Università degli Studi di Udine
Dipartimento di Matematica e Informatica
Corso di Laurea in Tecnologie Web e Multimediali

Tesi di Laurea

Verifica di proprietà locali su BRS

Candidato:
Luca Geatti

Relatore:
Prof. Marino Miculan

Co-relatore:
Dott. Marco Peressotti

Anno Accademico 2014-2015

Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

Ai miei genitori

Indice

Elenco delle figure

1
Introduzione

Il lavoro riportato in questa tesi nasce dal problema di verificare delle proprietà in un Sistema Reattivo Bigrafico (BRS). In particolare, si è studiato il modo di controllare il sistema durante la sua evoluzione e dunque di fermare quest'ultima appena le proprietà desiderate siano state raggiunte. Questo tipo di verifica va sotto il nome di "Model Checking".

I Sistemi Reattivi Bigrafici (BRS) sono un nuovo formalismo con il quale si possono rappresentare sistemi distribuiti, di qualsiasi tipo essi siano: da un sistema di smartphones ad un sistema biologico [2]. I BRS sono basati su un' importante struttura matematica: i bigrafi. Sono questi che permettono una facile trattazione dei vari "oggetti distribuiti" che compongono il sistema, e di come essi interagiscono tra di loro.

L'importanza dei bigrafi si può riscontrare nella loro flessibilità: essi costituiscono un *meta-modello*, con cui è possibile rappresentare sistemi di qualsiasi dominio si voglia. Di recente i bigrafi sono stati usati per creare delle Reti di Petri [3], come anche per controllare un sistema di robot mobili [5].

Un altro punto di forza dei bigrafi sta nella loro capacità di evolversi, potendo così rappresentare lo stato del sistema anche quando questo cambia. Si ha così a disposizione un Sistema Reattivo Bigrafico.

Questa tesi tratta il problema di come poter sapere se un dato BRS rispetti certe proprietà. Per esempio: se rappresentiamo una rete con un BRS, ci possiamo chiedere se, dato uno stato iniziale in cui il pacchetto parte dal mittente A, esso arrivi o meno al destinatario B che si trova a vari router di distanza da A. Oppure, cambiando dominio del problema, ci possiamo domandare se un automa rappresentato tramite bigrafi riconosca o meno una data stringa.

Il problema affrontato in questa sede prescinde quindi dal particolare dominio del problema, ed offre una soluzione generale, valida per qualsiasi BRS. Per fare questo,

si sono dovute affrontare varie problematiche. Tra le più importanti figurano:

- quando due bigrafi sono uguali? Un BRS evolve senza memoria degli stati precedenti in cui si è trovato. Questo problema, in concreto, può potenzialmente causare evoluzioni infinite del BRS: per esempio, il pacchetto nella rete può girare all'infinito tra due router, perchè il BRS si "dimentica" da dove il pacchetto è arrivato.

- come rappresentare le proprietà da verificare nel BRS? In particolare, posso rappresentare con un solo formalismo vari tipi di proprietà, dall'arrivo a destinazione di un pacchetto al riconoscimento di una stringa? Il problema maggiore è il fatto che il modo per rappresentarle deve essere generale tanto quanto i BRS. In sostanza si è scelto un modo che astraesse ancora una volta dal dominio scelto.

La struttura della tesi rispetta dunque queste problematiche:

Nel capitolo 2 verranno presentate formalmente le nozioni di Bigrafo e di BRS. Con esse, verrà anche descritta un'algebra per creare nuovi bigrafi a partire da bigrafi base.

Nel capitolo 3 si affronta il primo dei due principali problemi, che va sotto il nome di "isomorfismo tra bigrafi". La risoluzione di questo problema ci permetterà di poter affermare quando due bigrafi sono in pratica uguali o meno, e quindi di evitare evoluzioni infinite del BRS.

Si affronterà nel capitolo 4 il secondo problema, cioè quello delle proprietà. Esse verranno espresse sul calcolatore tramite una semplice logica a predicati. Si potranno così esprimere tutte le proprietà desiderate, indipendentemente dal dominio del sistema. Grazie a queste proprietà, si arriverà all'implementazione di un Model Checker per i bigrafi.

Nel capitolo 5 verranno presentati alcuni esempi, presi da vari domini. Si potrà apprezzare l'importanza di avere un Model Checker e della semplicità con cui si possono esprimere le proprietà da verificare.

Infine si sono tratte le conclusioni sull'intero lavoro. Verranno presentate alternative per l'implementazione dell'isomorfismo e delle proprietà.

2

Bigrafi e BRS

In questo capitolo vengono presentate le descrizioni formali di bigrafo e di Sistema Reattivo Bigrafico. Si vedrà come l'importanza dei bigrafi risieda nel fatto di rappresentare contemporaneamente i concetti di *località* e *connessione*.

L'algebra dei bigrafi è stata per gran parte costruita sulla base della Teoria delle Categorie, che in questa sede non verrà introdotta. Le definizioni ed i teoremi sono stati presi da [4], a cui si rimanda per i dettagli sulla Teoria delle Categorie. Infine si rimanda all'Appendice A per la descrizione della terminologia usata.

2.1 Definizione informale di Bigrafo

L'idea fondamentale alla base della loro teoria, è che ogni bigrafo sia composto da due strutture del tutto *indipendenti* sullo *stesso* insieme di nodi. Queste due strutture si chiamano *Place Graph* e *Link Graph* e modellano rispettivamente la località e la connessione.

Nell'introduzione, si è accennato al fatto che i bigrafi sono flessibili e adatti a rappresentare ogni dominio. Questo è possibile grazie al concetto di *segnatura*, che è l'analogo ad una grammatica per un linguaggio.

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

Quindi, dare una segnatura ad un bigrafo significa associare ad ogni nodo sia un tipo sia il suo numero di porte. L'equivalente grafico consiste nel disegnare nodi diversi con simboli diversi.

Notazione (Segnatura)**.** *Da qui in avanti, una segnatura* (K, ar) *verrà indicata nel seguente modo:*

$$K = \{K_1 : a_1, \ldots, K_n : a_n\}$$

dove ogni nodo di tipo (controllo) K_i *ha arietà* a_i*.*

2.1.1 Esempio

Diamo un primo esempio informale di bigrafo.

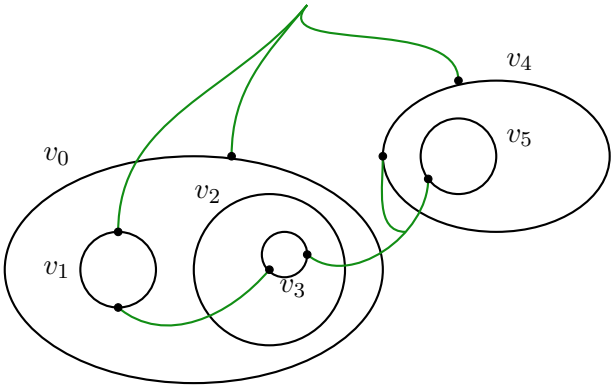
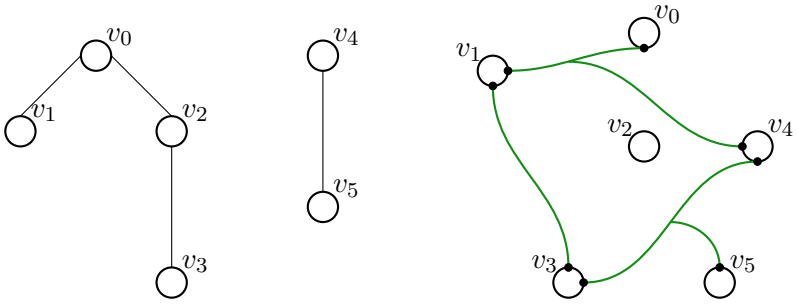


Figura 2.1: Un semplice bigrafo.

In figura 2.1 vediamo un bigrafo B, con una data segnatura. Si vede subito come ci siano dei controlli diversi. Per esempio, il nodo v_0 ha una sola porta mentre il nodo v_1 ne ha due. L'informazione che quest'ultimo sia contenuto in v_0 è rappresentata nel place graph di figura 2.2.a. Le interconnessioni dei vari nodi sono invece riportate nel rispettivo link graph di figura 2.2.b.



(a) Place Graph (b) Link Graph

Figura 2.2: Le due strutture ortogonali del bigrafo B.

Un'importante caratteristica dei bigrafi e della loro algebra è la possibilità di essere composti, cioè di formare un nuovo bigrafo da due bigrafi di partenza. Questo è equivalente al problema di considerare un bigrafo *parte* di un altro. Si vedrà che questa operazione va sotto il nome di *composizione*. Se prendiamo il bigrafo B di cui sopra, allora è possibile scrivere un'equazione di questo tipo:

$$B = A \circ C$$

Per renderla possibile dobbiamo aggiungere struttura ai bigrafi. Si introducono quindi le interfacce interne ed esterne, sia per il place graph sia per il link graph:

- L'interfaccia esterna ed interna del place graph sono un numero naturale n che possiamo trattare come un ordinale: l'interfaccia n indica l'insieme $\{0, \ldots, n-1\}$. Se l'interfaccia esterna è k, allora si dice che ci sono k *radici*. Se l'interfaccia interna è h, allora si dice che ci sono h *siti*.

- Per il link graph, invece, le interfaccie sono *insiemi* di nomi, come per esempio $\{x, y\}$. Se un link graph ha un interfaccia esterna del tipo $\{a, b, c\}$, allora diciamo che ci sono tre *outername* chiamati a, b e c. Se l'interfaccia interna è del tipo $\{x, y\}$, allora diciamo che il bigrafo ha due *innername* chiamati x e y.

Il concetto fondamentale delle interfacce è che servono per l'unione dei due bigrafi. Per esempio, prendendo il place graph, nell'operazione di composizione $A \circ C$, i siti di A dovranno *concordare* (vedremo una descrizione formale di questo concetto) con le radici di C. Per il link graph il concetto è lo stesso: gli innername di A dovranno unirsi con gli outername di C.

Si consideri per esempio il bigrafo della figura 2.1. Se vogliamo trovare due bigrafi A e C tali che $B = A \circ C$, allora dobbiamo rispettare le condizioni con cui si può effettuare l'operazione di composizione. Le figure 2.3 e 2.5 rappresentano rispettivamente i due bigrafi C ed A. Si noti ancora una volta che le radici di C si *uniscono* ai siti di A, e gli outername di C fanno lo stesso con gli innername di A.

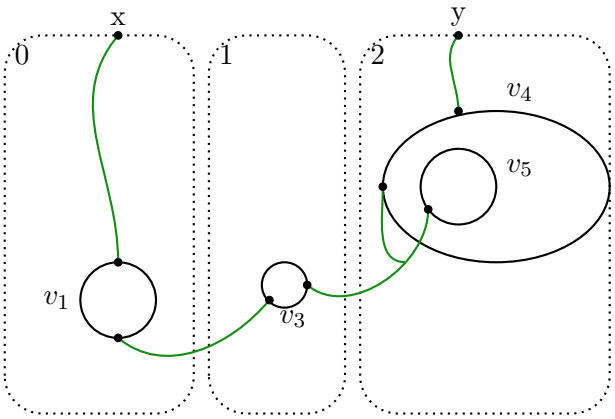
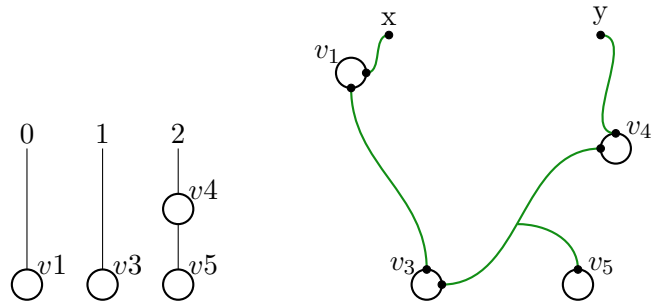


Figura 2.3: Il bigrafo C.



(a) Place Graph di C (b) Link Graph di C

Figura 2.4: Bigrafo C.

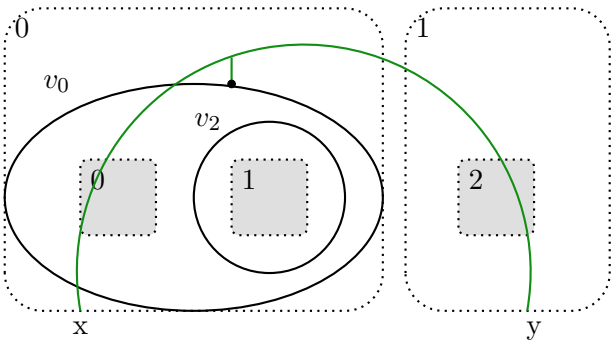
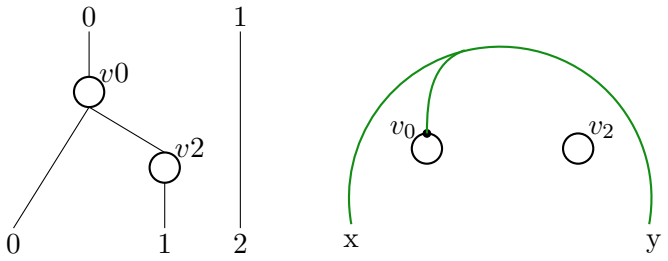


Figura 2.5: Il bigrafo A



(a) Place Graph di A (b) Link Graph di A

Figura 2.6: Bigrafo A.

2.2 Definizione formale di Bigrafo

Siamo ora pronti per definire separatamente i concetti di *Place Graph* e *Link Graph*. Quelli descritti, che noi chiameremo semplicemente bigrafi, in realtà si chiamano *concrete bigraphs*, per distinguerli da quelli astratti. In questa sede si tratterà solo di bigrafi concreti.

Si rimanda all'appendice A per la notazione usata.

2.2.1 Place Graph

Il place graph è una delle due strutture fondamentali di ogni bigrafo. E' una *foresta* e rappresenta l'informazione di *nesting*, ovvero quali nodi si trovano all'interno di altri. Come già notato in figura 2.2.a, le radici ed i siti sono *ordinati*, essendo essi rappresentati da un ordinale. In definitiva, quindi, il place graph è una foresta ordinata, in cui solo le radici ed i siti sono ordinati.

Definizione 2.2.1 (Place Graph)**.** *Un place graph*

$$F = (V_F, ctrl_F, prnt_F) : m \to n$$

è una tripla avente un'interfaccia interna m ed un'interfaccia esterna n, entrambe ordinali finiti.Queste indicano rispettivamente i siti *e le* radici *del place graph. F ha un insieme finito V_F di* nodi*, una* control map *$ctrl : V_F \to K$, e una* parent map

$$prnt : m \uplus V_F \to V_F \uplus n$$

che è aciclica, cioè se $prnt_F^i(v) = v$ allora $i = 0$.

Questa definizione formale ricalca ciò che è già stato notato nell'esempio della sottosezione 2.1.1. Infatti, si noti che il place graph F ha m siti ed n radici, entrambi ordinati, che costituiscono rispettivamente la sua interfaccia interna ed esterna.

La funzione $ctrl_F$ associa ad ogni nodo un controllo, cioè un nome. Come già notato, l'equivalente grafico sta nel disegnare con simboli diversi nodi con controllo diverso. Infine, la funzione $prnt_F$ associa ad ogni nodo interno o sito il suo genitore, che può essere a sua volta un altro nodo interno o una radice.

Questa funzione è di fondamentale importanza, in quanto è il cuore del place graph: rappresenta l'informazione di quali nodi si trovano all'interno di altri. É *aciclica*, nel senso che n sue *composizioni* non porteranno mai al nodo di partenza. In formule:

$$prnt_F \circ prnt_F \circ \cdots \circ prnt_F(v) \neq v$$

Questo è equivalente a dire che la struttura dati rappresentata dal place graph è una *foresta*.

2.2.2 Link Graph

La seconda struttura dati fondamentale è il link graph. Esso rappresenta l'informazione di *connessione* tra i nodi del bigrafo. E' un *ipergrafo*, infatti un arco può collegare due o più nodi.

Definizione 2.2.2 (Link Graph)**.** *Un link graph*

$$F = (V_F, E_F, ctrl_F, link_F) : X \to Y$$

è una quadrupla avente un'interfaccia interna X ed una interfaccia esterna Y, chiamate rispettivamente gli inner names *e* outer names *del link graph. F ha un insieme finito V_F di nodi e E_F di archi (*edges*), una* control map *$ctrl : V_F \to K$, e una* link map*:*

$$link_F : X \uplus P_F \to E_F \uplus Y$$

dove $P_F = \{(v, i) \mid v \in V_F \wedge i \in ar(ctrl_F(v))\}$ è l'insieme delle porte *di F. Quindi, (v, i) è l' i-esima porta del nodo v. Chiamiamo $X \uplus P_F$ i* punti *di F, mentre $E_F \uplus Y$ i suoi* link*.*

Notiamo subito come alcune nozioni rimangono invariate dal place graph. Per esempio, V_F e $ctrl_F$ non cambiano. Si aggiungono però due concetti:

- E_F: è l'insieme di archi del link graph F. Sono oggetti del tutto indipendenti dai nodi.

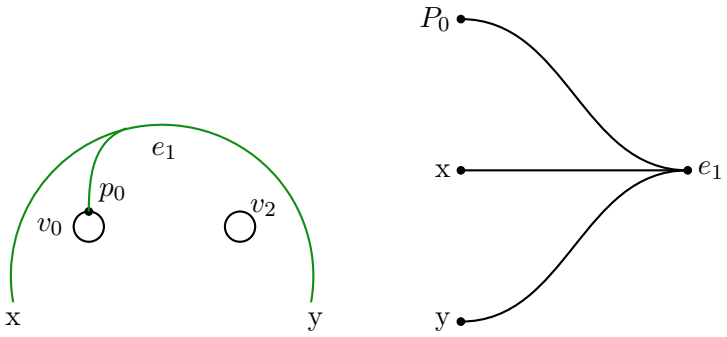
- $link_F$: è la funzione che consente di creare l'*ipergrafo*.

Dalla definizione 2.1, sappiamo che un dato controllo ha un ben preciso numero di porte. La funzione $ar : K \to \mathbb{N}$ consente di calcolare il numero di porte di un dato controllo: $ar(ctrl_F(v)$ restituisce un numero naturale, che è il numero di porte del controllo del nodo v. Grazie alla funzione *ar*, nel link graph ci sleghiamo totalmente dal concetto di *nodo*, e siamo in grado di sostituirlo con il concetto di *porta*.

Introduciamo ora due concetti:

- l'insieme dei *punti* di F è l'insieme che comprende tutte le sue porte (P_F) e tutti i suoi inner names (X).
- l'insieme dei *link* di F è l'insieme che comprende tutti i suoi archi (edges) e tutti i suoi outer names (Y).

La funzione $link_F$ avrà come dominio l'insieme dei punti e come codominio l'insieme dei link. Possiamo quindi pensarla come è raffigurata in figura 2.7.b. La funzione $link_F$ è perciò quella che crea il vero e proprio ipergrafo: associa ad ogni punto *uno ed un solo* link. Sfruttando la *non-iniettività* della funzione possiamo creare l'ipergrafo.



(a) Notazione grafica (b) Notazione funzionale

Figura 2.7: Link Graph decomposto.

In figura 2.7.a si vede il link graph nella sua notazione usuale, cioè con anche i nodi disegnati. Invece, in 2.7.b si vede la notazione sotto forma di funzione, dove si specifica dominio (p_0, x e y) e codominio (e_1).

2.2.3 Bigrafo

Un bigrafo è semplicemente l'unione del place graph e del link graph. E' importante notare che esse condividono lo *stesso* insieme di nodi.

Definizione 2.2.3 (Bigrafo)**.** *Un bigrafo*

$$F = (V_F, E_F, ctrl_F, prnt_F, link_F) : \langle n, X \rangle \to \langle m, Y \rangle$$

consiste in un place graph $F^P = (V_F, ctrl_f, prnt_F) : n \to m$ *e in un link graph* $F^L = (V_F, E_F, ctrl_F, link_F) : X \to Y$*.*

Notazione (Bigrafo)**.** *Un bigrafo* F *viene spesso indicato tramite le sue interfacce:*

$$F = \langle n, X \rangle \to \langle m, Y \rangle$$

stando ad indicare che F *ha* n *siti,* X *inner names,* m *radici e* Y *outer names.*

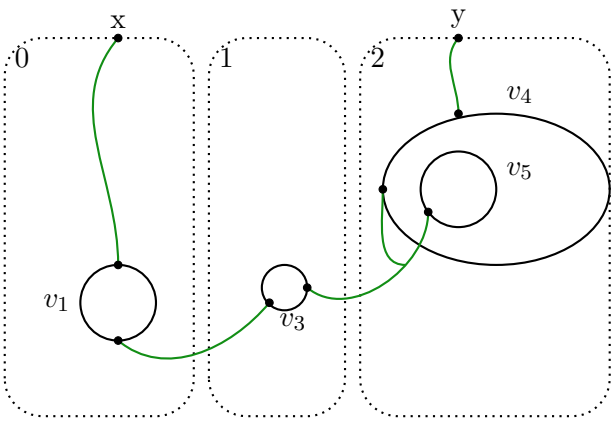


Figura 2.8: Esempio di bigrafo.

Il bigrafo della figura 2.8, rispettando la notazione, è scritto in questo modo: $F = \langle 0, \emptyset \rangle \to \langle 3, \{x, y\} \rangle$.

2.3 Operazioni sui Bigrafi

Nelle sezioni precedenti, si è definita formalmente la nozione di bigrafo. Ora, si studiano le varie operazioni possibili, ovvero come ottenere un nuovo bigrafo da due bigrafi di partenza. Queste operazioni saranno utili per la prossima sezione, che tratterà dell'algebra dei bigrafi.

2.3.1 Traduzione di Supporto

La prima operazione tratta di come si può ottenere un nuovo bigrafo avendo a disposizione un solo bigrafo base e una funzione biettiva. Si dimostra che applicando tale biiezione ai nodi e archi del bigrafo di partenza si determina *unicamente* il bigrafo risultante.

Definizione 2.3.1 (Traduzione di Supporto)**.** *Per ogni place graph, link graph e per ogni bigrafo è assegnato un insieme finito $|F|$, il suo* supporto*. Per un place graph, definiamo $|F| = V_F$, mentre per un link graph o un bigrafo definiamo $|F| = V_F \uplus E_F$. Per due bigrafi F e G, una* traduzione di supporto *$\rho : |F| \to |G|$ da F a G consiste in due biiezioni $\rho_V : V_F \to V_G$ and $\rho_E : E_F \to E_G$ che rispettano la struttura, nel seguente senso:*

- *ρ preserva i controlli, cioè: $ctrl_G \circ \rho_V = ctrl_F$. Ne segue che ρ induce una biiezione sulle porte $\rho_P : P_F \to P_G$, definita da $\rho_P((v, i)) = (\rho_V(v), i)$.*
- *ρ modifica le mappe sulla struttura (prnt e link) in questo modo:*

$$prnt_G \circ (id_m \uplus \rho_V) = (id_n \uplus \rho_V) \circ prnt_F$$
$$link_G \circ (id_X \uplus \rho_P) = (id_Y \uplus \rho_E) \circ link_F$$

Come detto prima, data la biiezione ρ e il bigrafo F, queste condizioni determinano unicamente G, che denotiamo con $\rho \cdot F$ e chiamiamo *traduzione di supporto* di F tramite ρ. Chiamiamo F e G *support equivalent* ($F \simeq G$) se esiste una tale traduzione di supporto.

Dati due bigrafi, il problema di trovare una traduzione di supporto tra i due è equivalente al problema di stabilire quando essi sono uguali. Per cui, da qui in seguito si userà il termine *traduzione di supporto* come sinonimo di *isomorfismo.*

Definizione 2.3.2 (Isomorfismo)**.** *Due bigrafi F e G si dicono isomorfi se e solo se esiste una traduzione di supporto tra F e G, cioè se e solo se F e G sono* support equivalent *($F \simeq G$).*

2.3.2 Composizione

L'operazione di *composizione* è denotata dal simbolo $\circ$ e permette di scrivere equazioni del tipo $B = A \circ C$, come nell'esempio 2.1.1. La composizione necessita del concetto di interfaccia: come già notato, l'interfaccia interna di A deve concordare con l'interfaccia esterna di C. Con una prima approssimazione, possiamo dire che C deve essere incluso *dentro* A.

Definizione 2.3.3 (Composizione)**.** *Si trattano separatamente i casi del place graph e del link graph:*

- Place Graph*: Se $F : k \to m$ e $G : m \to n$ sono due place graph con supporti disgiunti ($|F| \# |G|$), la loro composizione*

$$G \circ F = (V, ctrl, prnt) : k \to n$$

ha i nodi $V = V_F \uplus V_G$ *e la control map* $ctrl = ctrl_F \uplus ctrl_G$*. La sua parent map* $prnt$ *è definita come segue: se* $w \in k \uplus V_F \uplus V_G$ *è un sito o un nodo di* $G \circ F$*, allora :*

$$prnt(w) \stackrel{def}{=} \begin{cases} prnt_F(w), & se\ w \in k \uplus V_F \ \wedge \ prnt_F(w) \in V_F \\ prnt_G(j), & se\ w \in k \uplus V_F \ \wedge \ prnt_F(w) = j \in m \\ prnt_G(w), & se\ w \in V_G \end{cases}$$

Il place graph identità *su* m *è* $id_m \stackrel{def}{=} (\emptyset, \emptyset, id_m) : m \to m$

Si noti come l'unione delle due interfacce (quella esterna di A *e quella interna di* B*) sia modellata dalla seconda riga. In particolare,* $prnt_F(w) = j$ *sta ad indicare che* j *è una radice di* F *e* w *è uno dei suoi figli. Dato che* j *è una radice, essa sarà un intero nell'insieme* $\{0, \ldots, m-1\}$*, e dovrà quindi appartenere anche all'interfaccia interna di* G*, dove* j *sarà un sito. La parent map di* $G \circ F$*su* w *sarà quella di* G *sul suo sito* j*. Quindi la seconda riga modella la seguente azione: mantengo la parent map per tutti i nodi interni dei due bigrafi, e al momento dell'unione delle due interfacce unisco la radice i-esima di* F *con il sito i-esimo di* G*, per ogni* $i \in \{0, \ldots, m-1\}$*.*

- Link Graph*: Se* $F : X \to Y$ *e* $G : Y \to Z$ *sono due link graph con supporti disgiunti (* $|F| \# |G|$*), la loro composizione*

$$G \circ F = (V, E, ctrl, link) : X \to Z$$

ha $V = V_F \uplus V_G$*,* $E = E_F \uplus E_G$*,* $ctrl = ctrl_F \uplus ctrl_G$ *e la sua link map* $link$ *è definita come segue: se* $q \in X \uplus P_F \uplus P_G$ *è un punto di* $G \circ F$*, allora*

$$link(q) \stackrel{def}{=} \begin{cases} link_F(q), & se\ q \in X \uplus P_F \ \wedge \ link_F(q) \in E_F \\ link_G(y), & se\ q \in X \uplus P_F \ \wedge \ link_F(q) = y \in Y \\ link_G(q), & se\ q \in P_G \end{cases}$$

Il link graph identità *su* X *è* $id_X \stackrel{def}{=} (\emptyset, \emptyset, \emptyset, id_X) : X \to X$ *Anche qui, si noti come la seconda riga modelli l'unione tra le interfacce. In particolare, se* $link_F(q) = y$ *e* $y \in Y$*, si ha che il punto* q *del bigrafo* F *è collegato al suo outer name* y*. Dato che* Y *è anche l'interfaccia interna di* G*, si ha che* y *è un inner name di* G*. La seconda riga quindi rappresenta l'unione tra gli outername di* F *e gli inner name di* G *che hanno lo stesso nome.*

- Bigrafo*: Se $F : I \to J$ e $G : J \to K$ sono due bigrafi con supporti disgiunti ($|F|\#|G|$), la loro composizione*

$$G \circ F \stackrel{def}{=} \langle G_P \circ F_P, G_L \circ F_L \rangle : I \to K$$

ed il bigrafo identità *su $I = \langle m, X \rangle$ è $\langle id_m, id_x \rangle$.*

Si noti come l'operazione di composizione *unisca* le interfacce dei due bigrafi, creando un unico bigrafo risultato, cioè un'unica funzione *prnt* ed un'unica funzione *link*.

2.3.3 Giustapposizione

Si definisce ora un'altra operazione per creare un bigrafo da altri due base. Si chiama *giustapposizione* e consiste nell'affiancare un bigrafo ad un altro. Questa operazione è possibile solo se i due bigrafi sono *disgiunti*. Spesso viene anche chiamata *prodotto*.

Definizione 2.3.4 (Bigrafi disgiunti)**.** *Due place graphs $F_i (i = 0, 1)$ sono disgiunti se $|F_0|\#|F_1|$. Due link graph $F_i : X_i \to Y_i$ sono* disgiunti *se $X_0\#X_1$, $Y_0\#Y_1$ e $|F_0|\#|F_1|$. Due bigrafi $F_i (i = 0, 1)$ sono* disgiunti *se $F_0^P \# F_1^P$ e $F_1^L \# F_1^L$.*

L'operazione di giustapposizione è monoidale, cioè è associativa ed ha un unità. Si definiranno quindi le proprietà dell'operazione e le sue unità.

Definizione 2.3.5 (Giustapposizione)**.** *Definiamo separatamente i casi del place graph e del link graph:*

- Place Graph*: la giustapposizione di due interfacce $m_i (i = 0, 1)$ è $m_0 + m_1$ e l'unità è 0. Se $F_i = (V_i, ctrl_i, prnt_i) : m_i \to n_i$ sono place graphs disgiunti (i= 0, 1), la loro giustapposizione $F_0 \otimes F_1 : m_0 + m_1 \to n_0 + n_1$ è data da:*

$$F_0 \otimes F_1 = (V_0 \uplus V_1, ctrl_0 \uplus ctrl_1, prnt_0 \uplus prnt_1^{'}),$$

dove $prnt_1^{'}(m_0 + i) = n_0 + j$ ogni volta che $prnt_1(i) = j$.

Informalmente, affianco due place graph avendo come risultato una foresta. Le radici aumenteranno quindi di numero. Per questo motivo è necessaria la funzione $prnt_1^{'}$: supponiamo che F_0 abbia due radici e F_1 una sola. I nodi figli diretti di F_1 non possono più puntare alle loro vecchie radici, per esempio la radice 0, perché ora essa è la radice di F_0. La parent map di F_1 dovrà quindi venire traslata in questo modo: tutti i nodi di F_1 che puntano alla sua radice 0, ora punteranno alla nuova radice $0 + 2 = 2$. Questo è quello che fa la funzione $prnt_1^{'}$.

- Link Graph*: la giustapposizione di due interfaccie disgiunte di due link graph è* $X_0 \uplus X_1$ *e l'unità è* $\emptyset$*. Se* $F_i = (V_i, E_i, ctrl_i, link_i) : X_i \to Y_i$ *sono due link graph disgiunti (*$i = 0, 1$*), la loro giustapposizione* $F_0 \otimes F_1 : X_0 \uplus X_1 \to Y_0 \uplus Y_1$ *è data da*

$$F_0 \otimes F_1 = (V_0 \uplus V_1, E_0 \uplus E_1, ctrl_0 \uplus ctrl_1, link_0 \uplus link_1)$$

- Bigrafi*: la giustapposizione di due interfacce disgiunte* $I_i = \langle m_i, X_i \rangle (i = 0, 1)$ *è* $\langle m_0 + m_1, X_0 \uplus X_1 \rangle$ *e l'unità è* $\varepsilon = \langle 0, \emptyset \rangle$*. Se* $F_i : I_i \to J_i$ *sono bigrafi disgiunti (*$i = 0, 1$*), la loro giustapposizione* $F_0 \otimes F_1 : I_0 \otimes I_1 \to J_0 \uplus J_1$ *è data da:*

$$F_0 \otimes F_1 = \langle F_0^P \otimes F_1^P, F_0^L \otimes F_1^L \rangle$$

Tutte queste tre operazioni saranno utili per definire l'algebra dei bigrafi della prossima sezione.

2.4 L'algebra dei bigrafi

In questa sezione verrà illustrato come poter ottenere nuovi bigrafi da bigrafi base tramite le operazioni di composizione, identità e prodotto (giustapposizione). Si definisce quindi una vera e propria *algebra* per la teoria dei bigrafi, in cui si dimostra che ogni bigrafo può essere derivato da alcuni bigrafi base. Si incomincia nelle prime tre sezioni ad illustrare i bigrafi base su cui si appoggerà l'algebra, ovvero placing, linking e ioni.Infine, si enuncia un importante risultato che è la decomponibilità di ogni bigrafo in una sua forma normale.

Si rimanda all'appendice A per i dettagli sulla notazione usata.

2.4.1 Placing elementari

Definizione 2.4.1 (Placing, Permutazioni, Merge)**.** *Un bigrafo senza nodi e senza link viene detto* placing *(*$\varnothing$*). Un placing che è biiettivo dai siti alle radici è detto* permutazione *(*π*). Un placing sono una radice e n siti è denotata da* $merge_n$*.*

Un placing ha quindi solo siti e radici.

Il risultato importante è che si può ottenere qualsiasi *placing* dai tre *placing elementari* di figura 2.9, tramite le operazioni di composizione, identità e prodotto. In particolare:

(a) $\gamma_{1,1} : 2 \to 2$ (b) $1 : 0 \to 1$ (c) $join : 2 \to 1$

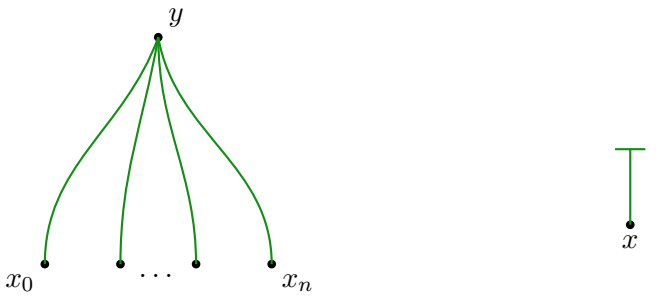
Figura 2.9: Placing elementari.

- ogni permutazione π può essere ottenuta dalla simmetria elementare $\gamma_{1,1}$. Per esempio, se si vuole ottenere la permutazione di ordine 3 (π_3), allora si può scrivere la seguente equazione: $\pi_3 = (\gamma_{1,1} \circ \gamma_{1,1}) \otimes id_1$.

- ogni placing $\varnothing$ (e quindi anche ogni merge) può essere ottenuto dai tre placing elementari $\gamma_{1,1}, 1$ e $join$. Per esempio: $merge_0 = 1$ e $merge_{n+1} = join \circ (id_1 \otimes merge_n)$.

2.4.2 Linking elementari

Definizione 2.4.2 (Linking, Sostituzioni, Chiusure)**.** *Un bigrafo senza nodi e senza places è detto* linking *(δ). Una* sostituzione *(σ) è il prodotto tra sostituzioni elementari (**??**). Una sostituzione biiettiva è detta* rinomina *(α). Una* chiusura *è il prodotto tra chiusure elementari(**??**).*



(a) ${}^{y}/X : X \to y$ (b) $/x : x \to \varepsilon$

Figura 2.10: Linking elementari.

Un linking ha solo inner names e outer names.

Il risultato importante è che ogni *linking* può essere generato tramite composizione, prodotto e identità a partire dai due *linking elementari* di figura 2.10: sostituzioni elementari ${}^{y}/X$ e chiusure elementari $/x : x \to \varepsilon$.

2.4.3 Ioni

Si descrive un altro tipo di bigrafo base: questa volta esso contiene nodi, a differenza dei placing e dei linking.

Definizione 2.4.3 (Ione)**.** *Per ogni controllo $K : n$, il bigrafo $K_{\vec{x}} : 1 \to \langle 1, \{\vec{x}\}\rangle$ avente un singolo nodo di tipo K le cui porte sono collegate biiettivamente con n distinti nomi $\vec{x}$, è chiamato un* ione discreto*.*

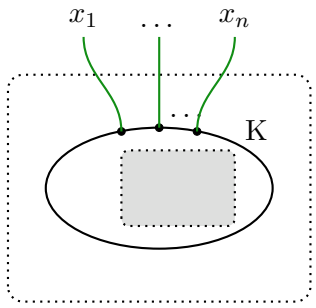


Figura 2.11: $K_{\vec{x}}$.

Riassumendo, i tre tipi di *bigrafi elementari* sono i placing, i linking e gli ioni. Tramite questi possiamo esprimere algebricamente ogni bigrafo in termini di composizione, prodotto e identità. Per esempio:

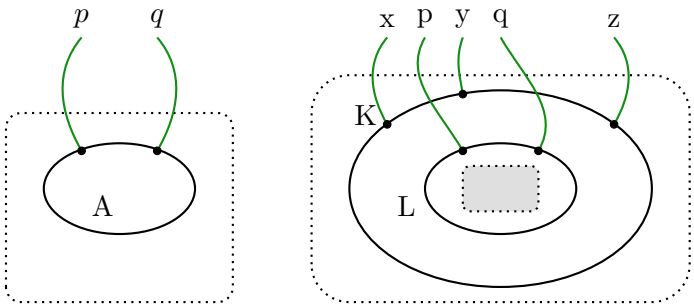
- per formare un atomo (vedi definizione 2.4.5) usiamo la composizione tra un ione ed un placing (2.12.a).
- per formare una molecola (vedi definizione 2.4.6) usiamo anche qui la composizione tra un ione e un bigrafo discreto, ottenuto a sua volta ricorsivamente usando solo placing, linking e ioni (2.12.b).

Definizione 2.4.4 (Bigrafo Discreto)**.** *Un bigrafo si dice discreto se non ha link chiusi e la sua link map è biiettiva.*

Definizione 2.4.5 (Atomo)**.** *Se il sito di un K-ione viene riempito dal placing $1 : 1 \to 0$ (vedi 2.9.b), il risultato è un* atomo discreto *$K_{\vec{x}} \circ 1$.*

Definizione 2.4.6 (Molecola)**.** *Se il sito di un K-ione viene riempito da un bigrafo discreto $G : I \to \langle 1, Y\rangle$, il risultato è una* molecola discreta *$(K_{\vec{x}} \otimes id_Y) \circ G$.*

Teorema 2.4.1. *Ogni bigrafo può essere costruito a partire da placing elementari, linking elementari e ioni tramite le tre operazioni di composizione, identità e giustapposizione.*



(a) $A_{pq} \circ 1$ (b) $(K_{xyz} \otimes id_{pq}) \circ L_{pq}$

Figura 2.12: Atomo e molecola discreti.

2.4.4 Forma normale discreta

Si è ora arrivati al risultato principale: la decomponibilità di ogni bigrafo in più bigrafi base. Per descrivere questo enunciato, sono necessarie le definizioni di *bigrafo discreto* (definizione 2.4.4) e di *bigrafo primo*.

Definizione 2.4.7 (Bigrafo primo)**.** *Un bigrafo si dice* primo *se non ha inner names ed ha un'interfaccia esterna unaria. Esso prende la segue forma:* $m \to \langle X \rangle$.

Un importante esempio di bigrafo primo è $merge_n : n \to 1$, dove $n \geq 0$. L'assenza di inner names nei bigrafi primi è fondamentale: essa assicura che ci sia una ed una sola decomposizione di ogni bigrafo in linking e primi discreti, come segue:

Proposizione 2.4.1 (Forma normale discreta)**.** *Ogni bigrafo* $G : \langle m, X \rangle \to \langle n, Z \rangle$ *può essere espresso univocamente, con al più una rinomina su* Y*, come:*

$$G = (id_n \otimes \lambda) \circ D$$

dove $\lambda : Y \to Z$ *è un linking e* $D : \langle m, X \rangle \to \langle n, Y \rangle$ *è un bigrafo discreto. Inoltre, ogni bigrafo discreto* D *può essere fattorizzato univocamente, con al più una permutazione dei siti di ogni fattore, come*

$$D = \alpha \otimes ((P_0 \otimes \cdots \otimes P_{n-1}) \circ \pi)$$

dove α *è una rinomina, ogni* P_i *è primo e discreto, e* π *è una permutazione di tutti i siti.*

Questa proposizione è cruciale per dimostrare la completezza della teoria algebrica dei bigrafi, che risulta quindi sia corretta che completa.

2.5 Bigraphical Reactive Systems

Uno degli aspetti fondamentali di un bigrafo è la sua capacità di evolversi sulla base di regole ben precise. Un BRS (Bigraphical Reactive System) consente di fare evolvere lo stato di un sistema (bigrafo iniziale) e di dedurre nuove informazioni con specifiche regole di reazione. Un BRS costituisce quindi la *dinamica* dei bigrafi.

La definizione di un BRS richiede prima la definizione di altri concetti, come quello delle *occorrenza* e di *regola di reazione parametrica*.

Definizione 2.5.1 (Occorrenza)**.** *Un bigrafo F occorre in un bigrafo G se l'equazione $G = C_1 \circ (F \otimes id_I) \circ C_0$ esiste per qualche interfaccia I e per qualche bigrafo C_0 e C_1.*

L'identità id_I è importante: consente a C_1 di avere figli in C_0 e in F, e consente a C_0 e C_1 di condividere link che non riguardano F.

Il secondo concetto importante è quello di *regola di reazione*.

Definizione 2.5.2 (Regola di reazione)**.** *Una regola di reazione è della forma*

$$R \to R'$$

dove R si dice essere il redex *della regola, cioè il pattern da trovare e cambiare nel bigrafo a cui si applica la regola, e R' è il* reactum*, cioè il pattern che andrà a sostituire il redex.*

Redex e reactum sono entrambi bigrafi, e possono occorrere (secondo la definizione 2.5.1) nel bigrafo a cui si applica la regola di reazione. Le regole di reazione possono coinvolgere sia il link graph sia il place graph, come si può vedere da questo esempio.

2.5.1 Esempio

Nella figura 2.13 , è rappresentato un bigrafo che indica lo stato di un ambiente. Esso è formato da due costruzioni B (buildings), da quattro stanze R (rooms), da cinque agenti A e da quattro computer C. Il bigrafo ha le seguente segnatura: $K = \{A : 2, B : 1, C : 2, R : 0\}$.

Il bigrafo rappresenta lo stato dell'ambiente, in cui i cinque agenti A stanno tenendo una video conferenza: gli agenti che stanno partecipando sono collegati dal link e_0. Ogni agente che vuole partecipare deve essere collegato ad un computer. A

sua volta i vari computer di una stessa costruzione sono collegati insieme, formando una LAN.

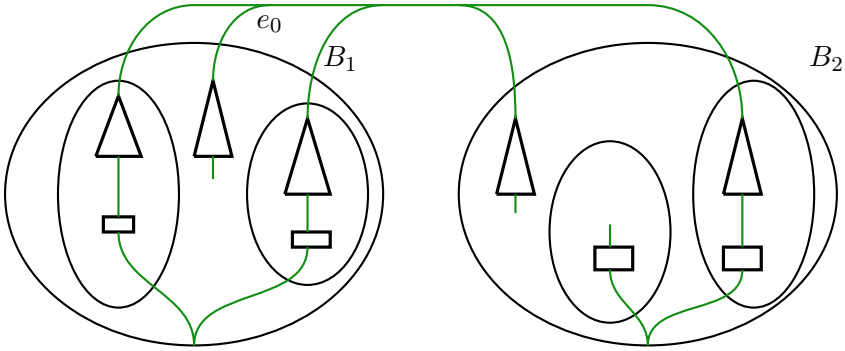


Figura 2.13: Esempio.

Alcune regole di reazione possono essere quelle delle figure 2.14, 2.15, 2.16. Esse agiscono sul bigrafo di partenza (figura 2.13) e sono così definite:

- La prima è la più semplice: un agente può lasciare la video chiamata. Il redex è la parte a sinistra della regola e può matchare qualsiasi agent. I due outername del redex indicano che le porte dell'agent devono essere collegate (prima dello scatto della regola) a zero o più porte. Se l'agent è collegato ad altri in una videochiamata, egli verrà disconnesso, mantenendo però attivo il collegamento al computer: questa informazione è espressa dal reactum, la parte destra della regola.

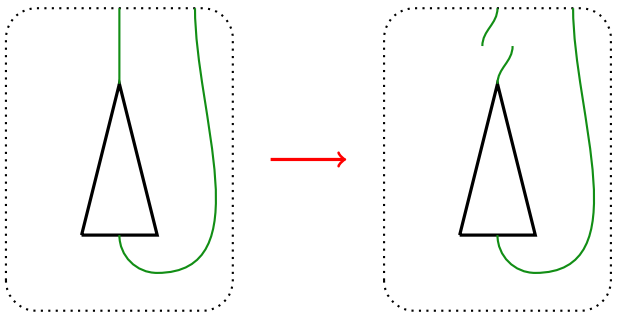


Figura 2.14: Prima regola di reazione

- La seconda regola prevede di matchare solo gli agent che non sono collegati a nessun computer e che si trova nella loro stessa stanza. Il reactum prevede che l'agent si ricolleghi a un tale computer. Se ci sono più computer nella stanza, la regola non si esprime su quale computer l'agent si debba ricollegare. Si noti

un importante dettaglio: agent e computer si trovano sotto la stessa radice, e quindi la regola scatta solo se essi si trovano dentro la stessa stanza, o dentro entrambi dentro lo stesso building.

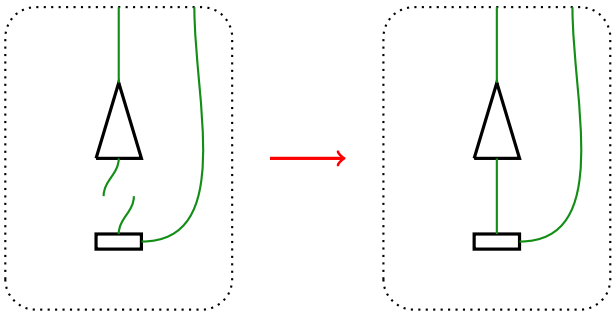


Figura 2.15: Seconda regola di reazione

- Mentre le prime due regole modificavano solamente il link graph, questa terza regola agisce anche sul place graph. Essa prevede lo spostamento di un agent in una stanza. Si noti ancora una volta che l'agent e la stanza si trovano sotto la stessa radice. Questo implica che se assumiamo che ogni stanza sia dentro una costruzione (building), allora questa regola scatta se e solo se l'agent si trova nella stesso building della stanza. In altre parole, non è possibile che un agent fuori da un building entri direttamente in una stanza. Infine, si noti il sito presente sia nel redex che nel reactum: esso rappresenta i parametri della regola. Dentro questo sito ci possono essere altri computer o altri agent. Se si togliesse tale sito dal redex, allora la regola scatterebbe solo se in tale stanza non ci fossero nè computer nè agenti, cioè solo se fosse vuota.

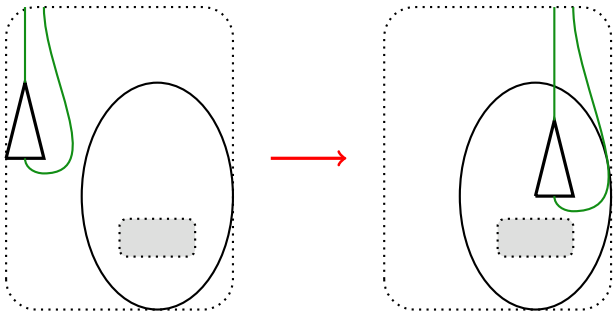


Figura 2.16: Terza regola di reazione

2.5.2 Regole di reazione parametriche

Definizione 2.5.3 (Matching)**.** *Il problema di trovare una o più occorrenze di un* redex R *all'interno di un bigrafo B si chiama* matching*. Nel caso in cui un* match *venga trovato, esso viene denotato tramite la seguente equazione:*

$$B = C \circ (R \otimes id_I) \circ D$$

dove C è detto il contesto *e D sono i* parametri *del match.*

I bigrafi e la loro teoria ereditano ed estendono molte caratteristiche del CCS (*Calculus of Communicating Systems*). Una di queste è la possibilità di isolare le zone che possono evolversi, cioè determinare quali nodi del bigrafo possono essere soggetti al processo di matching. Introduciamo quindi la nozione di *segnatura dinamica.*

Definizione 2.5.4 (Segnatura dinamica)**.** *Una segnatura è* dinamica *se assegna ad ogni controllo K uno stato nell'insieme $\{attivo, passivo\}$. Diciamo che un K-nodo è* attivo *se il suo controllo è assegnato allo stato attivo; lo stesso vale per lo stato passivo.*

Un bigrafo $G : \langle m, X\rangle \to \langle n, Y\rangle$ è attivo su $i \in m$ se ogni nodo antenato del sito i è attivo. Un bigrafo G si dice attivo, se è attivo su ogni sito.

Prima di scattare, una regola di reazione necessita del processo di matching, che trova l'occorrenza del redex della regola nel bigrafo. L'occorrenza sarà della forma $B = C \circ (R \otimes id_I) \circ D$. Se il contesto C è attivo, allora la regola scatta, sostituendo l'occorrenza del redex con il reactum. Altrimenti, la regola non scatta. In questo senso si riesce ad isolare parti del sistema che non si vogliono fare evolvere.

Si fornisce ora una definizione precisa di *regola di reazione.* Si tenga a mente l'esempio 2.5.1, in cui si presta particolare attenzione alla sostituzione del redex con il reactum.

Definizione 2.5.5 (Regola di reazione)**.** *Una regola di reazione $R \to R'$, dove il redex R e il reactum R' hanno la stessa interfaccia, è una trasformazione che, se applicata al bigrafo B, produce un nuovo bigrafo B' secondo questa relazione:*

$$B : C \circ (R \otimes id_I) \circ D \qquad \to \qquad B' : C \circ (R' \otimes id_I) \circ D$$

dove il contesto C è attivo.

Si noti che questa definizione richiede che R e R' abbiano la stessa interfaccia. Questo è in genere un vincolo troppo forte, che obbliga il progettista a scrivere regole di reazione limitative. Per esempio, non potrei copiare il contenuto, perché i siti del reactum sarebbero uno in più di quelli del redex, rendendo la loro interfaccia interna diversa. Per indebolire questo vincolo, introduciamo il concetto di *mappa di istanziazione* che porterà al concetto di *regole di reazione parametriche.*

Definizione 2.5.6 (Mappa di istanziazione)**.** *Siano $\langle m, X\rangle$ e $\langle m', X\rangle$ le interfacce interne rispettivamente di R e R'. Una* mappa di istanzazione *è una funzione $\eta : m \to m'$ che mappa siti di R' in siti di R.*

Definizione 2.5.7 (Regola di riscrittura parametrica)**.** *Una regola di reazione parametrica è una tripla della forma:*

$$(R : m \to I, R' : m' \to I, \eta)$$

dove R e R' sono rispettivamente il redex e il reactum della regola. Questa volta devono concordare solo sull'interfaccia esterna e sugli inner names.

La regola di reazione parametrica, se applicata al bigrafo B, produce la reazione:

$$B = C \circ (R \otimes id_I) \circ D \qquad \to \qquad B' = C \circ (R' \otimes id_I) \circ \overline{\eta}(D)$$

in cui la funzione $\overline{\eta}$ è definita come segue:

sia $g : \langle m, X\rangle$ un bigrafo la cui FND è $g = \lambda \circ (d_0 \otimes \cdots \otimes d_{m-1})$, allora

$$\overline{\eta}(g) = \lambda \circ (d'_0 || \ldots || d'_{m-1})$$

dove $d'_i \eqsim d_{\eta(i)}$

2.5.3 BRS

Si sono date tutte le nozioni necessarie per definire formalmente la dinamica di un bigrafo. Essa è costituita da un BRS (*bigraphical reactive system*), che fa evolvere il bigrafo di partenza sulla base di regole ben precise, contenute al suo interno. Ogni BRS è costituito da una segnatura e da un insieme di regole: risulta quindi essere un sistema deduttivo, perché composto rispettivamente da sintassi e semantica.

Definizione 2.5.8 (BRS)**.** *Un sistema reattivo bigrafico (BRS) è definito attraverso una coppia (K, R), dove K è una segnatura e R è un insieme di regole di reazione, e si indica con $BG(K, R)$.*
*L'insieme delle regole R è chiuso rispetto all'*equivalenza sul supporto*: se $R \eqsim S$ e $R' \eqsim S'$ e $(R, R', \eta) \in R$ per un certo η, allora $(S, S', \eta) \in R$.*

Si indichi ora con $B \to *B'$ il fatto che il bigrafo B' sia raggiungibile da B con zero o più regole di reazione. Si dimostra che, dato un bigrafo B la cui segnatura è K, il BRS denotato da $BG(K, R)$ computa ogni bigrafo B' tale che $B \to *B'$. Se si considerano quindi tutti i possibili B', si ha l'insieme di tutti i possibili stati in cui il sistema B può evolversi secondo le regole R.

2.5.4 Esempio

Vediamo ora un ultimo esempio, in cui si useranno le regole di reazione parametriche. Si vuole rappresentare l'operazione di moltiplicazione tramite i bigrafi. Incominciamo con la segnatura: ci sono tre controlli

- Mul, che rappresenta l'operazione di moltiplicazione. Ha arietà 0. Attivo.
- Num, che rappresenta il numero, anch'esso 0 porte. Passivo.
- One, rappresenta l'unità, con arietà 0. Passivo.

L'idea principale è che il nodo *mul* contiene due numeri, cioè due nodi *num*, che saranno quelli che dovranno essere moltiplicati. Si può anche optare per una versione ricorsiva, permettendo al nodo *mul* di contenere anche altri nodi *mul*. Infine, ogni numero è rappresentato come unione di varie unità. Per esempio, il numero 4 verrà rappresentano tramite il nodo *num* contenente quattro nodi *one*.

Se vogliamo operare la moltiplicazione fra 4 e 2, allora il bigrafo equivalente sarà quello di figura 2.17.

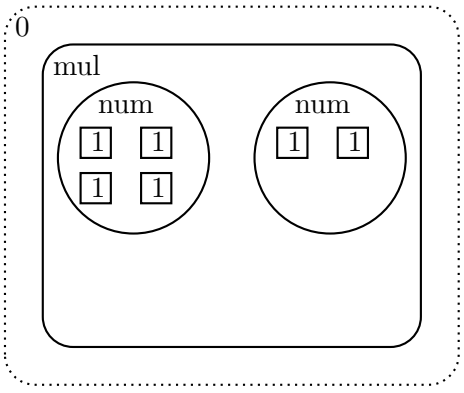


Figura 2.17: Moltiplicazione tramite bigrafi

Per eseguire la moltiplicazione si usano solamente due regole di reazione, ma in maniera *ricorsiva*. Esse ricalcano nel formalismo dei bigrafi questa semplice ope-

razione: $a * b = (a - 1) * b + b$. Distinguiamo quindi fra caso ricorsivo e caso base.

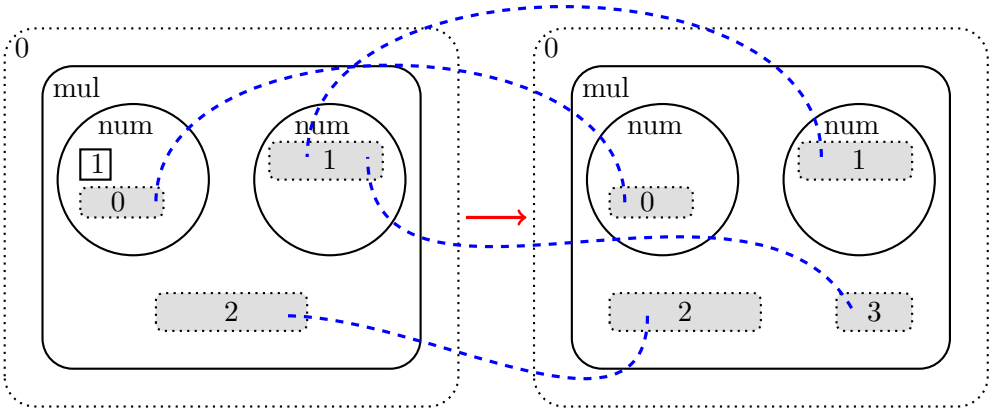


Figura 2.18: Caso ricorsivo

La regola del caso ricorsivo mette bene in evidenza l'importanza delle regole parametriche. Un sito del redex (1) viene copiato in una regione diversa, cioè dentro *mul*. Questo ci consente di *copiare* tutto il contenuto del secondo nodo *num* dentro il nodo *mul*. Infatti, il contenuto di *mul* al di fuori dei nodi *num* rappresenta la somma $(+b)$ dell'uguaglianza $a * b = (a - 1) * b + b$.

Si noti che questa regola va applicata ricorsivamente fino a quando il primo nodo *num* ha contenuto vuoto, cioè fino a quando il caso base è applicabile.

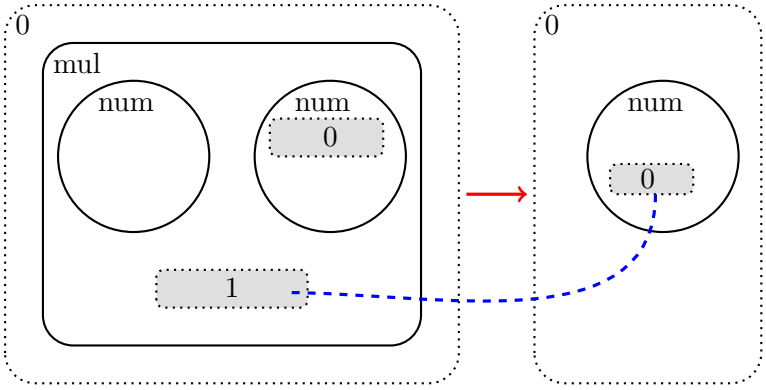


Figura 2.19: Caso Base

Il caso base copia il contenuto di *mul* dentro un nuovo nodo *num*, che rappresenta il risultato della moltiplicazione. Nell'aritmetica classica, per risolvere ricorsivamente la moltiplicazione $2 * 4$ si operano questi passi:

$$result = 4 * 2 = (4-1) * 2 + 2 = 3 * 2 + 2 \qquad \text{(caso ricorsivo)}$$
$$3 * 2 = (3-1) * 2 + 2 = 2 * 2 + 2 \qquad \text{(caso ricorsivo)}$$
$$2 * 2 = (2-1) * 2 + 2 = 1 * 2 + 2 \qquad \text{(caso ricorsivo)}$$
$$1 * 2 = (1-1) * 2 + 2 = 0 * 2 + 2 = 2 \qquad \text{(caso base)}$$

Seguendo la catena di uguaglianze, si trova che il risultato finale della moltiplicazione è:

$$result = 4 * 2 = (((0 * 2 + 2) + 2) + 2) + 2 = 2 + 2 + 2 + 2 = 8.$$

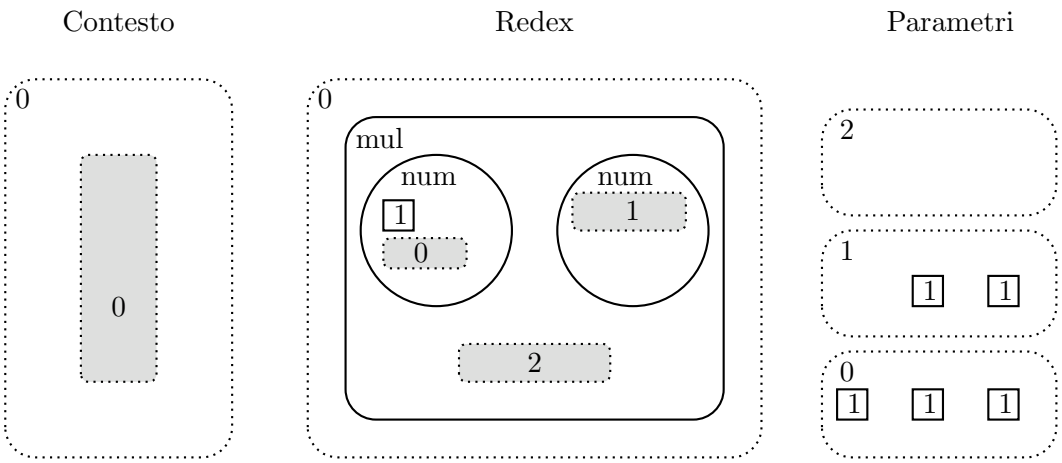


Figura 2.20: Match decomposto

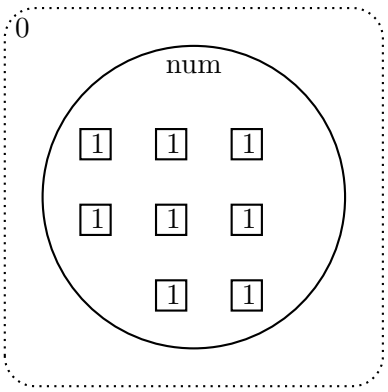


Figura 2.21: Bigrafo finale

Vediamo ora l'applicazione della prima regola ricorsiva sul bigrafo di partenza (figura 2.17). La prima operazione da eseguire è trovare l'occorrenza del redex nel bigrafo, che formalmente significa rispettare l'equazione $B = C \circ (R \otimes id_I) \circ D$, dove C è il contesto, R è il redex e D sono i parametri. Dato che tutti e tre sono bigrafi, possiamo disegnarli, come è stato fatto in figura 2.20.

Il bigrafo finale, dopo 3 applicazioni della regola ricorsiva ed una della regola base, sarà quello di figura 2.21.

L'implementazione la si può trovare in [1]. Questo esempio mostra bene come le regole di reazione che formano il BRS rappresentano la semantica del sistema. Per evitare di sprecare memoria sul calcolatore, è necessario progettare con cura la segnature e l'insieme di regole del BRS. Nell'esempio di cui sopra, le regole erano soltanto due e potevano essere applicate ricorsivamente. Questo è preferibile ad avere k regole e applicarle una dopo l'altra, appunto a causa dello spreco di memoria che si otterrebbe.

3

Isomorfismo tra bigrafi

Si è visto nel precedente capitolo come un bigrafo sia capace di evolversi all'interno di un BRS. In alcune situazioni, si vuole evitare evoluzioni infinite di un bigrafo, perché per esempio porterebbero sempre a stati uguali fra di loro.

Per tenere traccia all'istante t_k di tutti gli stati precedentemente assunti da un BRS $(t_0 \dots t_{k-1})$, si è costruita una struttura dati a grafo, dove ogni nodo è uno stato, cioè un bigrafo: si chiamerà questo grafo "*grafo degli stati*". Se un BRS parte dallo stato S_0, è possibile che dopo K regole di reazione lo stato S_k sia uguale allo stato iniziale S_0. Questo significa che S_0 e S_k hanno la stessa *semantica*, cioè rappresentano lo stesso stato del sistema. Per cui, nel grafo degli stati, essi dovranno essere lo stesso medesimo nodo.

Questo è equivalente a scrivere: "se lo stato S_k è stato ottenuto da S_{k-1} tramite la regola R e S_k e S_0 sono isomorfi, allora da S_{k-1} si ottiene S_0"

$$\text{se } S_{k-1} \xrightarrow{R} S_k \text{ e } S_k \simeq S_0 \qquad \text{allora } S_{k-1} \xrightarrow{R} S_0$$

Capire quando due bigrafi sono isomorfi e quindi *semanticamente equivalenti* è di fondamentale importanza se si vogliono evitare i così detti *loop* fra regole: per evitare che fra S_i e S_{i+1} si continuino ad applicare sempre le due stesse regole R_1 e R_2 all'infinito, devo capire che:

- da S_i, tramite la regola R_1, ottengo un bigrafo isomorfo a S_{i+1}
- da S_{i+1}, tramite la regola R_2, ottengo un bigrafo isomorfo a S_i

Questo concetto verrà chiarito con gli esempi sottostanti.

3.1 Esempio

Si prenda l'esempio di una rete modellata tramite un bigrafo: si vuole fare in modo che, dato un pacchetto iniziale che ha come mittente l'host A, esso arrivi al destina-

tario B. Ci saranno quindi delle regole di reazione per i router, che permetteranno di inoltrare i pacchetti verso le sue interfacce di uscita. Poichè, utilizzando i soli bigrafi, servirebbero troppe regole di reazione per modellare il fatto che se il destinario è X allora l'interfaccia di uscita del pacchetto è la numero N, si può pensare di inoltrare il pacchetto verso tutte le uscite in modo non deterministico. Così facendo, il pacchetto arriverà sicuramente al destinatario. I problemi sono ora due:

- il pacchetto arriverà a destinatari non corretti. Si può introdurre una regola di reazione che elimini dall'host ogni pacchetto che non ha come destinatario l'host stesso.

- se prendiamo il k-esimo router R_k, allora il pacchetto ritornerà al $(k-1)$-esimo router R_{k-1} , che a sua volta lo inoltrerà verso tutte le sue interfacce, e quindi anche nuovamente verso R_k. Si ha così un ciclo infinito di pacchetti tra R_k e R_{k-1}.

In questa sede si tratterà il secondo di questi problemi, che ha un'apparentemente semplice soluzione: capire quando due bigrafi sono uguali. In questo esempio, è facile capire che per risolvere il problema basta verificare se gli stati S_{k+i} e S_{k-1} sono uguali: se S_k è il bigrafo in cui il pacchetto è nel router R_k, allora si applica la regola di inoltro e si generano tanti stati $(S_{k+1} \dots S_{k+j})$ quanti sono i router vicini a R_k; tra di questi ci sara anche R_{k-1}. Quindi, se supponiamo che nello stato S_{k+i} il pacchetto torni indietro al router R_{k-1}, allora il problema è capire che S_{k+i} è uguale S_{k-1}, cioè verificare che i due bigrafi siano isomorfi. In questo modo si sa che dallo stato S_{k-1} non si dovrà più applicare la regola di inoltro verso S_k, perché questo causerebbe un ciclo infinito.

3.2 Formulazione del problema

Si è visto che il concetto di isomorfismo ci aiuta a capire quando due bigrafi sono uguali. Riportiamo di nuovo la sua definizione formale:

Definizione 3.2.1 (Isomorfismo)**.** *Due bigrafi F e G si dicono isomorfi se e solo se esiste una traduzione di supporto tra F e G, cioè se e solo se F e G sono* support equivalent *($F \simeq G$).*

La definizione di isomorfismo può far credere che esso sia un termine puramente sintattico, cioè che riguardi solamente la struttura dei due bigrafi. L'isomorfismo è

invece un'operazione che riguarda tanto la sintassi quanto la semantica: due bigrafi strutturalmente uguali sono due bigrafi semanticamente equivalenti (figura 3.3).

Il problema consiste quindi nel trovare un funzione biettiva che ha come dominio i nodi e gli archi del primo bigrafo, come codominio quelli del secondo bigrafo e che ne rispetti la struttura del primo. Come già visto, tale funzione si chiama *traduzione di supporto*.

Si noti come la traduzione di supporto consenta di ritenere isomorfi due bigrafi che sono uguali *modulo permutazione*: tale funzione può infatti operare una permutazione sui nomi dei nodi e degli archi, così come sugli inner names e outer names. In altre parole, non c'è nessun vincolo sui nomi, ma solo sulla struttura del bigrafo. Si capirà meglio questo concetto negli esempi che seguiranno.

3.2.1 Complessità

Stabilire la complessità del problema dell'isomorfismo tra bigrafi è molto importante, perché relazionandolo con altri problemi è possibile trovare un algoritmo efficiente per risolverlo. Facciamo un esempio: ipotizziamo che il problema dell'isomorfismo tra bigrafi (che chiameremo P_{big}) si trovi nella classe C, e che il problema P_c sia *completo* rispetto a questa classe. Per definizione di *completezza*, ogni istanza di ogni problema appartenente alla classe C, quindi anche ogni istanza di P_{big}, può essere tradotta (cioè *ridotta*) con una certa complessità al problema P_c.

L'importanza di questa considerazione risiede in questo fatto: se si riesce a scoprire un algoritmo che operi in tempo polinomiale per risolvere il problema P_c allora tutti i problemi della classe C, incluso P_{big}, sarebbero risolvibili in tempo polinomiale.

Sia P_{graph} il problema dell'isomorfismo tra grafi. Dato che P_{graph} è in NP ma non è stato ancora dimostrato che sia completo, introduciamo la classe di complessità GI, definita come l'insieme dei problemi che possono essere ridotti in tempo polinomiale al problema P_{graph}. Si dimostrerà il seguente risultato:

Teorema 3.2.1 (Complessità di P_{big})**.** *Il problema dell'isomorfismo tra bigrafi è GI-completo.*

Dimostrazione. Per dimostrare l'equivalenza, procederemo in due direzioni: nella prima dimostreremo che esiste una riduzione in tempo polinomiale da P_{graph} a P_{big}, cioè che ogni istanza del problema P_{graph} può venire tradotta in un'istanza del problema P_{big}. Invece, nella seconda direzione, faremo l'inverso. Si dimostreranno

quindi che esistono le seguenti riduzioni:

$$\begin{cases} P_{graph} \leq P_{big} \\ P_{big} \leq P_{graph} \end{cases}$$

- $P_{graph} \leq P_{big}$:
  Ogni grafo G può essere trasformato in un bigrafo B tramite la seguente trasformazione. Innanzitutto, definiamo la segnatura di B: servono solamente due controlli, $Node$ con 0 porte e $Port$ con una sola ($\{Node : 0, Port : 1\}$). Traduciamo ogni nodo n_G di G in un nodo n_B di tipo $Node$ di B; in seguito, per ogni arco uscente o entrante di n_G inseriamo in n_B un nodo di controllo $Port$ e colleghiamo la sua porta all'altro nodo n_B di B. Questa trasformazione consente quindi di trasformare ogni grafo in un bigrafo: dato che è possibile implementarla come una visita sul grafo, questa riduzione è in tempo polinomiale (*Karp-riduzione* da P_π a P_{big}). Un esempio è rappresentato in figura 3.1.

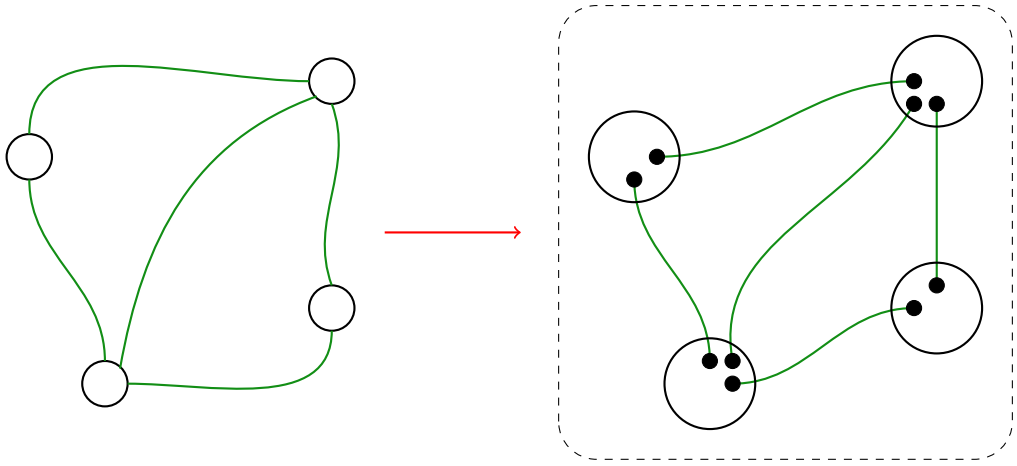


Figura 3.1: Riduzione da P_{graph} a P_{big}

- $P_{big} \leq P_{graph}$:
  Il primo approccio potrebbe essere quello di trattare i casi del place graph e del link graph separatamente, ma questa sarebbe una soluzione sbagliata: come si vedrà dagli esempi seguenti, due bigrafi che hanno sia place graph sia link graph isomorfi potrebbero non essere isomorfi. Per questo motivo, si deve trasformare ogni bigrafo in un grafo tramite questa trasformazione, di cui si è riportato un esempio in figura 3.2. Ogni nodo, radice, sito, porta, innername,

arco e outername viene trasformato in un nodo del grafo: in altre parole ogni entità del bigrafo (compresi gli archi) viene trasformata in un nodo.

Ora, l'idea è quella di sfruttare le funzione *prnt* e *link* che compongono rispettivamente ogni place graph ed ogni link graph. Nel primo caso, se il nodo n_1 si trova all'interno di n_2 allora ci sarà un link tra questi due nodi. Nel secondo caso, se le porte $p1$ e p_2 sono collegate tramite l'arco e_0, allora nel grafo il nodo e_0 sarà collegato sia al nodo p_1 sia al nodo p_2. Come si vede bene dalla figura 3.2, si costruisce un grafo che tenga conto sia della struttura del place graph (archi rossi) sia di quella del link graph (archi verdi).

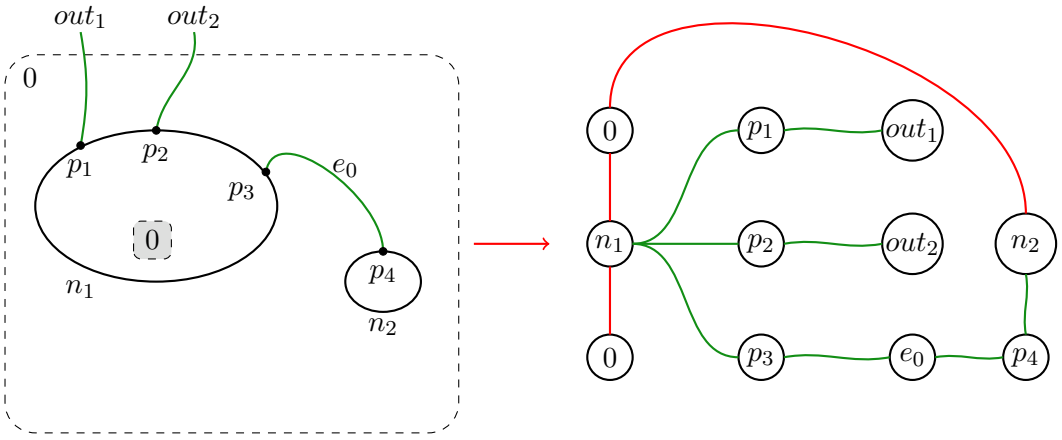


Figura 3.2: Riduzione da P_{big} a P_{graph}

Si è quindi dimostrata l'equivalenza fra il problema dell'isomorfismo tra bigrafi e quella tra grafi, concludendo quindi che il problema P_{big} è equivalente al problema P_{graph}.

Corollario 3.2.1. Il problema dell'isomorfismo tra bigrafi è **equivalente** al problema dell'isomorfismo tra grafi.

Quindi si ha che $P_{big} \in GI$. Come già accennato, l'importanza di trovare la classe di complessità a cui appartiene un dato problema è enorme: se si trovasse un algoritmo che risolva in tempo polinomiale un problema NP-completo (cioè se la famosa domanda *P is NP?* avesse risposta affermativa), allora tutti i problemi appartenenti alla classe NP, incluso P_{big}, potrebbero essere ridotti in tempo polinomiale al problema NP-completo di cui si ha l'algoritmo polinomiale. In questo modo, anche l'isomorfismo tra bigrafi sarebbe risolvibile in tempo polinomiale.

3.3 Strategia di soluzione

Dal punto di vista teorico, si è trattato questo problema come una *rete di flusso*: esse consentono di specificare in modo molto preciso delle condizioni sulla *struttura* dei grafi. Qui l'obbiettivo è trovare una traduzione di supporto, che conservi la struttura e sia biettiva: quest'ultima caratteristica per esempio può essere espressa come il fatto che ogni nodo del primo bigrafo deve essere associato ad uno ed un solo nodo del secondo bigrafo. Tramite la rete di flusso, si può specificare questa condizione dicendo che il flusso in uscita da ogni nodo deve essere esattamente 1. Questo è un primo esempio del motivo per cui si è scelta una rete di flusso per la modellizzazione.

Si è scelto di trattare questo problema tramite la programmazione a vincoli. Questo paradigma permette di rappresentare al calcolatore un sistema di equazioni, che esso risolverà. Le equazioni saranno i vincoli che la rete di flusso dovrà rispettare. Nel nostro caso, ci basterà capire se esiste una soluzione al sistema, e non ci interessa sapere quali sono le sue soluzioni. Tutte le equazioni scritte sono lineari, con la conseguenza che si avrà un sistema lineare: questo permette di abbassare la complessità dell'algoritmo.

Si è scelta il paradigma del *Constraint Programming* per i seguenti motivi:

- la programmazione a vincoli consente di ridurre notevolmente il tempo di sviluppo dell'algoritmo e il numero di errori, specie per problemi NP, come l'isomorfismo tra bigrafi.

- il motore che risolve il sistema di equazioni è di solito soggetto a molti test, ed è quindi affidabile. C'è anche la possibilità di imporre delle euristiche che migliorano notevolmente le prestazioni, rendendolo quindi anche efficiente.

- il motore può venire considerato come una *black box*, astraendosi quindi alla sua implementazioni interna, che può per esempio venire cambiata o migliorata nel tempo, evitando di affliggere il software che lo usa.

E' possibile che esista un algoritmo che risolva in modo più efficiente questo problema: se per esempio sarà inventato l'algoritmo che risolve in tempo polinomiale un isomorfismo tra grafi, allora per quanto dimostrato nella sottosezione 3.2.1 anche questo problema potrà essere risolto in tempo polinomiale. Fino ad allora, la programmazione a vincoli, che può non risultare il metodo migliore per risolvere questo

problema, crea un ottimo compromesso tra complessità dell'algoritmo e tempo di sviluppo.

Si è precedentemente visto che ogni bigrafo è formato da due strutture ortogonali e totalmente indipendenti, il place graph ed il link graph. Il problema dell'isomorfismo verrà quindi trattato separatamente per le due strutture: ci saranno dei vincoli solamente per il place graph, ed altri solamente per il link graph. Infine, gli ultimi vincoli serviranno per conciliare le due soluzioni: si vedrà che senza di questi il calcolatore riconoscerà come uguali due bigrafi che hanno lo stesso link graph e lo stesso place graph modulo permutazione ma che non sono isomorfi (figura 3.4).

3.3.1 Esempi

Non è sempre banale capire quando due bigrafi sono isomorfi. Si forniscono quindi alcuni esempi per acquisire familiarità con questo concetto.

Il primo riprende il bigrafo di figura 2.17 per la moltiplicazione tra numeri naturali. Si vuole mostrare che, semanticamente, l'operazione $2 * 4$ è equivalente all'operazione $4 * 2$. In altre parole, non importa l'ordine dei nodi.

Incominciamo notando che i due link graph sono isomorfi, perché non hanno archi e presentano lo stesso numero di nodi. Prendiamo ora in esame il place graph: se si ritorna alla sua definizione, si nota subito come i due place graph siano uguali perché i nodi interni non sono ordinati.

Il fatto di dimostrare che i due bigrafi sono isomorfi e semanticamente equivalenti, dimostra che la proprietà commutativa della moltiplicazione vale anche nella sua versione bigrafica.

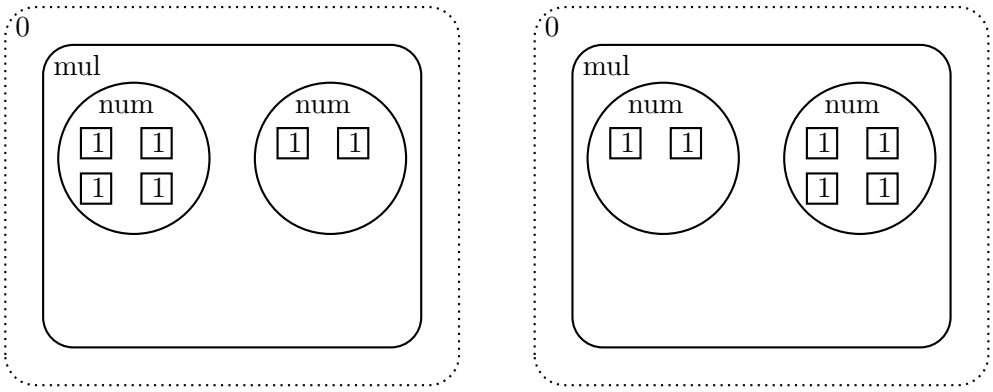


Figura 3.3: Bigrafi Isomorfi: $(4 * 2) = (2 * 4)$

Presentiamo un altro esempio, questa volta su un caso negativo. Si è specificato che per risolvere il problema si trattano separatamente i casi del place graph e del link graph: questo approccio però necessita di avere dei vincoli di "coerenza" che uniscano le due soluzioni. Senza questi vincoli, si può incorrere nel seguente problema.

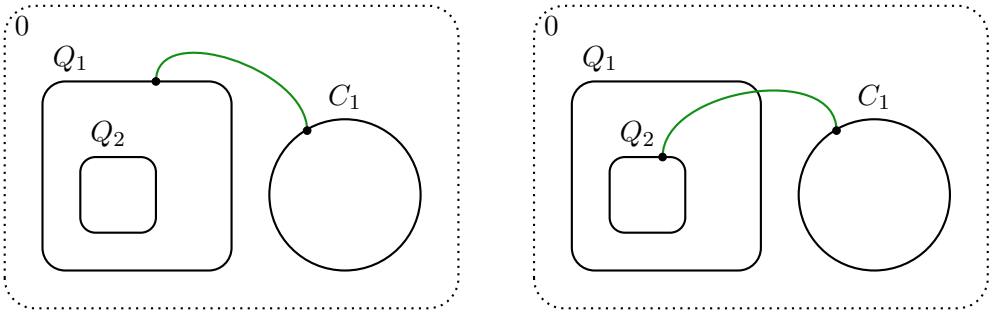


Figura 3.4: Bigrafi non Isomorfi

(a) Place Graph isomorfi

(b) Link Graph isomorfi

Figura 3.5: Place Graph e Link Graph del bigrafo 3.4

Si considerino i due bigrafi di figura 3.4 e i loro relativi place graph e link graph (figura 3.5). Si nota subito che i due bigrafi *non* sono isomorfi. Prendendo però i due place graph ci si accerta che essi lo sono. Lo stesso vale per i due link graph: sono entrambi formati da tre nodi, in cui c'è solamente un edge che collega il cerchio ad un quadrato.

I casi separati del link graph e del place graph sembrano quindi suggerire che i due bigrafi siano isomorfi, mentre si vede subito che non è così. Quello che non

abbiamo considerato sono appunto i *vincoli di coerenza*, che informalmente dicono che il quadrato collegato al cerchio (nel primo link graph) deve essere quello che contiene un altro quadrato (nel primo place graph). Aggiunta questa condizione, si nota che le due soluzioni sono *incompatibili*, avendo quindi che i due bigrafi *non* sono isomorfi.

3.4 Vincoli

In questa sezione si presentano le equazioni necessarie per risolvere il problema dell'isomorfismo. Come già anticipato, si lavorerà sulle *reti di flusso*, che sono grafi orientati pesati. Distinguiamo le equazioni per il place graph, per il link graph e per la così detta coerenza.

3.4.1 Vincoli per il place graph

L'isomorfismo tra place graphs è un isomorfismo tra foreste. Si vuole infatti vedere quando la prima foresta è isomorfa alla seconda, a meno di permutazioni delle radici e dei siti. Una rete di flusso per questo problema è quella in figura 3.6.

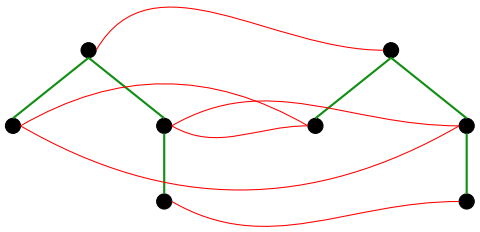


Figura 3.6: Rete di flusso per l'isomorfismo tra place graphs.

Alle due foreste (cioè ai due place graph, costituiti da archi di colore verde), che chiameremo P_F e P_G, si sono aggiunti altri archi (quelli rossi) *solamente* tra nodi della stessa altezza, creando così un grafo. Questa è la vera e propria rete di flusso per il problema dell'isomorfismo tra place graph. Ogni arco rosso è *orientato*, perché va dai nodi della prima foresta ai nodi della seconda, e *pesato*, perché gli è assegnato un numero naturale $p \in \{0, 1\}$.

Nell'implementazione, trattata tramite la programmazione a vincoli, ogni arco rosso è una variabile. L'insieme delle variabili per il place graph sarà quindi:

$$M_{d,m,n} \in \{0, 1\} \qquad 0 \leq d \leq depth$$

$$m \in P_F^d$$
$$n \in P_G^d$$

dove *depth* è l'altezza massima della prima foresta, e P_F^d e P_G^d indicano l'insieme di nodi/radici/siti che si trovano all'altezza d rispettivamente nel place graph P_F e P_G. In altre parole, creo una variabile che può assumere valore 0 o 1 per ogni coppia di nodi (a, b) che si trovano sulla stessa altezza, dove a appartiene al primo place graph mentre b al secondo.

I vincoli dovranno essere tali che, dopo l'esecuzione del sistema sul calcolatore, le variabili che assumeranno il valore 1 saranno quelle che *formeranno* la vera e propria traduzione di supporto. Ovvero:

Proposizione 3.4.1. *La variabile $M_{d,m,n}$ assumerà il valore 1 se e solo se esiste una traduzione di supporto ρ tale che $\rho(m) = n$.*

Tutte le altre variabili dovranno assumere il valore 0. Un altro modo di vedere la soluzione è questa: la funzione di traduzione di supporto sarà definita da tutte e sole le variabili con valore 1. Infine:

Terminologia. *Quando la variabile $M_{d,m,n} = 1$ diremo che m e n costituiscono un match.*

Nell'esempio di figura 3.6, la soluzione esiste ed è definita come in figura 3.7, dove i numeri sopra le variabili (archi rossi) indicano i valori che esse hanno assunto dopo la risoluzione del sistema di equazioni.

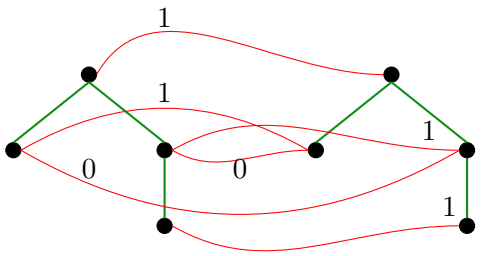


Figura 3.7: Soluzione della rete di flusso

Riassumendo, dobbiamo tradurre in vincoli il fatto che la funzione di traduzione di supporto sia biiettiva e che conservi la struttura del place graph P_F. Distingueremo quindi in vincoli di flusso, che serviranno per il primo problema, e in vincoli strutturali, per il secondo.

Incominciamo con i **vincoli strutturali**: si è adottata una versione ricorsiva per i vincoli. Distinguiamo quindi in caso base e passo ricorsivo.

- *Caso Base*: due nodi della stessa altezza che hanno un numero diverso di figli *non* possono costituire un match. In formule:

$$M_{d,m,n} = 0 \qquad \text{se } |prnt_F^{-1}(m)| \neq |prnt_G^{-1}(n)| \\ \forall d \leq depth - 1 \\ \forall m \in P_F^d \text{ e } n \in P_G^d$$

  Nella figura sottostante, si vede subito che la variabile $M_{0,r_0,r_0} = 0$ perché $prnt_F^{-1}(r_0) = 1 \neq 2 = prnt_G^{-1}(r_0)$.

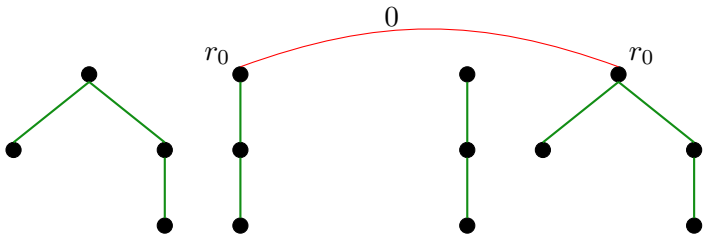


Figura 3.8: Esempio per il primo vincolo

  Si noti che questo vincolo viene applicato a tutti i nodi tranne ai siti, come specificato dalla condizione $\forall d \leq depth - 1$.

- *Caso Ricorsivo*: se due nodi alla stessa altezza *non* costituiscono un match, allora neanche i loro figli lo fanno. In formule:

$$M_{d,m,n} \leq M_{d-1,prnt(m),prnt(n)} \qquad \forall d \geq 1 \\ \forall m \in P_F^d \\ \forall n \in P_G^d$$

  Questo vincolo modella la seguente implicazione:

$$M_{d-1,prnt(m),prnt(n)} = 0 \qquad \Rightarrow \qquad M_{d,m,n} = 0$$

  Esso costituisce il caso ricorsivo perché vale per tutti i nodi tranne le radici, e le radici sono ricoperte dal caso base. Si veda la figura 3.9: la variabile $M_{1,n_1,n_2} = 0$ perché dalla figura 3.8 sappiamo che $M_{0,r_0,r_0} = 0$ e perché $prnt_F(n_1) = r_0$ e $prnt_G(n_2) = r_0$. Lo stesso ragionamento vale per la variabile M_{1,n_1,n_3}.

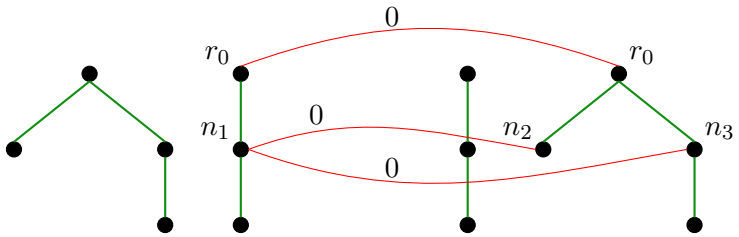


Figura 3.9: Esempio per il secondo vincolo

Questi vincoli non sono però sufficienti per determinare un isomorfismo tra i due place graph P_F e P_G. In particolare, la traduzione di supporto deve essere biiettiva, cioè associare un nodo/radice/sito di P_F ad uno ed un solo nodo/radice/sito di P_G. Si sono quindi aggiunti i **vincoli di flusso**.

- *Flusso in uscita*: il flusso totale in uscita da ogni nodo deve essere esattamente pari a 1.

  Notazione. *Questo vincolo lo indicheremo con la notazione:* $\delta^+(m) = 1$, *dove* $m \in P_F^d$.

  Esso si traduce nel fatto che la somma di tutte le variabili in uscita da ogni nodo/radice/sito di P_F deve essere 1, ovvero: ogni nodo di F può costituire un match solamente con uno ed un solo altro nodo di G. In formule:

$$\sum_n M_{d,m,n} = 1 \qquad \begin{array}{l} 0 \leq d \leq depth \\ m \in P_F^d \\ n \in P_G^d \end{array}$$

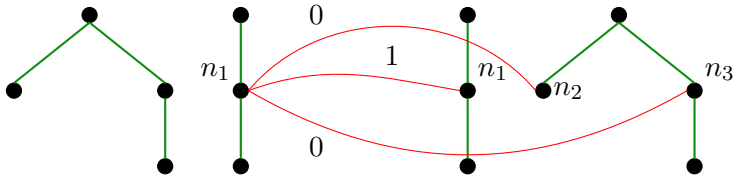


Figura 3.10: Esempio per il vincolo sul flusso in uscita

  In figura 3.10 si può vedere come la somma degli archi in uscita dal nodo n_1 sia 1, ovvero: $M_{1,n_1,n_1} + M_{1,n_1,n_2} + M_{1,n_1,n_3} = 1$.

- *Flusso in entrata*: il flusso totale in entrata da ogni nodo deve essere esattamente pari a 1.

Notazione. *Questo vincolo lo indicheremo con la notazione:* $\delta^-(n) = 1$*, dove* $n \in P_G^d$.

Esso è equivalente a dire che la somma di tutte le variabili in entrata da ogni nodo/radice/sito di P_G deve essere 1. In altre parole: ogni nodo di G può costituire un match solamente con uno ed un solo altro nodo di F. In formule:

$$\sum_m M_{d,m,n} = 1 \qquad \begin{array}{l} 0 \leq d \leq depth \\ m \in P_F^d \\ n \in P_G^d \end{array}$$

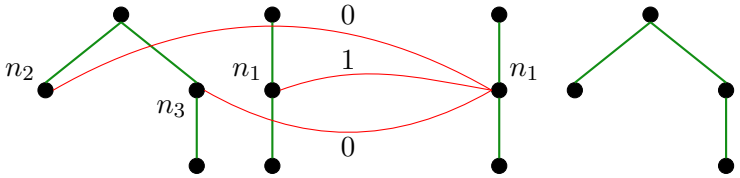


Figura 3.11: Esempio per il vincolo sul flusso in entrata

In figura 3.11, è stato espresso il vincolo che il flusso in entrata verso il nodo n_1 deve essere 1, cioè: $M_{1,n_1,n_1} + M_{1,n_2,n_1} + M_{1,n_3,n_1} = 1$.

Si capisce bene come questi due ultimi vincoli assicurino che la traduzione di supporto sia rispettivamente iniettiva e suriettiva, rendendola quindi *biiettiva* come si voleva.

3.4.2 Vincoli per il link graph

L'isomorfismo tra link graph è un isomorfismo tra ipergrafi. In questo problema, sfruttiamo soprattutto la definizione di link graph: la sua struttura è definita dalla funzione *link*, che collega Punti a Handles. I primi sono l'insieme delle porte e degli inner names, mentre i secondi sono l'insieme degli archi e degli outer names (sottosezione 2.2.2). Si può quindi vedere ogni link graph come una funzione che ha come dominio i Punti e come codominio gli Handles.

Sulla base di queste osservazioni, possiamo costruire la *rete di flusso* per questo problema nel seguente modo: innanzitutto chiamiamo i due link graph rispettivamente L_F e L_G, e le loro funzioni come $link_F$ e $link_G$ (definite dagli archi verdi di figura 3.12). Possiamo collegare tutti gli elementi del dominio di $link_F$ a tutti gli elementi del dominio di $link_G$, creando così archi *orientati*, perché vanno da punti

di L_F a punti di L_G, e *pesati*, perché possono assumere un valore $p \in \{0, 1\}$. Infine, facciamo lo stesso con i loro Handle: colleghiamo tutti gli elementi del codominio di $link_F$ a tutti gli elementi del codominio di $link_G$.
Si è così creata la rete di flusso in figura 3.12, dove D_X (con $X \in \{F, G\}$) indica il dominio di $link_X$ e C_X il suo codominio.
Per chiarezza visiva, si sono omessi alcuni archi rossi, ma si deve immaginare che ogni elemento di D_F abbiamo tre archi verso ognuno degli elementi di D_G. Lo stesso vale per C_F.

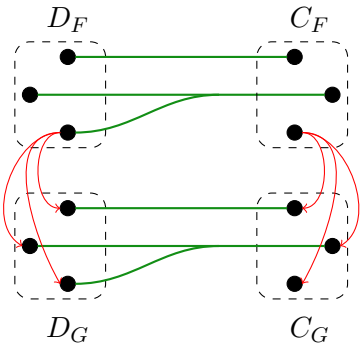


Figura 3.12: Esempio di rete di flusso per il problema di isomorfismo tra link graphs.

Incominciamo quindi con definire le variabili del sistema di equazioni:

$$x_{h,h'} \in \{0,1\} \qquad \begin{aligned} h &\in C_F = Y_F \uplus E_F \\ h' &\in C_G = Y_G \uplus E_G \end{aligned}$$

$$y_{p,p'} \in \{0,1\} \qquad \begin{aligned} p &\in D_F = X_F \uplus P_F \\ p' &\in D_G = X_G \uplus P_G \end{aligned}$$

Si sono distinte le variabili che vanno dal dominio D_F a D_G, che vengono chiamate y, da quelle che vanno dal codominio C_F a C_G, chiamate x. Le soluzioni del sistema hanno lo stesso significato che avevano nel place graph: dopo l'esecuzione, le variabili che assumeranno il valore 1 saranno solamente quelle che costituiranno la vera e propria traduzione di supporto. Quindi, anche per questo problema, valgono la proposizione 3.4.1 e la notazione 3.4.1.

I vincoli che si devono scegliere hanno il compito di "costituire" la funzione di traduzione di supporto, e devono quindi assicurare che essa sia biiettiva e che mantenga la struttura del primo link graph. Perciò, anche in questo caso, distinguiamo in *vincoli di flusso*, per il primo problema, e in *vincoli strutturali*, per il secondo.

Incominciamo con il definire i **vincoli strutturali**. Essi hanno il compito di definire una funzione che va dal primo link graph al secondo, che sia in grado di mantenere la struttura del primo. In altre parole, devono controllare che le due strutture siano *compatibili*. I vincoli strutturali fanno riferimento al caso negativo, cioè descrivono nel sistema quando due punti o due handle *non* possono costituire un match. Da qui i due vincoli:

- *Primo vincolo strutturale*: due handles (il primo di L_F e il secondo di L_G) che hanno un numero diverso di pre-immagini *non* possono costituire un match. In altre parole: se l'handle h è immagine di due punti ma l'handle h' lo è di uno solo, allora h e h' non possono essere associati. In formule:

$$x_{h,h'} = 0 \qquad |link_F^{-1}(h)| \neq |link_G^{-1}(h')|$$
$$h \in C_F = Y_F \uplus E_F$$
$$h' \in C_G = Y_G \uplus E_G$$

  In figura 3.13, si può vedere come la variabile x_{h_3,h_1} sia vincolata ad assumere il valore 0, infatti: $|link_F^{-1}(h_3)| = 0$ ma $|link_G^{-1}(h_1)| = 1$, quindi $x_{h_3,h_1} = 0$.

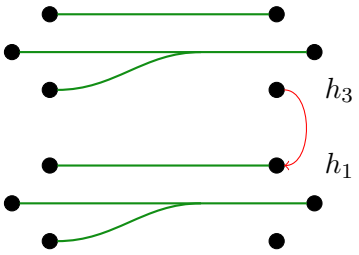


Figura 3.13: Esempio per il primo vincolo.

- *Secondo vincolo strutturale*: se due handles ($h \in C_F$ e $h' \in C_G$) non costituiscono un match, allora neanche i punti che hanno h e h' come immagine lo fanno. Questo vincolo modella la seguente implicazione:

$$x_{h,h'} = 0 \qquad \Rightarrow \qquad y_{p,p'} = 0$$

  dove $p \in link_F^{-1}(h)$ e $p' \in link_G^{-1}(h')$. Esso può essere tradotto tramite la seguente equazione.

$$y_{p,p'} \leq x_{link_F(p),link_G(p')} \qquad p \in D_F = X_F \uplus P_F$$
$$p' \in D_G = X_G \uplus P_G$$

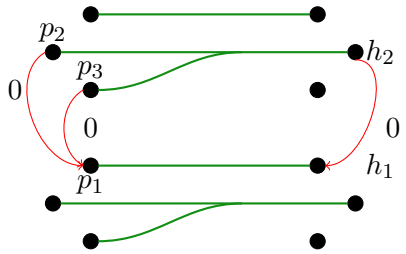


Figura 3.14: Esempio per il secondo vincolo.

Si noti che questa equazione equivale all'implicazione precedente: se $x = 0$, allora deve per forza essere che anche $y = 0$.

In figura 3.14, si può vedere un esempio per questo vincolo. Dal primo vincolo sappiamo che $x_{h_2,h_1} = 0$ perché $|link_F^{-1}(h_2)| = 2$ mentre $|link_G^{-1}(h_1)| = 1$. Aggiungendo il vincolo appena descritto, ricaviamo quindi che $y_{p_2,p_1} = 0$ e $y_{p_3,p_1} = 0$. Infatti, si vede subito che i punti p_2 e p_3 non possono costituire un match con p_1.

Descriviamo adesso i **vincoli di flusso**, che consentono di avere una funzione biiettiva. Il caso è analogo a quello per il place graph e la notazione rimane la stessa.

- *Flusso in uscita*:
  - $\delta^+(p) = 1$: Dobbiamo assicurare che ogni punto di D_F sia associato ad uno e un solo punto di D_G, che si traduce in questa equazione:

$$\sum_{p'} y_{p,p'} = 1 \qquad p \in X_F \uplus P_F \qquad p' \in X_G \uplus P_G$$

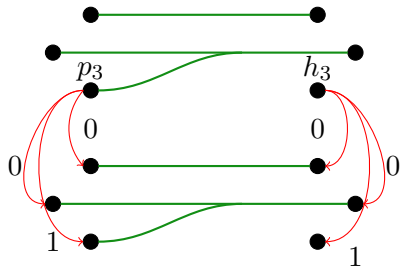


Figura 3.15: Esempio per i due vincoli sul flusso in uscita.

  - $\delta^+(h) = 1$: E' l'analogo del caso precedente. Ogni handle di C_F deve essere associato ad uno e un solo handle di C_G. La rispettiva equazione è:

$$\sum_{h'} y_{h,h'} = 1 \qquad h \in Y_F \uplus E_F \qquad p' \in Y_F \uplus E_F$$

La figura 3.15 mostra un esempio di questi due vincoli.

- *Flusso in entrata*:
  - $\delta^-(p') = 1$: ogni punto di D_G può essere associato ad uno e un solo punto di D_F. In formule:

$$\sum_{p} y_{p,p'} = 1 \qquad p \in X_F \uplus P_F \qquad p' \in X_G \uplus P_G$$

  - $\delta^-(h') = 1$: ogni handle di C_G può essere associato ad uno e un solo handle di C_F. In formule:

$$\sum_{h} y_{h,h'} = 1 \qquad h \in Y_F \uplus E_F \qquad p' \in Y_G \uplus E_G$$

  La figura 3.16 mostra un esempio per questi due vincoli.

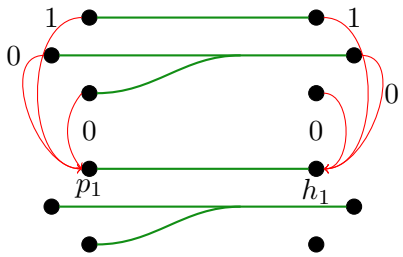


Figura 3.16: Esempio per i due vincoli sul flusso in entrata.

Si noti che i quattro vincoli di flusso assicurano la iniettività e la suriettività della funzione di traduzione di supporto, rendendola quindi *biiettiva*.

3.4.3 Vincoli di coerenza

Nella sottosezione 3.3.1, dedicata ad alcuni esempi, la figura 3.4 mostrava due bigrafi che, pur avendo i place graphs ed i link graphs isomorfi, *non* erano tali. Questo perché il sistema di equazioni non era completo: non bastano cioè i vincoli visti fino ad ora, e bisogna integrarli con dei **vincoli di coerenza** che sono dedicati all'unione delle due soluzioni.

Nell'esempio citato, si era già notato informalmente che il quadrato collegato al cerchio nel primo link graph deve essere quello più esterno nel primo place graph. Diamo ora le definizioni formali di questi vincoli:

- *Primo vincolo di coerenza*: due porte costituiscono un match nei due link graphs *se e solo se* i rispettivi nodi lo fanno nei due place graphs e le due porte hanno lo stesso indice. In formule:

$$y_{p,p'} = M_{d,m,m'} \qquad \begin{array}{ll} p = (m,i) & p' = (m',i) \\ p \in P_F & p' \in P_G \\ m \in V_F & m' \in V_G \\ i \in \mathbb{N} & \end{array}$$

  In altre parole, se due nodi m e m' non costituiscono un match nei due place graphs, allora neanche le loro porte lo fanno. D'altra parte, se m e m' costituiscono un match, cioè $M_{d,m,m'} = 1$, allora le porte con lo stesso indice devono costituire un match: $y_{p,p'} = 1$.

- *Secondo vincolo di coerenza*: due porte con indici diversi non possono costituire un match. In formule:

$$y_{p,p'} = 0 \qquad \begin{array}{ll} p = (m,i) & p' = (m',i) \\ p \in P_F & p' \in P_G \\ m \in V_F & m' \in V_G \\ i, i' \in \mathbb{N} & i \neq i' \end{array}$$

- *Terzo vincolo di coerenza*: se due punti sono di tipo diverso, cioè il primo è una porta e il secondo un inner name, allora essi non possono costituire un match.

$$y_{p,p'} = 0 \qquad p \in P_F \qquad p' \in X_G$$

  Banalmente, vale anche il caso speculare, dove il primo punto è un inner name mentre il secondo è una porta:

$$y_{p,p'} = 0 \qquad p \in X_F \qquad p' \in P_G$$

- *Quarto vincolo di coerenza*: se due handles sono di tipo diverso, cioè il primo è un arco e il secondo un outername, allora essi non possono costituire un match.

$$y_{h,h'} = 0 \qquad h \in E_F \qquad h' \in Y_G$$

  Il caso speculare, dove il primo handle è un outer name name mentre il secondo è un arco, è dato dalla formula:

$$y_{h,h'} = 0 \qquad h \in Y_F \qquad h' \in E_G$$

- *Quinto vincolo di coerenza*: un nodo può costituire un match solo con un altro nodo. Lo stesso vale per le radici e per i siti. Perciò, è impossibile per esempio che esista una variabile con valore 1 da una radice verso un nodo. I vincoli che coprono tutti i possibili casi sono:

$$\begin{aligned} M_{d,a,b} = 0 \quad &\text{se} \quad a \in n \quad b \in V_G \\ &\text{oppure se} \quad a \in n \quad b \in m \\ &\text{oppure se} \quad a \in m \quad b \in V_G \\ &\text{oppure se} \quad a \in m \quad b \in n \\ &\text{oppure se} \quad a \in V_F \quad b \in m \\ &\text{oppure se} \quad a \in V_F \quad b \in n \end{aligned}$$

- *Sesto vincolo di coerenza*: due nodi con controlli diversi non possono costituire un match. Per esempio, in figura 3.4 il quadrato più grande del primo bigrafo non può essere associato con il cerchio del secondo.

$$M_{d,a,b} = 0 \qquad ctrl_F(a) \neq ctrl_G(b)$$

3.4.4 Implementazione

Tutti i vincoli sono stati espressi sul calcolatore tramite la libreria Java *Choco v3.3.1*. Essa permette di rappresentare un sistema di equazioni che poi la stessa libreria risolverà. Forniamo quindi una panoramica della libreria ed alcuni esempi di vincoli che sono stati espressi.

La prima fase, che precede la definizione dei vincoli, riguarda la creazione delle variabili. Ad ogni variabile si deve obbligatoriamente assegnare anche un dominio, che, per quanto grande sia, deve essere finito. A titolo di esempio, in *Choco* possiamo creare le variabili x e y entrambe con dominio $\{0, 1, 2, 3, 4\}$ con le seguenti istruzioni:

```
IntVar x = VariableFactory.enumerated("x", 0, 4, solver);
IntVar y = VariableFactory.enumerated("y", 0, 4, solver);
```

Si noti come ogni variabile venga associata ad un **solver**, che è il motore interno che risolverà il sistema. Sia x che y sono associate allo stesso solver: questo equivale a dire che apparterranno allo stesso sistema di equazioni. La sintassi dei metodi è chiara: come primo argomento c'è il nome della variabile, come secondo ci può essere

un vettore (che enumera tutti gli elementi del dominio) oppure il primo estremo dell'intervallo del dominio; come terzo argomento abbiamo il limite superiore del dominio, ed infine c'è il solver al quale le variabili dovranno fare riferimento.

Esprimiamo ora degli esempi di vincoli, che sono appunto equazioni. Tutte le equazioni che appartengono allo stesso sistema, dovranno fare riferimento allo stesso solver. Nel solver di *Choco*, le equazioni vengono aggiunte al sistema tramite il suo metodo *post()*. Il sistema

$$\begin{cases} x + y = 7 \\ x \leq 3 \\ y > 1 \end{cases}$$

si può rappresentare con le seguenti istruzioni:

```
IntVar[] sumV = {x,y};
IntVar seven = VariableFactory.fixed("seven", 7, solver);
solver.post(ICF.sum(sumV, "=", seven));
solver.post(ICF.arithm(x,"<=",3));
solver.post(ICF.arithm(y,">",1));
```

Si noti come tutti i vincoli facciano riferimento a "solver", e che quindi faranno parte dello stesso sistema. Inoltre, con il metodo *sum* si è dovuto creare una costante che rappresentasse il numero 7 (nel metodo *arithm* questo non è necessario). Vediamo brevemente come si usano questi metodi: il metodo *fixed* serve per creare una costante; è simile ad *enumerated* e vuole come argomenti il nome della variabile, il valore della costante ed il solver al quale fare riferimento.

Il metodo *sum* accetta come primo argomento un vettore formato da tutte le variabili che costituiscono la somma, come secondo argomento l'operatore (per esempio $=$ oppure $\leq$) ed infine la variabile che nell'equazione sta a destra dell'operatore.

Il metodo *arithm* consente di esprimere le più comuni equazioni aritmetiche: il primo argomento è una variabile, il secondo un operatore ($<, \leq, =, \geq, >$) ed il terzo una variabile o un numero naturale.

Con il seguente comando:

```
solver.findSolution();
```

la libreria cerca di risolvere il sistema di equazioni. Se esiste una soluzione ritorna True, altrimenti False. Si noti come in questa sede non ci interessino le soluzioni del sistema, che però potrebbero venire calcolate con il metodo:

```
var.getValue() //dove var è una IntVar
```

per ogni variabile *var* che fa parte del sistema.

Da quanto appena visto, l'implementazioni dei vincoli per l'isomorfismo tra bigrafi è una traduzione quasi immediata delle formule della sezione precedente. Forniamo quindi solo alcuni esempi:

- Il vincolo sul flusso in uscita ($\delta^+(p) = 1$) da ogni punto del primo link graph è $\sum_{p'} y_{p,p'} = 1$. La sua traduzione nella libreria *Choco vs 3.3.1* usa il metodo *sum* visto prima. Si guardi la figura 3.17: se vogliamo esprimere questo vincolo su p_3 allora la corrispondente istruzione sarà:

```
BoolVar p3_p1 = VF.bool("P3-P1", solver);
BoolVar p3_p2 = VF.bool("P3-P2", solver);
BoolVar p3_p3 = VF.bool("P3-P3", solver);
IntVar[] redArrows = {p3_p1 , p3_p2 , p3_p3};
IntVar one = VF.fixed("one", 1, solver);
solver.post(ICF.sum(redArrows, "=", one));
```

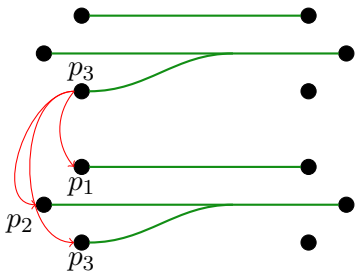


Figura 3.17: Implementazione del vincolo sul flusso in uscita.

VF è la semplice abbreviazione per *VariableFactory*. Come prima cosa, si sono create le tre variabili che corrispondo agli archi rossi della figura. Tutte e tre hanno dominio $\{0, 1\}$, per cui possiamo dichiararle direttamente come variabili booleane. Dato che c'è una sommatoria, dobbiamo usare il metodo *sum*: creiamo quindi il vettore con tutti i membri della somma (*redArrows*) e una costante per il numero 1 (*one*). L'ultima istruzione, aggiunge l'equazione $\sum_{p'} y_{p,p'} = 1$ al sistema.

- Il vincolo $y_{p,p'} \leq x_{link_F(p),link_G(p')}$ modella una semplice implicazione, come visto nella sezione precedente. Si prenda la figura 3.18: presi i due punti p_2 e p_1 rispettivamente del primo e del secondo link graph, si ha che questo vincolo può essere espresso tramite queste istruzioni:

```
BoolVar p2_p1 = VF.bool("P2-P1", solver);
BoolVar h2_h1 = VF.bool("H2-H1", solver);
solver.post(ICF.arithm(p2_p1, "<=", h2_h1));
```

Come si può vedere, la traduzione è quasi immediata: il metodo *arithm* consente di scrivere direttamente l'equazione, risultando quindi molto comodo.

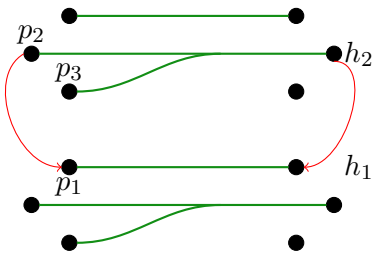


Figura 3.18: Implementazione del secondo vincolo strutturale

3.5 Benchmarks

Nella sottosezione 3.2.1 si è vista la complessità teorica del problema dell'isomorfismo tra bigrafi. Ora si vedranno dei test (benchmark) volti a misurare le prestazioni dell'algoritmo proposto. Tutti i test sono stati eseguiti su una macchina con queste caratteristiche:

- processore Intel Core i5 a 2.4GHz
- IDE: Eclipse Luna 4.4.2
- libreria per il constraint programming: *Choco vs 3.3.1*

La preparazione dei dati è stata condotta con attenzione: si è implementato un metodo che potesse generare bigrafi casuali con il numero di nodi richiesto, per fare in modo che il tempo di risoluzione non dipendesse dalla struttura del bigrafo, che può essere semplice o complessa. Questo metodo è stato implementato in un modo molto semplice:

- per prima cosa si è costruito il place graph, che è equivalente a generare un albero casuale. All'inizio del metodo, si può specificare il numero di porte che gni nodo dell'albero possiede.
- in seguito, ogni porta di ogni nodo del place graph è stata collegata ad un outername scelto casualmente da un insieme precedentemente generato.

Si vedranno due grafici:

- il primo rappresenta il *tempo di preparazione* (figura 3.19): è la misura di quanto l'algoritmo impiega per creare tutte le variabili ed i vincoli
- il secondo rappresenta il *tempo d'esecuzione* (figura 3.20), cioè quanto tempo impiega la libreria *Choco* per risolvere il sistema di equazioni

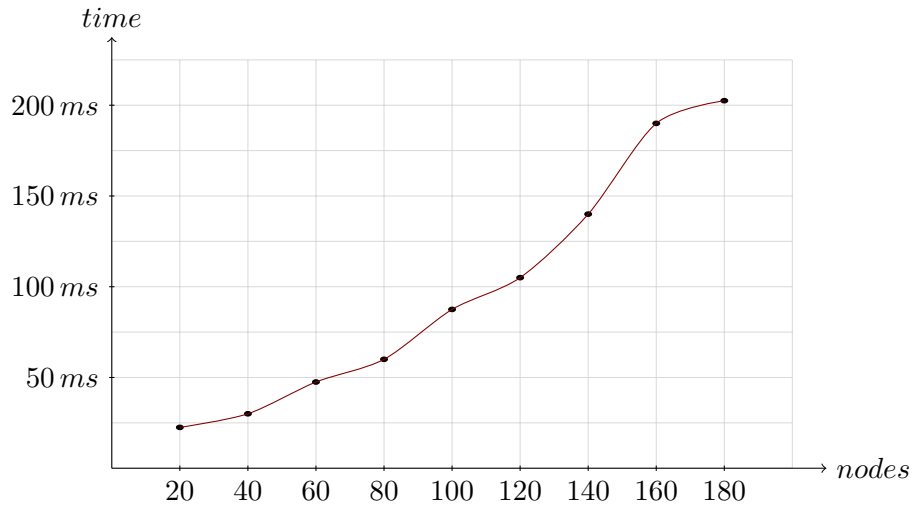


Figura 3.19: Loading Time

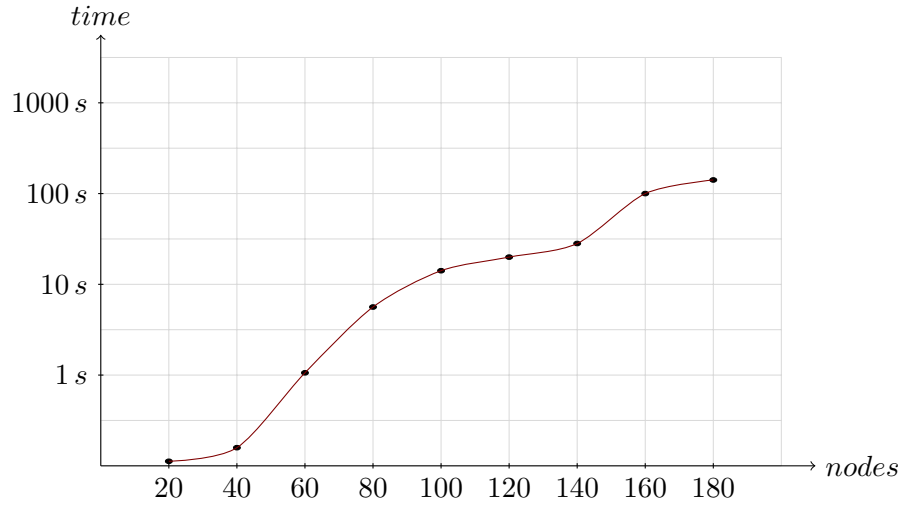


Figura 3.20: Working Time

Ogni grafico ha sulle ascisse il numero di nodi del problema: in altre parole, il problema testato in $x = n$ verifica l'isomorfismo tra due bigrafi con $n/2$ nodi ciascuno. Sulle ascisse c'è invece il tempo espresso in secondi. Il secondo grafico è in scala

logaritmica: l'aumento del numero di nodi causa un esplosione combinatoria tra i due bigrafi, che porta ad un aumento esponenziale del numero di vincoli e quindi anche del relativo tempo d'esecuzione.

Si noti come il tempo di preparazione cresca linearmente con il numero di nodi, rimanendo sempre sotto un secondo. Il tempo d'esecuzione invece aumenta drasticamente, arrivando fino a due minuti nel caso di due bigrafi con 90 nodi ciascuno.

4

Model Checker per bigrafi

In questo capitolo si vedrà come l'isomorfismo introdotto nel capitolo precedente consentirà di riconoscere quando due bigrafi sono uguali e di arrestare l'esecuzione del BRS, evitando sue evoluzioni infinite.

Inoltre si vedrà una soluzione per l'altro principale problema di questa tesi: come poter verificare date proprietà sul BRS. Riprendendo l'esempio della rete, una proprietà che potremmo verificare è l'arrivo a destinazione di un pacchetto, oppure assicurarci che nessun pacchetto non autorizzato passi attraverso un firewall.

Come per l'isomorfismo, anche questo problema necessita di una soluzione generale, che prescinde dal dominio che i bigrafi rappresentano. Si è creata quindi una semplice logica a predicati, con cui è possibile esprimere le proprietà che si vuole verificare. Essa andrà a formare la *politica* per il Model Checker, che servirà a verificare le proprietà sul grafo degli stati visto nel capitolo precedente.

4.1 Grafo degli stati

Sono stati elencati tutti i vincoli necessari per determinare l'isomorfismo tra bigrafi. Si è già visto che l'utilità di sapere quando due bigrafi sono uguali risiede nel fatto che è possibile fermare l'esecuzione del BRS, evitando così cicli infiniti tra due bigrafi uguali. Per memorizzare tutti gli stati assunti da un BRS durante la sua evoluzione, si è creato il *grafo degli stati*: è una struttura dati a grafo dove ogni nodo è a sua volta un bigrafo.

Come si può vedere dalla figura 4.1, ogni nodo è etichettato con la stringa S_i, perché si tratta di uno stato, cioè di un bigrafo. All'interno di un BRS, abbiamo visto che un bigrafo può evolversi tramite le regole di reazione, motivo per cui ogni arco orientato del grafo degli stati è etichettato con il nome della regola che ha portato dal primo stato al secondo. Nell'esempio di cui sopra, dal bigrafo iniziale S_0 si passa a S_1 tramite la regola R_1.

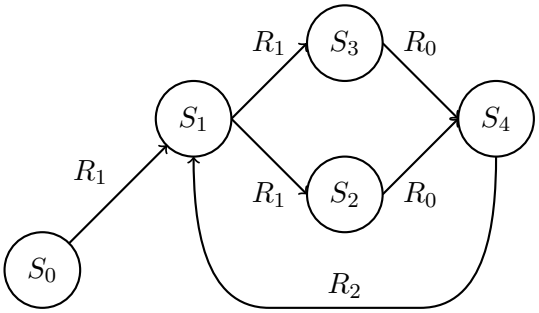


Figura 4.1: Esempio di grafo degli stati.

Si noti un aspetto molto importante: da S_0 a S_1 la regola R_1 ha generato un solo nuovo stato. Questo vuol dire che il redex di R_1 ha una sola occorrenza in S_0, ed essa è stata sostituita dal reactum di R_1, creando così lo stato S_1. Però, se prendiamo in considerazione quest'ultimo stato, si vede che ora la regola R_1 porta a due nuovi stati. Questo perché in S_1 ci sono due occorrenze del redex di R_1, che vengono sostituite dal suo reactum, creando rispettivamente i due nuovi stati S_2 e S_3. In altre parole: una regola di reazione può generare un diverso numero di stati a seconda del bigrafo a cui è applicata.

Infine, si presti attenzione all'arco tra S_4 e S_1. Il significato è il seguente: applico la regola R_2 al bigrafo S_4, generando un nuovo stato S_5. Esso è però isomorfo a S_1, cioè: S_1 e S_5 hanno la stessa *semantica*, per cui collego S_4 a S_1. Se non ci fossimo accorti di questa propietà, allora avremmo continuato ad applicare le regole R_0, R_1 e R_2 all'*infinito*, generando sempre nuovi stati, come in figura 4.2. Ora invece, dato che ci siamo accorti che da S_4 siamo ritornati ad S_1, non applichiamo più nessuna regola, avendo quindi un grafo degli stati *finito*, cioè quello di figura 4.1.

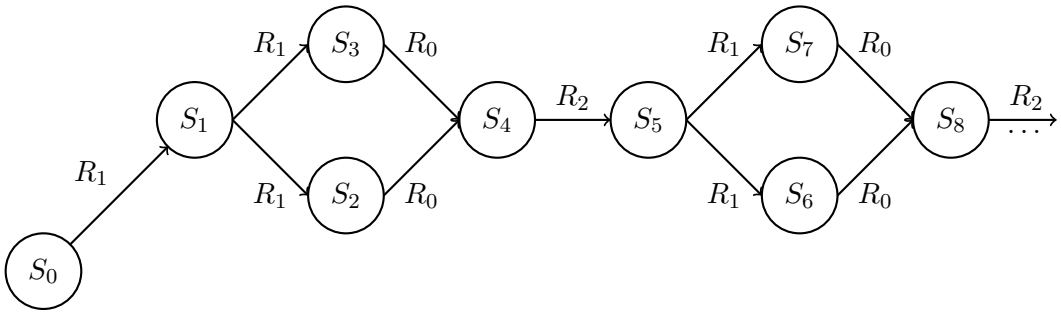


Figura 4.2: Grafo degli stati infinito.

4.1.1 Esempio

In questa sottosezione, viene fornito un esempio concreto di grafo degli stati, con particolare attenzione al problema dell'evoluzione infinita. Si riprende l'esempio della rete della sottosezione 4.1.1. L'implementazione la si può trovare in [1].

Si introduce solo la segnatura del pacchetto, del router e del dominio: $K = \{pacchetto : 2, \quad router : 2, \quad dominio : 0\}$. Chiameremo rispettivamente *Encap* e *Decap* le regole per l'incapsulamento e il decapsulamento dei pacchetto nei vari strati, per esempio da Http a Tcp, o da Ip a Ethernet. Ci concentriamo sulla regola di inoltro tra router, che è quella in figura 4.3.

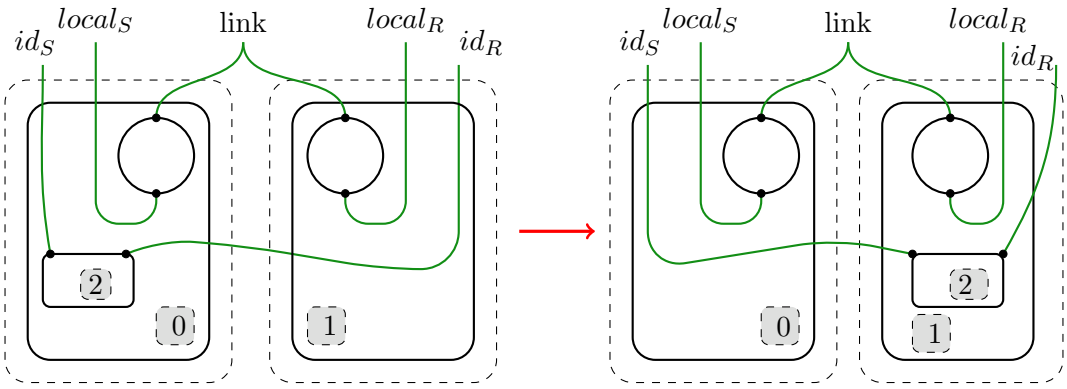


Figura 4.3: Regola di inoltro tra router.

La regola è molto semplice. Innanzitutto, incominciamo con i vari controlli. Figli diretti delle due radici sono i due rettangoli: essi sono i domini di cui fanno parte i due router. Questi ultimi, a loro volta, sono disegnati come circonferenze con due porte: la prima collegata a "*link*" e la seconda a "*Local*". Infine, il pacchetto è rappresentato da un rettangolo, anch'esso con due porte: la prima, che è quella più a sinistra, è collegata al mittente, mentre la seconda al destinatario.

Passiamo ora a definire il ruolo degli outername. "*Link*" è l'outername dedicato alla connessione tra router: se un terzo router volesse collegarsi a questi due, allora dovrebbe collegare (nel vero senso della parola) la sua porta superiore con l'outername "*link*". Si ricordi quanto detto nella sottosezione 4.1.1: i router inoltrano non deterministicamente ogni pacchetto verso tutte le sue interfacce. Se tre router sono collegati all'outername "*link*", allora uno di questi inoltra il pacchetto verso tutti gli altri due. Per cui "*link*" rappresenta l'insieme di tutte le interfacce di un router.

Gli outername "$Local_S$" e "$Local_R$" sono a disposizione solamente per gli host del dominio corrente. Per esempio, a "$Local_S$" (che sta per *Local Sender*) si possono

collegare solo gli host del dominio in cui si trova il primo router. Nella regola di figura 4.3, questi host vengono inglobati dal sito numero 0. Il medesimo discorso vale per "$Local_R$".

Il pacchetto è collegato a due outername. Il primo, quello più a sinistra, è "id_S" (*id Sender*), mentre il secondo è "id_R" (*id Receiver*). Nella realtà, questi due outername saranno gli identificativi del mittente e del destinatario. Se per esempio il pacchetto in questione è IP, allora id_S sarà l'indirizzo IP del mittente, mentre id_R quello del destinatario.

Infine, si noti anche il sito numero 2: si trova dentro il pacchetto e permette di astrarre al tipo di pacchetto. Per esempio: se è un pacchetto IP, allora è probabile che abbia incapsulato al suo interno un pacchetto TCP, che a sua volta contiene un pacchetto HTTP. Onde evitare di scrivere regole ad hoc per ogni tipo di pacchetto, si introduce il sito numero 2, così che la regola trovi un occorrenza (e quindi scatti) per qualsiasi tipo di pacchetto.

Vediamo ora un esempio reale, proposto in figura 4.4. Siano D_S e D_R i domini rispettivamente del mittente (sender) e del destinatario (receiver). Indicheremo con il triangolo il controllo per un host. Quindi h_1 e h_2 saranno rispettivamente all'interno di D_S e D_R. Lo stesso vale per i router, cioè R_S e R_R. Supponiamo che il pacchetto sia di tipo IP, e che h_1 sia collegato per esempio a 158.110.3.46, e h_2 a 158.110.144.31. Infine, per quanto detto prima a proposito di "$Local_S$" e "$Local_R$", alla porta inferiore di R_S si collegherà h_1, mentre a quella di R_R si collegherà id_2.

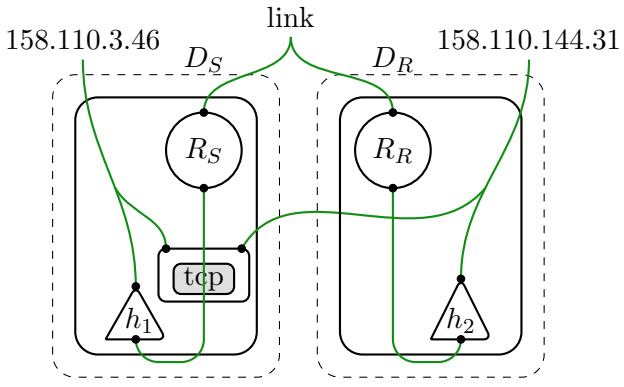


Figura 4.4: Bigrafo di partenza

Chiamiamo il bigrafo della figura 4.4 come S_0, cioè lo stato iniziale del grafo degli stati. Il nostro BRS è formato da una sola regola, cioè quella di figura 4.3, che chiameremo R_0. Si noti come il redex di R_0 trovi una ed una sola occorrenza in S_0, e quindi genererà un solo stato, ovvero S_1, dove il pacchetto si è spostano in D_R,

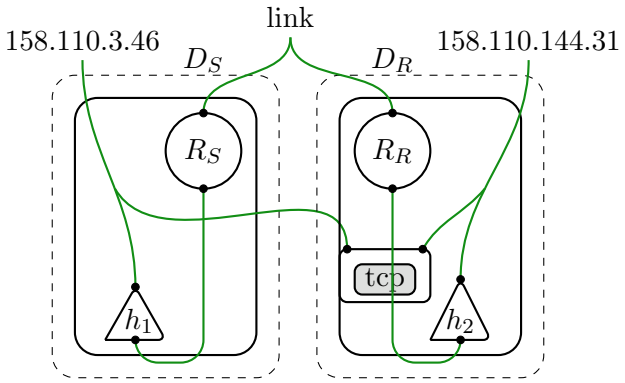


Figura 4.5: Bigrafo dopo l'applicazione della regola R_0

arrivando a destinazione. Si può vedere questo risultato nel bigrafo di figura 4.5, che sarà quindi S_1.

Di per sè, il BRS si dimentica degli stati precedenti, in questo caso S_0. Per cui, da S_1 scatterebbe di nuovo la regola R_0, creando lo stato S_2. A sua volta, da S_2, eseguirebbe di nuovo R_0 e causerebbe così un'esecuzione infinita, dando luogo al grafo degli stati in figura 4.6.

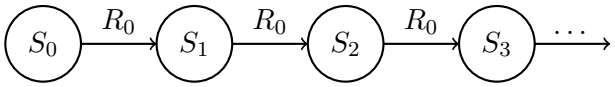


Figura 4.6: Grafo degli stati infinito.

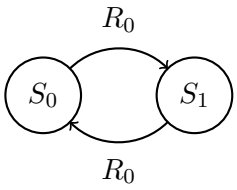


Figura 4.7: Grafo degli stati finito.

Grazie all'isomorfismo tra bigrafi, riusciamo a collegare lo stato S_1 a S_0. Poichè da S_1 tramite la regola R_0 viene creato lo stato S_2, siamo ora in grado non doverlo più memorizzare, in quanto è uguale (o meglio, è semanticamente equivalente) allo stato S_0. Per cui, ritrovandoci di nuovo in S_0, sappiamo che non dobbiamo applicare più nessuna regola, pena un'esecuzione infinita del BRS. Tramite l'isomorfismo, si è potuto generare il *grafo degli stati* della figura 4.7.

4.2 Model Checker

Un Model Checker (MC) è un metodo per verificare delle proprietà in un sistema formale. Nel nostro caso, si è costruito un MC basato sul grafo degli stati (come quello di figura 4.8): il problema sarà capire se un nodo rispetti le proprietà specificate.

Si è visto come nel *grafo degli stati* ogni nodo sia a sua volta un bigrafo. In figura 4.8 c'è il grafo degli stati dell'esempio 4.1.1 sullo scambio di pacchetti tra due router. Ci possiamo chiedere se in uno dei due nodi il pacchetto sia arrivato a destinazione, cioè se uno dei due stati S_i ($i \in \{0, 1\}$) il pacchetto sia nello stesso dominio dell' host destinazione.

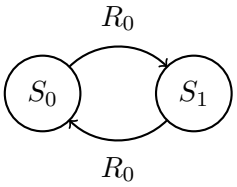


Figura 4.8: Grafo degli stati.

Nei MC queste proprietà sono di solito esprimibili attraverso una qualche logica, per cui possiamo esprimere formalmente cosa significa che un MC verifichi una certa proprietà.

Proposizione 4.2.1. *Il problema della verifica di una proprietà da parte di un MC è esprimibile come:*

$$MC, S_0 \models p$$

dove MC è un model checker, S_0 è lo stato iniziale e p è una proprietà espressa in una qualche logica.

Ovviamente, dallo stato S_0 il MC evolverà secondo precise regole per formare tutti i possibili stati $S_0 \dots S_n$: nel nostro caso, ogni arco tra due nodi del *grafo degli stati* è una regola di reazione. Per cui il model checker controllerà l'intero grafo: appena trova uno stato S_i che soddisfa p ($MC, S_i \models p$) ritorna True, altrimenti, cioè nel caso in cui *tutti* gli stati del grafo non rispettino la proprietà, ritorna False.

Queste considerazioni ci portano a definire il comportamento del MC:

Proposizione 4.2.2. *Il comportamento di un model checker MC è definito dalla seguente relazione:*

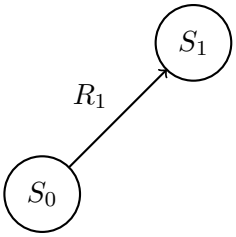
$$\begin{cases} return \;\; True & if \;\; \exists S_i \in MC \;\; : \;\; MC, S_i \models p \\ return \;\; False & otherwise \;\; (\forall S_i \in MC \;\; (MC, S_i \not\models p)) \end{cases}$$

Nell'esempio 4.8, il problema è quindi banale: il model checker ritorna vero se e solo se è vera la formula $MC, S_0 \models p \vee MC, S_1 \models p$.

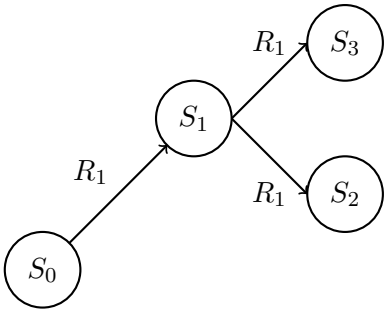
4.2.1 Generazione degli stati

Si è appena visto che in un model checker si possono esprimere delle proprietà: ogni MC ha però anche un altro grado di libertà, che riguarda la generazione degli stati. Come creare il grafo degli stati? E con che ordine?

(a) Primo passo



(b) Secondo passo



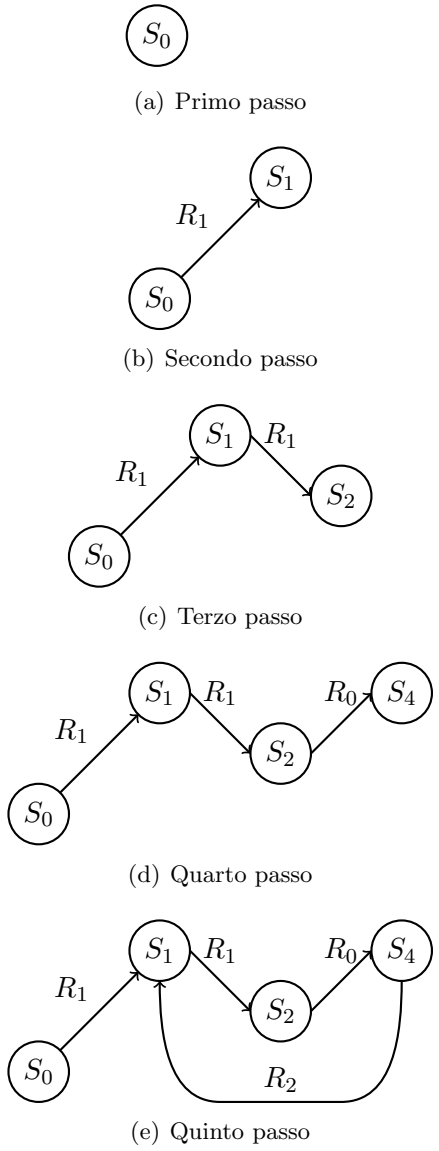
(c) Terzo passo

Figura 4.9: Esempio di generazione Breadth First.

Nel MC costruito per questa tesi, che chiameremo MC_{big}, ci sono varie strategie ed ognuna è adatta per certi scopi. Vediamone alcune:

- *Strategia Breadth First*: per ogni stato S_i vengono generati tutti gli stati possibili adiacenti ad S_i. Per esempio, in figura 4.9, si mostrano i primi tre passi della strategia Breadth First. Il suo vantaggio è che non si tralascia nessuno stato, ottenendo un grafo degli stati *completo*. Inoltre, se un nodo porta ad un

vicolo cieco (cioè se non genera nessun bigrafo tramite nessuna regola) allora questo viene semplicemente tolto dalla coda. Lo svantaggio è che può essere molto lenta: se da ogni stato si generano k stati (con k molto alto), allora prima di verificare la proprietà potrebbero volerci molte iterazioni.



(a) Primo passo

(b) Secondo passo

(c) Terzo passo

(d) Quarto passo

(e) Quinto passo

Figura 4.10: Esempio di generazione random.

- *Strategia Random*: se dallo stato S_i si possono applicare k regole allora viene scelto in maniera random un numero naturale $m \in \{1 \dots k\}$ e si genera solamente lo stato S_m. Questo consente di non memorizzare l'intero grafo degli stati (che in certi casi può essere molto grande) e verificare al momento della generazione di S_k se $MC, S_k \models p$. Lo svantaggio è quello che l'esecu-

zione potrebbe andare avanti all'infinito, infatti il grafo *non è completo*. Per cui potenzialmente potrebbero occorrere infinite evoluzioni prima di verificare una proprietà. Un esempio di generazione con la strategia random è quello in figura 4.10.

E' possibile creare altre strategie a seconda degli scopi: in questa sede, se non specificato altrimenti, si assumerà che la strategia sia sempre quella Breadth First, che consente di computare l'intero grafo degli stati.

4.3 Logica per i bigrafi

Nella precedente sezione si è vista la struttura base del model checker per i bigrafi (MC_{big}): il grafo degli stati. L'altro aspetto importante di ogni MC sono le proprietà: necessitiamo quindi di un linguaggio per esprimerle. Il primo problema riscontrato è stato quello riguardante la flessibilità: come fare ad avere un unico linguaggio che astraesse dal dominio scelto e che potesse essere adatto per qualsiasi BRS? In altre parole, è necessario scegliere un linguaggio che sia flessibile e allo stesso tempo espressivo.

La linea guida seguita per la scelta del linguaggio è stata quindi la sua *universalità*. Per esempio: si prendano gli esempi 2.5.4 sulla moltiplicazione e 4.1.1 sulla rete. Sia la proprietà p_1 definita come "Il risultato è il numero 8". Sia la proprietà p_2 definita come "Il pacchetto è arrivato a destinazione". Si noti come i domini dei due esempi siano totalmente differenti: il nostro linguaggio deve permettere di esprimere le due proprietà p_1 e p_2, senza dover ricorrere ad altri formalismi.

Si capisce bene come un tale linguaggio così generale sia molto comodo per esprimere le proprietà da fare verificare al model checker MC_{big}. Infatti, in questo modo si crea uno strumento generale **valido per qualsiasi BRS**.

Si è scelto di usare una logica a predicati, esprimibile attraverso il linguaggio generato da una grammatica *Context Free*.

4.3.1 Sintassi

Incominciamo con il descrivere la sintassi del linguaggio.

Proposizione 4.3.1. *Il linguaggio $L(G)$ per il model checker MC_{big} è generato dalla grammatica $G = (V, T, P, S)$, dove:*

- $V = \{\varphi, \sigma\}$ *è l'insieme di variabili*

- $U = \{T, \wedge, \text{`(', `)', `,' }, \neg, \mathcal{W}, \pi, A \dots Z\}$ *è l'insieme di simboli terminali*
- P *è l'insieme di* produzioni*, definito dalle seguenti relazioni:*

$$\varphi \to T \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathcal{W}_\sigma(\varphi, \varphi, \varphi) \mid \pi_\sigma$$
$$\sigma \to A\sigma \mid \cdots \mid Z\sigma \mid \varepsilon$$

- $S = \{\varphi\}$ *è il simbolo iniziale*

Il nostro linguaggio sarà quindi definito dall'insieme

$$L(M) = \{w \in U^* \ : \ S \Rightarrow_*^G w\}.$$

Seguendo l'usuale definizione di grammatica CF, diamo ora alcuni esempi di stringhe generabili dalla grammatica G, cioè di formule appartenenti al linguaggio L(G):

- $\varphi = \mathbb{W}_B(T, T, T) \wedge \pi_C$
- $\varphi = \mathbb{W}_B(\pi_X, \pi_Y, \pi_Z)$
- $\varphi = \neg\pi_B \wedge \pi_C$

4.3.2 Semantica

Definiamo ora la semantica del linguaggio, cioè specifichiamo il significato di ogni predicato. Si vedrà che quella presentata è una logica spaziale e non temporale, motivo per cui le proprietà verranno chiamate **proprietà locali**. Spesso nei model checker si usano logiche temporali o spazio-temporali. Nell'implementazione, si è comunque dato spazio a tali logiche, rendendo le classi flessibili. In futuro, sarà quindi possibile aggiungere una nuova logica a MC_{big}.

La semantica è definita come segue (ricordiamo che ogni stato S_i è un bigrafo):

Proposizione 4.3.2. *Siano S uno stato e φ una proprietà espressa nel linguaggio L(G). La relazione $S \models \varphi$ (lo stato S soddisfa la proprietà φ) è definita per ricorsione sulla complessità di φ:*

- $S \models T$ *sempre*
- $S \models \varphi_1 \wedge \varphi_2 \quad \Leftrightarrow \quad \begin{cases} S \models \varphi_1 \\ S \models \varphi_2 \end{cases}$
- $S \models \neg\varphi \quad \Leftrightarrow \quad S \not\models \varphi$

- $S \models \mathcal{W}_\sigma(\varphi_1, \varphi_2, \varphi_3) \quad \Leftrightarrow \quad \begin{cases} \exists C, D \ : \ S = C \circ (\sigma \otimes id_I) \circ D \\ C \models \varphi_1 \quad \sigma \models \varphi_2 \quad D \models \varphi_3 \end{cases}$
- $S \models \pi_\sigma \quad \Leftrightarrow \quad S \doteq \sigma$

Si dirà che la formula φ costituisce la *politica* per il model checker.

Proposizione 4.3.3. *Ogni formula del linguaggio $L(G)$ viene chiamata* ***proprietà locale*** *perché deriva da una logica spaziale e quindi può fare riferimento solamente ad aspetti spaziali/locali di un bigrafo.*

Analizziamo ora i vari predicati. I primi tre consentono le usuali operazioni della logica proposizionale, mentre il terzo è un predicato ad-hoc per questa logica: $\mathcal{W}_\sigma$ è detto "Wario Predicate"[1], e usa l'operazione di Match per controllare le tre proprietà che ha come argomento. Facciamo un esempio: sia $\mathcal{W}_B(T, T, T)$ un Wario Predicate. Lo stato S soddisfa questo predicato ($S \models \mathcal{W}_B(T, T, T)$) se e solo se esiste un match M di B nel bigrafo S tale che rispetti queste condizioni: il contesto del match M deve soddisfare φ_1, il redex di M deve soddisfare φ_2 mentre i parametri di M devono soddisfare φ_3. Poichè $\varphi_1 = \varphi_2 = \varphi_3 = T$, si ha che $S \models \mathcal{W}_B(T, T, T)$ se e solo se esiste un match di B in S. Il Wario Predicate consente quindi di isolare contesto, redex e parametri e verificare le proprietà in modo indipendente per ognuno di questi tre bigrafi.

L'ultimo predicato, π_σ, controlla se esiste un isomorfismo tra due bigrafi. Per esempio, lo stato S_i soddisfa il predicato π_A (in formule $S_i \models \pi_A$) se e solo se S_i è isomorfo al bigrafo A, cioè $S_i \doteq A$. Questo predicato è di particolare importanza: esso funge da simbolo di uguaglianza tra bigrafi, rendendo quindi la nostra logica una *logica con uguaglianza.*

Si osservino tutti e cinque i predicati: dai i primi tre è possibile derivare ogni formula della logica proposizionale. Per esempio: se si vuole esprimere la formula $\varphi_1 \vee \varphi_2$, allora si possono usare le leggi di De Morgan e scrivere $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$. Oppure, se si vuole esprimere il falso, basterà la formula $\neg T$. Gli ultimi due predicati sono invece propri dei bigrafi. Si osservi la loro definizione: si può notare che il Wario Predicate fa riferimento alla struttura interna del bigrafo consentendo infinite scomposizioni. Tramite questo predicato posso quindi *isolare* qualsiasi parte del bigrafo ed esprimere proprietà su di essa. Si può pensare alla sua funzionalità in

[1]Nella storia della logica, le lettere dei nuovi predicati sono spesso state scelte rovesciando l'iniziale del predicato. É questo l'esempio di $\exists$ (per il predicato *Exist*) e di $\forall$ (per il predicato *for All*). Volendo mantenere la tradizione ed essendo il *Matching* la caratteristica peculiare di questo predicato, si è scelto il simbolo $\mathcal{W}$.

questo modo: $\mathbb{W}_\sigma$ permette di spostarci all'interno del bigrafo, scegliere una sua parte (σ) e verificare se essa soddisfa una certa proprietà. Il predicato π_σ, come abbiamo già notato, ci consente di avere una logica con uguaglianza, permettendo quindi di aumentare la sua espressibilità.

4.4 Dettagli Implementativi

Nel codice sorgente di questa tesi, i bigrafi e i BRS, nonchè la loro evoluzione, è stata modellata tramite *JLibbig*[6], una libreria Java che tra le altre cose consente di specificare le varie regole di reazione ed eseguirle sul bigrafo.

4.4.1 Property Matcher

JLibbig mette a disposizione la possibilità di assegnare ad ogni nodo delle proprietà. Per esempio, ai due router dell'esempio 4.1.1 si possono assegnare delle stringhe (in realtà qualsiasi tipo di oggetto) che descrivano il loro nome, per esempio R_S (Router Sender) e R_R (Router Receiver). Inoltre, è possibile estendere la classe Matcher e creare il proprio Matcher personale: in questa sede, si era interessati a definire un matcher in cui due nodi potessero *costituire* un match se e solo se avessero le stesse proprietà.

Il *Property Matcher* è molto comodo per esprimere le proprietà per il model checker: per esempio, potremmo essere interessati a sapere quando il pacchetto con destinazione 158.110.144.31 arrivi all'host con tale indirizzo IP. Quindi si potrebbe creare un Wario Predicate che consenta di capire quando il pacchetto e il destinatario sono dentro lo stesso dominio. Però, senza il *Property Matcher*, la presenza di più pacchetti creerebbe confusione. Infatti non sapremmo più a quale pacchetto fare riferimento. Essendo interessati *solamente* al pacchetto destinato a 158.110.144.31, ci occorre il *Property Matcher*.

4.4.2 Regole di Reazione con Proprietà

In *JLibbig* le regole di reazione non fanno alcun riferimento alle proprietà, il che significa che dopo l'applicazione di una regola ogni nodo coinvolto (cioè attivo) perde le sue proprietà. Si è creata quindi una classe che ne consenta il *mantenimento* anche dopo lo scatto della regola. Chiaramente, il modo in cui le proprietà si devono conservare è lasciato da definire all'utente, perché è impossibile definirlo a priori. Per esempio: prendiamo la regola di figura 4.11, che inoltra e duplica un pacchetto.

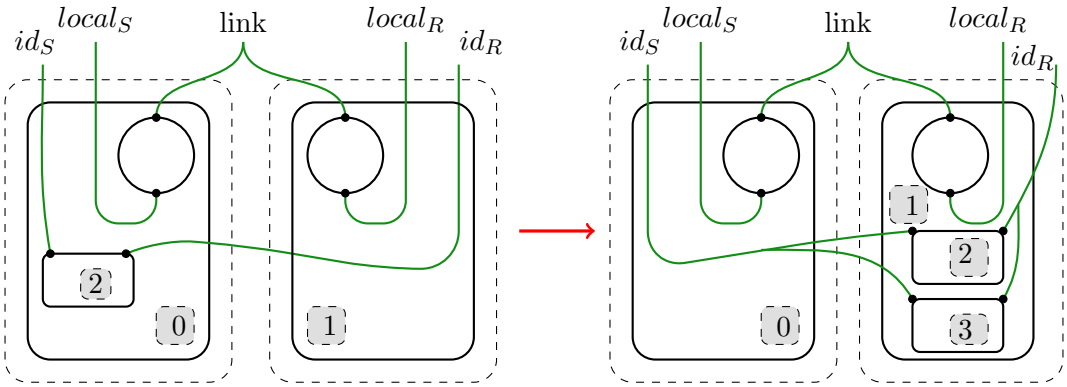


Figura 4.11: Regola di inoltro tra router.

Secondo la definizione di regola di reazione, il redex viene sostituito dal reactum: esso però è un *nuovo* bigrafo e quindi ha nuovi nomi per tutti gli outer e inner names, e tutti i suoi nodi sono privi di proprietà. Supponiamo che prima dell'esecuzione della regola il pacchetto del redex abbia come proprietà la stringa P_1. Dopo l'esecuzione si vuole che il primo pacchetto del reactum abbia tutte le proprietà di P_1, mentre il secondo abbia la nuova proprietà New_Packet. E' ovvio che tale scelta è arbitraria, ed è questo il motivo per cui la definizione del modo in cui le proprietà si conservano è stata lasciata all'utente.

Infine, creando delle regole di reazione che consentano il mantenimento delle proprietà, è possibile usare il *Property Matcher* anche dopo l'esecuzione di varie regole: di conseguenza lo possiamo usare anche nel model checker MC_{big}.

4.4.3 MC_{big}

Si è voluto implementare il model checker MC_{big} in maniera che fosse il più flessibile possibile. I suoi gradi di libertà sono principalmente il modo in cui si esprimono le proprietà e il modo in cui esso genera i nuovi nodi del grafo degli stati. Si è visto che il primo problema è stato risolto introducendo una logica spaziale; il secondo problema invece ha richiesto l'introduzione di una particolare classe chiamata *simulazione*.

Ogni simulazione usa una particolare strategia, scelta tra quelle della sottosezione 4.2.1, per generare i nuovi nodi. Per esempio, si ha che la simulazione della classe *BreathFirstSim* usa la strategia *Breath First*. Si è quindi lasciato all'utente il compito di scegliere in che modo i nuovi stati vengano generati, in quanto le prestazione del model checker dipendono molto da questa scelta.

Per usare MC_{big} occorre quindi scrivere una particolare proprietà e scegliere una *simulazione* per la generazione dei nuovi stati. Un esempio di codice è riportato qui

sotto:

```
Predicate trueP = new TruePredicate();
Predicate notP = new NotPredicate(trueP);
Sim simulation = new BreadthFirstSim(bigraph, rules);
ModelChecker mc = new ModelChecker(sim, trueP);
```

dove *bigraph* è un dato bigrafo e *rules* è un array di regole di reazione. Infine, per verificare la proprietà specificata basta la seguente istruzione:

```
mc.modelCheck();
```

che ritorna True o False, a seconda che esista o meno un nodo del grafo degli stati che rispetti la proprietà.

4.5 Esempi

Si vedranno ora degli esempi di formule e di come poterle usare con il model checker MC_{big}.

4.5.1 Moltiplicazione

Il primo esempio che si propone riprende la moltiplicazione tra numeri naturali della sottosezione 2.5.4. Se rappresentiamo la moltiplicazione $x * y$, ci possiamo chiedere se il BRS funzioni correttamente con le regole che abbiamo definito e verificare che il risultato sia corretto. La proprietà da verificare è quindi questa: "Dati due numeri x e y, il risultato della loro moltiplicazione deve essere il numero $x * y$". Vediamo ora la formula corrispondente: sappiamo che con la segnatura iniziale (vedi 2.5.4) un numero naturale n è rappresentato da un nodo di tipo *num* che contiene n nodi di tipo 1. Per esempio, il numero 8 è il bigrafo di figura 4.12.

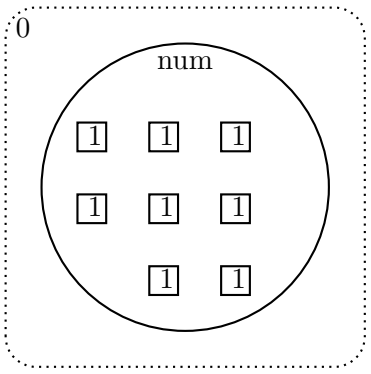


Figura 4.12: Bigrafo per il numero 8

Chiamiamo B il bigrafo di figura 4.12. La proprietà di cui sopra si può esprimere con la seguente formula logica: $\varphi = \pi_B$.

Diamo ora al model checker la formula φ. MC_{big} incomincerà a generare il grafo degli stati con la strategia Breadth First (se non specificato altrimenti), e per ogni nuovo stato S_i controllerà se $MC, S_i \models \varphi$. In questo esempio, il grafo generato sarà quello di figura 4.13. Per gli stati S_0 e S_1 il model checker troverà che la proprietà non è soddisfatta, perché nessuno di questi stati è isomorfo a B. Arrivando però a S_2, la proprietà φ sarà soddisfatta, ovvero $MC, S_2 \models \varphi$, e MC_{big} ritornerà True.

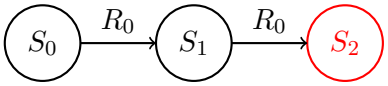


Figura 4.13: Model Checker.

4.5.2 Router

Il secondo esempio riprende quello della sottosezione 4.1.1. Vogliamo modellare una rete con quattro domini, come quella di figura 4.14. L'host h_1 vuole comunicare con h_4, inviando un pacchetto IP. Tra questi due host ci sono due domini, con due router ciascuno. Prendiamo in considerazione il dominio D_2: l'arco tra $R_{2.1}$ e $R_{2.2}$ significa che i due router sono collegati e quindi, essendo nello stesso dominio, ogni pacchetto che arriverà a $R_{2.1}$ arriverà anche a $R_{2.1}$.

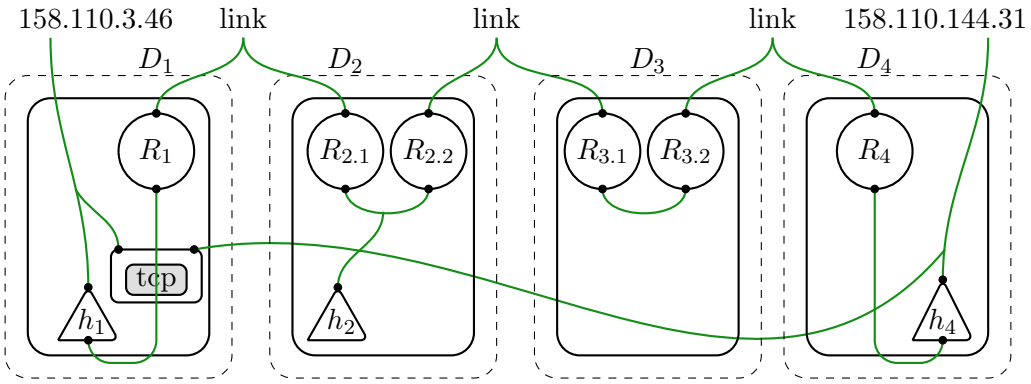


Figura 4.14: Bigrafo di partenza

Si ricordi che un router inoltra non deterministicamente ogni pacchetto verso tutte le uscite (interfacce). Questo permette di creare un compromesso tra numero di regole ed efficienza del BRS. Perciò, l'unica regola di cui abbiamo bisogno è quella di figura 4.15 (si rimanda alla sottosezione 4.1.1 per la sua descrizione).

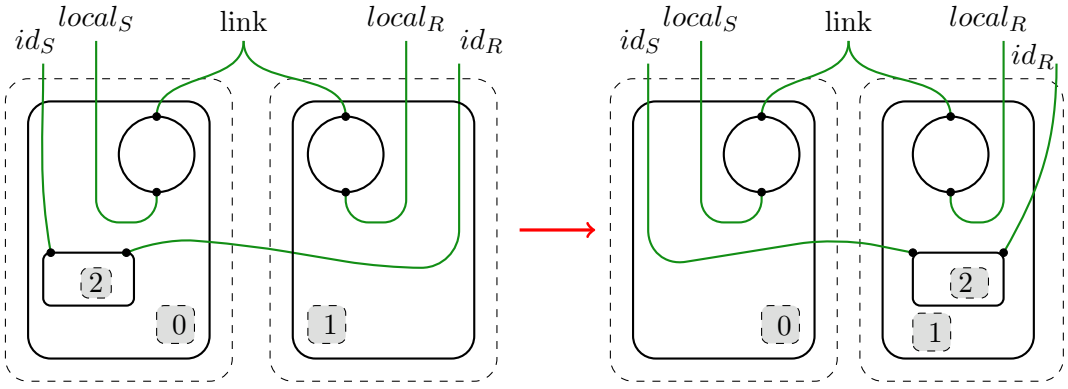
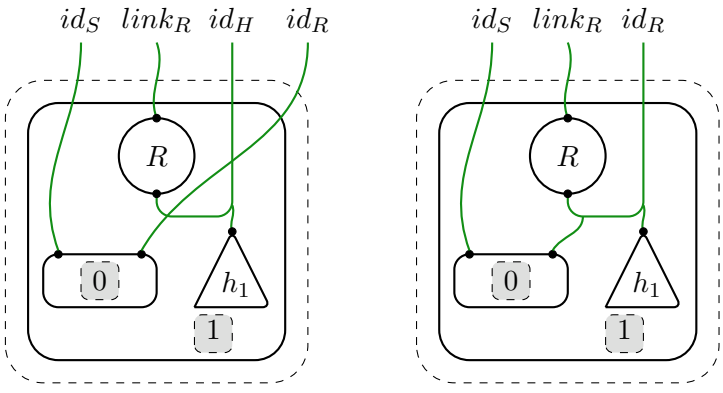


Figura 4.15: Regola di inoltro tra router.

Ora il problema di capire quando un pacchetto è arrivato a destinazione diventa più interessante. Ci sono tre host, e bisogna prestare attenzione a quale sia il corretto destinatario.

Per esempio, non si deve fare il seguente errore: vogliamo esprimere la proprietà "Il pacchetto è arrivato a destinazione". Nella logica descritta precedentemente, si potrebbe sbagliare e creare una formula del genere: $\varphi = \mathbb{W}_B(T, T, T)$, dove B è il bigrafo di figura 4.16.a.



(a) Predicato Sbagliato (b) Predicato Corretto

Figura 4.16: Esempi di predicati

Questa proprietà **non** verifica l'arrivo a destinazione di un pacchetto, infatti è soddisfatta anche esso si trova per esempio nel dominio D_2. Quello che non abbiamo

considerato sono gli outernames: quello del pacchetto e quello dell'host devono essere lo stesso outername; solo così il *Wario Predicate* sarà soddisfatto. Il predicato corretto è quindi quello di figura 4.16.a.

Testiamo ora la formula $\varphi = \mathbb{W}_B(T, T, T)$ sul bigrafo di partenza, cioè quello di figura 4.14, che sarà chiamato S_0. Per prima cosa, il model checker controlla se in questo stato iniziale la formula sia soddisfatta, ovvero se $MC, S_0 \models \varphi$. Non essendo soddisfatta continua. Inizialmente la regola R_0 (in figura 4.15) trova un solo match, per cui il pacchetto viene inoltrato dal dominio D_1 a D_2, dando origine allo stato S_1. E' utile guardare il grafo degli stati di figura 4.17 per tenere traccia di tutte le esecuzioni.

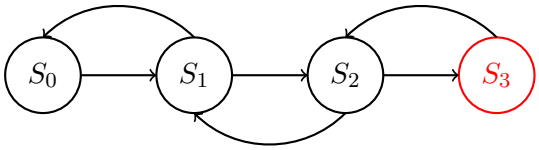


Figura 4.17: Grafo degli stati

In S_1 la formula non è soddisfatta ($MC, S_1 \not\models \varphi$), quindi MC_{big} continua. Dallo stato S_1, la regola R_0 trova ora due match: il primo coinvolge il router $R_{2.2}$ e manderà il pacchetto nel dominio D_3, mentre il secondo riguarda il router $R_{2.1}$ e manderà il pacchetto indietro verso D_1. Quindi il grafo degli stati avrà rispettivamente gli archi: $S_1 \to S_2$ e $S_1 \to S_0$. Il model checker ora dovrà duplicare le computazioni, cioè seguire sia il primo che il secondo arco.

Seguendo l'arco $S_1 \to S_0$, il MC trova il nodo S_0 grazie all'algoritmo per l'isomorfismo e riconosce che appartiene già al grafo degli stati. La computazione di questo ramo **viene interrotta** perché il model checker capisce che continuando si causerebbe un'esecuzione infinita.

Seguendo l'arco $S_1 \to S_2$, si genera invece un nuovo stato S_2. Il model checker cerca in tutto il grafo degli stati un nodo isomorfo a S_2 e, se non lo trova, capisce che esso è un *nuovo* stato. Dato che $MC, S_2 \not\models \varphi$, la computazione di questo ramo quindi continua.

Applicando R_0 a S_2 il discorso è lo stesso: la computazione che segue l'arco $S_2 \to S_1$ si interrompe, mentre quella che segue $S_2 \to S_3$ continua. Si noti come la strategia *Breadth First* consenta di interrompere da subito le computazioni che causano cicli infiniti: è uno dei vantaggi di questa strategia.

Infine, lo stato S_3 è il bigrafo in cui il pacchetto è dentro il dominio D_4. La regola R_0 trova solamente un match, ed eseguendola rimanda il pacchetto indietro verso il dominio D_3, creando nel grafo degli stati l'arco $S_3 \to S_2$. Questa volta però lo stato S_3 soddisfa la proprietà desiderata: il model checker trova che

$$MC, S_3 \models \mathbb{W}_B(T, T, T) \qquad \text{dove B è il bigrafo di figura 4.16.b}$$

e quindi ritorna True.

Un'ultima osservazione: nel caso di più pacchetti si deve prestare attenzione ad un altro dettaglio. Se siamo interessati a verificare che il pacchetto spedito da 158.110.3.46 a 158.110.144.31 sia arrivato a destinazione, dobbiamo usare il Property Matcher. Definiremo quindi la formula sempre come $\varphi = \mathbb{W}_B(T, T, T)$, ma ora il bigrafo B avrà nomi specifici nei suoi outernames, come in figura 4.18.

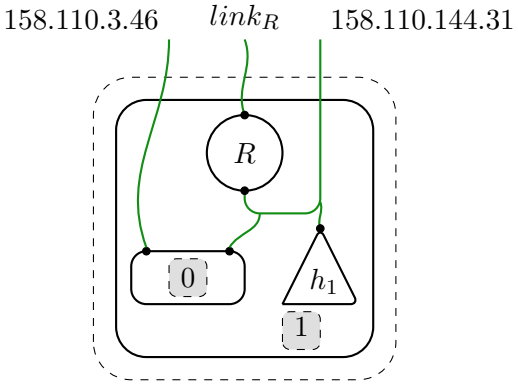


Figura 4.18: Predicato con proprietà

5

Casi di studio

In questo capitolo vedremo degli esempi che riguarderanno principalmente il model checker MC_{big} e la sua logica. Si cambierà spesso dominio, mostrando come i bigrafi siano flessibili per rappresentare vari tipi di sistema.

5.1 NFA

In questa sezione si propone una codifica in bigrafi degli NFA (*non-deterministic finite automata*). Come noto, ogni automa A denota un linguaggio L(A). Vedremo come l'implementazione di un automa con i bigrafi consenta di avere una sorta di analizzatore lessicale il cui motore interno funziona tramite BRS. In questa sede, si è costruito un modulo che accetta dall'utente una stringa x e restituisce True se $x \in L(A)$, altrimenti False.

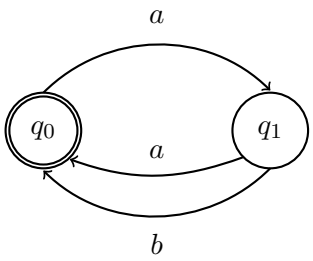


Figura 5.1: Automa per il linguaggio $(a(a+b))^*$

Prendiamo l'automa di figura 5.1, che chiameremo A. Definiamolo formalmente:

Proposizione 5.1.1. *L'automa A è una quintupla $(Q, \Sigma, \delta, q_0, F)$, dove:*

- $Q = \{q_0, q_1\}$ *è l'insieme finito di stati*
- $\Sigma = \{a, b\}$ *è l'alfabeto di input*
- $\delta : Q \times \Sigma \to Q$ *è la funzione di transizione definita come:*

$$\delta(q_0, a) = \{q_1\}$$
$$\delta(q_1, a) = \{q_0\}$$
$$\delta(q_1, b) = \{q_0\}$$

- q_0 *è lo stato iniziale*
- $F = \{q_0\}$ *è l'insieme di stati finali*

A riconosce il linguaggio $L(A) = (a(a + b))^*$.

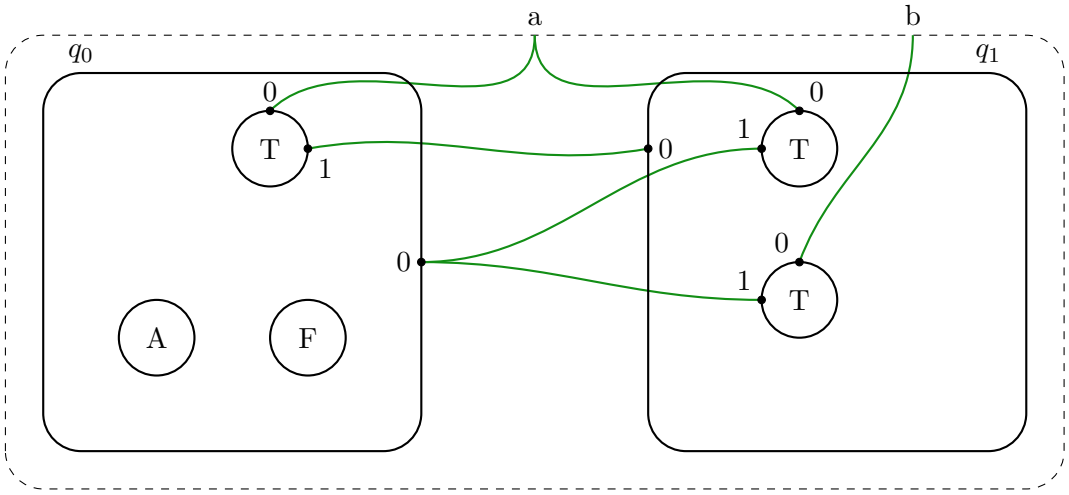


Figura 5.2: Bigrafo per l'automa A

Trattiamo ora il problema di come rappresentare gli automi tramite i bigrafi. Per prima cosa, guardiamo la figura 5.2. I due stati q_0 e q_1 sono stati modellati con dei rettangoli aventi una sola porta. Le transizioni sono i nodi di tipo T. Ogni stato q_i contiene al suo interno tutte le transizioni che partono da esso: per esempio q_0 contiene solo un nodo T perché la sola transizione che parte da q_0 è $q_0 \xrightarrow{a} q_1$, mentre q_1 contiene due nodi T, che modellano le transizioni $q_1 \xrightarrow{a} q_0$ e $q_1 \xrightarrow{b} q_0$.

Ogni transizione è quindi modellata da un nodo di controllo T con due porte: la prima è collegata al carattere che fa scattare la transizione, mentre la seconda è collegata allo stato destinazione. Nella precedente figura, si prenda in considerazione il nodo T dentro q_0: esso simboleggia la transizione $q_0 \xrightarrow{a} q_1$ perché:

- T è dentro q_0
- la prima porta di T è collegata all'outername 'a'
- la seconda porta di T è collegata a q_1

Lo stesso discorso vale per le altre transizioni. Si noti come l'alfabeto Σ sia rappresentato tramite *l'insieme degli outernames*. Il nodo A (Active) indica che lo stato q_i che lo contiene è quello attivo, ovvero: l'automa si trova nello stato q_i se e solo se q_i contiene il nodo A. Infine, lo stato q_i è uno stato finale ($q_i \in F$) se e solo se contiene al suo interno il nodo di tipo F (Final). Queste considerazioni ci portano a definire la segnatura del bigrafo:

Proposizione 5.1.2 (Segnature per gli NFA)**.** *La segnatura del bigrafo rappresentante un generico NFA è definita come segue, dove la notazione* $Node : n :$ *significa che il nodo* $Node$ *ha* n *porte:*

- *State : 1*
- *Transition : 2*
- *ActiveState : 0*
- *FinalState : 0*
- *String : 0*
- *Input : 3*

Dalla segnatura di sopra si scopre che ci sono due nuovi controlli: *String* e *Input*. Il nodo di tipo *String* è quello che dovrà contenere la stringa che l'utente immetterà per farla riconoscere dall'automa, mentre ogni nodo di tipo *Input* è un carattere della stringa.

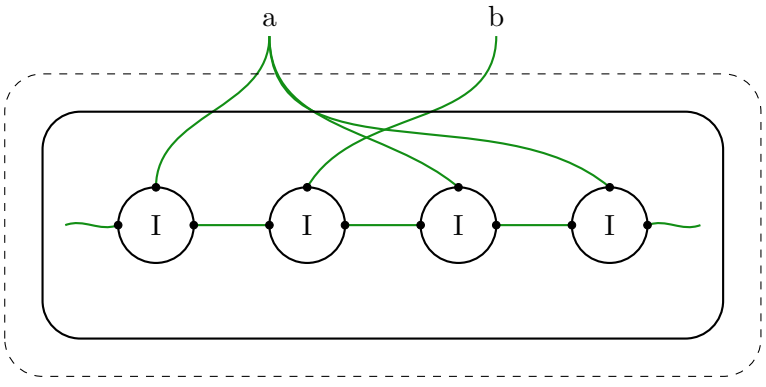


Figura 5.3: Bigrafo per la stringa "abaa"

Si prenda in considerazione la figura 5.3: ogni nodo di tipo *Input* ha tre porte, numerate da sinistra verso destra con 0, 1 e 2. La porta numero 0 è collegata al

carattere precedente: per esempio, il primo carattere a non è collegato a nessun altro nodo. La porta numero 2 è collegata al carattere successivo. Infine la porta numero 1 è collegata alla lettera (che è un outername) che il carattere simboleggia. Così facendo si crea una *lista* di caratteri che forma la vera e propria stringa. Il nodo di tipo *String* raggruppa tutti questi caratteri al suo interno. Il bigrafo di figura 5.3 rappresenta dunque la stringa "abaa". Nell'implementazione, si è costruito un modulo che accetta dall'utente una stringa e la trasforma nel bigrafo equivalente, secondo le regole appena citate.

Ora che si è definito come modellare un NFA e una sua stringa, rappresentiamo una istanza del problema. In particolare, costruiamo il bigrafo che ha come NFA l'automa di figura 5.2 e come stringa di input il bigrafo di figura 5.3. Il risultato è il bigrafo di figura 5.4.

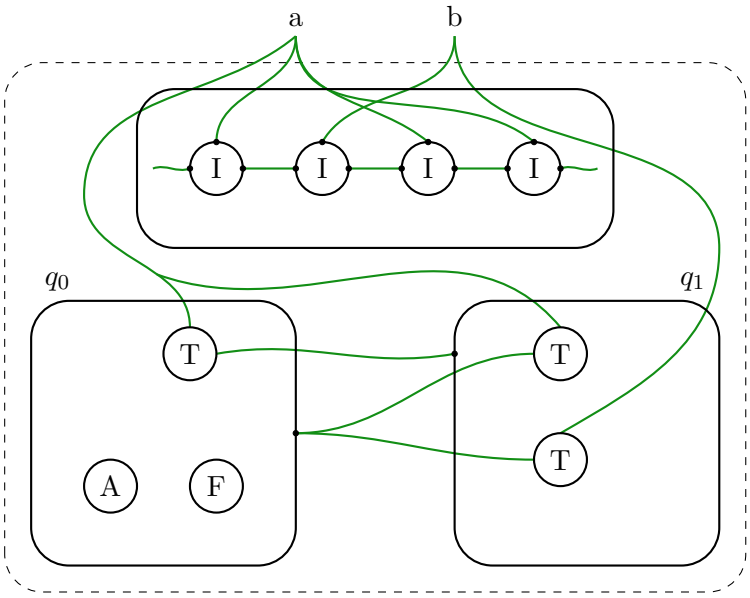


Figura 5.4: Istanza del problema: la stringa "abaa" viene accettata dall'automa A?

Vediamo ora come il bigrafo si può evolvere. L'idea principale è di far "consumare" al sistema un carattere alla volta. Si veda la regola della figura 5.5 che rappresenta la regola di reazione R_0: se il primo carattere della stringa è la lettera σ e se lo stato attivo possiede una transizione tramite la lettera σ, allora si elimina tale carattere dal bigrafo e si sposta il nodo *Active* nello stato destinazione.

La regola R_0 è molto intuitiva: se per esempio prendiamo in considerazione il bigrafo di figura 5.4, allora applicando la regola si ha che il primo carattere (a) viene eliminato ed il nodo A passa dentro lo stato q_1. Quindi il BRS per questo problema

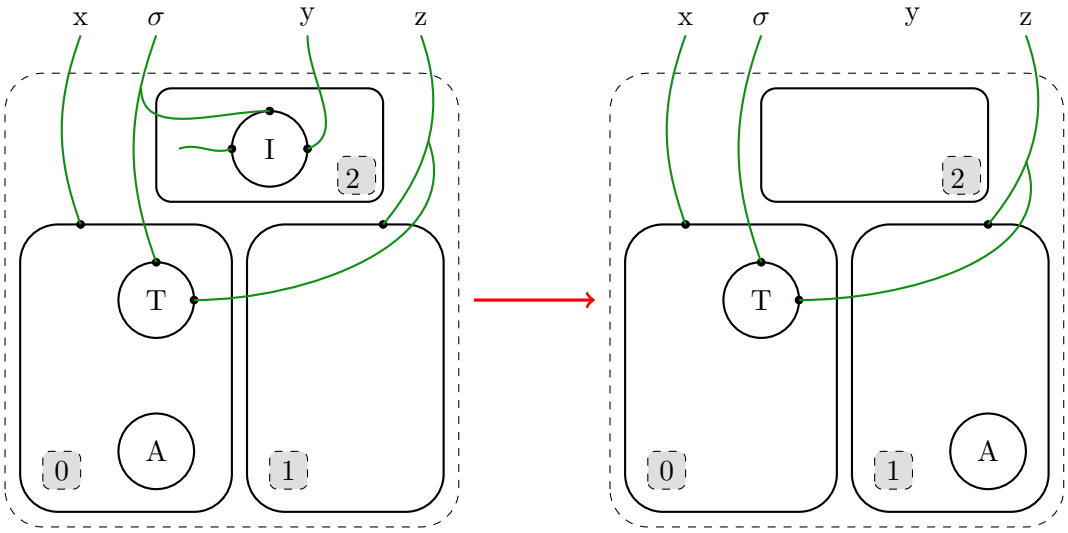


Figura 5.5: Regola di reazione R_0

è definito da una sola regola (R_0), che itera finchè trova un match nel bigrafo, cioè si ferma solo quando il nodo di tipo *String* **non** contiene più nessun nodo *Input*.

Ora che abbiamo creato il BRS, è possibile usare il model checker MC_{big} definendo la formula che esso andrà a verificare. Nella teoria degli automi, vale la seguente preposizione:

Proposizione 5.1.3. *Una stringa x viene accettata se e solo se alla fine di essa l'automa si trova in uno stato finale.*

Nella nostra segnatura, tutti gli stati finali ($q_F \in F$) vengono distinti tramite un nodo F all'interno di essi: essendo passivi, nessuna regola di reazione può modificarli. Per cui la proprietà da verificare sarà la seguente: "Il nodo di tipo *String* non deve contenere nessun altro nodo e il nodo A e il nodo F si devono trovare dentro lo stesso stato S". Nella logica di MC_{big}, questo si traduce nella formula:

$$\varphi = \mathbb{W}_B(T, T, T)$$

dove B è il bigrafo di figura 5.6.

Si noti la semplicità di quest'ultima formula: il nodo di tipo *String* deve essere vuoto, il che significa che tutti i caratteri sono stati "consumati" dall'automa. Inoltre, il nodo attivo e il nodo finale si devono trovare nello stesso stato, assicurando che l'automa dopo aver letto tutta la stringa è finito in uno stato finale. Quindi possiamo affermare che:

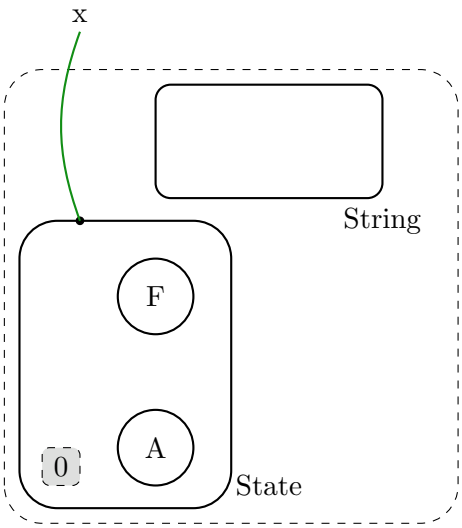


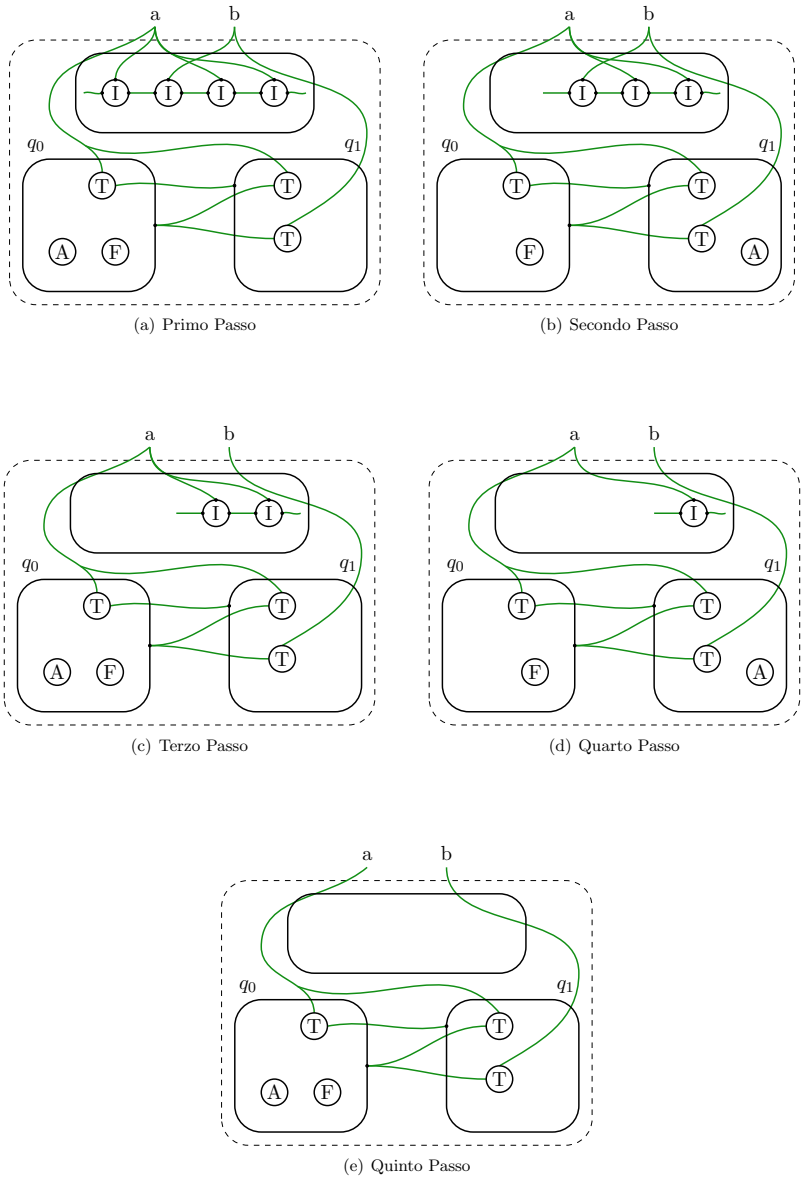
Figura 5.6: Bigrafo B

Proposizione 5.1.4. *Una stringa x viene accettata dall'automa A se e solo se* $MC, S_i \models \varphi$ *per qualche i, dove* $\varphi = \mathcal{W}_B(T, T, T)$.

Infine, si noti come l'automa modellato tramite bigrafi sia *non deterministico*: le transizioni sono modellate tramite regole di reazione che scattano dopo aver trovato un match. Il processo di matching però è per natura non deterministico, il che vuol dire che se nel bigrafo di partenza la regola R_0 trova più di un match, allora sceglie uno dei due in maniera non deterministica. Si prenda il bigrafo di figura 5.4: se si aggiunge nel nodo q_0 un'altra transizione collegata all'outername 'a' che porta ad un terzo stato q_2, allora l'automa diventa non deterministico, perché la regola R_0 troverà sempre due match nel bigrafo e ne sceglierà uno in maniera casuale. Riassumendo: nella versione bigrafica, gli automi non deterministici si distinguono da quelli deterministici *solamente* perché nei primi esiste almeno un nodo che ha due o più archi uscenti con la stessa etichetta.

Prendiamo l'istanza del problema in figura 5.4. Seguiremo una traccia d'esecuzione, cioè faremo tutti i passi che fa MC_{big} per verificare la proprietà desiderata. Vedremo come il grafo degli stati sarà molto semplice: questo spesso è un indice di buona progettazione. Vuol dire che il sistema è stato modellato correttamente, scegliendo poche regole ed evitando di costruire regole ad-hoc per casi particolari. Il nostro BRS è formato da una sola regola, il che evita per esempio che in uno stato S_i venga applicata la regola sbagliata creando nodi inutili nel grafo degli stati.

In figura 5.7, c'è la traccia d'esecuzione di MC_{big}, mentre in 5.8 c'è il corrispondente grafo degli stati.



(a) Primo Passo

(b) Secondo Passo

(c) Terzo Passo

(d) Quarto Passo

(e) Quinto Passo

Figura 5.7: Traccia d'esecuzione di MC_{big}

Si noti come l'ultimo stato S_4 soddisfi la formula φ: il Wario Predicate $\mathbb{W}_B(T,T,T)$ è soddisfatto dallo stato S_4, in formule $MC, S_4 \models \mathbb{W}_B(T,T,T)$, perché il model checker trova in S_4 un match del bigrafo B. Poichè tutti gli argomenti del Wario Predicate sono True, basta che questo match esista perché il predicato sia soddisfatto. Quindi, la proposizione 5.1.4 è rispettata.

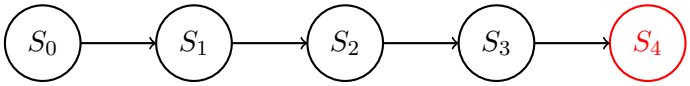


Figura 5.8: Grafo degli stati

5.1.1 Implementazione

Si è costruito un modulo che accetta dall'utente una stringa e costruisce il rispettivo bigrafo. Se vogliamo sapere se una data stringa è riconosciuta dal *NFA*, basta innanzitutto estendere la classe *NFA* costruendo il proprio automa, e in seguito inserire una stringa appena il sistema lo chiede. Ora la stringa verrà processata e trasformata in bigrafo. Per esempio, se vogliamo risolvere il problema di figura 5.4, allora dobbiamo estendere la classe NFA costruendo il bigrafo di figura 5.2 ed infine inserire la stringa "abaa":

```
Insert the string:
abaa
Does this NFA recognize the string "abaa"? YES
```

Appena inseriamo la stringa, il software crea il bigrafo rappresentante l'NFA che abbiamo precedentemente costruito, il quale è affiancato ad un altro bigrafo che contiene la stringa, proprio come in figura 5.4. In questo caso, l'automa riconosce correttamente la parola. Se invece scegliamo una stringa che non appartiene al suo linguaggio, allora il sistema non la riconosce:

```
Insert the string:
aaba
Does this NFA recognize the string "aaba"? NO
```

Questo esempio è molto importante perché mostra due aspetti fondamentali:

- il primo è la flessibilità dei bigrafi: tramite una sola regola di reazione si è riusciti a modellare un NFA. Inoltre, si presti attenzione al grafo degli stati: è molto semplice e lineare. Come già scritto, questo è un indizio di buona progettazione, perché significa che abbiamo creato una sola regola per tutti i casi possibili. Se avessimo creato una regola per casi particolari, allora ci sarebbero state diramazioni del grafo che avrebbero portato a vicoli ciechi, cioè rami in cui la foglia non rispetta la proprietà φ.

- il secondo è la generalità della logica per MC_{big}: si è espressa una proprietà di uno specifico dominio usando la logica generale creata per il model checker. In

questo esempio si è usato il Wario Predicate, che permette di spostarsi all'interno del bigrafo e di verificare se una sua parte esiste e rispetta determinate proprietà. Nel NFA ci è bastato verificare che esistesse la parte denotata dal bigrafo B.

In questo esempio si è potuto apprezzare la comodità del model checker e della sua logica per verificare una generica proprietà, che rappresentano quindi uno strumento generale adatto per ogni BRS.

5.2 Problema dei filosofi a cena

In questa sezione si fornisce una codifica in bigrafi del problema dei filosofi a cena, introdotto da Dijkstra nel 1965 per esporre un problema di concorrenza tra processi paralleli.

Definizione 5.2.1 (Filosofi a cena)**.** *Cinque filosofi sono seduti a cena ad una tavola rotonda. Ogni filosofo ha davanti il piatto in cui mangiare e due forchette, una a destra e l'altra a sinistra: per cui nel tavolo sono presenti cinque filosofi, cinque piatti e cinque forchette. Ogni filosofo alterna periodi in cui mangia ad altri in cui pensa. Per mangiare, ha bisogno di entrambe le forchette, ma deve prenderle una per volta. Quando ha finito di mangiare, lascia le forchette e continua a pensare.*

Si progetti un algoritmo che eviti deadlock *o* starvation.

Il problema chiede di progettare un algoritmo che eviti queste due situazioni:

- ogni filosofo ha una forchetta e aspetta l'altra dal suo vicino: la situazione si trova in uno stato di stallo (*deadlock*)
- una parte di filosofi riesce a mangiare e pensare ripetute volte, a discapito di un'altra parte che non riesce mai a mangiare perché non ha mai due forchette, morendo di inedia (*starvation*).

Il problema è una metafora, dove i filosofi sono processi paralleli in un calcolatore: il momento in cui devono prendere le forchette corrisponde alla lettura dei dati, che possono per l'appunto essere condivisi tra più processi come le forchette; il momento in cui un filosofo mangia corrisponde al momento in cui un processo consuma i dati che ha appena recuperato. Si vedranno due strategie: la prima causerà una situazione di deadlock, mentre la seconda lo eviterà e sarà una soluzione al problema.

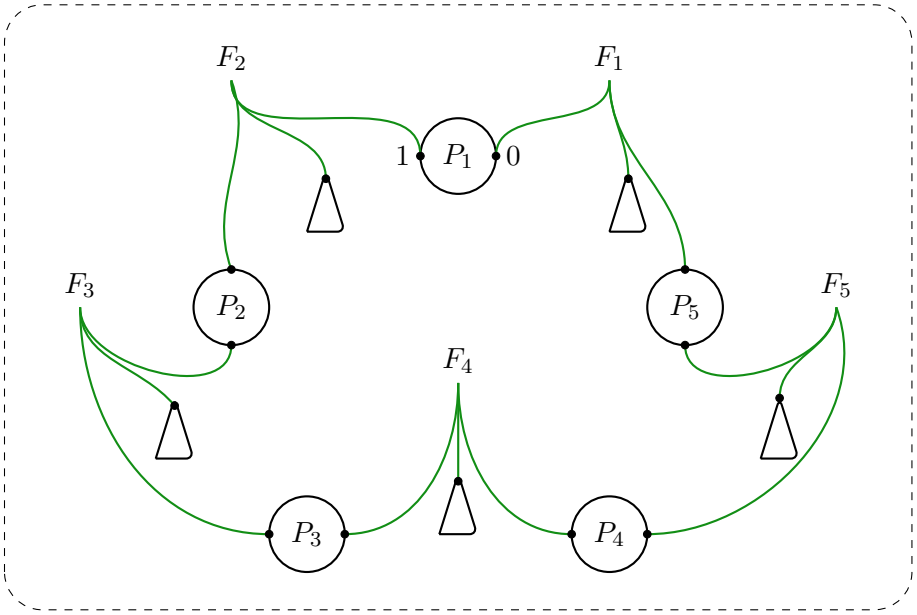


Figura 5.9: Codifica in bigrafi di un'istanza del problema

Per prima cosa, si definirà come il problema può essere tradotto in bigrafi. Si consideri la figura 5.9: ogni filosofo è rappresentato da un nodo circolare con due porte; si immagini che ogni filosofo sia rivolto verso il centro del tavolo: si ha che la porta a sinistra *dal punto di vista del filosofo* rappresenta la mano sinistra e la chiameremo *porta 0*, mentre la porta a destra rappresenta la mano destra (*porta 1*). Per esempio: la mano sinistra le filosofo P_1 è quella collegata a F_1, mentre la mano destra è quella collegata a F_2.

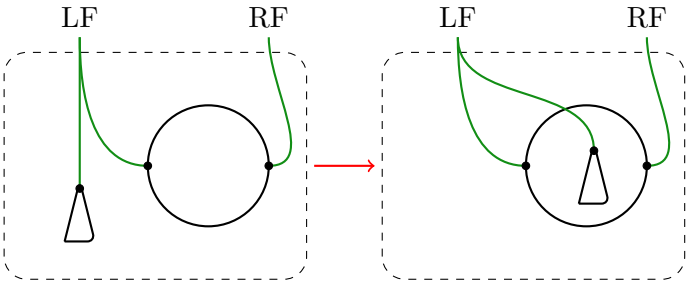
Le forchette sono rappresentate come nodi triangolari e sono identificate tramite un outername, per esempio F_4. Modelliamo il fatto che la forchetta sinistra di un filosofo sia F_i collegando la sua porta 0 all'outername F_i. Per esempio: il filosofo P_4 ha come forchetta sinistra F_4 perché la sua porta 0 è collegata a questo outername, mentre come forchetta destra F_5 perché la sua porta 1 punta a F_5.

Inizialmente tutte le forchette sono posizionate sul tavolo. Modelliamo il fatto che un filosofo P_i abbiamo preso la forchetta F_k spostando quest'ultima all'interno di P_i.

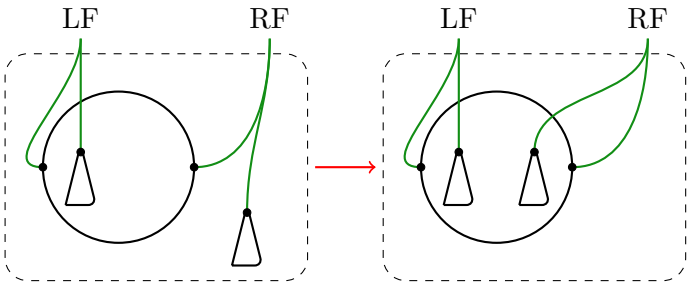
5.2.1 Prima strategia

Tramite la prima strategia, che causerà situazioni di stallo, vogliamo fare vedere come il model checker MC_{big} riesca ad individuare un deadlock. Essa prevede che ogni filosofo, per riuscire a mangiare, debba prendere prima la forchetta sinistra

e poi quella destra, e le rimetta in ordine sul tavolo (prima la sinistra e poi la destra). Questa strategia causa un deadlock perché P_i prende prima la forchetta F_i per $i \in \{1, 2, 3, 4, 5\}$, e quindi ogni filosofo P_i aspetta che P_{i+1} liberi la forchetta causando uno stallo.



(a) Prima Regola



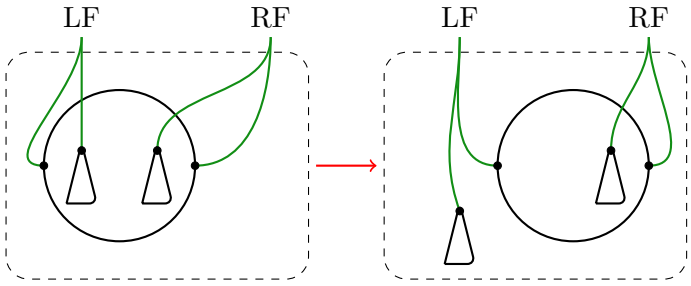
(b) Seconda Regola

Figura 5.10: Regole per prendere le forchette

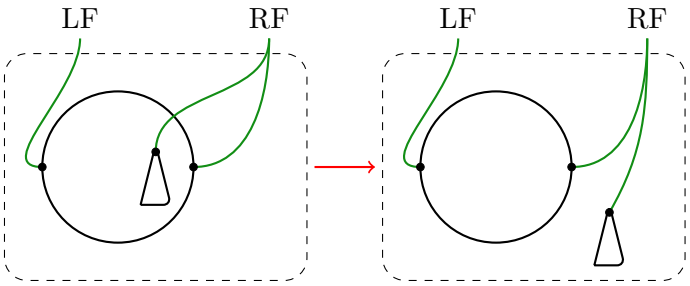
In figura 5.10, ci sono le regole che consentono ad un filosofo di prendere le forchette. Si noti come si debba prendere prima la forchetta sinistra e poi la destra: la seconda regola infatti scatta se e solo se il filosofo possiede già la forchetta sinistra, che a sua volta ha potuto prendere se e solo se non possedeva ancora nessuna forchetta (nella regola 5.10.a non c'è nessun sito dentro il nodo del filosofo).

Le regole per lasciare le forchette sono similari: quando un filosofo ha entrambe le forchette significa che ha mangiato e, grazie alla regola 5.11.a, lascia prima la forchetta a sinistra, come vuole la nostra strategia. Infine, tramite 5.11.b, lascia la forchetta destra.

Queste quattro regole andranno a formare il BRS per il nostro problema. Si noti come un BRS di questo tipo sia particolarmente adatto per il problema della cena tra filosofi: quest'ultimo è infatti un problema di sincronizzazione tra processi paralleli che viene modellato perfettamente dalle regole non deterministiche del BRS. Ogni filosofo decide autonomamente quando incominciare a mangiare: per cui può essere che incominci il terzo filosofo così come il primo. Questa è la stessa situazione in cui



(a) Prima Regola



(b) Seconda Regola

Figura 5.11: Regole per lasciare le forchette

operano le regole: per esempio, in figura 5.9 la regola 5.10.a trova cinque match e ne sceglie in modo non deterministico uno solo. In altre parole, il BRS modella bene il caso reale in cui i processi decidono di leggere dati autonomamente, senza che ci sia alcun ordine tra di loro.

Ora si userà MC_{big} per capire se con questa strategia c'è pericolo di deadlock. In questo caso, la proprietà da fare verificare al model checker è una disgiunzione tra due *IsoProperty*:

$$\varphi = \pi_G \vee \pi_{G'}$$

dove G è il bigrafo di figura 5.12 e G' quello di figura 5.13. Nel primo, ogni filosofo P_i contiene solamente la forchetta F_i, mentre nel secondo ogni filosofo P_i contiene solamente F_{i+1}.

Dato che la grammatica di MC_{big} non contiene il simbolo $\vee$ per la disgiunzione, si è usata l'equivalenza di De Morgan, avendo quindi che:

$$\varphi = \pi_G \vee \pi_{G'} = \neg(\neg\pi_G \wedge \neg\pi_{G'})$$

Infatti, nell'implementazione le righe corrispondenti alla creazione di questa proprietà sono queste riportate di seguito:

```
Predicate p1 = new IsoPredicate(getAim1(n));
Predicate p2 = new IsoPredicate(getAim2(n));
Predicate notP1 = new NotPredicate(p1);
Predicate notP2 = new NotPredicate(p2);
Predicate andNP1NP2 = new AndPredicate(notP1,notP2);
Predicate prop = new NotPredicate(andNP1NP2);
```

da cui si vede bene come si sia usata la legge di De Morgan.

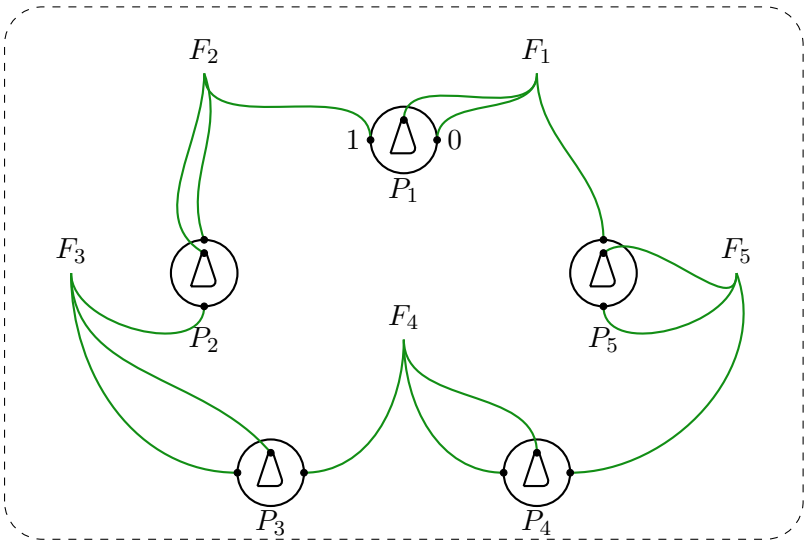


Figura 5.12: Bigrafo G: prima proprietà di stallo

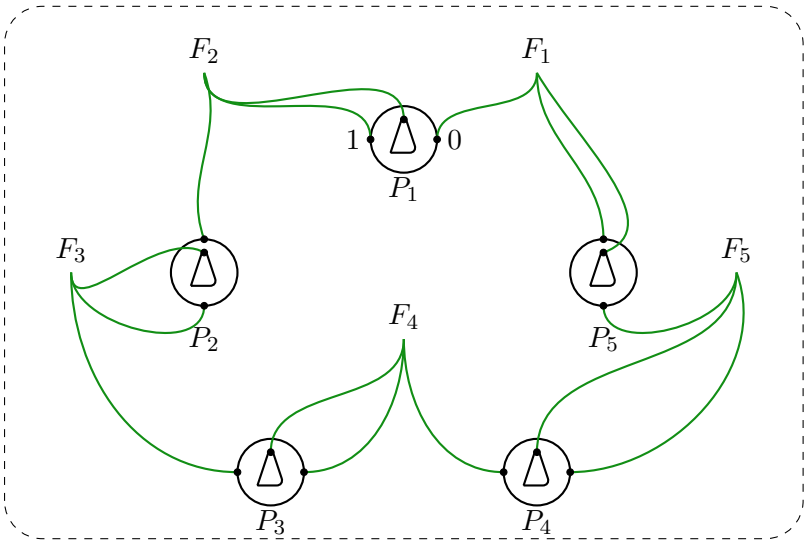


Figura 5.13: Bigrafo G': seconda proprietà di stallo

MC_{big} incomincia a computare l'intero grafo degli stati: esso è **teoricamente**

infinito perché le quattro regole viste prima possono essere applicate un numero arbitrario di volte; per esempio, posso applicarle una dopo l'altra sempre sul filosofo P_1, facendolo mangiare e pensare all'infinito. Nella prossima sottosezione, vedremo invece che nella pratica il grafo degli stati è finito, pur rappresentando processi infiniti.

Con la strategia di questa sottosezione è impossibile trovarsi nella situazione di figura 5.13. Quando MC_{big} trova la situazione di stallo di figura 5.12, il model checker smette di generare il grafo degli stati e ritorna False, perché si è causato un deadlock:

Proposizione 5.2.1. *MC_{big} ritorna False se e solo se esiste almeno una situazione di deadlock.*

Per convincersi che il bigrafo B rappresenti una situazione di deadlock, si noti come su di esso non si possa applicare più nessuna delle quattro regole: poichè la proprietà è una *IsoProperty*, MC_{big} cerca un nodo che sia uguale a B, cioè a cui non si possano applicare più regole. Un tale nodo del grafo degli stati si chiama **stato finale**.

Si noti come ad ogni passo ogni regola trovi molti match: il grafo degli stati sarà molto grande. Si consideri la figura 5.9: la regola 5.10.a può scattare su ognuno dei 5 filosofi; per esempio, ipotizziamo che scatti sul primo filosofo P_1. Ora, le possibilità sono molteplici:

- P_1 può prendere anche la forchetta destra: quindi scatta la regola 5.10.b
- uno qualsiasi degli altri quattro filosofi può prendere la propria forchetta sinistra: quindi statta la regola 5.10.a

per un totale di 6 stati possibili. Si vede bene come l'**esplosione combinatoria** faccia aumentare in modo esponenziale il numero di nodi del grafo degli stati.

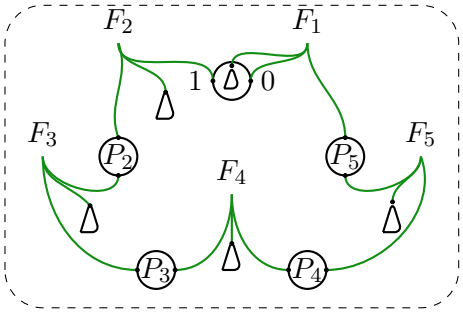
```
Insert the number of philosophers:
5


STRATEGY: Every philosopher takes first the left fork.
Are deadlocks avoided? NO
Number of states of the Model Checker: 93
Resolution Time: 4.786207234 seconds
```

Quello mostrato qui sopra è il risultato dell'esecuzione del software: si è implementato un modulo che costruisce il bigrafo con un numero di filosofi scelto dall'utente. In questo caso si è scelto di avere cinque filosofi come in figura 5.9. In seguito, il sistema informa quale strategia si sta adottando: per ora ci interessa solamente la prima strategia. Nella riga seguente il software ritorna il risultato (cioè se c'è o non c'è un deadlock), ed infine il numero di nodi del grafo degli stati.

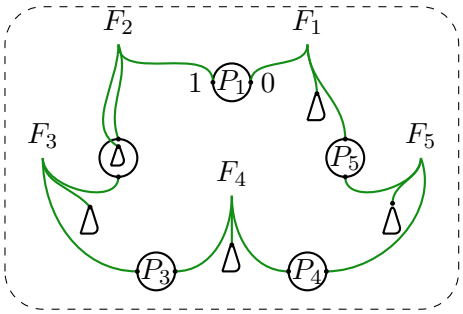
Per verificare che la strategia in uso causa deadlock, con solamente cinque filosofi il model checker ha creato un grafo degli stati con 93 nodi. Questo è il risultato dell'esplosione combinatoria, che causa appunto un aumento esponenziale del numero di nodi che il model checker deve controllare.

In questo esercizio si è usato l'isomorfismo con proprietà (*Property Isomorphism*), il cui funzionamento, che è poco differente dal normale algoritmo, viene riportato qui sotto:

Proposizione 5.2.2 (Property Isomorphism)**.** *Due nodi costituiscono un match nel Property Isomorphism se e solo se lo fanno nel normale Isomorphism e hanno le stesse proprietà.*



(a) Primo Bigrafo



(b) Secondo Bigrafo

Figura 5.14: Due bigrafi isomorfi

Nella libreria *JLibbig*, c'è la possibilità di allegare ad ogni nodo delle proprietà. Si sono quindi dati degli identificativi per ogni filosofo ed ogni forchetta. In seguito, si sono usate le regole con proprietà introdotte nella sottosezione 4.4 per manterle anche dopo l'esecuzione delle quattro regole viste prima.

La creazione del *Property Isomorphism* è stato indispensabile: senza di esso il model checker avrebbe riconosciuto come uguali i due bigrafi in figura 5.14, causando un comportamento anomalo nella verifica delle proprietà.

Dalla figura 5.14 si capisce immediatamente che senza il *Property Isomorphism* il grafo degli stati sarebbe molto più piccolo, perché non si esplorerebbero tutte le possibilità. Se S_0 è lo stato iniziale, allora ipotizziamo che S_1 sia lo stato in cui P_1 prende la sua forchetta sinistra F_1. Ora però, da S_0 si può anche applicare la regola sul filosofo P_2, che prenderà la forchetta F_2 dando vita allo stato S_2. Senza l'uso del *Property Isomorphism*, il model checker troverà che $S_1 = S_2$, memorizzando solamente il primo. Quindi il model checker non distingue tra le due situazioni, creando un grafo degli stati **incompleto**.

5.2.2 Seconda strategia

Vediamo ora un modo per evitare i deadlock: si enumerino tutte le forchette presenti sul tavolo (nel nostro esempio consideriamo il pedice di F_i); per mangiare, ogni filosofo deve prendere per prima la forchetta con indice minore. Una traccia d'esecuzione basata sulla figura 5.9 potrebbe essere questa: tutti i filosofi P_i ($i \in \{1, 2, 3, 4\}$) prenderanno la forchetta F_i tranne l'ultimo (P_5), che dovrà prendere F_1, che ha indice minore; questa forchetta però è già occupata da P_1 e quindi P_5 dovrà aspettare, lasciando libera F_5 che potrà essere utilizzata da F_4. Quest'ultimo riuscirà a mangiare e dopo un po' lascerà libera la forchetta F_4, che verrà presa da P_3, e così via. Infine, quando si libererà F_1, anche il filosofo P_5 riuscirà a mangiare. Con questa soluzione, si evitano quindi sia i deadlock sia le situazioni di starvation.

Le regole sono uguali a quelle della strategia precedente con un'eccezione: si noti come l'ultimo filosofo debba prendere la prima forchetta (quella con indice minore) con la mano destra, mentre la seconda con la sinistra. Quindi si creano altre quattro regole ad-hoc per l'ultimo filosofo: deve prendere per prima la forchetta destra e per seconda quella sinistra, a differenza di tutti gli altri. Lo stesso vale per posare le forchette, per un totale di quattro nuove regole. La nuova strategia è quindi identica alla prima, modulo l'aggiunta di queste quattro regole. Il nodo finale viene identificato tramite un controllo L (Last) al suo interno.

Il BRS per questa strategia ha otto perciò regole. La proprietà da verificare resta la stessa: $\varphi = \pi_B$. Si vedrà però che ora MC_{big} non riuscirà ad identificare nessuna situazione di deadlock.

Prima però si deve precisare un'aspetto importante: si sarebbe tentati di pensare che in assenza di deadlock il model checker continui all'infinito a generare stati. Infatti, se al bigrafo di partenza, che chiameremo B_1, applichiamo le quattro regole sempre al filosofo P_1, allora quest'ultimo (in ordine):

- prende la forchetta sinistra
- prende la forchetta destra
- posa la forchetta sinistra
- posa la forchetta destra

Dopo la quarta regola, si ritorna ad uno stato uguale al primo, che chiameremo B_2, e nulla vieta che queste quattro regole continuino ad eseguire all'infinito. Tramite il *Property Isomorphism* si riesce ad evitare questa situazione: questo algoritmo riesce infatti a verificare che $B1$ e B_2 sono uguali, evitando quindi di tornare ad eseguire tutte le regole. Grazie ad esso, si è riusciti ad avere **un grafo degli stati finito per un processo infinito**.

Riprendendo quanto detto nella sottosezione precedente sull'*Property Isomorphism*, si ha che esso porta i seguenti benefici:

- il model checker computa un grafo degli stati **completo**, senza dimenticare nessuna possibile situazione
- il model checker verifica le proprietà di un processo infinito su un grafo degli stati **finito**.

MC_{big} riesce quindi a computare l'intero grafo degli stati e, dato che per ogni suo nodo S_i si ha che $MC, S_i \not\models \varphi$, ritorna True, a significare che la tecnica è priva di situazioni di deadlock o starvation.

Proposizione 5.2.3. *MC_{big} ritorna True se e solo se ogni stato non rappresenta una situazione di deadlock.*

```
Insert the number of philosophers:
5
```

```
STRATEGY: all the forks are enumerated. Every philosopher takes first the
    fork with the lower index.
Are deadlocks avoided? YES
Number of states of the Model Checker: 189
Resolution Time: 14.129189577 seconds
```

Il risultato ora è quello della seconda strategia. Si noti come l'assenza di deadlock causi un aumento del numero di stati, che ora ha raggiunto il valore 189. Aumentando il numero di filosofi ci si rende conto dell'andamento esponenziale dell'esplosione. Infine si prendano in considerazione i tempi di risoluzione del problema: MC_{big} impiega 20 minuti per capire che questa strategia è priva di deadlock.

```
Insert the number of philosophers:
7


STRATEGY: Every philosopher takes first the left fork.
Are deadlocks avoided? NO
Number of states of the Model Checker: 801
Resolution Time: 214.26274781 seconds


STRATEGY: all the forks are enumerated. Every philosopher takes first the
    fork with the lower index.
Are deadlocks avoided? YES
Number of states of the Model Checker: 1701
Resolution Time: 1210.612988337 seconds
```

5.3 Politiche di sicurezza

In quest'ultimo esempio vedremo come tramite i bigrafi ed il model checker MC_{big} si possano testare delle politiche di sicurezza. In particolare, faremo riferimento alla seguente situazione: un edificio appartenente ad un'azienda contiene varie stanze, ognuna con un computer al suo interno. L'azienda ha importanti file segreti (token) da mantenere al sicuro dentro ogni computer. Ha anche vari dipendenti, che possono entrare liberamente in ogni stanza e collegarsi ai vari computer. Uno di questi dipendenti si chiama Alice. Lei ed il suo complice Bob vogliono rubare un segreto dell'azienda, seguendo questo piano:

- Alice, che essendo una dipendente può collegarsi ad un computer, ruba il file segreto e lo trasferisce sul proprio smartphone.
- Alice esce dalla stanza, pur rimanendo all'interno dell'azienda
- Alice tramite il suo smartphone stabilisce un collegamento con quello di Bob, che si trova al di fuori dell'azienda, e trasferisce il file segreto, che ora è di dominio pubblico

Il nostro compito è quello di stabilire una politica di sicurezza che eviti a qualsiasi token dell'azienda di diventare pubblico, in questo caso di diventare in possesso di Bob. Si vedranno quindi due politiche: la prima consentirà a Bob di ottenere il token, mentre la seconda lo eviterà assicurando la sicurezza dell'azienda.

Tutte le possibili situazioni in cui Alice e Bob possono agire per rubare i file segreti verranno calcolate dal model checker MC_{big}, che alla fine dell'esecuzione dirà se la politica scelta sarà sicura o meno. Infine, si è costruito un modulo che offre all'utente la possibilità di visitare il grafo degli stati, consentendogli di ripercorrere tutte le azioni che hanno consentito a Alice e Bob di ottenere il file segreto.

5.3.1 Segnatura

Incominciamo con il definire la segnatura del bigrafo che modella il problema. Si consideri la figura 5.15: il nodo più grande è "*Building*" e rappresenta l'edificio dell'azienda. In questo esempio, sono presenti due stanze, ognuna con un computer all'interno. Ogni computer contiene un "*Token*", che è uno dei file segreti dell'azienda.

I nodi per Alice e Bob sono modellati tramite due circonferenze, che però si trovano sotto radici diverse: Alice è dentro l'azienda, in quanto è una dipendente e può entrare a tutti gli effetti sia dentro il "*Building*" sia dentro le "*Room*". Invece Bob si trova al di fuori dell'azienda, perché non è un dipendente e quindi il suo accesso è vietato. Ai fini della verifica da parte del model checker, tutti i dipendenti verranno considerati con cattive intenzioni, ovvero: ogni dipendente sarà modellato da un nodo di controllo "*Alice*". Allo stesso modo, ogni persona al di fuori dell'azienda verrà modellata con un nodo di controllo "*Bob*". Sia Alice che Bob possiedono uno smartphone, che useranno per trasferire il "*Token*".

Ogni smartphone ed ogni computer possiedono una porta per consentire il collegamento: uno smartphone si può collegare ad un altro smartphone oppure ad un

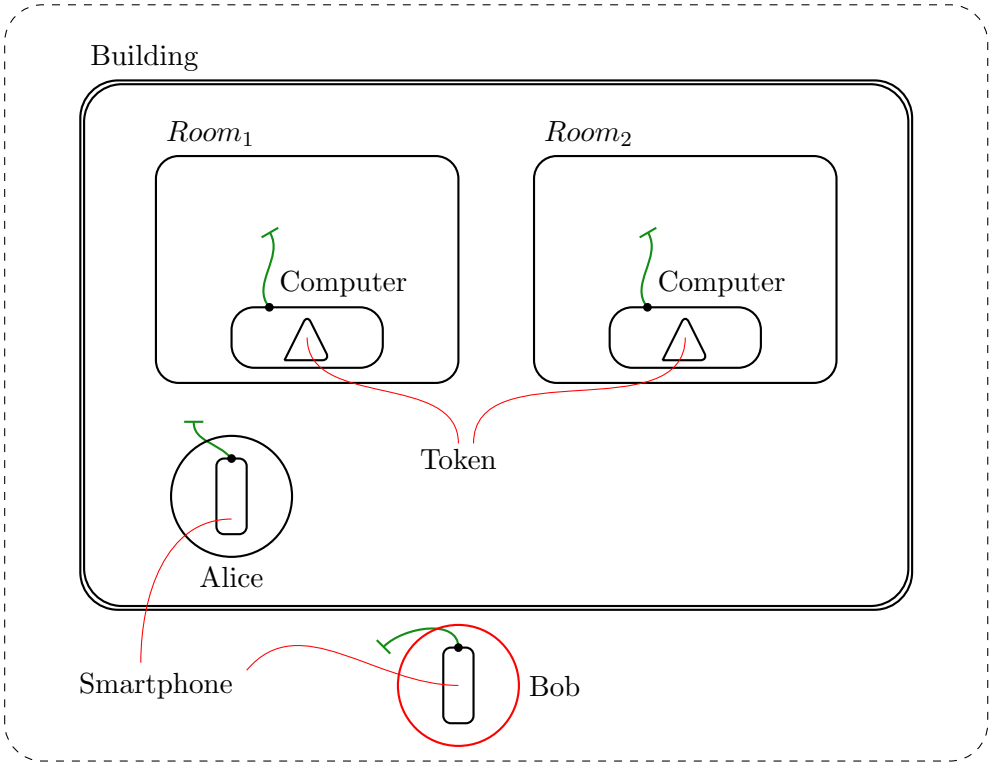


Figura 5.15: Bigrafo per il problema della sicurezza dell'azienda

computer. Stabilito il collegamento, è possibile trasferire il Token. Per distinguere il "*Building*" dalle "*Room*", si è disegnato il primo tramite doppie linee.

5.3.2 Prima politica

Vediamo ora la prima politica, che come già anticipato **non** assicurerà la sicurezza dell'azienda. Nei BRS, una politica si traduce in un insieme di regole di reazione, che descrivono come ogni nodo deve comportarsi. Per esempio, una regola potrebbe coinvolgere una "*Room*", obbligandola a chiedere un badge ad ogni dipendente che vuole entrare.

Le regole R_0 e R_1 modellano rispettivamente l'entrata e l'uscita da una stanza all'interno dell'azienda. Si noti come ogni dipendente, ovvero ogni nodo circolare all'interno dell'azienda, possa entrare liberamente in ogni stanza. La politica di sicurezza è quindi **minima**, perché non vengono fatti controlli e ogni dipendente può accedere al computer: in particolare, può collegarlo al suo smartphone e trasferire il Token. Se togliessimo il sito numero 1, allora si potrebbe entrare nella stanza solamente senza oggetti pericolosi: in altre parole, ogni dipendente non dovrebbe avere niente con sè per poter entrare in una stanza.

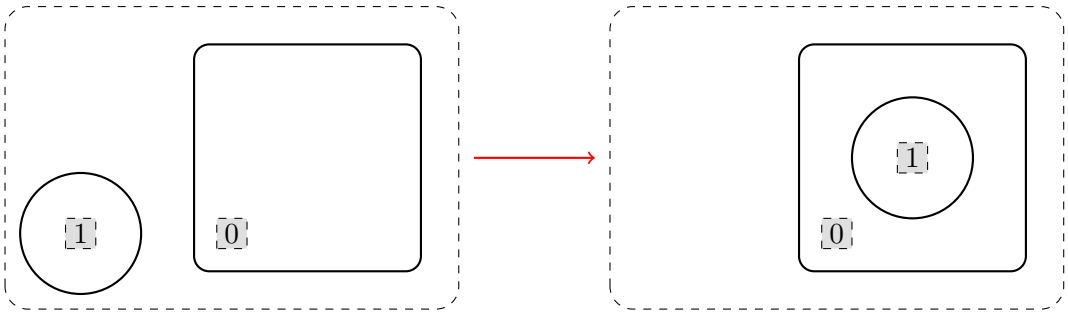


Figura 5.16: regola per l'entrata in una stanza (R_0)

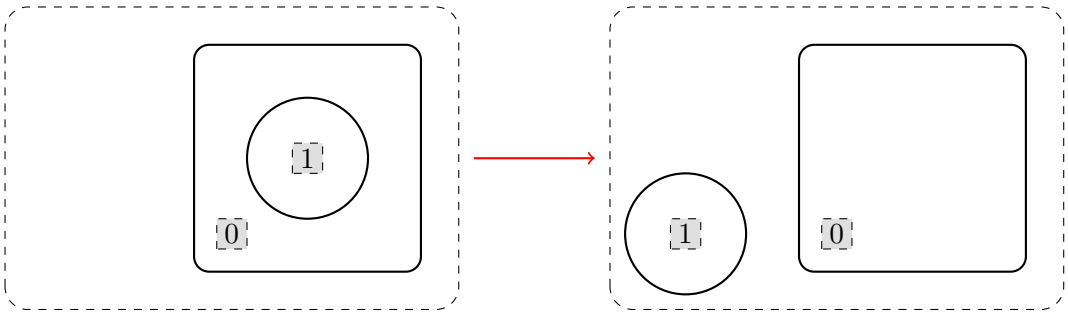


Figura 5.17: regola per l'uscita da una stanza (R_1)

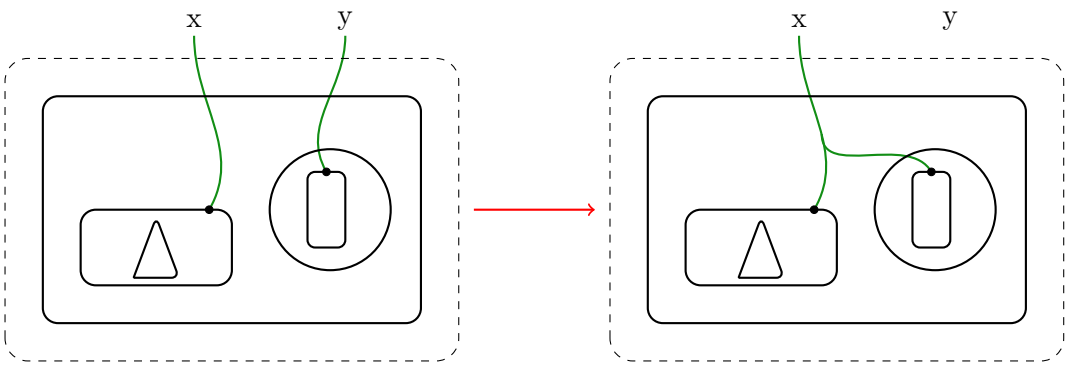


Figura 5.18: regola per il collegamento ad un computer (R_2)

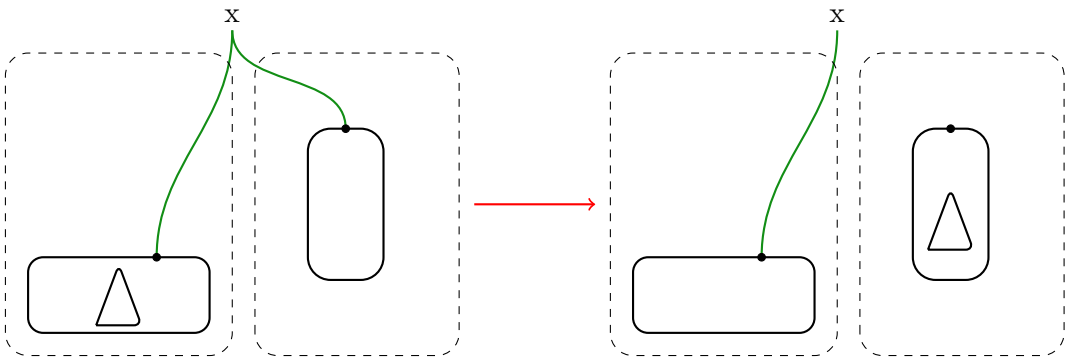


Figura 5.19: regola per il trasferimento di un token da un computer ad uno smartphone (R_3)

Le regola R_2 e R_3 trattano il trasferimento di un Token dal computer ad uno smartphone: la prima stabilisce una connessione mentre la seconda esegue lo spostamento del file segreto. Si noti come lo smartphone del dipendente non debba contenere nessun altro token: se Alice ha già rubato un file segreto, allora il prossimo computer a cui si connette se ne accorgerà e non le darà il file segreto al suo interno.

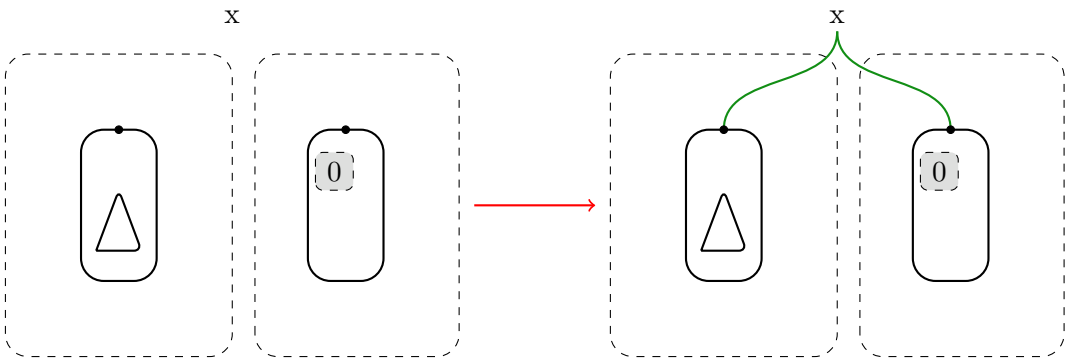


Figura 5.20: regola per iniziare una connessione tra due smartphones (R_4)

Infine, le regole R_4 e R_5 consentono allo smarthphone di Alice di stabilire una connessione con quello di Bob e di trasferire il Token, che ora si trova al di fuori dell'azienda. Alice può chiamare solo se ha ottenuto il Token, come suggerisce il redex della regola R_4. Si noti come, una volta ottenuto il Token, Alice possa connettersi a Bob anche dall'interno dell'azienda.

Vediamo ora l'applicazione delle regole sul bigrafo di figura 5.15, che chiameremo

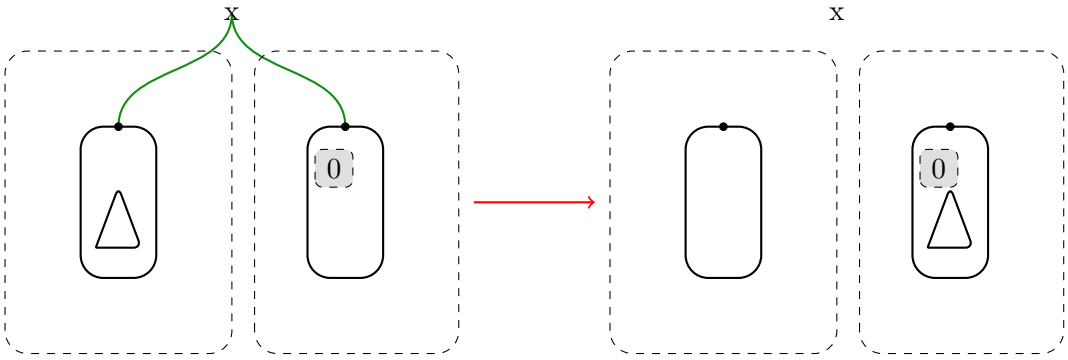


Figura 5.21: regola per trasferire un Token fra due smartphones (R_5)

S_0. Applicando in ordine le regole R_0, R_2 ed R_3 al bigrafo S_0 si ottiene il bigrafo di figura 5.22: Alice è entrata nella prima stanza, ha collegato il suo smartphone al computer ed ha trasferito un file segreto dell'azienda dal computer allo smartphone. Dato che abbiamo applicato tre regole, chiameremo questo bigrafo S_3.

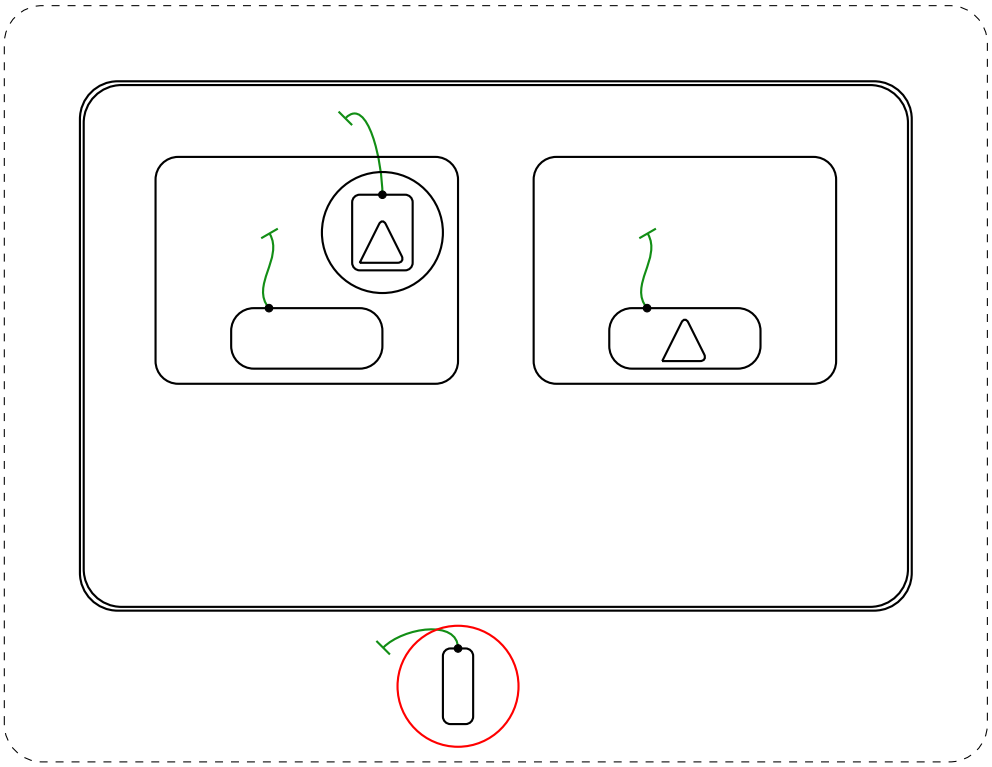


Figura 5.22: Applicazione delle regola R_0,R_2 e R_3

In figura 5.23, è raffigurato il bigrafo S_6, ottenuto tramite l'applicazione delle regole R_1,R_4 e R_5 al bigrafo S_3. In ordine: Alice è uscita dalla stanza con il Token, ha stabilito una connessione con Bob ed ha trasferito il file segreto. Si noti come non

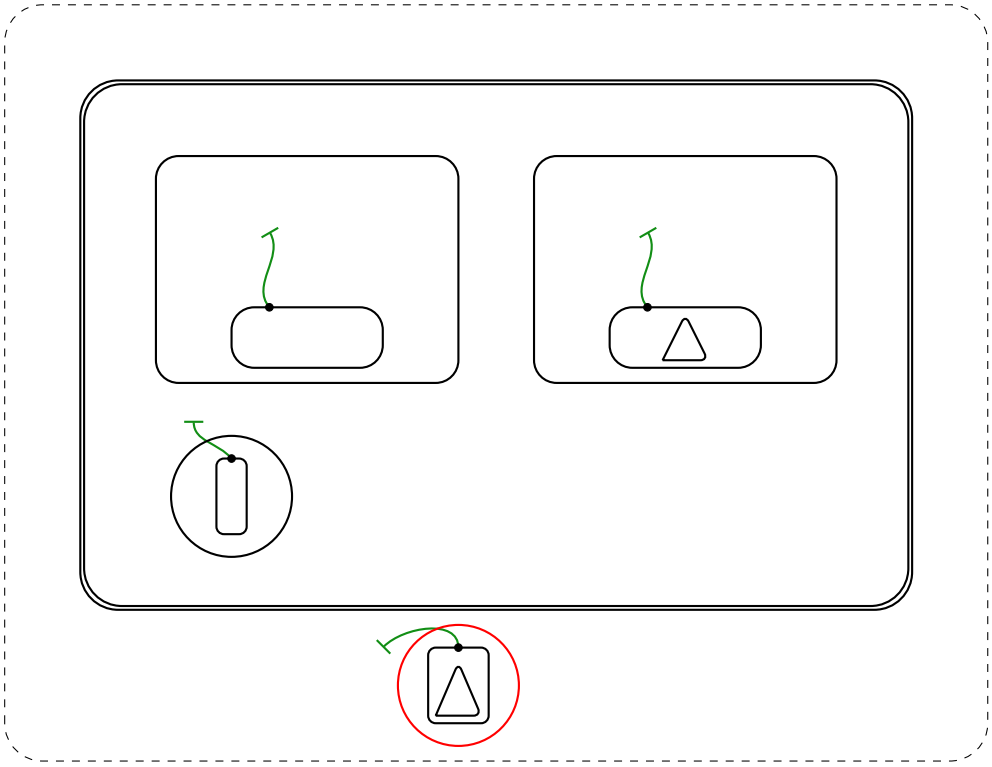


Figura 5.23: Applicazione delle regole R_1,R_4 e R_5

fosse necessario che Alice uscisse dalla stanza per chiamare Bob. Questa politica di sicurezza è quindi fallace, perché consente a Bob o a qualsiasi persona esterna di ottenere un file segreto dell'azienda.

In molti casi l'azienda non ha un edificio semplice come quello di figura 5.15, e quindi c'è bisogno di uno strumento automatico, che affronti il problema anche quando nell'edificio sono presenti numerose stanze, computers e soprattutto numerosi dipendenti. Per questi motivi si è usato MC_{big} per la verifica: esso computa l'intero grafo degli stati, cioè considera tutte le possibili situazioni in cui l'azienda si può trovare, ed appena trova una situazione di pericolo ritorna False, a significare che la politica adottata dall'azienda non è sicura.

Proposizione 5.3.1. *Il model checker MC_{big} ritorna True se e solo se non c'è* nessuna *situazione di pericolo.*

La proprietà da far verificare al model checker è molto semplice:

$$\varphi = \mathbb{W}_B(T, T, T)$$

dove B è il bigrafo di figura 5.24. Quindi la precedente proposizione, la possiamo tradurre in:

Proposizione 5.3.2. *Il model checker MC_{big} ritorna False $\Leftrightarrow \exists S_i$ t.c. $MC, S_i \models \mathcal{W}_B(T,T,T)$. Equivalentemente, ritorna True $\Leftrightarrow \forall S_i$ $(MC, S_i \not\models \mathcal{W}_B(T,T,T))$.*

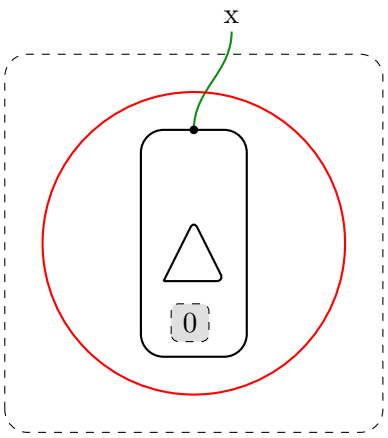


Figura 5.24: Applicazione delle regole R_1,R_4 e R_5

5.3.3 Seconda strategia

Si vedrà ora una strategia che, tramite l'introduzione di due nuove regole, riesce a garantire la sicurezza dell'azienda. Le regole sono le stesse del caso precedente, eccetto R_0 e R_1, che ora sono state sostituite da $secureR_0$ e $secureR_1$, in figura 5.25 e 5.26.

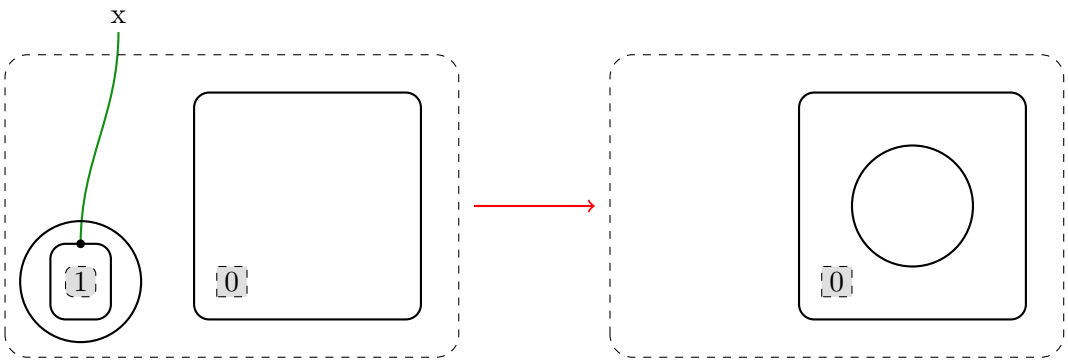


Figura 5.25: regola per l'entrata sicura in una stanza ($secureR_0$)

Ora Alice non può più trasferire il Token sul suo smartphone, e di conseguenza le uniche due regole applicabili sono $secureR_0$ e $secureR_1$. La proprietà che MC_{big} dovrà verificare sarà sempre $\varphi = \mathcal{W}_B(T,T,T)$: ora nessuno stato S_i del grafo sarà

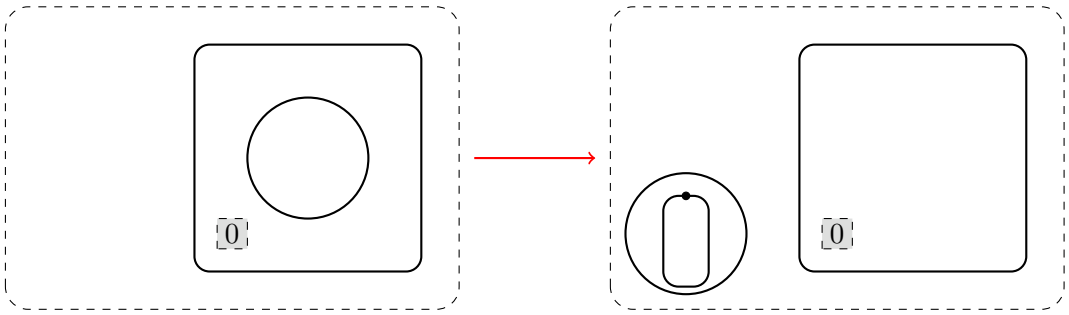


Figura 5.26: regola per l'uscita sicura da una stanza ($secureR_1$)

tale che $MC, S_i \models \varphi$, per cui, data la proposizione 5.3.2, il model checker ritornerà True. Questa politica rende sicura l'azienda.

5.3.4 Implementazione

Si sono costruite delle classi flessibili, in modo che l'utente possa definire il proprio edificio (per quanto grande esso sia) senza ridefinire le regole di reazione della politica di sicurezza. Per esempio, in [1] sono presenti le classi astratte *Building* e *SecureBuilding*, che modellano rispettivamente la prima e la seconda strategia. L'edificio rappresentato nelle figure precedenti è stato costruito nelle due classi *MyBuilding* e *MySecureBuilding*, che estendono le prime due, in modo da poter simulare entrambe.

Se modelliamo il problema di figura 5.15 e lanciamo l'esecuzione, allora il sistema avrà il seguente output in cui informa che la prima politica non è sicura mentre la seconda lo è:

```
1. First building: the policy does not ensure the safety
Are the tokens inside the building safe? NO


2. Second building: the policy ensures the safety
Are the tokens inside the building safe? YES
```

Un'importante caratteristica di questo esempio, è il modulo che consente di poter navigare nel grafo degli stati: per esempio, se il titolare dell'azienda vuole sapere quali sono stati i singoli passi che hanno eseguito Alice e Bob per rubare il file segreto, allora può navigare nel grafo scegliendo di volta in volta che regola di reazione applicare.

```
Do you want to visit the first graph? (yes,no)
yes
40 - 0
```

```
----- Printing Bigraph Root -----
edge E_412{ N_411:Phone; }
edge E_418{ N_417:Phone; }
edge E_414{ N_413:Computer; }
edge E_416{ N_415:Computer; }
root 0 {}
   node N_40D:Bob {}
      node N_411:Phone { port0: Edge E_412;}
   node N_40C:Building {}
      node N_40F:Room {}
         node N_415:Computer { port0: Edge E_416;}
            node N_41A:Token {}
      node N_40E:Room {}
         node N_413:Computer { port0: Edge E_414;}
            node N_419:Token {}
      node N_410:Alice {}
         node N_417:Phone { port0: Edge E_418;}
----- Done Printing Root ------

Choose a branch:
0- enter_room
1- enter_room
```

Nell'output di cui sopra, il sistema chiede se si vuole visitare il grafo. Nel caso di risposta affermativa, stampa il primo nodo (S_0) del grafo degli stati da cui è partita l'esecuzione e ci chiede quale regola vogliamo applicare. In questo caso, Alice può entrare nella prima o nella seconda stanza: se scriviamo 0 allora Alice, entrerà nella prima stanza e l'output sarà:

```
----- Printing Bigraph enter_room -----
edge E_4CB{ N_4CA:Phone; }
edge E_4E3{ N_4E2:Computer; }
edge E_4D4{ N_4D3:Computer; }
edge E_4E5{ N_4E4:Phone; }
root 0 {}
   node N_4C7:Building {}
      node N_4CE:Room {}
         node N_4D3:Computer { port0: Edge E_4D4;}
            node N_4D9:Token {}
      node N_4DC:Room {}
         node N_4DD:Alice {}
            node N_4E4:Phone { port0: Edge E_4E5;}
         node N_4E2:Computer { port0: Edge E_4E3;}
            node N_4E6:Token {}
   node N_4C6:Bob {}
      node N_4CA:Phone { port0: Edge E_4CB;}
----- Done Printing enter_room ------

Choose a branch:
0- comp_connect
```

```
1- leave_room
```

Alice è entrata nella stanza, e ora può semplicemente lasciarla (1) o connettersi al computer(0).

In questo modo, si consente all'utente di navigare all'interno del grafo degli stati e di poter vedere gli effetti di ogni regola di reazione. Quindi, il model checker MC_{big} è ora più informativo: non dice solamente se le proprietà sono rispettate o meno, ma fornisce anche una feature per seguire dei percorsi sul grafo.

Quest'ultimo esempio mostra particolarmente bene con i bigrafi ed i BRS siano flessibili per rappresentare qualsiasi dominio e la loro utilità in casi reali come questo. MC_{big} e la sua logica, implementati per questa tesi, sono quindi di aiuto sia per bigrafi che modellano un formalismo (come quello per gli NFA) sia per i bigrafi *domain specific* (come quest'ultimo esempio), rendendo l'isomorfismo tra bigrafi, il model checker e la sua relativa logica degli strumenti generali adatti ad ogni dominio.

Conclusioni

L'obbiettivo principale di questa tesi è la creazione di uno strumento per la verifica di proprietà all'interno di un BRS, che è stato implementato risolvendo due problemi principali: l'isomorfismo tra bigrafi e la creazioni di un model checker con una logica generale.

Il requisito principale su questo strumento di verifica è la sua *generalità*: esso deve permettere di esprimere le proprietà su qualsiasi dominio il BRS rappresenti. Questa tesi ha rispettato il requisito appena citato, grazie soprattutto alla semantica della logica creata, che risulta quindi sufficientemente generale. D'altra parte, potrebbe non risultare molto espressiva: essa è principalmente una logica spaziale, e quindi non tiene conto di fattori temporali che potrebbero aumentare la sua espressibilità. Si è però dato spazio a future logiche che potranno facilmente estendere MC_{big}.

L'isomorfismo tra bigrafi ha permesse invece la costruzione dello scheletro per il model checker: qualsiasi strumento di verifica deve permettere di evitare esecuzioni infinite quando è possibile. L'isomorfismo ha permesso quindi il raggiungimento di questo obbiettivo. Il problema è stato risolto tramite la *programmazione a vincoli*, che ha ridotto notevolmente il tempo di sviluppo dell'algoritmo ma che potrebbe non essere la soluzioni migliore in termini di complessità.

Il contributo principale di questa tesi è stato il model checker MC_{big}: i bigrafi sono considerati un *meta-modello* con cui si possono esprimere vari formalismi, come per esempio gli automi a stati finiti (che sono stati implementati in questa sede nell'esempio 5.1), il π-calcolo, il λ-calcolo e molti altri, come le Reti di Petri. Inoltre è possibile esprimere anche un bigrafo appartenente a un dominio specifico (come l'esempio 5.3). Avere a disposizione uno strumento come MC_{big} permette di verificare proprietà su tutti questi formalismi, sia *general-purpose* (come gli NFA) sia a *domain-specific* (come la rappresentazione di un edificio), risultando quindi **universale e flessibile** come lo sono i bigrafi.

A

Notazione dei vari capitoli

Sono riportate le principali notazioni delle varie sezioni.

A.1 Notazione della sezione 2.2

Il più delle volte si tratterà un intero come l'insieme di tutti i suoi precedenti, così che se si scriverà m in realtà si intende $\{0, \ldots, m-1\}$.
Scriviamo $S\#T$ per dire che i due insiemi S e T sono disgiunti. Scriviamo $S \uplus T$ per descrivere l'unione dei due insiemi S e T dopo che sappiamo o assumiamo che siano disgiunti.
Se f ha dominio S e $S' \subset S$, allora denotiamo con $f \restriction S'$ la restrizione di f su S'.
Per due funzioni f e g con dominio disgiunto S e T, scriviamo $f \uplus g$ per la funzione con dominio $S \uplus T$ tale che $(f \uplus g) \restriction S = f$ e $(f \uplus g) \restriction T = g$.

A.2 Notazione della sezione 2.4

I *posti* di $G : \langle m, X \rangle \to \langle n, Y \rangle$ sono i suoi siti m, i suoi nodi e le sue radici n.
I *punti* di G sono le sue porte e i suoi inner names X.
I *link* di G sono i suoi archi (edge) e i suoi outer names Y.
Gli edges sono *link chiusi* mentre gli outer names sono *link aperti*. Un punto è detto *aperto* se il suo punto è aperto, altrimenti è detto chiuso. G è detto *aperto* se tutti i suoi link sono aperti (ovvero se non ha edges).
Un place senza figli o un link senza punti viene detto *idle*. Due places con lo stesso genitore o due link con lo stesso punto vengono detti *fratelli*.
Se un' interfaccia $I = \langle m, X \rangle$ ha $X = \emptyset$, allora scriveremo I come m; se $m = 0$ o $m = 1$, scriveremo I come X e $\langle X \rangle$ rispettivamente.
Si definisce l'unità ε come $\varepsilon = \langle 0, \emptyset \rangle$.
L'unico bigrafo con supporto vuoto in $\varepsilon \to I$ viene scritto come I. Un bigrafo

$g : \varepsilon \to I$, con dominio ε, viene detto *ground*. Useremo lettere minuscole per i bigrafi ground e li scriveremo come $g : I$.

Spesso si omette '$\cdots \otimes id_I$' in una composizione del tipo $(F \otimes id_I) \circ G$, dove F non possiede un'interfaccia sufficiente per comporre con G. Si scriverà dunque, qualora non presenti ambiguità, $F \circ G$. Dato un linking $\lambda : Y \to Z$, potremmo volerlo applicare ad un bigrafo G con interfaccia esterna $\langle m, X \rangle$ avente meno nomi, i.e. $Y = X \uplus X'$. Scriveremo allora $\lambda \circ G$ per indicare $(id_m \otimes \lambda) \circ (G \otimes X')$, quando m e X' possono essere capiti dal contesto.

Bibliografia

[1] E. Calligaris e L. Geatti. Brs analysis, 2015. https://alichino.dimi.uniud.it/theses/BRSAnalysis.

[2] Troels C. Damgaard e Jean Krivine. A generic language for biological systems based on bigraphs, 2008.

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

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

[5] Eloi Pereira, Pedro Marques da Silva, Clemens Krainer, Christoph M Kirsch, Jose Morgado, e Raja Sengupta. A networked robotic system and its use in an oil spill monitoring exercise.

[6] M. Peressotti. Jlibbig, 2015. http://mads.uniud.it/wordpress/downloads/libbig/.