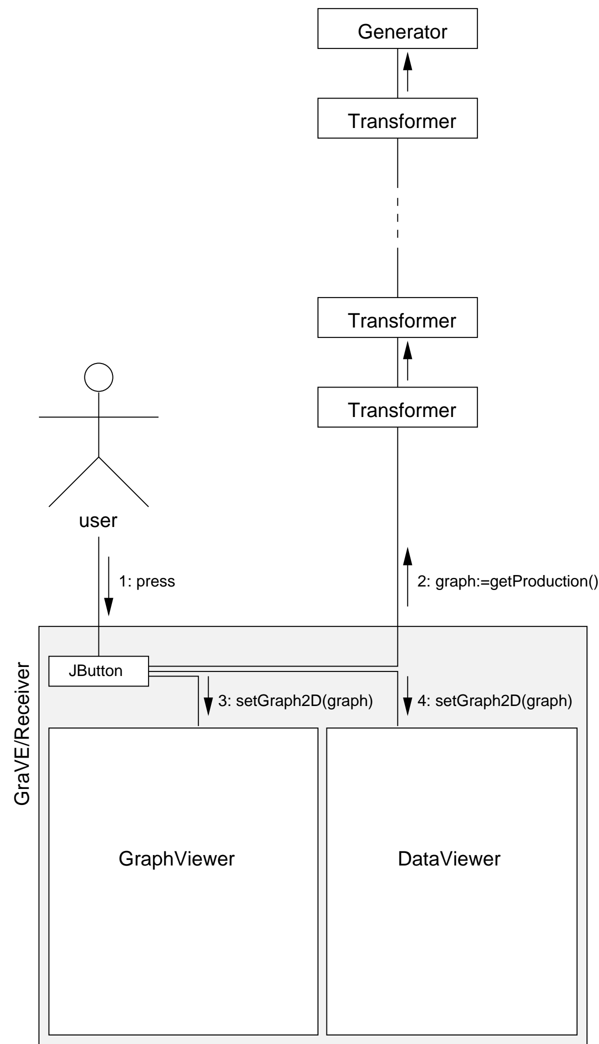



**Figura 5.5:** *Diagramma di collaborazione di GraVE*

I programmi che vogliamo realizzare con *GraVE* compiono spesso operazioni molto simili: applicazione di un layout ad un grafo, etichettatura dei nodi e degli archi, modifica di proprietà grafiche quali il colore e la dimensione dei nodi e degli archi. É quindi importante disporre di un meccanismo che non richieda ogni volta di *reinventare la ruota* ma che permetta di applicare determinate modifiche ad un grafo combinando componenti già esistenti, e limitando la programmazione solo ai nuovi componenti che non possono essere ottenuti dalla combinazione di quelli esistenti. Tale problema è stato risolto creando un meccanismo mediante il quale è possibile applicare trasformazioni complesse connettendo in sequenza oggetti che applicano trasformazioni più semplici. Ciascun elemento della sequenza riceve un grafo dal componente precedente, applica la trasformazione per cui è stato programmato, e passa il grafo al componente successivo. Tale catena di elementi costituisce il componente per la generazione del grafo, la cui architettura verrà descritta in 5.1.

Lo stile più appropriato per visualizzare il grafo e le informazioni associate ai nodi è legata all'ambito applicativo del programma che si vuole realizzare con *GraVE*. Non è infatti possibile stabilire a priori un componente universale per la visualizzazione di un grafo, nè tantomeno uno per la visualizzazione delle informazioni contenute nei nodi. Nel caso di *SHE* ad esempio i nodi rappresentano degli ipergrafi. Al contrario, nel caso del programma che visualizza gli *URL* correlati, si potrebbe desiderare conoscere maggiori dettagli sull'*URL* in oggetto, o magari visualizzare il documento collocato in tale *URL*.

Non essendo possibile creare nè un componente universale per la visualizzazione del grafo nè uno per la visualizzazione delle informazioni contenute nei nodi, si è scelto di realizzare tali componenti mediante due interfacce,

che rappresentano un costrutto dei linguaggi ad oggetti per descrivere solo le operazioni, intese come nome dell'operazione, argomenti e tipi di ritorno, che si possono compiere su un oggetto, senza legarle ad un'implementazione specifica. Le interfacce realizzate a tale scopo sono:

- grave.graph.GraphViewer che rappresenta oggetti capaci di *visualizzare* un grafo,

- grave.data.DataViewer che rappresenta oggetti capaci di *visualizzare* informazioni associate ai nodi selezionati in un grafo.

Tali interfacce vanno implementate per realizzare dei componenti adatti agli specifici ambiti applicativi. Per *SHE*, ad esempio, è stata creata una classe concreta, denominata InteractiveHypergraphViewer, che implementa l'interfaccia DataViewer e visualizza un ipergrafo in base al contenuto del nodo selezionato nel grafo degli stati.

L'eseguibile che rappresenta il programma *GraVE* è grave.Grave. Tale programma richiede che siano specificati da linea di comando due file, il primo che elenca gli elementi per la generazione del grafo, e il secondo che specifica la coppia di GraphViewer e DataViewer che si desidera utilizzare all'interno di *GraVE*. Nella sezione 5.5 è presente una descrizione del formato di tali file, e il contenuto di tali file per l'utilizzo di *GraVE* all'interno di *SHE*.

All'interno di *GraVE* è stato inoltre implementato un meccanismo avanzato per la selezione di nodi nel grafo. Tale sistema si basa sull'elaborazione da parte dell'utente di espressioni booleane, sia in maniera *grafica* che in maniera *testuale*. I nodi selezionati nel grafo sono tutti quelli che soddisfano l'espressione elaborata. *GraVE* fornisce alcuni operatori base per le espressioni, tra cui alcuni tradizionali operatori booleani (and, or, not) e operatori

che riguardano le caratteristiche *topologiche* di un nodo (ad esempio l'operatore source, che risulta vero per tutti quei nodi che non hanno archi entranti). Tuttavia, poichè *GraVE* si pone come un sistema *generale* di esplorazione di grafi, il sistema è realizzato in maniera che sia possibile implementare nuovi operatori.

Osserviamo inoltre che, poichè per alcuni componenti può essere utile fornire un menu attraverso cui l'utente imposti alcuni parametri di configurazione (ad esempio un componente potrebbe applicare tipi diversi di layout ad un grafo in base alla scelta dell'utente), è stata creata l'interfaccia grave.MenuProvider. Se un componente che viene inserito all'interno di *GraVE* (tra gli elementi che generano il grafo $G$, il visualizzatore del grafo e il visualizzatore delle informazioni sui nodi selezionati) desidera utilizzare un menu per la propria configurazione, è sufficiente che implementi tale interfaccia, e *GraVE* provvederà ad inserire un'apposita voce nel proprio menu.

Il resto del capitolo è strutturato nel seguente modo: nella sezione 5.1 descriviamo la generazione di grafi nel sistema *GraVE*, nelle sezioni 5.2 e 5.3 analizziamo rispettivamente l'interfaccia GraphViewer e l'interfaccia DataViewer e mostriamo alcuni esempi. Nella sezione 5.4 analizziamo il modulo di *GraVE* per la selezione dei nodi mediante interrogazioni. Infine, nella sezione 5.5 presentiamo una breve guida al formato dei file di configurazione da fornire all'avvio di *GraVE*.

## 5.1 La generazione di grafi con *GraVE*

Il package chaintk è la componente di *GraVE* progettata per generare il grafo e applicargli una serie di trasformazioni prima della visualizzazione. In tali trasformazioni rientrano ad esempio l'applicazione di un layout al grafo,

l'eliminazione di nodi con determinate caratteristiche (in *SHE* è necessario eliminare dal grafo i nodi che non rappresentano *stati validi*), la decorazione degli archi mediante etichette. Ciò che ci interessava era un'architettura mediante cui fosse facile includere/escludere trasformazioni da applicare al grafo e con cui fosse possibile combinare insieme singole trasformazioni per ottenere trasformazioni più complesse.

La soluzione implementata è ispirata alle *pipe* di UNIX [MPT78]. Tale soluzione consiste in una serie di oggetti disposti in sequenza in cui ognuno riceve il proprio input dal precendente e invia il proprio output al successivo. Con tale approccio è raggiunto l'obiettivo di ottenere trasformazioni complesse dalla combinazione di trasformazioni più semplici, aumentando inoltre la riusabilità di questi componenti. Infatti, dopo aver realizzato un componente che applica un layout ad un grafo ricevuto in input, esso potrà essere utilizzato in qualsiasi sequenza a condizione che in input gli venga fornito un grafo. Osserviamo che, a differenza delle *pipe*, nella nostra architettura la comunicazione:

- non è basata su un flusso continuo di byte ma sullo scambio di oggetti della classe Object,
- è bidirezionale.

Per tali motivi, nel seguito ci riferiremo a tale architettura con il nome di *chain*. Notiamo inoltre che, poichè il package chaintk non è limitato ad essere utilizzato per trasformazioni di grafi, ma di oggetti della classe Object, esso può essere usato in contesti diversi da quello di *GraVE*.

In figura 5.6 è mostrata l'architettura delle classi principali del sistema. La classe *astratta* ChainLink rappresenta un generico elemento della *chain*. La classe Generator rappresenta l'elemento iniziale di una *chain*, la classe

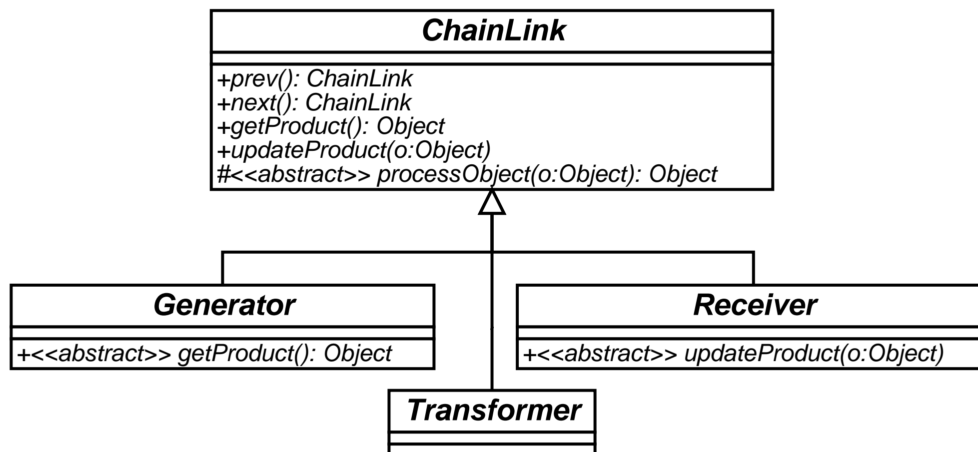



**Figura 5.6:** *Architettura delle classi principali del package* chaintk

Transformer gli elementi intermedi, la classe Receiver l'ultimo elemento, ossia l'elemento destinato a *ricevere* il risultato dell'intera elaborazione. All'interno di *GraVE*, il programma principale grave.Grave è il Receiver della *chain* di generazione del grafo. Ogni ChainLink può riferire rispettivamente l'elemento precedente e l'elemento successivo con i metodi prev e next. Osserviamo che il numero di Transformer di una *chain* è arbitrario e può essere zero.

Il metodo *astratto* processObject è destinato a processare l'oggetto ricevuto come parametro e ritornare come risultato l'oggetto elaborato.

Il metodo getProduct viene chiamato su un ChainLink per ottenere il risultato della sua elaborazione. L'implementazione di default di tale metodo restituisce il risultato dell'applicazione del metodo processObject all'oggetto ottenuto chiamando getProduct sul ChainLink precedente. Tuttavia tale metodo è astratto nella classe Generator, poichè un Generator è il primo elemento di una *chain* ed è suo compito la generazione dell'oggetto Java iniziale. In figura 5.7 vediamo un diagramma di sequenza in cui il Generator della *chain*

crea l'oggetto iniziale della sequenza leggendo un file da disco.

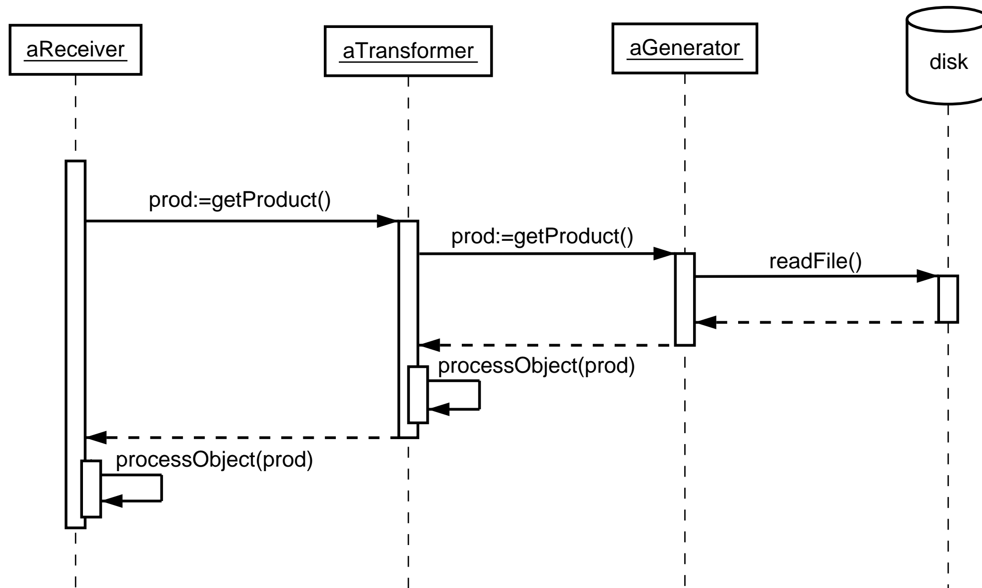



**Figura 5.7:** *Diagramma di sequenza di una chiamata a* getProduct

Il metodo updateProduct serve a comunicare ad un oggetto nella *chain* che sono state effettuate delle modifiche nella porzione di *chain* che lo precede. Immaginiamo di utilizzare, all'interno di *GraVE*, una *chain* di generazione del grafo in cui il Generator crea un grafo in base a informazioni prelevate da una base di dati. Ogni volta che viene premuto il bottone generate nell'interfaccia di *GraVE* il Generator della *chain* si connette alla base di dati, genera un grafo e lo passa al successivo ChainLink della *chain*. Dopo che sono state effettuate una serie di trasformazioni ad opera di elementi Transformer il grafo viene visualizzato all'interno di *GraVE*. Se però i dati nella base di dati cambiano molto di frequente, tali cambiamenti non verranno visualizzati subito, ma solo quando verrà premuto nuovamente il tasto generate, momento in cui il Generator si riconnetterà nuovamente alla base di dati e creerà

un nuovo grafo. Se invece il Generator interrogasse periodicamente la base di dati, potrebbe, nel caso in cui i dati sono cambiati rispetto all'ultima interrogazione, creare *immediatamente* il nuovo grafo, e passarlo al ChainLink successivo mediante il metodo updateProduct. Ciò provocherà una sequenza di chiamate del metodo updateProduct sui vari elementi della *chain* fino ad effettuare l'update del grafo finale visualizzato da *GraVE*, senza alcun intervento dell'utente. In figura 5.8 vediamo un diagramma di sequenza della chiamata a updateProduct.

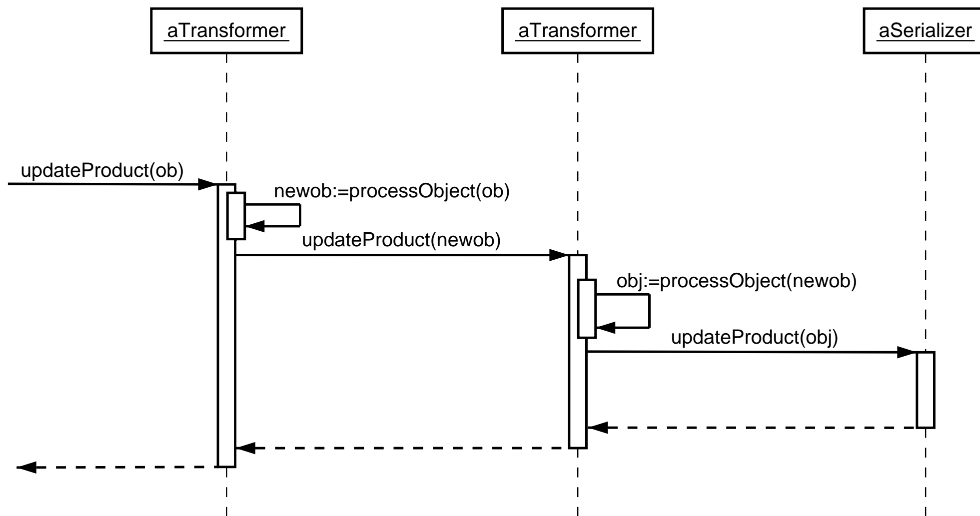



**Figura 5.8:** *Diagramma di sequenza della chiamata a* updateProduct

Vediamo ora un esempio di un Transformer utilizzato all'interno di *SHE*. Ricordiamo brevemente che il grafo degli stati di Mur$\varphi$ contiene degli stati, detti *validi*, che rappresentano degli stati di una riscrittura in *SG*, mentre gli altri stati non hanno un corrispettivo in *SG*. All'interno di *SHE* desideriamo visualizzare soltanto il grafo degli stati *validi*, cioè un grafo in cui i nodi siano

solo gli stati *validi* e ci sia un arco tra due stati *validi* se c'è un cammino tra di essi composto solo di stati non validi.

Per risolvere tale problema è stato implementato un Transformer, detto MurphiValidTransformer che prende in input un grafo degli stati di Mur$\varphi$ *completo* e restituisce un grafo degli stati *ripulito*, che contiene solo stati *validi*. Quello che il metodo processObject di tale Transformer realizza è:

- individuare gli stati validi,
- per ogni coppia $V_1, V_2$ di stati validi, creare un arco diretto da $V_1$ a $V_2$ se e solo se esiste un cammino da $V_1$ a $V_2$ composto soltanto di stati non validi,
- eliminare gli stati non validi.

In figura 5.9 vediamo un esempio di applicazione di tale Transformer, in cui gli stati *validi* sono mostrati con un cerchio più grande.

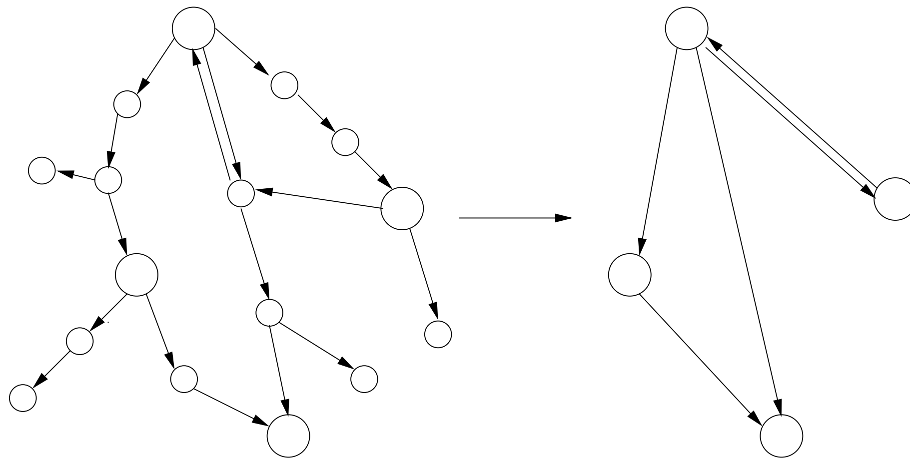


**Figura 5.9:** *applicazione di un* MurphiValidTransformer

Osserviamo che tale Transformer implementa l'interfaccia MenuProvider che, ricordiamo, consente di aggiungere una voce al menu di *GraVE*. In tal modo questo Transformer consente all'utente di scegliere se applicare o meno tale *filtro* al grafo (figura 5.10).

**Figura 5.10:** *Menu del* MurphiValidTransformer *per escludere il filtro*

## 5.2 L'interfaccia GraphViewer

La seguente interfaccia:

```
public interface GraphViewer {

    public Graph2D getGraph2D();

    public void setGraph2D(Graph2D graph);

    public JComponent getView();

}
```

rappresenta oggetti capaci di *mostrare* un grafo. Il metodo getView() di un tale oggetto deve restituire un JComponent[3], che rappresenta un componente grafico che può essere inserito nell'interfaccia principale di *GraVE*, in cui è possibile visualizzare in qualche forma il grafo passato come argomento al

[3] http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JComponent.html

metodo setGraph2D(). Alcuni GraphViewer concreti sono forniti all'interno di *GraVE* nel package grave.graph:

- InteractiveViewer è un GraphViewer il cui compito è quello di fornire un ambiente per l'eplorazione visuale di un grafo. Oltre a mostrarne una rappresentazione grafica, l'InteractiveViewer consente di compiere operazioni di *zoom* sul grafo. Permette inoltre di selezionare e spostare gli archi e i nodi del grafo visualizzato con l'uso del cursore. Un esempio di utilizzo di un InteractiveViewer è presente in figura 5.1.

- JTreeTreeViewer è un GraphViewer per la visualizzazione di alberi. La visualizzazione avviene mediante un javax.swing.JTree[4], che consente di espandere/collassare i nodi dell'albero che non sono foglie. Un esempio di utilizzo di un JTreeTreeViewer è presente in figura 5.3.

## 5.3 L'interfaccia DataViewer

Per visualizzare le informazioni sui nodi selezionati in un grafo è necessario un meccanismo per conoscere variazioni riguardo ai nodi selezionati dall'utente nel grafo, in maniera da mostrare sempre le informazioni relative ai nodi *effettivamente* selezionati. Tale meccanismo è implementato nella libreria yfiles mediante la realizzazione del *pattern* Observer [GHJV95]. Un oggetto Graph2D rappresenta il *soggetto* dell'osservazione, mentre oggetti di tipo Graph2DSelectionListener rappresentano gli *osservatori*. Ogni volta che viene modificato l'insieme di nodi/archi selezionati nel *soggetto*, questo invoca il metodo onGraph2DSelectionEvent su ciascun *osservatore*. In tal modo

[4] http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JTree.html

è possibile ricevere la comunicazione di ogni variazione nella selezione del grafo.

L'interfaccia degli oggetti capaci di mostrare le informazioni sui nodi selezionati nel grafo è la seguente:

```
public interface DataViewer
    extends Graph2DSelectionListener,
            GraphViewer{
}
```

Alcuni DataViewer concreti sono forniti all'interno di *GraVE* nel package grave.data:

- GraphInfoViewer è un DataViewer che mostra in una JLabel[5] informazioni sul numero totale di nodi e archi presenti nel grafo. Nel caso in cui fosse selezionato un solo nodo mostra anche il numero di archi entranti, l'etichetta del nodo selezionato e il numero di archi uscenti. All'interno di GraVE è sempre presente un GraphInfoViewer e viene mostrato nella parte inferiore della finestra. Nelle figure 5.1, 5.2 e 5.3 è possibile vedere un GraphInfoViewer.

- InteractiveHypergraphViewer è un DataViewer che fornisce una rappresentazione visuale dell'ipergrafo *contenuto* in un nodo. L'ipergrafo deve essere memorizzato in una NodeMap[6] del grafo, che rappresenta una tabella in cui le chiavi sono i nodi del grafo. Tale DataViewer è utilizzato all'interno di *SHE*.

- FileViewerData è un DataViewer che visualizza il contenuto del file associato al nodo selezionato nel grafo. Mostra inoltre il nome del file, il

[5] http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JLabel.html
[6] http://www.yworks.com/products/yfiles/doc/api/y/base/NodeMap.html

numero di linee e il numero di caratteri che lo compongono. Per utilizzare tale DataViewer è necessario che ci sia una NodeMap del grafo in cui ai nodi sono associati ogggeti della classe File[7]. In figura 5.3 è possibile vedere un FileViewerData in azione.

- MultipleDataViewer è un DataViewer che permette di visualizzare più DataViewer al suo interno. Questi DataViewer vengono organizzati in un JTabbedPane[8] a cui è possibile aggiungerne di nuovi e rimuovere quelli esistenti. In figura 5.1 è possibile vedere un MultipleDataViewer in cui è stato selezionato il tab che contiene un InteractiveHypergraphViewer, e in cui è possibile vedere i tab corrispondenti ad altri DataViewer.

## 5.4 Selezione di nodi

*GraVE* fornisce una funzionalità avanzata per la selezione dei nodi. Tale selezione avviene mediante l'elaborazione di espressioni booleane sui nodi. I nodi selezionati nel grafo sono quelli per cui risulta vera l'espressione elaborata. Tale funzionalità può risultare utile quando si vogliono individuare in un grafo i nodi che soddisfano determinate caratteristiche affidando tale ricerca al sistema. Immaginiamo ad esempio, all'interno di *SHE*, di aver selezionato un nodo nel grafo degli stati e di voler conoscere quali stati è possibile raggiungere da questo stato. Per un grafo "piccolo" tale compito è piuttosto semplice. Ma in un grafo con un numero molto elevato di archi, in cui questi spesso si intersecano, può risultare difficoltoso. Mediante il modulo di selezione avanzata fornito da *GraVE* è invece possibile elaborare una

[7] http://java.sun.com/j2se/1.4.2/docs/api/java/io/File.html
[8] http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JTabbedPane.html

semplice espressione che seleziona automaticamente i nodi raggiungibili dal nodo correntemente selezionato nel grafo.

La creazione delle espressioni può avvenire sia attraverso un linguaggio *grafico* che attraverso uno *testuale*. I due linguaggi sono equivalenti per quanto riguarda le espressioni che si possono comporre. Il linguaggio testuale rappresenta il formato *nativo* poichè in questa forma le espressioni vengono anche salvate su file; il linguaggio grafico è stato introdotto perchè può fornire una comprensione più immediata del significato di un'espressione.

Per avviare il modulo di composizione delle espressioni di selezione, è necessario premere il bottone select nella schermata principale di *GraVE*. Ciò farà apparire sullo schermo la finestra di figura 5.11. La parte superiore della finestra è dedicata alla *composizione* dell'espressione grafica, la parte inferiore a quella testuale. In figura 5.12 è possibile vedere sia la rappresentazione grafica che quella testuale dell'espressione "not(or(source,end))", che è un'espressione che seleziona tutti i nodi che non sono nè nodi sorgente nè nodi pozzo. Ogni modifica nell'espressione grafica si riflette in tempo reale in quella testuale. Modificando invece l'espressione testuale, è necessario selezionare la voce update per riflettere le modifiche nella parte grafica. Dopo aver composto un'espressione, è necessario selezionare la voce show per mostrare nel grafo i nodi per i quali l'espressione è verificata.

**Linguaggio grafico.** La creazione di espressioni con il linguaggio grafico avviene attraverso il *disegno* di grafi. Nel seguito indicheremo con *grafo oggetto* il grafo principale mostrato all'interno di *GraVE*, per distinguerlo dal grafo dell'espressione realizzata con il linguaggio grafico.

Un grafo che rappresenta un'espressione valida sintatticamente è un albero orientato in cui ogni nodo ha al massimo un arco uscente. In questi grafi

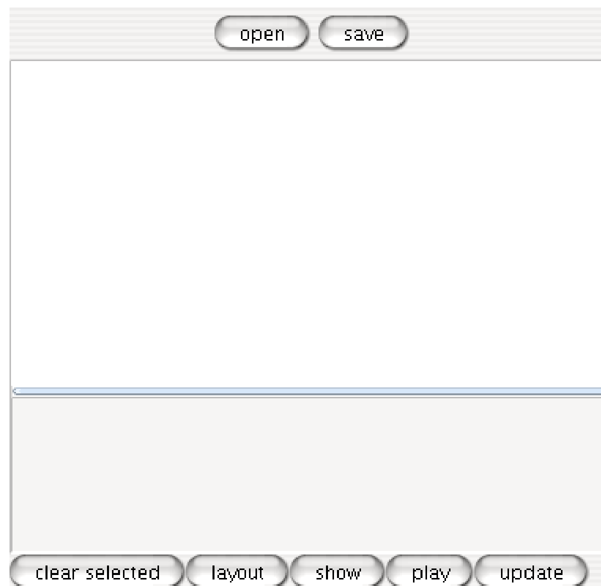



**Figura 5.11:** *Finestra di selezione*

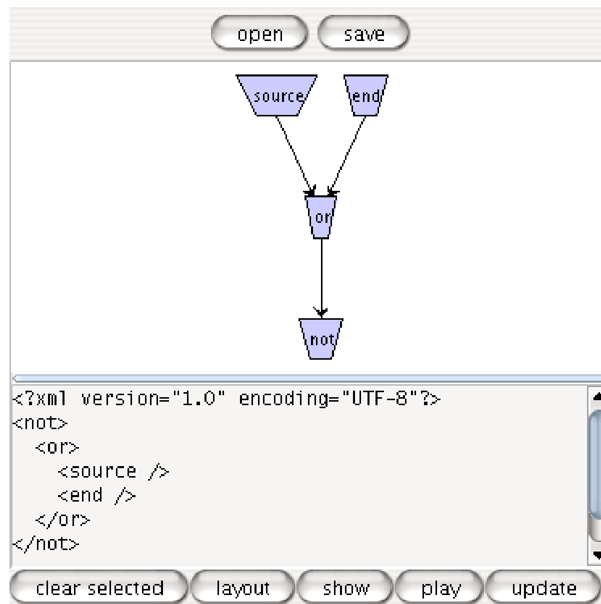



**Figura 5.12:** *Esempio di selezione*

esiste sempre un solo nodo che ha zero archi uscenti. Infatti se non esistesse un nodo con zero archi uscenti, allora ci sarebbero $n$ nodi e almeno $n$ archi e quindi il grafo non sarebbe un albero. D'altra parte, supponendo per assurdo che ci siano almeno due nodi con zero archi uscenti, ci sarà, nel grafo non diretto, un cammino che li congiunge. Allora lungo tale cammino nel grafo diretto ci sarà almeno un nodo con due archi uscenti, contraddicendo l'ipotesi che ogni nodo ha al massimo un arco uscente. Il nodo con zero archi uscenti viene detto *radice*.

I nodi di un'espressione vengono chiamati *selettori* e rappresentano operatori capaci di restituire un sottoinsieme dei nodi del grafo oggetto. Ad esempio, in figura 5.12 il nodo etichettato con source è un selettore che restituisce l'insieme dei nodi sorgente del grafo oggetto. Ciascun selettore accetta un numero fisso (eventualmente anche zero) di selettori in input; tale numero viene detto *fan-in*. Il fatto che esista l'arco diretto $(s_1, s_2)$ indica intuitivamente che il selettore $s_2$ riceve l'insieme di nodi calcolati da $s_1$ e li usa per la propria elaborazione. I selettori con *fan-in* uguale a zero sono detti detti *foglie* o selettori *foglia*. In figura 5.12 il nodo etichettato con not rappresenta un selettore con *fan-in* uguale a uno, che restituisce i nodi del grafo oggetto che *non* appartengono all'insieme restituito dal selettore or.

Analizziamo ora più in dettaglio l'espressione in figura 5.12. Tale espressione presenta due selettori foglia, source e end, in cui il primo seleziona i nodi sorgente e il secondo i nodi pozzo. Il fatto che esistano degli archi tra questi due selettori e il selettore or indica che quest'ultimo utilizza il nodi calcolati dai due precendenti per fornire un nuovo insieme; in particolare tale selettore effettua l'unione dei due insiemi. Infine l'insieme dei nodi generato dal selettore or viene passato al selettore not, che effettua il complemento di tale insieme rispetto a i nodi del grafo oggetto. Essendo not la radice

del grafo, i nodi selezionati nel grafo oggetto sono i nodi ritornati da questo selettore: vengono selezionati i nodi che non sono nè sorgenti nè pozzi.

**Linguaggio testuale.** I selettori possono essere descritti anche in modalità testuale. Quando un utente elabora graficamente un selettore, la sua rappresentazione testuale compare nella parte inferiore del modulo di selezione. Il formato utilizzato nel linguaggio testuale è l'XML 1.0 [BPSM97]. Ogni selettore è rappresentato da un elemento XML. Il nome dell'elemento XML corrispondente ad un selettore è uguale a quello utilizzato per etichettare il selettore nella rappresentazione grafica.

Se un selettore $S$ riceve l'input dai selettori $S_1, \ldots, S_n$, gli elementi XML corrispondenti a $S_1$, ..., $S_n$ saranno figli dell'elemento XML corrispondente a $S$. La descrizione testuale del selettore radice è quindi un documento XML ben formato. In figura 5.12 è possibile vedere contemporaneamente la rappresentazione grafica e quella testuale di un selettore.

*GraVE* mette a disposizione dell'utente diversi tipi di selettori, tra i quali alcuni operatori insiemistici (unione, intersezione e complemento), selettori di nodi sorgenti, nodi pozzo, selettori pseudo-casuali. É inoltre possibile programmare nuovi selettori e utilizzarli all'interno di *GraVE*. I selettori realizzati con l'editor di *GraVE* possono inoltre essere salvati su un file e caricati da esso. Ciò che viene salvato su file è *esattamente* la rappresentazione testuale del selettore. In tal modo è possibile archiviare selettori e scambiarli con altri utenti. Inoltre, il fatto che il selettore venga salvato in formato XML, permette una comprensione abbastanza immediata del suo significato, senza necessità di avviare *GraVE* per utilizzare il selettore o per vederne la rappresentazione grafica.

## 5.5 File di configurazione

Per utilizzare GraVE è necessario fornire al programma principale due file: il primo deve contenere l'elenco dei ChainLink che devono formare la catena di generazione del grafo e l'altro deve contenere il nome di un GraphViewer e quello di un DataViewer. In entrambi i file le linee che iniziano con il simbolo "#" vengono ignorate. Ecco ad esempio il contenuto dei due file forniti a GraVE per l'utilizzo all'interno di *SHE*:

```
#file per la catena di generazione del grafo degli stati
#
#generator per la creazione di un grafo
#a partire da un file graphml
grave.graph.GraphMLGraphGenerator
#transformer per visualizzare il grafo
#degli stati validi
grave.graph.MurphiValidTransformer
#transformer che applica un layout al grafo
grave.graph.LayoutTransformer
```

```
#file per la vista e l'esplorazione del grafo
#degli stati in she
grave.graph.InteractiveViewer
grave.data.SHEViews
```

Durante l'uso di GraVE è comunque possibile modificare sia la catena di generazione che il GraphViewer e il DataViewer. Tale modifica avviene fornendo a GraVE dei file simili a quelli precedenti. Per selezionare una nuova catena di generazione è necessario selezionare la voce del menu general→chain→open (figura 5.13). Per selezionare una nuova coppia GraphViewer/ DataViewer è necessario selezionare la voce general→ui→open (figura 5.14).

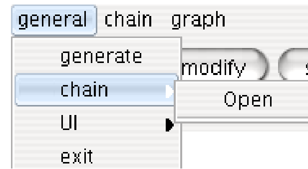



**Figura 5.13:** *Selezione di una nuova chain*

**Figura 5.14:** *Selezione di una nuova coppia* GraphViewer, DataViewer

# Capitolo 6

# Casi studio

In questo capitolo presentiamo tre casi studio implementati con *SHE*. Il primo affronta il problema classico dell'ordinamento di una sequenza di interi. Un aspetto interessante è la semplicità della soluzione proposta, in cui non viene *implementato* un particolare algoritmo di ordinamento ma il problema viene risolto in maniera *descrittiva*. In tale applicazione mostreremo inoltre che non sempre la rappresentazione visuale degli ipergrafi è di facile comprensione, e che può essere più utile, in alcuni casi, una descrizione testuale di alcune proprietà degli ipergrafi. Ciò che mostreremo è, dunque, come visualizzare, invece dell'ipergrafo, la sequenza di interi che l'ipergrafo rappresenta.

Nella seconda applicazione risolviamo un problema classico dei sistemi operativi: l'uso in scrittura di un area di memoria condivisa da più processi concorrenti. Tale esempio mostra come *SHE* possa essere usato dall'utente per costruire in maniera *incrementale* un sistema *SG*. In tale applicazione proporremo una prima soluzione che, dall'analisi delle computazioni con *SHE*, si rivelerà sbagliata. A questo punto, sempre con *SHE*, modificheremo alcune produzioni, verificando che le modifiche proposte risolvono effettivamente il

problema dato.

Nel terzo caso studio affrontiamo il problema di un insieme di processi divisi in due livelli di sicurezza, un livello *high* e un livello *low* che comunicano scambiandosi messaggi e, per motivi di sicurezza, desideriamo che non ci sia flusso di informazione dal livello *high* a quello *low*. In questo caso svilupperemo una soluzione e utilizzeremo *SHE* per verificare che il sistema soddisfi la condizione di sicurezza richiesta.

## 6.1 Ordinamento di interi

Il problema è quello classico dell'ordinamento di una sequenza di interi. L'idea è di modellare ogni intero $n$ con un arco, etichettato con $n$, che incide su due nodi (predecessore e successore) e una sequenza di interi con una sequenza di archi. Data la sequenza di interi $n_1 n_2 \ldots n_m$, questa sarà dunque rappresentata mediante il grafo $n_1(x_1, x_2)|n_2(x_2, x_3)| \ldots |n_m(x_m, x_{m+1})$. In figura 6.1 vediamo l'editor dello stato iniziale in cui è stato composto un grafo che rappresenta la sequenza 2,4,3.

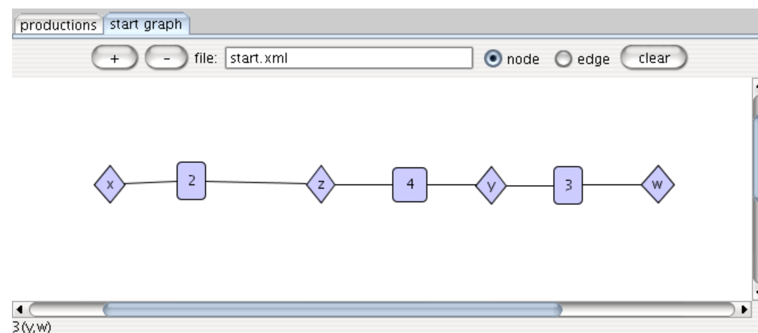



**Figura 6.1:** *Sequenza* 2,4,3

Con tali grafi è possibile implementare l'ordinamento mediante le seguenti

produzioni:

$$n(x, y) \xrightarrow{(y,m,xz)} n(y, z) \; m < n \tag{6.1}$$

$$n(x, y) \xrightarrow{(x,\overline{n},zy)} n(z, x) \tag{6.2}$$

$$n(x, y) \longrightarrow n(x, y) \tag{6.3}$$

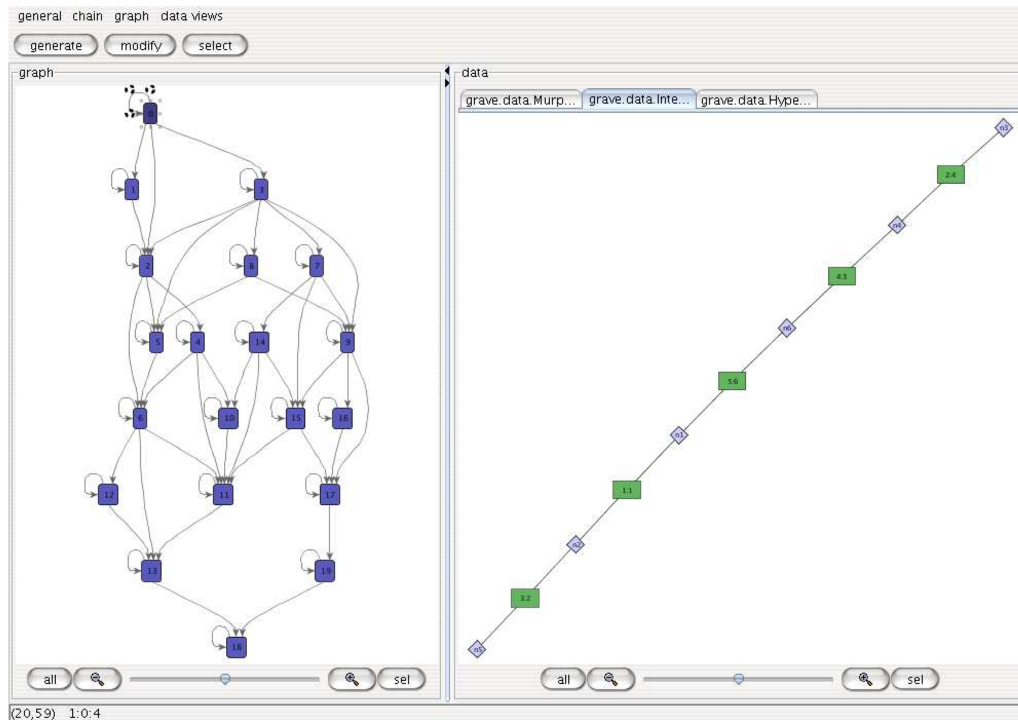
La produzione (6.1) è utilizzata da un arco che ne precede uno con etichetta minore e serve a *scavalcare* tale arco nella sequenza. La produzione (6.2) è utilizzata da un arco che ne precede uno con etichetta maggiore e serve a *farsi scavalcare*. La produzione (6.3) è utilizzata da archi che non si spostano nella sequenza, e viene utilizzata quando l'arco si trova già tra un arco con etichetta inferiore e uno con etichetta superiore, oppure quando non è possibile effettuare uno scambio con uno degli archi adiacenti, poichè questi sono già a loro volta coinvolti in uno *scambio*. L'aspetto interessante di queste produzioni è che non stiamo implementando nessun algoritmo di ordinamento, ma stiamo programmando in maniera *dichiarativa*: con le produzioni fornite non specifichiamo un metodo di ordinamento ma affermiamo che, quando un arco $n$ è seguito da un arco $m$ con $m < n$, i due archi si *devono* scambiare di posto nella sequenza.

Tuttavia quelle che abbiamo fornito non sono vere e proprie produzioni, ma piuttosto *schemi* di produzione, che non possono essere disegnati *direttamente* all'interno di *SHE*. Ciò che possiamo comporre all'interno di *SHE* sono istanze di questi schemi. Supponendo di avere un grafo in cui la massima etichetta è $k$, sono quindi necessarie: due produzioni per l'arco etichettato con 1, tre per l'arco etichettato con 2, ..., $k$ per l'arco etichettato con $k - 1$ e $k + 1$ per l'arco etichettato con $k$. Il numero di produzioni è quindi:

$$\sum_{i=1}^{k}(i + 1) = \sum_{i=1}^{k} i + k = \frac{k(k + 3)}{2}$$

Osserviamo che, sebbene tali produzioni siano tutte molto simili e semplici da scrivere con l'editor delle produzioni, soprattutto grazie alla possibilità di copiare e modificare produzioni già esistenti, sarebbe molto più comodo poter definire all'interno dell'editor direttamente gli schemi di produzione (6.1),(6.2) e (6.3). Tale discorso verrà ripreso nel capitolo 7.

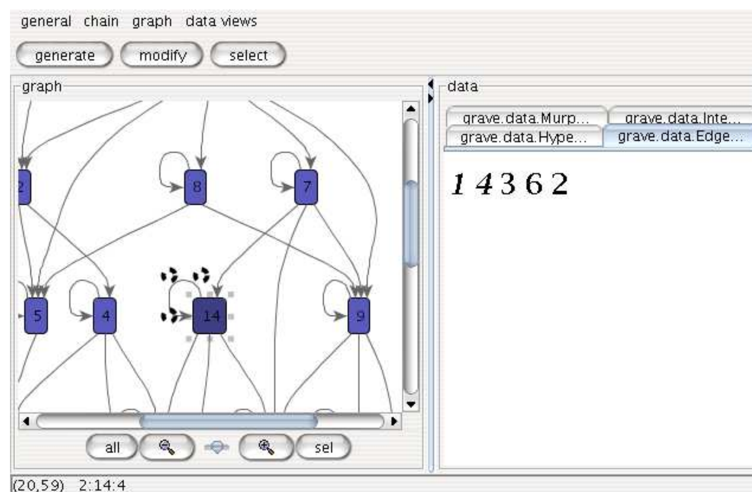
Dopo aver specificato con *SHE* un insieme di produzioni per archi con etichette da 1 a 6 e come grafo iniziale quello corrispondente alla sequenza 4,3,6,1,2, generiamo il grafo degli stati. Esplorando tale grafo all'interno di GraVE si ottiene quello che vediamo in figura 6.1, dove è stato selezionato lo stato iniziale. Dalla figura è possibile individuare la presenza di un unico




stato finale, che corrisponde proprio alla sequenza ordinata.

Osserviamo che la visualizzazione dell'ipergrafo non consente di compren-

dere immediatamente l'ordine degli archi, e quindi l'ordinamento dei valori. Per questo motivo è possibile implementare un apposito DataViewer, che non visualizza l'ipergrafo ma la sequenza che questo rappresenta. In figura 6.1 è possibile vedere *GraVE* che esplora lo stesso grafo di figura 6.1 ma che utilizza il DataViewer grave.data.EdgeArrayViewer, che visualizza la sequenza di interi rappresentata dall'ipergrafo. Inoltre con tale DataViewer, selezionando di seguito due stati, vengono evidenziati i valori che sono stati *spostati* nella sequenza, corrispondenti ad archi che hanno cambiato posizione nel grafo.




## 6.2 Modifica di un'area di memoria condivisa

In questa sezione risolviamo un problema tipico dei sistemi operativi. Consideriamo un insieme di processi $\mathcal{P}$ e un dato $d$ collocato in un'area di memoria condivisa da tutti i processi. Ciascun processo può copiare il dato dalla memoria condivisa in un'area di memoria privata, modificare eventualmente il valore in memoria privata e successivamente scrivere nell'area di memoria

condivisa il nuovo valore. Nel seguito considereremo di avere un dato $d$ di tipo intero che può assumere i valori da 0 a $m$.

Vediamo ora come possiamo modellare tale problema in *SG*. La soluzione è di rappresentare un dato condiviso con gli archi $D_i(x)$ con $i = 0, \ldots, m$, dove $i$ rappresenta il valore memorizzato nell'area di memoria. Tali archi accettano due tipi di operazioni, una per la lettura e una per la scrittura:

- $D_i(x) \xrightarrow{(x,\overline{read_i})} D_i(x)$ con $i = 0, \ldots, m$
- $D_i(x) \xrightarrow{(x,\overline{write_j})} D_j(x)$ con $i = 0, \ldots, m$ e $j = 0, \ldots, m$

I dati all'interno della memoria privata di un processo vengono rappresentati con gli archi $V_i(x)$ con $i = 0, \ldots, m$ e dispongono di produzioni per l'incremento, il decremento, la lettura e la scrittura del valore del valore:

- $V_i(x) \xrightarrow{(x,\overline{inc})} V_{i+1}(x)$ con $i = 0, \ldots, m-1$
- $V_i(x) \xrightarrow{(x,\overline{dec})} V_{i-1}(x)$ con $i = 1, \ldots, m$
- $V_i(x) \xrightarrow{(x,\overline{read_i})} V_i(x)$ con $i = 0, \ldots, m$
- $V_i(x) \xrightarrow{(x,\overline{write_j})} V_j(x)$ con $i = 0, \ldots, m$ e $j = 0, \ldots, m$

Etichettiamo con $P$ un arco che rappresenta un processo. Il primo nodo di un arco $P$ rappresenta il nodo in comune con l'arco che rappresenta il dato condiviso. In figura 6.2 è mostrato l'editor del grafo iniziale di *SHE* in cui sono rappresentati tre processi che condividono il dato $D_1$.

Vediamo le produzioni che un processo $P$ può utilizzare per creare in memoria privata una copia del dato condiviso:

- $P(y, \vec{x}) \xrightarrow{(y,read_i)} P(y, \vec{x}, z)|V_i(z)$ con $i = 0, \ldots, m$

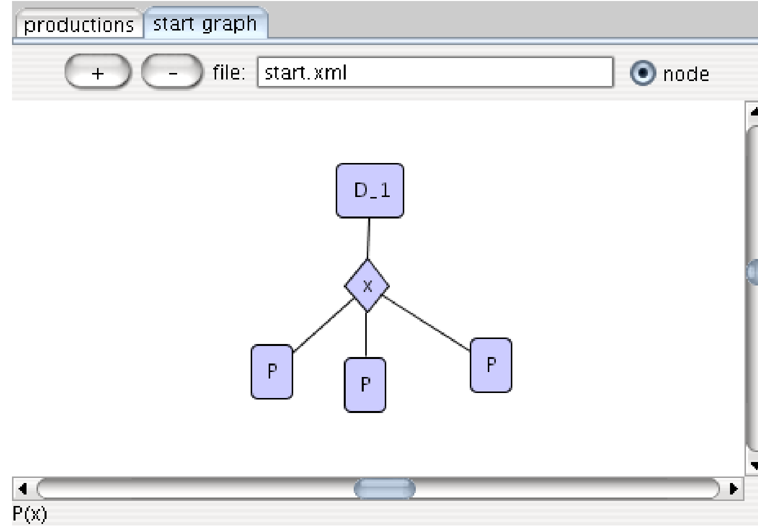



**Figura 6.2:** *Un grafo che rappresenta tre processi che codividono il dato* $D_1$

Vediamo ora invece le produzioni utilizzate per copiare un valore dalla memoria privata a quella condivisa:

- $P(y,\vec{x}) \xrightarrow[]{\substack{(y,write_i)\\(x_j,read_i)}} P(y,\vec{x})$ con $i = 0,\ldots,m$ e $x_j \in \vec{x}$

Inoltre un arco $P$ può compiere operazioni di incremento e decremento sulle variabili locali:

- $P(y,\vec{x}) \xrightarrow{(x_j,inc)} P(y,\vec{x})$ con $x_j \in \vec{x}$
- $P(y,\vec{x}) \xrightarrow{(x_j,dec)} P(y,\vec{x})$ con $x_j \in \vec{x}$

Modelliamo ora un processo che incrementa il valore del dato condiviso. Le sue produzioni sono:

- $P(x) \xrightarrow{(x,read_i)} P_1(x,y)|V_i(y)$ con $i = 0,\ldots,m$
- $P_1(x,y) \xrightarrow{(y,inc)} P_2(x,y)$ con $i = 0,\ldots,m$

- $P_2(x,y) \xrightarrow{\substack{(y,read_i)\\(x,write_i)}} P_3(x,y)$ con $i = 0, \ldots, m$

Disegnando con l'editor delle produzioni di *SHE* le produzioni appena enunciate, oltre alle produzioni identità per tutti gli archi, e specificando come grafo iniziale il grafo $D_0(x)|P(x)$, possiamo vedere in figura 6.3 il grafo degli stati della computazione, in cui è stato selezionato lo stato di partenza. Esplorando tale grafo è possibile vedere che il dato $D_0$ viene copiato in

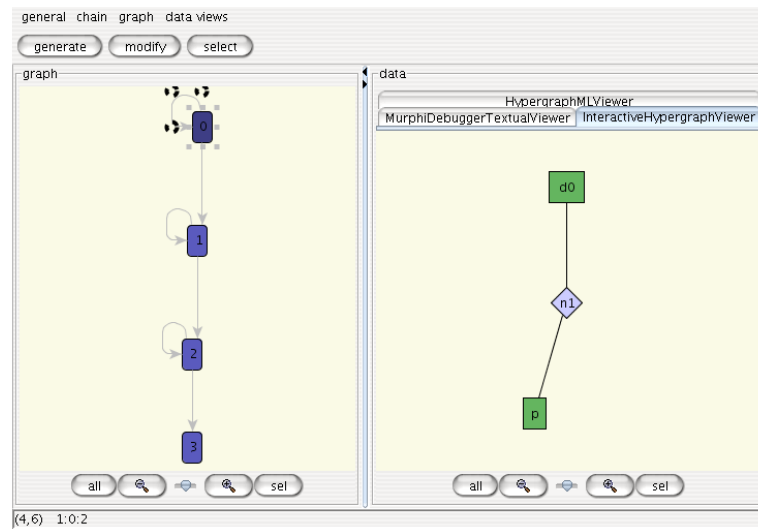



**Figura 6.3:** *Grafo degli stati con lo stato iniziale selezionato*

locale da $P$, incrementato e successivamente ricopiato sul dato condiviso. In figura 6.4 è mostrato lo stato finale della computazione, in cui è presente il dato condiviso che è diventato $D_1$ e in cui è possibile vedere l'arco $V_1$, che rappresenta la variabile privata di $P$.

Impostiamo ora $D_0(x)|P(x)|P(x)$ come grafo iniziale. Ciò che vorremmo ottenere, alla fine della computazione, è che il dato condiviso sia diventato $D_2(x)$, dopo aver subito due incrementi da parte dei due archi $P$. In figura

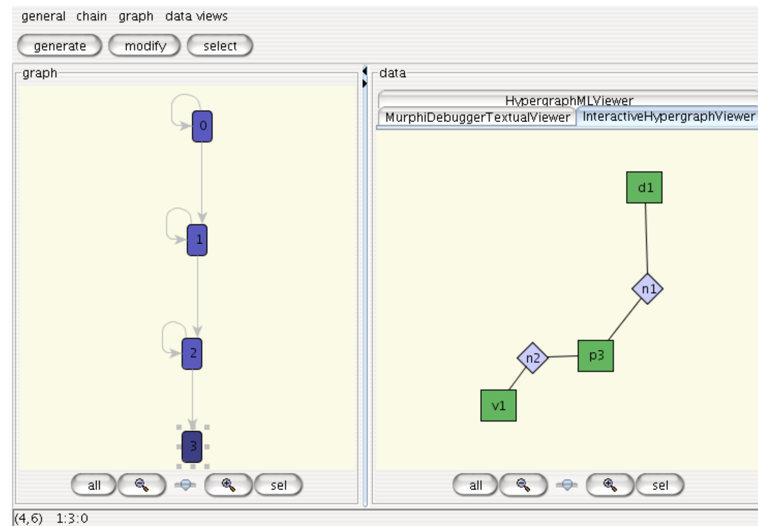



**Figura 6.4:** *Lo stato finale della computazione*

è possibile vedere il grafo degli stati. In particolare notiamo la presenta di *quattro* stati finali. In due di essi il dato condiviso è stato incrementato due volte, come è possibile vedere in figura 6.6. Negli altri due stati finali invece, il dato condiviso è $D_1$ (vedi figura 6.7). Esplorando le computazioni che conducono a questa situazione, si nota che il problema deriva dal fatto che dopo che un processo $P$ ha letto il dato condiviso, ma *prima* che questo scriva sul dato condiviso il risultato della sua computazione, un altro processo $P$ legge il dato condiviso. Quindi entrambi i processi leggono 0 e scrivono 1.

Una soluzione a tale problema è quella di *inibire* l'accesso simultaneo al dato condiviso. Ciò può essere fatto in maniera *trasparente* ai processi, simulando in tal modo un controllore delle risorse condivise. Per fare ciò, modifichiamo le produzioni per il dato condiviso:

- $D_i(x) \xrightarrow{(x,\overline{read_i})} Locked_i(x)$ con $i = 0, \ldots, m$

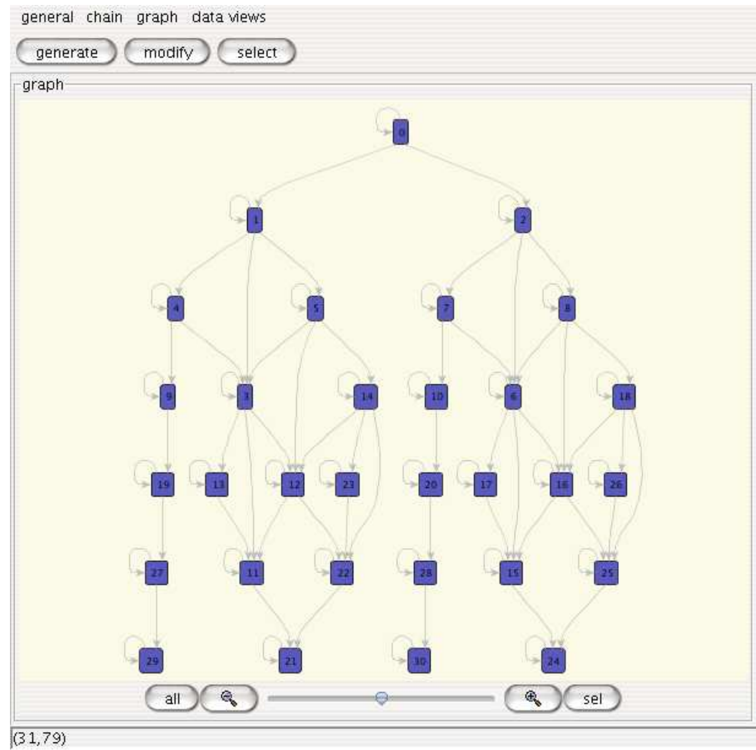



**Figura 6.5:** *Il grafo degli stati*

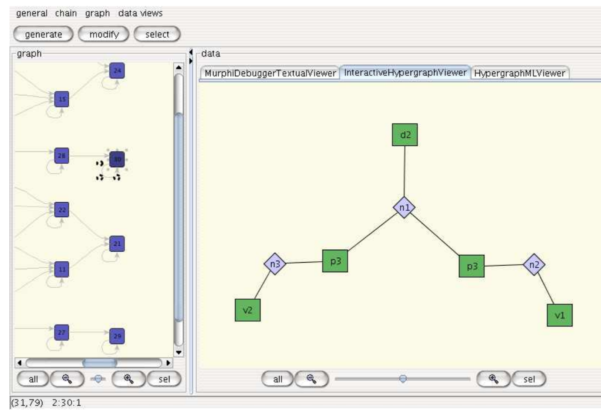



**Figura 6.6:** *Uno stato finale in cui il dato condiviso vale* $D_2$

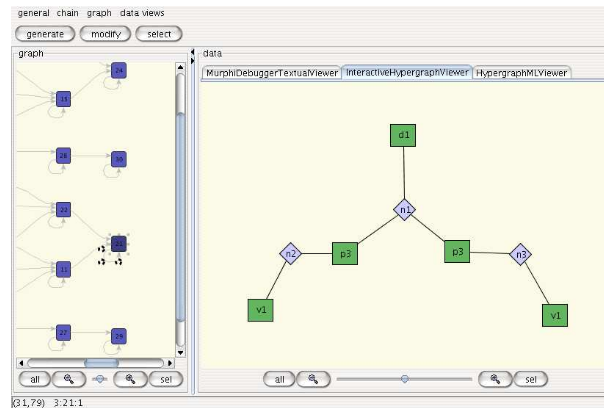



**Figura 6.7:** *Uno stato finale in cui il dato condiviso vale* $D_1$

- $Locked_i(x) \xrightarrow{(x,\overline{write_j})} D_j(x)$ con $i = 0, \ldots, m$ e $j = 0, \ldots, m$

Ritorniamo quindi all'editor delle produzioni di *SHE* e modifichiamo le produzioni suddette. A questo punto generiamo il nuovo grafo degli stati. Analizzandolo con *GraVE* si ottiene il grafo mostrato in figura 6.8, e si scopre che tale grafo ha due stati finali, e in entrambi questi stati il dato condiviso è etichettato con $D_2$, così come ci aspettavamo.

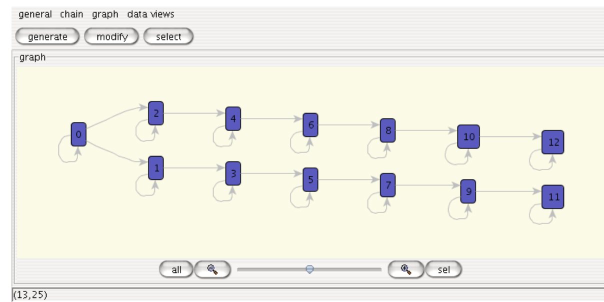



**Figura 6.8:** *Il grafo degli stati dopo la modifica delle produzioni per il dato condiviso*

In questo caso studio abbiamo dunque mostrato come è possibile programmare con *SHE* in maniera *incrementale*. Inizialmente abbiamo descritto un insieme di produzioni per risolvere un problema. Con *GraVE* abbiamo analizzato il grafo degli stati e abbiamo scoperto che alcune computazioni producevano dei risultati inaspettati. Analizzando tali computazioni abbiamo compreso il problema ed elaborato una soluzione. Tramite l'editor delle produzioni tale soluzione è stata implementata e poi, di nuovo analizzando gli stati finali della computazione, ci siamo accorti che effettivamente la soluzione proposta risolveva il problema.

Osserviamo che in generale non tutti i requisiti di un sistema possono essere verificati mediante un'analisi del contenuto dei singoli stati esplorati, poichè alcuni requisiti possono riguardare le relazioni esistenti tra i diversi stati. Il caso studio successivo mostra l'uso di *SHE* per verificare una proprietà di sicurezza non verificabile dall'analisi locale di uno stato.

## 6.3 Verifica di una semplice proprietà di sicurezza

In questa sezione utilizziamo *SHE* per la verifica di una proprietà di sicurezza all'interno di un sistema. Il problema che affrontiamo è quello di un insieme di processi, che comunicano tra loro inviando e ricevendo messaggi, divisi in livelli di sicurezza ordinati. Per motivi di sicurezza si richiede che:

- un processo invii messaggi solo a processi di livello non inferiore,
- un processo riceva messaggi sono da processi di un livello non superiore.

Tali sistemi sono particolarmente utilizzati in ambito militare in cui ad esempio si possono avere i livelli di sicurezza: *non classificato*, *segreto* e *top secret*. Per le condizioni enunciate, un processo a livello di sicurezza *non classificato* non deve mai ricevere informazioni dai livelli *segreto* e *top secret*, ma deve potere inviare messaggi a processi in tali livelli, mentre uno di livello *segreto* può inviare messaggi ad un processo *top secret* ma non può ricevere messaggi da esso.

Quello che mostriamo è come rappresentare tale problema in *SHE* e come verificare, mediante gli strumenti messi a disposizione da *SHE*, che non vengano violate le condizioni enunciate.

Nella soluzione che proponiamo sia i processi che i messaggi sono rappresentati con degli archi. Gli archi che rappresentano i processi incidono su tre nodi:

- il primo è comune a tutti i processi e rappresenta il loro canale di comunicazione,
- sul secondo incidono gli archi che rappresentano i messaggi da inviare,
- sul terzo incidono gli archi che rappresentano i messaggi ricevuti.

In figura 6.9 vediamo un sistema in cui il processo $P$ desidera inviare il messaggio $m$ e il processo $Q$ ha ricevuto il messaggio $n$.

Nel seguito consideriamo un sistema con due livelli di sicurezza: *high* e *low*. I processi del livello *low* possono inviare messaggi ad altri processi del livello *low* oppure a processi del livello *high* e ricevere messaggi da processi del livello *low*. I processi del livello *high* possono inviare messaggi solo a processi dello stesso livello e ricevere messaggi sia da processi del livello *high* che da

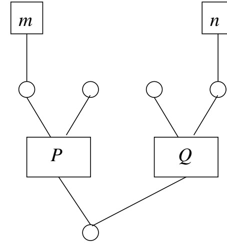



**Figura 6.9:** *Un sistema con due processi*

quelli del livello *low*. I processi del livello *high* sono etichettati con $H$, quelli del livello *low* con $L$.

I messaggi sono etichettati con $M$ e sono dotati della seguente produzione:

- $M(x) \xrightarrow{(x,\overline{s},y)} M(y)$.

I processi del livello *high* hanno una produzione per inviare un messaggio ad un processo dello stesso livello e una per ricevere un messaggio:

- $H(x,y,z) \xrightarrow{\substack{(x,h,w)\\(y,s,w)}} H(x,y,z)$
- $H(x,y,z) \xrightarrow{(x,\overline{h},z)} H(x,y,z)$

Le azioni $h$ e $\overline{h}$ vengono emesse da due archi $H$ che desiderano comunicare. L'arco che desidera ricevere un messaggio emette l'azione $\overline{h}$ e comunica il nodo su cui andrà ad incidere il messaggio ricevuto. L'arco che desidera inviare un messaggio si sincronizza con il messaggio da inviare mediante l'azione $s$ e comunica un nodo nuovo. Il nodo nuovo viene comunicato inoltre anche nella sincronizzazione con l'arco $H$ che riceve il messaggio, che verrà unificato con il nodo comunicato da questo, provocando lo *spostamento* dell'arco $M$ dal primo nodo dell'arco $H$ che invia il messaggio, al secondo nodo dell'arco $H$

che lo riceve. In figura 6.10 è presente un esempio di una transizione in cui avviene lo scambio di un messaggio.

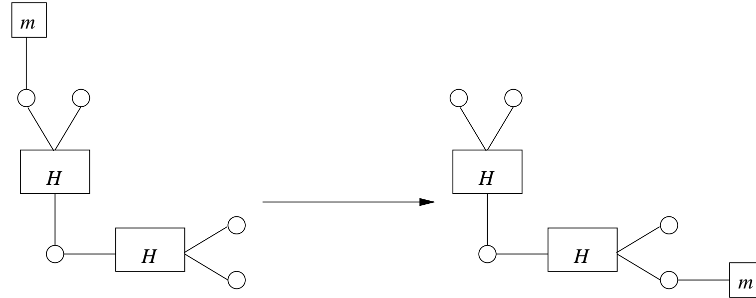



**Figura 6.10:** *Scambio di un messaggio tra due processi* high

I processi del livello *low* hanno una produzione per inviare un messaggio ad un processo dello stesso livello, una per inviare un messaggio ad un processo del livello *high* e una per ricevere un messaggio:

- $L(x,y,z) \xrightarrow[]{\substack{(x,l,w)\\(y,s,w)}} L(x,y,z)$
- $L(x,y,z) \xrightarrow[]{\substack{(x,h,w)\\(y,s,w)}} L(x,y,z)$
- $L(x,y,z) \xrightarrow{(x,\bar{l},z)} L(x,y,z)$

Per tutti gli archi assumiamo inoltre la presenza della produzione identità.

Disegnando in *SHE* queste produzioni e come grafo iniziale il grafo mostrato in figura 6.11, si ottiene il grafo degli stati mostrato in figura 6.12. Mediante *SHE* è possibile analizzare i singoli stati del sistema ma, soprattutto a causa dell'elevato numero di stati e di transizioni di stato, è difficile verificare che nessun messaggio passi da un processo *high* ad uno *low*. Per tale scopo è possibile utilizzare il modulo di verifica di *SHE*.

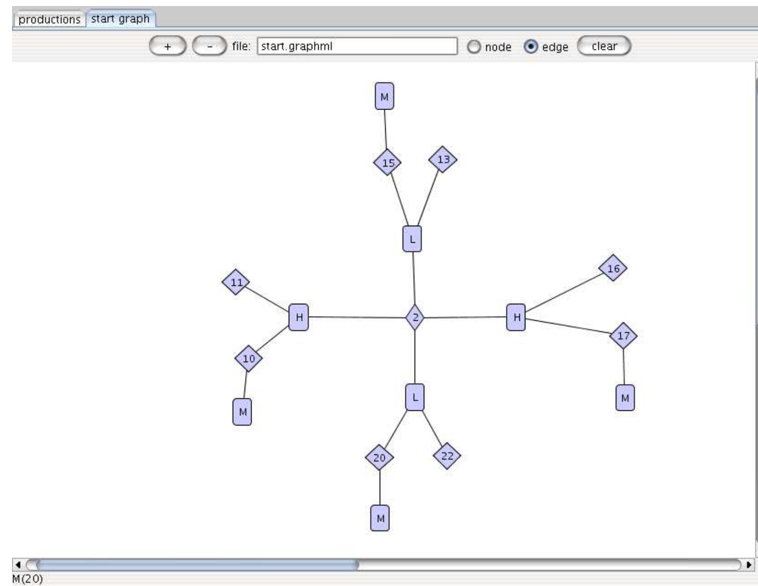



**Figura 6.11:** *Grafo iniziale*

Selezioniamo il pulsante select dell'interfaccia di *GraVE* e creiamo un nodo etichettato MessagePass (vedi figura 6.13). Tale espressione permette di calcolare le transizioni in cui avviene il passaggio di un messaggio tra processi. In particolare, l'utente specifica due livelli di sicurezza (non necessariamente diversi), che chiamiamo *sorgente* e *destinazione*, e il sistema seleziona, nel grafo degli stati, tutte le transizioni in cui c'è il passaggio di un messaggio da un processo appartenente al livello di sicurezza sorgente ad uno appartenente a quello di destinazione. Alla pressione del tasto show nel pannello di selezione, ci viene chiesto di specificare i due livelli sorgente e destinazione. Ad esempio, specificando L per entrambi i livelli, verranno selezionate le transizioni in cui c'è uno scambio di messaggi tra processi del livello *low* (figura 6.14).

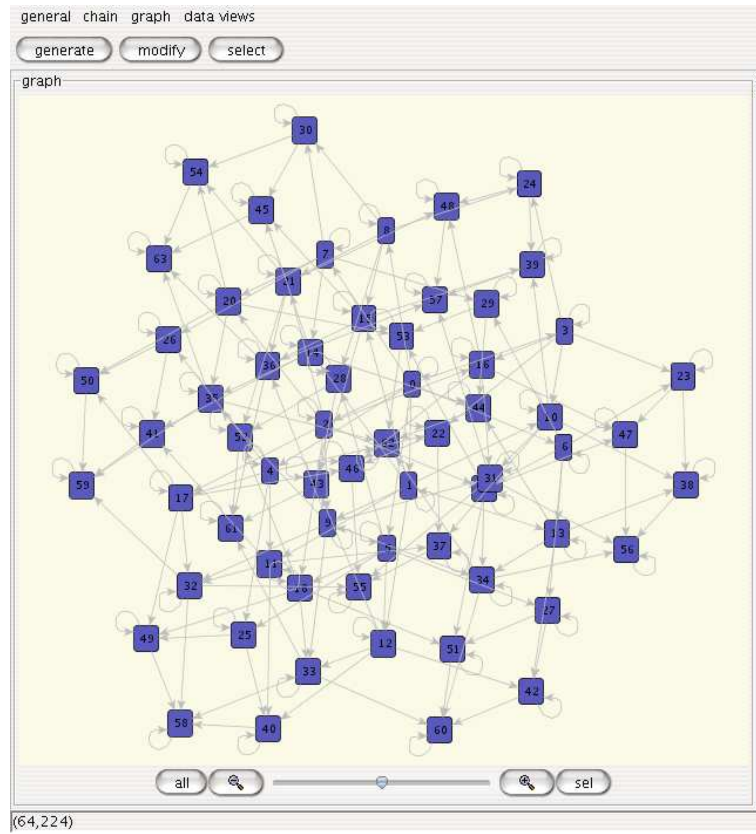



**Figura 6.12:** *Grafo degli stati*

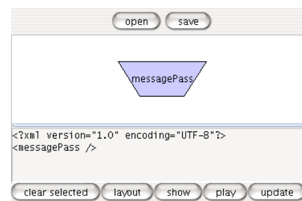



**Figura 6.13:** *Espressione per la verifica*

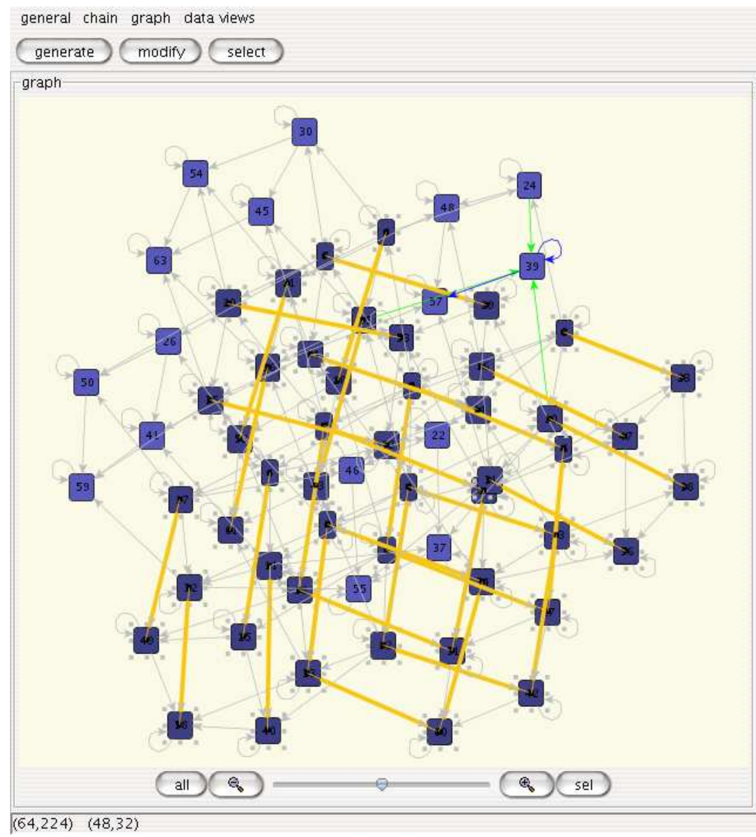



**Figura 6.14:** *Transizioni in cui c'è comunicazione tra messaggi del livello low*

Tale meccanismo può essere utilizzato per *verificare* che il sistema descritto soddisfi le caratteristiche richieste, cioè che nessun messaggio viene passato da un processo del livello *high* ad uno del livello *low*. Per verificare tale proprietà è necessario specificare H come livello sorgente e L come livello destinazione. Premendo il pulsante show non viene selezionata nessuna transizione nel grafo degli stati, verificando quindi la correttezza del sistema implementato.

Osserviamo che la verifica che è necessario compiere in tale problema non riguarda una condizione di *safety* degli stati, come può essere la presenza in ogni stato di un arco con una determinata etichetta, oppure il fatto che due archi non condividano mai un nodo, bensì una condizione che riguarda le relazioni tra stati consecutivi. Per verificare la condizione enunciata, è necessario controllare, in ogni transizione $A \rightarrow B$, che nessuno degli archi che rappresentano messaggi connessi ad un arco *high* in $A$, si trovi connesso ad un arco *low* in $B$. Osserviamo quindi che, sebbene per condizioni di *safety* sui singoli stati si potrebbe scegliere di utilizzare le *invarianti* di Mur$\varphi$, una verifica come quella appena vista risulterebbe di difficile implementazione con le invarianti, poiché i controlli effettuati da queste riguardano esclusivamente il contenuto degli stati e non relazioni tra stati consecutivi.

# Capitolo 7

# Conclusioni

In questo lavoro abbiamo realizzato un sistema di supporto alla programmazione distribuita. Questo sistema integra la tecnologia del model checking (generalmente usata per verificare sistemi già completamente sviluppati) all'interno di un ambiente di *sviluppo* software, dove l'attività di verifica di soluzioni *parziali* può indirizzare il progettista verso raffinamenti opportuni. Questo è uno degli aspetti sperimentali di *SHE*.

Un secondo importante aspetto è l'uso di un modello basato su riscrittura di grafi, che pone l'enfasi sugli aspetti topologici delle applicazioni, particolarmente cruciali nella computazione distribuita. I casi studio presentati mostrano comunque che l'approccio grafico è utile in generale anche per applicazioni non distribuite.

Un terzo aspetto che vogliamo sottolineare del lavoro svolto riguarda le scelte architetturali. Nella progettazione di *SHE* abbiamo deciso di rendere le diverse parti del sistema indipendenti e riusabili. Questo approccio è stato seguito sia nel *progetto generale* , ossia *SHE*, sia nelle parti che lo compongono, ed in particolar modo in *GraVE*. In tal modo, singole componenti

possono essere modificate, migliorate ed espanse, senza dover effettuare modifiche al resto. Inoltre, per la sua estrema riusabilità, *GraVE* si presenta uno strumento capace in intervenire in diversi ambiti applicativi, di cui *SHE* è solo un esempio.

Nel resto del capitolo analizziamo i possibili sviluppi di *SHE*. In particolare proponiamo delle modifiche all'editor delle produzioni per consentire di esprimere *schemi* di produzione, che semplificherebbero al programmatore la realizzazione di un sistema all'interno di *SHE*. Inoltre mostriamo come *GraVE* possa essere utilizzato per esplorare generiche computazioni Mur$\varphi$. Concludiamo proponendo l'implementazione di *SG* con restrizione all'interno di *SHE*, che consentirebbe di aumentare il potere espressivo del sistema, e illustrando come *SHE* possa essere utilizzato come sistema per la programmazione in calcoli diversi da *SG* sfruttando un meccanismo di traduzione in *SG* dei suddetti calcoli.

**Schemi di produzione.** Come visto nell'esempio 6.1 sarebbe comodo poter esprimere all'interno di *SHE schemi* di produzione, oltre a semplici produzioni. L'esempio dell'ordinamento, discusso in 6.1 può infatti essere espresso completamente da soli tre schemi di produzione, mentre per utilizzare tali schemi all'interno di *SHE* è necessario istanziarli e, nel caso in cui la massima etichetta nel grafo sia $n$, il numero di produzioni da creare è dell'ordine di $n^2$. Sebbene il processo sia semplice è tuttavia tedioso e può portare facilmente ad introdurre errori nel sistema. Risulterebbe quindi interessante includere all'interno di *SHE* la possibilità di esprimere direttamente schemi di produzione. Per fare ciò si dovrebbe modificare l'editor in maniera da permettere all'utente di disegnare schemi di produzione e generare da questi una serie di produzioni che rappresentano le istanze dello schema. L'editor

dovrebbe quindi dare la possibilità di introdurre delle variabili nel disegno di una produzione per le quali l'utente specifica l'insieme dei possibili valori. A questo punto l'editor si occuperebbe di generale le produzioni, istanziando ogni schema di produzione con tutti i possibili valori ammessi per le variabili.

***SHE* come esploratore di computazioni Mur$\varphi$.** Poichè il grafo generato da *SHE* contiene (nella modalità di *debugging*) il contenuto di ogni stato incontrato da Mur$\varphi$ (vedi 4.3.3), *SHE* può essere utilizzato per esplorare *qualsiasi* computazione Mur$\varphi$.

Sebbene Mur$\varphi$ offra già la possibilità di conoscere il contenuto dei singoli stati e le regole applicate per passare da uno stato ad un altro, è tuttavia difficile ricostruire una sequenza di cambiamenti di stato affidandosi soltanto a tale output. In tal caso *SHE* visualizza il grafo degli stati, etichettando gli archi con la regola applicata per la transizione di stato e consentendo di vedere il contenuto di ciascuno stato. Tuttavia, poichè il contenuto di tale stato verrebbe visualizzato all'interno di *GraVE* nel formato testuale restituito da Mur$\varphi$, è possibile implementare un DataViewer che analizzi lo stato e ne fornisca una diversa rappresentazione.

***SG* con restrizione.** Nei precedenti capitoli abbiamo illustrato il modello dei *Synchronizing Graphs* che è alla base di *SHE*, ed abbiamo mostrato come questo sia stato implementato usando un model-checker. L'implementazione di *SG* in Mur$\varphi$ risulta in generale abbastanza intuitiva: la struttura dati è semplice e la corrispondenza tra *SG* e Mur$\varphi$ è di una regola per ogni produzione più la regola sync. Ciò è il risultato dell'estrema semplicità del modello [CTT04] che, rispetto alla *riscrittura sincronizzata di ipergrafi* (*SHR*) [FMT01], riduce drasticamente la complessità del calcolo. Un primo

tentativo di implementare l'*SHR* in Mur$\varphi$ aveva presentato subito notevoli difficoltà, derivate principalmente dall'estrema complessità del calcolo. Tuttavia la versione di *SG* utilizzata all'interno di *SHE* non presenta la possibilità di restrizione sui nodi. Una versione dei *Syncronizing Graphs* che include la restrizione è presentata in [CT04, Tib04]. Un'interessante evoluzione per *SHE* è quindi quella di utilizzare *SG* con l'operatore di restrizione.

***SHE* come framework comune.** Come è stato dimostrato in [Tib04] è possibile, mediante il modello *SG*, simulare il *CCS Distribuito* [RH01] e il calcolo dei *Mobile Ambients* [CG98]. Inoltre in [CT04] la versione con restrizione di *SG* viene utilizzata per simulare computazioni del *Fusion Calculus* [PV98]. Tali risultati supportano la proposta di utilizzare *SG* come *framework* unico per modellare calcoli differenti. Sarebbe dunque interessante integrare all'interno di *SHE* la possibilità di lavorare *direttamente* in tali calcoli, ossia simulare i suddetti calcoli mediante riscritture di ipergrafi senza però mostrare all'utente di *SHE* tale livello di simulazione. Ciò può essere realizzato creando un opportuno modulo di composizione, da sostituire a quello di default fornito da *SHE*, che consenta di esprimere termini di uno dei suddetti calcoli e che generi in output una serie di produzioni *SG* e un grafo iniziale. Tali file seguirebbero a questo punto lo stesso processo a cui sono sottoposti i file generati disegnando produzioni *SG*, producendo infine il grafo degli stati. Mediante *GraVE* sarebbe dunque possibile analizzare tale computazione *SG*, che rappresenta tuttavia una computazione del calcolo simulato. Inoltre, implementando dei DataViewer che, prendendo in input un ipergrafo, restrituiscano termini del calcolo simulato, è possibile *nascondere* completamente il livello degli ipergrafi. In tal modo l'utente non vedrebbe mai la struttura sottostante basata sugli ipergrafi ma solo i termini del calcolo

prescelto.

Ricapitolando, per utilizzare *SHE* come ambiente per la programmazione con un generico calcolo $L$, è necessario fornire:

- un modulo per la composizione di termini di $L$ che produca delle produzioni *SG* e un ipergrafo iniziale
- un DataViewer che effettui la traduzione di un ipergrafo in un termine del calcolo $L$

In tal modo *SHE* diventerebbe un editor per diversi linguaggi, con un motore di calcolo basato su *SG*. Tale modello sarebbe inoltre espandibile con nuovi linguaggi fornendo opportuni moduli di traduzione.

# Bibliografia


[BEH+01] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. Graphml progress report: Structural layerproposal. In *9th Intl. Symp. Graph Drawing (GD '01)*, Lecture Notes in Computer Science, pages 501–512. Springer-Verlag, 2001.

[BPSM97] T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible Markup Language. *World Wide Web Journal*, 2(4):29–66, Winter 1997. `http://www.w3.org/TR/REC-xml`.

[CD93] C.N. Ip and D.L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[CM88] K. Mani Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1988.

[CT04] P. Cenciarelli and A. Tiberi. Synchronizing graphs. 2004. Submitted to *Express 2004*.

[CTT04] P. Cenciarelli, I. Talamo, and A. Tiberi. Ambient Graph rewriting. In *Proceedings of WRLA 2004*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[D. 96] D. L. Dill. The Murphi Verification System. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 390–393, New Brunswick, NJ, USA, 1996. Springer Verlag.

[DDHY92] David L. Dill, Andreas J. Drexler, A. J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society, pp. 522-525*, IEEE Computer Society, 1992.

[FMT01] G. Ferrari, U. Montanari, and E. Tuosto. A LTS semantics of ambients via graph synchronization with mobility. *ITCS*, 2001.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[LGM+95] L.Yang, D. Gao, J. Mostoufi, R. Joshi, and P.Loewenstein. System design methodology of UltraSPARC™-I. In *32nd Design Automation Conference*, pages 7–12, 1995.

[Mit98] John C. Mitchell. Finite-state analysis of security protocols. In *Computer Aided Verification*, pages 71–76, 1998.

[MMS97] J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Mur$\varphi$. In *IEEE Symposium on Security and Privacy*, pages 141–153, 1997.

[MPT78] M.D. McIllroy, E.N. Pinson, and B.A. Tague. Unix time-sharing system forward. *The Bell System Technical Journal*, 57(6 (part 2)):p. 1902, 1978.

[MU] Murphi description language and verifier. `http://verify.stanford.edu/dill/murphi.html`.

[PITZ02] Giuseppe Della Penna, Benedetto Intrigila, Enrico Tronci, and Marisa Venturini Zilli. Exploiting transition locality in the disk based Mur$\varphi$ verifier. In *4th International Conference on "Formal Methods in Computer Aided Verification" (FMCAD)*. Springer-Verlag, 2002.

[PV98] Joachim Parrow and Bjorn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Logic in Computer Science*, pages 176–185, 1998.

[RH01] James Riely and Matthew Hennessy. Distributed processes and location failures. *Theoretical Computer Science*, 266(1–2):693–735, 2001.

[SD95] Ulrich Stern and David L. Dill. Automatic verification of the SCI cache coherence protocol. In *Conference on Correct Hardware Design and Verification Methods*, pages 21–34, 1995.

[Tib04] A. Tiberi. Computazione distribuita come riscrittura di ipergrafi. Master's thesis, Università La Sapienza - Roma, 2004.