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

Tesi di Laurea

Simulazione di Sistemi Reattivi Bigrafici

Candidato:
Elia Calligaris

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

A coloro che mi hanno donato la parola,
ed a coloro che mi hanno insegnato ad usarla.

Sommario

I bigrafi sono un meta-modello le cui potenzialità sono ancora oggetto di ricerca. In questa tesi si esporrà una parte di teoria dei bigrafi, per poi analizzarne un'implementazione in Java in grado di simulare sistemi basati su bigrafi. Per tali sistemi si vedrà come generare tutti gli stati possibili e come verificare delle proprietà in modo automatico. Verranno dati alcuni esempi.

Ringraziamenti

Qualcuno disse che scrivere una tesi di laurea è una prova di maturità scientifica; senza ombra di dubbio, raggiungere tale maturità è un percorso lungo e non banale. Pertanto i miei ringraziamenti vanno soprattutto a coloro che mi hanno permesso di arrivare fin qui: i miei genitori, tutti i maestri e professori, gli amici, i compagni di classe e di corso.

Un ringraziamento particolare va a Luca Geatti: il parallelismo dei nostri lavori di tesi lo ha reso il mio collega più stretto in tutti questi mesi di studio, analisi, programmazione e scrittura. Il suo contributo è stato fondamentale.

Indice

Elenco delle figure

Listings

1

Introduzione

2

Teoria dei bigrafi

In questo capitolo si daranno definizioni formali dei bigrafi, delle loro componenti, e delle relative operazioni elementari e derivate; si vedrà come costruire un bigrafo a partire da bigrafi elementari, e si darà la definizione di *BRS* (*Sistema Reattivo Bigrafico*).

La teoria è estratta dall'ampia trattazione di Robert Milner [2], e adattata per le ridotte necessità di questa tesi.

2.1 Bigrafi e le loro componenti

Un *bigrafo* è un tipo particolare di grafo in cui i nodi possono essere annidati l'uno dentro l'altro, e gli archi sono *iper-archi*, cioè collegano un punto di partenza a più punti di arrivo. Pertanto, come la parola *bi-grafo* suggerisce, siamo in presenza di una struttura composta da due grafi indipendenti, dei quali uno descrive la topologia, e l'altro i collegamenti tra nodi. Inoltre si possono definire diversi tipi di nodo con diverso significato: questo rende i bigrafi un *meta-modello*, in modo simile all'XML.

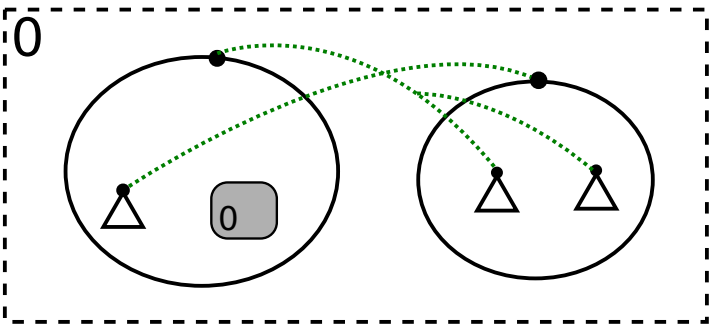


Figura 2.1: Esempio di bigrafo.

2.1.1 Segnature e controlli

Data la generalità dei bigrafi, è innanzitutto necessario definire il significato dei nodi: per far ciò gli si assegnano dei *controlli*.

Definizione 2.1.1 (Controllo)**.** Un *controllo* K è il tipo di un nodo.

Ogni nodo ha un solo controllo, ed ogni controllo è tipicamente definito con un nome. La possibilità di creare ed assegnare controlli ai nodi permette di definirne il significato a piacere, garantendo una grande libertà semantica. Per fare un paragone con la programmazione orientata agli oggetti, un nodo sta al suo controllo come un'istanza sta alla sua classe.

Terminologia (K-nodo)**.** Con l'espressione *K-nodo* si intende un nodo il cui controllo è K.

Ora, siccome ogni bigrafo avrà (tipicamente) più di un tipo di nodo, è necessario definirne l'insieme dei controlli.

Definizione 2.1.2 (Segnatura)**.** Una *segnatura* è una coppia (K, ar), dove K è un insieme di controlli e ar è una funzione $K \to \mathbb{N}$, che mappa ad ogni nodo la sua arietà (numero di porte).

Definizione 2.1.3 (Porta)**.** Una *porta* è un "punto" dal quale partono o terminano i collegamenti (archi).

Osservazione. Tramite la segnatura, si accoppia il controllo di un nodo al suo numero di porte biunivocamente; le porte sono ordinate e non intercambiabili.

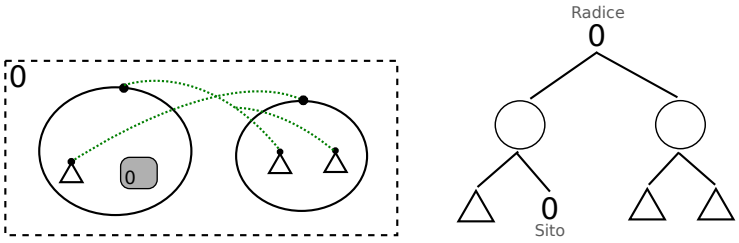
Notazione. Per semplicità, le segnature possono essere descritte come segue:

$K = \{K_1 : a_1, \ldots, K_n : a_n\}$ intendendo che il controllo K_i ha arietà a_i.

Quindi tramite la segnatura si ha già una prima caratterizzazione del bigrafo in termini di semantica ed arietà dei nodi. Con questo fondamento si può passare ad analizzare la topologia e le relazioni tra nodi.

2.1.2 Place graph

Il place graph è una foresta che descrive la topologia del bigrafo; in altre parole, indica come sono annidati i nodi. Ogni albero della foresta inizia con una *radice* (o *regione*); ogni radice è etichettata con un numero che va da 0 a $n - 1$, dove n è il numero di radici. Ogni nodo può contenere un *sito*, che indica la possibilità che vi siano altri nodi all'interno di quel nodo.



(a) Bigrafo (b) Place graph

Figura 2.2: Esempio di bigrafo e relativo place graph.

Terminologia (Posto)**.** Gli elementi del place graph sono detti *posti*.

Notazione. Con $A \uplus B$ si intende l'unione di insiemi che si sa o si assume essere fra loro disgiunti.

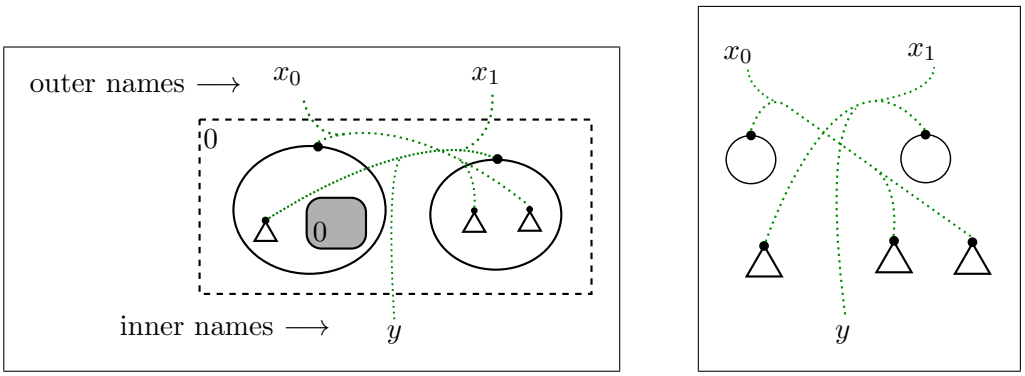
Definizione 2.1.4 (Place graph)**.** Un *place graph* è una tripla $(V, ctrl, prnt) : m \to n$ dove:

- V è l'insieme dei nodi;
- $ctrl$ è la mappa $V \to K$ che assegna ad ogni nodo un controllo;
- m è l'*interfaccia interna* ed n l'*interfaccia esterna*; rappresentano rispettivamente il numero di siti e di radici, pertanto sono entrambe rappresentate da numeri finiti;
- $prnt$ è la mappa (aciclica) $m \uplus V \to V \uplus n$, che assegna ad ogni nodo il suo genitore (se c'è).

2.1.3 Link graph

Il link graph è un *ipergrafo* (cioè un grafo i cui archi possono connettere più nodi) non orientato che descrive il collegamento fra i vari nodi del bigrafo (ovvero le loro relazioni). Il link graph descrive anche dei potenziali collegamenti con altri bigrafi grazie alle interfacce descritte dagli *inner names* e dagli *outer names*: gli outer names indicano l'interfaccia verso l'esterno, a cui possono collegarsi altri bigrafi; gli inner names (interfaccia interna) descrivono l'interfaccia dei bigrafi che possono aggiunti per composizione (si veda il paragrafo 2.2.2).

Terminologia (Punti e link)**.** Inner names e porte vengono chiamati *punti*; outer names e archi vengono chiamati *link*. Gli archi sono detti *link chiusi*, mentre gli outer names sono detti *link aperti*.



(a) Bigrafo (b) Link graph

Figura 2.3: Esempio di bigrafo e relativo link graph.

Definizione 2.1.5. Un link graph è una tupla $(V, E, ctrl, link) : X \to Y$ dove:

- V ed E sono, rispettivamente, gli insiemi (finiti) dei nodi e degli (iper)archi;
- $ctrl$ è la mappa dei controlli $V \to K$;
- X e Y sono, rispettivamente, le *interfacce interna* e *esterna*, composte dagli *inner names* e dagli *outer names*;
- $link$ è la mappa $X \uplus P \to E \uplus Y$, dove $P = \{(v, i) : v \in V, i \in ar(ctrl(v))\}$ è l'insieme delle porte.

Terminologia (Idle)**.** Un posto senza figli o un link senza punti è detto *idle*.

2.1.4 Bigrafo

Definendo le strutture che compongono i bigrafi, si è parlato più volte di *interfacce*. Ora verranno definite formalmente.

Definizione 2.1.6 (Interfaccia)**.** Un'*interfaccia* per bigrafi è una coppia $\langle m, X\rangle$, dove $m \in \mathbb{N}$ è detta *larghezza*, mentre X è l'insieme dei nomi.

Detto questo, si può dare una definizione formale di bigrafo.

Definizione 2.1.7 (Bigrafo)**.** Un *bigrafo* è una tupla

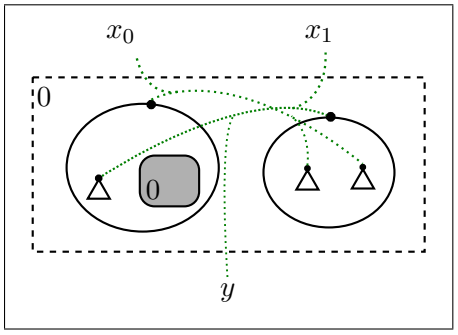
$$(V, E, ctrl, prnt, link) : \langle m, X\rangle \to \langle n, Y\rangle$$

dove $\langle m, X\rangle$ è detta *interfaccia interna*, mentre $\langle n, Y\rangle$ è detta *interfaccia esterna*.

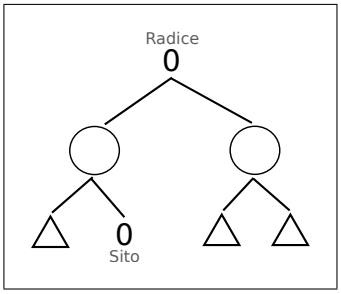
Osservazione. Un bigrafo B è composto da:

- un place graph $B^P = (V, ctrl, prnt) : m \to n$;
- un link graph $B^L = (V, E, ctrl, link) : X \to Y$.

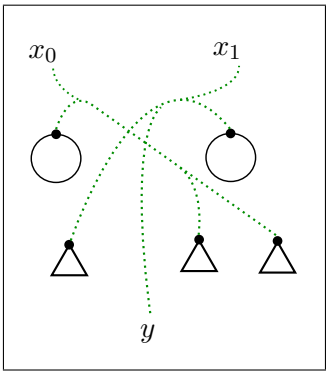
Che sono strutture ortogonali ed indipendenti. Pertanto, un bigrafo si può definire anche come una coppia $\langle B^P, B^L \rangle$.



(a) Bigrafo B



(b) Place graph B^P



(c) Link graph B^L

Figura 2.4: Esempio di bigrafo e sua scomposizione.

Terminologia. (Ground, agente) Un bigrafo la cui interfaccia interna è vuota viene detto *ground* o *agente*. Ne viene indicata solo l'interfaccia esterna.

Notazione. I bigrafi generici vengono indicati con lettere maiuscole (A,B,C,...), mentre gli agenti vengono indicati con lettere minuscole (a,b,c,...).

2.2 Operazioni sui bigrafi

Ora si andranno a definire alcune operazioni sui bigrafi e la loro algebra, con lo scopo di fornire il necessario per poter costruire bigrafi complessi a partire da

quelli elementari. Tali operazioni saranno fondamentali anche per poter descrivere l'evoluzione dei sistemi reattivi bigrafi (sez. 2.3).

2.2.1 Operazioni di base

Notazione. Con Id_I si indica la funzione identità sull'insieme I.

Definizione 2.2.1 (Supporto)**.** Ad ogni link graph, place graph, o bigrafo B viene assegnato un insieme finito $|B|$ detto *supporto*. Per un place graph $|B| = V$, mentre per link graphs e bigrafi $|B| = V \uplus E$.

Definizione 2.2.2 (Traduzione di supporto)**.** Per due bigrafi F e G, una *traduzione di supporto* $\rho : |F| \to |G|$ è una coppia di biiezioni $\rho_V : V_F \to V_G$ e $\rho_E : E_F \to E_G$ che ne rispetta la struttura, ovvero:

- ρ preserva i controlli, cioè $ctrl_G \circ \rho_V = ctrl_F$; ne consegue che ρ induce una biiezione $\rho_P : P_F \to P_G$ sulle porte, definita da $\rho_P((v, i)) \stackrel{\text{def}}{=} (\rho_V(v), i)$.
- ρ commuta con le mappe strutturali come segue:

  per i posti: $prnt_G \circ (id_m \uplus \rho_V) = (id_n \uplus \rho_V) \circ prnt_F$

  per punti e link: $link_G \circ (id_X \uplus \rho_P) = (id_Y \uplus \rho_E) \circ link_F$

  Dati F e la biiezione ρ, sotto queste condizioni è possibile determinare univocamente G; pertanto si dice che $G = \rho \cdot F$ è la traduzione di supporto di F per ρ.

Definizione 2.2.3 (Equivalenza sul supporto)**.** Diremo che F e G sono *support equivalent* (hanno supporti equivalenti) e si scriverà $F \simeq G$, se e solo se esiste una traduzione di supporto tra F e G.

2.2.2 Operazioni di costruzione

Verranno ora illustrate le principali operazioni di costruzione di bigrafi.

Definizione 2.2.4 (Composizione)**.** Dati due bigrafi con supporti disgiunti

$$F : \langle m, X \rangle \to \langle k, Z \rangle \ , \ G : \langle k, Z \rangle \to \langle n, Y \rangle$$

la composizione $H = G \circ F : \langle m, X \rangle \to \langle n, Y \rangle$ è il bigrafo

$$H = G \circ F = (V_F \uplus V_G, E_F \uplus E_G, ctrl_F \uplus ctrl_G, prnt_H, link_H)$$

dove:

$$prnt_H(w) = \begin{cases} prnt_F(w) & \text{se } w \in m \uplus V_F \wedge prnt_F(w) \in V_F \\ prnt_G(j) & \text{se } w \in m \uplus V_F \wedge prnt_F(w) = j \in k \\ prnt_G(w) & \text{se } w \in V_G \end{cases}$$

$$link_H(q) = \begin{cases} link_F(q) & \text{se } q \in X \uplus P_F \wedge link_F(q) \in E_F \\ link_G(z) & \text{se } q \in X \uplus P_F \wedge link_F(q) = z \in Z \\ link_G(q) & \text{se } q \in P_G \end{cases}$$

con P_F e P_G insiemi delle porte, come definiti nella sezione del link graph.

Osservazione. La composizione richiede che l'interfaccia esterna di F sia uguale a quella interna di G.

Terminologia (Contesto)**.** Un bigrafo composto con un agente viene detto *contesto*.

Definizione 2.2.5 (Bigrafo identità)**.** Il *bigrafo identità* su $\langle m, X\rangle$ è

$$id_{\langle m,X\rangle} = (\emptyset, \emptyset, \emptyset_K, id_m, id_X).$$

Definizione 2.2.6 (Bigrafi disgiunti)**.** Due bigrafi $F_i : \langle m_i, X_i\rangle \to \langle n_i, Y_i\rangle$ $(i = 0, 1)$ sono *disgiunti* se $|F_i|$,X_i e Y_i sono (rispettivamente) fra loro disgiunti.

Definizione 2.2.7 (Giustapposizione)**.** Dati due bigrafi disgiunti:

$$F : \langle m_F, X_F\rangle \to \langle n_F, Y_F\rangle \ , \ G : \langle m_G, X_G\rangle \to \langle n_G, Y_G\rangle$$

la loro giustapposizione $F \otimes G : \langle m_F + m_G, X_F \uplus X_G\rangle \to \langle n_F + n_G, Y_F \uplus Y_G\rangle$ è il bigrafo

$$F \otimes G = (V_F \uplus V_G,\ E_F \uplus E_G,\ ctrl_F \uplus ctrl_G,\ prnt_F \uplus prnt'_G,\ link_F \uplus link_G)$$

dove $prnt'_G(m_F + i) = n_F + j$ ogni qualvolta $print_G(i) = j$.

Osservazione. Nella giustapposizione tra bigrafi l'interfaccia unità è $\epsilon = \langle 0, \emptyset\rangle$.

2.2.3 Algebra dei bigrafi

Ora si mostrerà come i bigrafi possono essere costruiti a partire da bigrafi più piccoli, per via di composizioni, prodotti ed identità. Verranno descritti i bigrafi elementari e le loro forme normali.

Definizione 2.2.8 (Placing, permutazione, merge)**.** Un *placing* (ϕ) è un bigrafo senza né nodi né link. Un placing biiettivo dai siti alle radici (ovvero che per ogni radice ha un solo sito) è una *permutazione* (π). Un placing con una radice ed n siti è denotato con $merge_n$. Vengono definiti tre placing detti *elementari*:

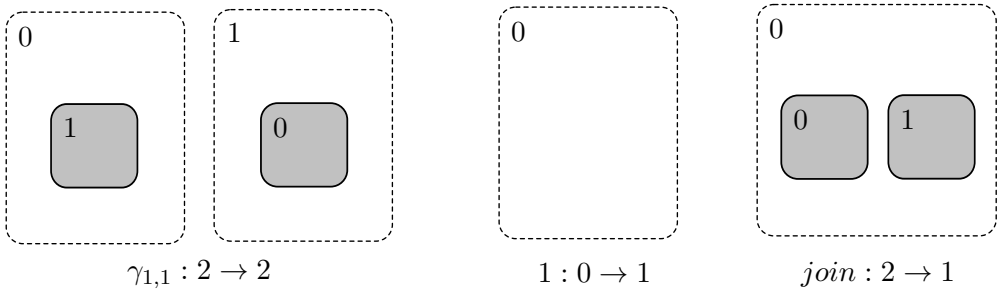


Figura 2.5: Placing elementari.

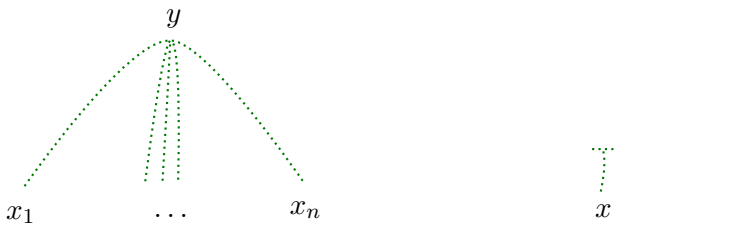
Osservazione. Tutte le permutazioni possono essere costruite a partire dalla simmetria elementare $\gamma_{1,1}$; tutti i placings possono essere costruiti a partire da $\gamma_{1,1}$, 1 e *join*.

Esempio. $merge_0 = 1$ e $merge_{n+1} = join \circ (id_I \otimes merge_n)$

Definizione 2.2.9 (Linking, sostituzione, chiusura)**.** Un *linking* (λ) è un bigrafo senza posti. I linkings sono generati per composizione, prodotto e identità a partire da *sostituzioni elementari* y/X e *chiusure elementari* $/x : x \to \epsilon$ (si veda la figura 2.6).

Una *sostituzione* (σ) è il prodotto di sostituzioni elementari; una *chiusura* è il prodotto di chiusure elementari.

La sostituzione vuota è $x : \epsilon \to x$.



(a) Sostituzione elementare (b) Chiusura elementare

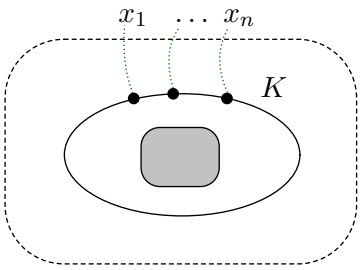
Figura 2.6: Linking elementari.

Terminologia (Rinomina)**.** Una sostituzione biiettiva è detta *rinomina* (α).

Proposizione 2.2.1 (Isomorfismo)**.** Isomorfismi tra place graphs e link graphs sono, rispettivamente, permutazioni π e rinomine α; isomorfismi tra bigrafi sono coppie $\langle \pi, \alpha \rangle$.

C'è solo un tipo di bigrafo elementare che introduce nodi: l'*ione*.

Definizione 2.2.10 (Ione)**.** Per ogni controllo $K : n$, il bigrafo $K_{\vec{x}} : 1 \to \langle 1, \{x_1, \ldots, x_n\}\rangle$ avente un unico nodo K le cui porte sono collegate biiettivamente con n distinti nomi ed un unico sito all'interno di K, è detto *ione discreto* (o più semplicemente, *ione*).



$K_{\vec{x}}$

Figura 2.7: Ione.

Definizione 2.2.11 (Primo, discreto)**.** Un bigrafo *primo* con interfaccia $m \to \langle X\rangle$ è privo di inner names, e ha una sola radice. Un bigrafo (o link graph) è *discreto* se non ha link chiusi, e la sua mappa dei link è biiettiva.

Osservazione. $merge_n : n \to 1$ è un primo importante, in quanto non ha nodi e mappa n siti in una singola radice.

Definizione 2.2.12 (Atomo, molecola)**.** Se il sito di un K-ione discreto è riempito da $1 : 0 \to 1$ (placing elementare), il risultato è un *atomo* discreto $K_{\vec{x}} \circ 1$; se è riempito da un bigrafo discreto $G : I \to \langle 1, Y\rangle$ è una *molecola* discreta $(K_{\vec{x}} \otimes id_Y) \circ G$.

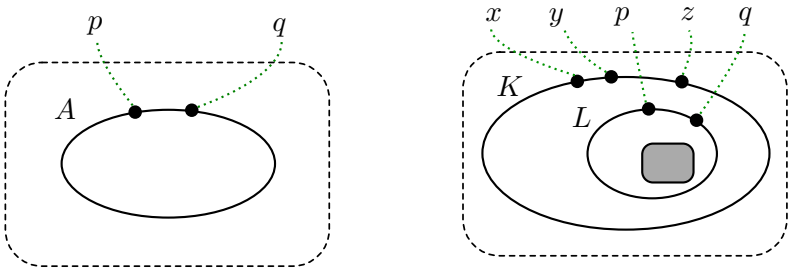
Osservazione. L'identità id_I è importante qua: permette ai nodi di C_1 di avere figli in C_2 ed F, e permette a C_1 e C_2 di condividere link che non coinvolgono F.

Proposizione 2.2.2 (Forma Normale Discreta)**.** Ogni bigrafo $G : \langle m, X\rangle \to \langle n, Z\rangle$ può essere espresso univocamente, a meno di rinomine 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 discreto. Inoltre, ogni bigrafo discreto D può essere fattorizzato univocamente, a meno di permutazione dei siti per ogni fattore, come

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



$A_{pq} \circ 1$ (a) Atomo

$(K_{xyz} \otimes id_{pq}) \circ L_{pq}$ (b) Molecola

Figura 2.8: Atomo e molecola.

con α un renaming, ogni P_i primo e discreto e π una permutazione di tutti i siti.

Si noti che il renaming α è discreto ma non primo, dato che ha $width = 0$ e possiede inner names. Nel caso in cui il grafo D sia ground, il risultato può essere semplificato come segue:

Corollario 2.2.1 (Forma Normale Discreta per Bigrafi Ground)**.** Un bigrafo ground $g : \langle n, Z \rangle$ è fattorizzabile in modo univoco, a meno di rinomine su Y, come

$$g = (id_n \otimes \lambda) \circ (d_0 \otimes \cdots \otimes d_{n-1})$$

dove $\lambda : Y \to Z$ sono linking e d_i sono bigrafi discreti e primi.

Questa scomposizione di un bigrafo in bigrafi discreti più piccoli è cruciale per dimostrare la completezza della teoria algebrica dei bigrafi ([2]).

2.2.4 Operazioni derivate

Notazione. 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.

Se $X = \{x_1, \ldots, x_n\}$ si scriverà $/X$ per intendere $/x_1 \otimes \cdots \otimes x_n$.

Definizione 2.2.13 (Prodotto parallelo)**.** Siano $G_i : \langle m_i, X_i\rangle \to \langle n_i, Y_i\rangle$ $(i = 0, 1)$ due bigrafi i cui supporti sono disgiunti e per cui $link_0 \cap link_1$ è una funzione; Allora il loro *prodotto parallelo*

$$G_0 \parallel G_1 \stackrel{\text{def}}{=} \langle m_0 + m_1, X_0 \cup X_1\rangle \to \langle n_0 + n_1, Y_0 \cup Y_1\rangle$$

è definito proprio come la giustapposizione, eccetto per il fatto che la mappa dei link permette la condivisione dei nomi.

Proposizione 2.2.3 (Proprietà di $\parallel$)**.** Il prodotto parallelo ha la *proprietà associativa*, ovvero $F \parallel (G \parallel H) = (F \parallel G) \parallel H$, ed ha id_ϵ come unità; inoltre soddisfa la *proprietà 'bifuntoriale'*, se ambo le parti sono definite:

$$(F_1 \parallel G_1) \circ (F_0 \parallel G_0) = (F_1 \circ F_0) \parallel (G_1 \circ G_0)$$

Dimostrazione. *La dimostrazione segue linearmente dalla definizione, una volta notato che la condizione sulle mappe dei link è soddisfatta da una parte se e solo se è soddisfatta anche dall'altra.* □

Definizione 2.2.14 (Annidamento)**.** Dati i bigrafi $F : I \to \langle m, X\rangle$ e $G : m \to \langle n, Y\rangle$, il loro *annidamento* $G.F : I \to \langle m, X \cup Y\rangle$ è definito come:

$$G.F \stackrel{\text{def}}{=} (id_X \parallel G) \circ F$$

Definizione 2.2.15 (Prodotto Merge)**.** Il prodotto merge $(|)$ di due bigrafi $G_i : \langle m_i, X_i\rangle \to \langle n_i, Y_i\rangle$ $(i = 0, 1)$, il cui prodotto parallelo esiste, è definito come:

$$G_0 \mid G_1 = merge_{n_0+n_1} \circ (G_0 \parallel G_1) : \langle m_0 + m_1, X_0 \cup X_1\rangle \to \langle 1, Y_0 \cup Y_1\rangle$$

Proposizione 2.2.4 (Proprietà di $|$)**.** Il prodotto merge gode della *proprietà associativa* e (su bigrafi di larghezza unitaria) ha 1 come unità.

2.3 Evoluzione dei bigrafi

Il uno dei motivi per cui i bigrafi sono un formalismo interessante è la loro capacità di evolvere.

Definizione 2.3.1 (Reazione)**.** Una *reazione* $F \to F'$ è una trasformazione da un bigrafo F ad un bigrafo F'.

Definizione 2.3.2 (Segnatura dinamica)**.** Una segnatura è detta *dinamica* se associa ad ogni controllo K uno *stato* nell'insieme $\{attivo, passivo\}$. Un K-nodo è

detto attivo se il suo controllo è attivo. Le regioni sono sempre attive. Un sito è attivo se tutti i suoi antenati nel place graph sono attivi. Un bigrafo è attivo se tutti i suoi siti sono attivi.

Osservazione. Un K-nodo passivo, e conseguentemente tutti i suoi discendenti, sono inibiti al prendere parte alle reazioni.

Definito cosa intendiamo per reazione, ora è necessario formalizzare un modo per rappresentare il *modo* in cui queste reazioni avvengono; ciò viene ottenuto definendo delle *regole di riscrittura*.

2.3.1 Regole di riscrittura

Definizione 2.3.3 (Regola di riscrittura)**.** Una *regola di riscrittura* $R \to R'$ definisce una reazione $a \to a'$, dove:

- $a = C \circ (R \otimes id_I) \circ d$;
- $a' = C \circ (R' \otimes id_I) \circ d$;
- C è attivo;
- a, a' e d sono $ground(agenti)$;
- R ed R' hanno una stessa interfaccia I.

Terminologia (Redex, Reactum)**.** R è detto *redex*, R' è detto *reactum*.

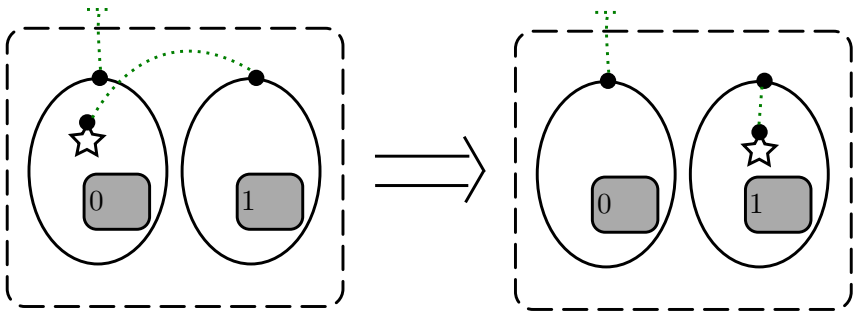


Figura 2.9: Esempio di regola di riscrittura. A sinistra della freccia c'è il redex, a destra c'è il reactum

In buona sostanza, quando si applica una regola di riscrittura ad un bigrafo B, si va a cercare un'*occorrenza* del redex in B, e la si sostituisce con il reactum. Ora si formalizzerà questo procedimento.

Definizione 2.3.4 (Occorrenza)**.** Un bigrafo F *occorre* in un bigrafo G se l'equazione $C_1 \circ (F \otimes id_I) \circ C_2$ è vera per qualche interfaccia I e bigrafo C_1 e C_2.

Terminologia (Matching, contesto, parametro)**.** La ricerca dell'occorrenza di un bigrafo F in uno G viene detta *matching*. I risultati del matching vengono chiamati *match*, indicati con $M = C \circ (F \otimes id_I) \circ D$; C viene detto *contesto* mentre D viene chiamato *vettore di parametri* (o, più semplicemente, *parametro*).

Osservazione. Con riferimento alla definizione 2.3.3, il fatto che R ed R' debbano avere la stessa interfaccia è un vincolo molto forte: infatti non ciò consente di fare, per esempio, la copia del contenuto di un nodo in un altro, o lo scambio dei contenuti di due nodi in regioni differenti (mantenendo inalterata la loro posizione).

Per rilassare questo vincolo, si va a definire una mappa tra le interfacce del redex e del reactum.

Definizione 2.3.5 (Mappa d'istanziazione)**.** Siano R ed R' due bigrafi con, rispettivamente, $\langle m, X\rangle$ e $\langle m', X\rangle$ come interfacce interne. Una *mappa d'istanziazione* è una funzione $\eta : m' \to m$ che mappa siti di R' in siti di R.

Definizione 2.3.6 (Funzione istanza/istanziamento)**.** Una *funzione istanza* $\bar{\eta}$, data una mappa d'istanziazione $\eta : m' \to m$, e dato un agente $g : \langle m, X\rangle$ la cui Forma Normale Discreta è $g = \lambda \circ (d_0 \otimes \cdots \otimes d_{m-1})$, è definita come:

$$\bar{\eta}(g) \stackrel{\text{def}}{=} \lambda \circ (d'_0 \parallel \cdots \parallel d'_{m'-1}), \text{ con } d'_i \eqsim d_{\eta(i)}.$$

Grazie alle mappe di istanziazione è possibile definire una forma più espressiva di regola di riscrittura.

Definizione 2.3.7 (Regola di riscrittura parametrica)**.** Una regola di riscrittura $R \to R'$, affiancata da una mappa d'istanziazione η, è detta *regola di riscrittura parametrica*. Dato un *agente* a, dove R occorre in a, la regola di riscrittura parametrica produce la seguente reazione $a \to a'$, dove:

- $a = C \circ (R \otimes id_I) \circ d$;
- $a' = C \circ (R' \otimes id_I) \circ \bar{\eta}(d)$;
- $\bar{\eta}$ è la funzione istanza su η.

Osservazione. Ogni regola di riscrittura si può scrivere come una regola di riscrittura parametrica che ha come mappa d'istanziazione la funzione identità.

Terminologia. In seguito, con la locuzione *regola di riscrittura*, si indicheranno le regole di riscrittura *parametriche*.

2.3.2 BRS - Sistema Reattivo Bigrafico

Definite le regole di scrittura, si vuole accoppiare la semantica del bigrafo (definita dalla segnatura) all'insieme di regole che ne causano l'evoluzione. Ciò viene fatto dal *BRS*.

Definizione 2.3.8 (Sistema Reattivo Bigrafico - BRS)**.** Un *Sistema Reattivo Bigrafico* (BRS) è definito da una coppia $(K, \mathcal{R})$, dove K è una segnatura ed $\mathcal{R}$ è un insieme di regole di riscrittura; pertanto viene indicato con $BG(K, \mathcal{R})$. L'insieme $\mathcal{R}$ è chiuso rispetto all'*equivalenza sul supporto*: se $R \eqsim S$ e $R' \eqsim S'$ e $(R, R', \eta) \in \mathcal{R}$ per un certo η, allora $(S, S', \eta) \in \mathcal{R}$.

Pertanto, dato un agente a con segnatura K, $BG(K, \mathcal{R})$ permette di calcolare tutti gli agenti a' che si possono ottenere facendo reagire a un numero indeterminato di volte, ovvero gli $a' : a \rightarrow_* a'$.

Nel capitolo 4 a pagina 23 sarà illustrato un modo di implementare e simulare il BRS, nonché di ottenere tutti gli stati possibili del sistema da esso.

3

Visualizzare i Bigrafi

Prima di trattare la simulazione dei sistemi reattivi bigrafici, sarebbe interessante trovare il modo di visualizzare i bigrafi su calcolatore nel modo migliore possibile. Tuttavia generare in modo automatico una rappresentazione grafica dei bigrafi è un problema complicato, che non è stato ancora trattato a sufficienza. Pertanto, come punto di partenza, in questo capitolo si fornirà un metodo di rappresentazione testuale; in seguito, si indicherà come produrre una rappresentazione grafica semplificata.

3.1 Rappresentazione testuale di bigrafi

La libreria che si utilizzerà per implementare i BRS (si veda la sez. 4.1 a pagina 24) fornisce già un metodo per rappresentare un bigrafo con del testo, ovvero il metodo `Bigraph.toString()`[1]; tuttavia il formato, come si può vedere sotto, è poco leggibile.

Codice 3.1: Risultato di `Bigraph.toString()` sul bigrafo 3.2 a pagina 20

```
1  Car Signature {car:(1,a), road:(1,a), fuel:(0,a), place:(1,a),
       target:(1,a)} :: <0,{}> -> <1,{}>
2  E_12:e <- {0@N_13:place, 0@N_1A:road, 0@N_1B:road}
3  E_14:e <- {0@N_15:place, 0@N_18:road}
4  E_16:e <- {0@N_17:place, 0@N_19:road}
5  E_1C:e <- {0@N_1E:car, 0@N_1D:target}
6  0 <- {N_13:place, N_15:place, N_17:place}
7  N_13:place <- {N_1E:car, N_18:road}
8  N_15:place <- {N_19:road, N_1A:road}
9  N_17:place <- {N_1B:road, N_1D:target}
10 N_1E:car <- {N_1F:fuel, N_20:fuel, N_21:fuel, N_22:fuel, N_23:
       fuel}
11 N_18:road <- {}
12 N_19:road <- {}
13 N_1A:road <- {}
14 N_1B:road <- {}
15 N_1D:target <- {}
16 N_1F:fuel <- {}
```

[1] `Bigraph` è la classe che modella i bigrafi, si veda la sez. 4.3 a pagina 25

```
17 N_20:fuel <- {}
18 N_21:fuel <- {}
19 N_22:fuel <- {}
20 N_23:fuel <- {}
```

Ci si propone di migliorare la situazione andando a definire una classe *pretty-printer*: tale classe legge un oggetto `Bigraph`, costruendone una sua rappresentazione interna (ad albero); tale struttura viene poi convertita in testo, secondo i seguenti criteri:

- si vogliono rappresentare nodi e relativi controllo, proprietà e porte;
- si vuole rappresentare la gerarchia dei nodi indentando il testo;
- si vogliono visualizzare in modo particolare alcune proprietà; in, particolare le propietà con nome del tipo *"*Name"*, vengono visualizzate come nome del relativo nodo.

Codice 3.2: *Pretty-print* del bigrafo 3.2 a pagina 20

```
1  ----- Printing Bigraph Car Example -----
2  edge E_12{ N_1B:road; N_1A:road; N_13:place; }
3  edge E_1C{ N_1D:target; N_1E:car; }
4  edge E_14{ N_15:place; N_18:road; }
5  edge E_16{ N_17:place; N_19:road; }
6  root 0 {}
7     node N_17:place { port0: Edge E_16;}
8        node N_1D:target { port0: Edge E_1C;}
9        node N_1B:road { port0: Edge E_12;}
10    node N_13:place { port0: Edge E_12;}
11       node N_18:road { port0: Edge E_14;}
12       node N_1E:car { port0: Edge E_1C;}
13          node N_1F:fuel {}
14          node N_20:fuel {}
15          node N_21:fuel {}
16          node N_22:fuel {}
17          node N_23:fuel {}
18    node N_15:place { port0: Edge E_14;}
19       node N_19:road { port0: Edge E_16;}
20       node N_1A:road { port0: Edge E_12;}
21 ----- Done Printing Car Example ------
```

Come si può notare, il formato prodotto è più leggibile, ma non con la stessa immediatezza di una rappresentazione grafica.

3.2 Rappresentazione grafica di bigrafi

Come anticipato, non esiste un algoritmo preciso ed affidabile per generare automaticamente la rappresentazione grafica dei bigrafi; al contrario, il campo della visualizzazione dei grafi è stato soggetto ad estesa ricerca, e sono disponibili algoritmi, linguaggi e programmi già pronti. Pertanto si è deciso di fornire una rappresentazione semplificata dei bigrafi in questo modo:

1. un *parser* trasforma il bigrafo in un grafo orientato, rappresentato in *DOT Language*;
2. si usa il tool *GraphViz* per trasformare il file *dot* in un'immagine vettoriale *svg*;
3. a questo punto si può visualizzare l'immagine con un *browser*, o un visualizzatore che ne supporta il formato; altrimenti si può convertirla nel formato ritenuto più opportuno.

Un esempio del risultato di questa procedura è rappresentato dalla figura 3.1 nella pagina successiva.

Strumenti impiegati Il *parser* citato al punto 1 è implementato dalla classe `DotLangPrinter`, che internamente funziona in modo simile al *pretty-printer*.

Il *DOT Language* è un formalismo nato per rappresentare grafi orientati e non orientati, permettendo di specificare proprietà per i nodi come colore, forma etichetta, etc...

GraphViz è un programma *open-source* che genera immagini di grafi, data la loro descrizione in DOT. È reperibile su `http://www.graphviz.org/`.

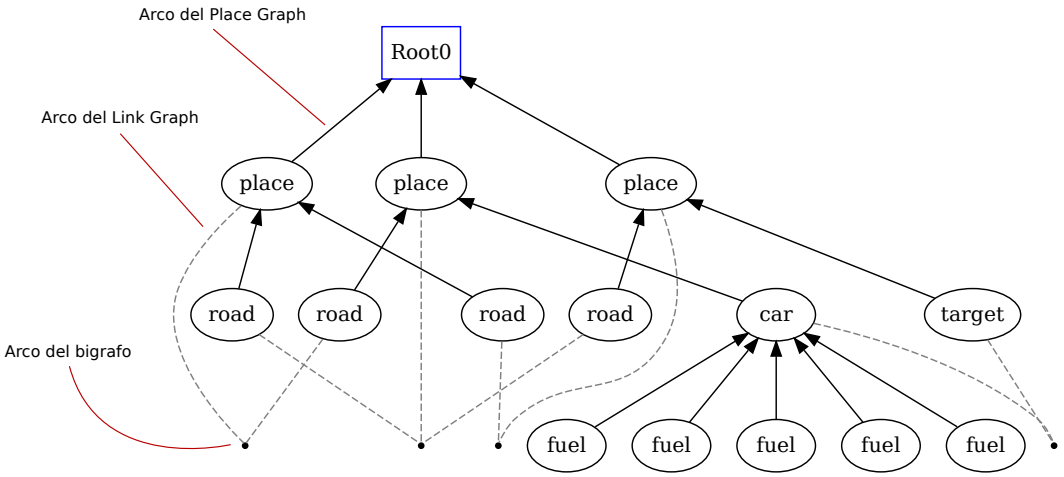


Figura 3.1: *Render* di GraphViz del codice 3.3 nella pagina successiva.

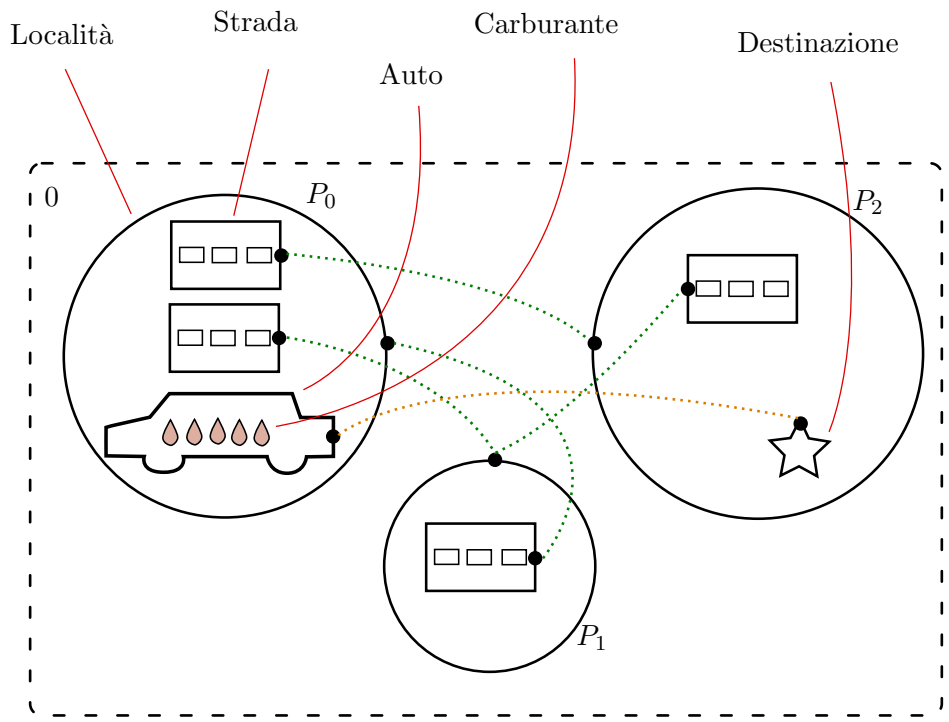


Figura 3.2: Bigrafo da rappresentare (si veda la sez. 6.1 a pagina 37).

Codice 3.3: Rappresentazione in DOT (ridotta) del bigrafo 3.2 nella pagina precedente.

```
digraph Car Example {
// Ranghi usati per allineare i nodi su piu' righe
ranksep=.75;{
outernames->rank0->rank1->rank2->rank3[style=invisible
   arrowhead=none];outernames[shape=none style=invisible];
[...] // Omissis
}

// Place graph e nodi

N_17 -> 0
N_1D -> N_17
N_1B -> N_17
N_13 -> 0
N_18 -> N_13
[...]
{
rank=same;
rank0;0[shape=box color=blue,label="Root0"];
}
{
rank=same;
rank1;N_17[shape=ellipse,label="place"];
N_13[shape=ellipse,label="place"];
N_15[shape=ellipse,label="place"];
}
[...]

// Link graph e archi
N_1B -> E_12[arrowhead=none color=grey style=dashed k=2];
N_1A -> E_12[arrowhead=none color=grey style=dashed k=2];
N_13 -> E_12[arrowhead=none color=grey style=dashed k=2];
E_12[shape=point];
[...]


// Outer Names
{
rank=same;
outernames;
}}
```

4

Simulazione di BRS

Si immagini di avere un sistema che si vuole rappresentare usando il formalismo dei bigrafi. Si immagini di aver definito (su carta) un'adeguata segnatura, un bigrafo rappresentante lo stato iniziale, e un insieme adeguato di regole di riscrittura. Si vuole ora costruire un programma che permetta di *simulare* l'evoluzione del sistema definito sopra. In questo capitolo si descrive un'implementazione di un simulatore e delle sue componenti ancillari.

Terminologia (Sistema, stato)**.** D'ora in avanti con il termine *sistema* si intende un sistema reattivo bigrafico (BRS), mentre con *stato* si intende un bigrafo.

Traccia della discussione Innanzitutto bisogna implementare la segnatura, i bigrafi e le regole: per questo ci si appoggia alla libreria JLibBig. Poi si pone il problema di come far evolvere il sistema, dato che le regole di riscrittura vengono applicate in modo non-deterministico: non esiste un solo percorso lineare di evoluzione, bensì ve ne sono molteplici, alcuni dei quali magari non hanno senso concretamente; possono addirittura formarsi cicli nell'evoluzione degli stati (p.e. reazioni $A \to B$ e $B \to A$), portando un eventuale processo di simulazione a non terminare. In buona sostanza, può risultare necessario "filtrare" le applicazioni delle regole di reazione e tenere traccia di di tutti gli stati generati, in modo da poter decidere in seguito quali ci interessano e quali no.

Modularità Si può già intuire che un un simulatore è un costrutto complesso: al fine di semplificarne la comprensione, nonché l'implementazione, si propone di suddividerlo in moduli, che verranno discussi uno ad uno. Saranno altresì discusse le basi implementative necessarie.

4.1 La libreria JLibBig

JLibBig è una libreria Java per modellare i bigrafi, reperibile dal sito del MADS (laboratorio di Modelli ed Applicazioni di Sistemi Distribuiti) dell'Università di Udine.[1] Essa fornisce la base solida su cui si appoggia la parte implementativa di questo lavoro di tesi. Un breve elenco delle *feature* più utili:

- Permette di modellare segnature e controlli;
- Permette di modellare bigrafi ed entità associate (nodi, punti, link, ...);
- Permette di agganciare delle proprietà (ovvero coppie $(chiave, valore)$) ai nodi dei bigrafi;
- Implementa le regole di riscrittura e il relativo matching.

Cosa non fornisce La libreria non dà strumenti per efficaci per visualizzare i bigrafi (trattati nel capitolo 3 a pagina 17) e gestire l'evoluzione di un sistema reattivo bigrafico nel suo insieme: in particolare non permette di tenere traccia di tutti gli stati generati. Ci si propone quindi di costruire delle classi che rendano la simulazione dei BRS un'operazione di alto livello, cioè non rendendo necessario occuparsi dei dettagli implementativi che essa comporta.

4.2 Rappresentare Controlli e Segnature

Le segnature possono essere facilmente costruite utilizzando la classe *factory* `SignatureBuilder`, a cui si aggiungono i controlli indicandone nome, stato(attivo/passivo) ed arietà con il metodo `add(String,boolean,int)`.

Codice 4.1: Esempio di definizione di segnatura in JLibBig per il bigrafo 4.1 a pagina 26

```
1  private static Signature generateSignature() {
2       SignatureBuilder sb = new SignatureBuilder();
3
4       sb.add("car", true, 1);
5       sb.add("fuel", true, 0);
6       sb.add("place", true, 1);
7       sb.add("road", true, 1);
8       sb.add("target", true, 1);
9
10      return sb.makeSignature("Car Signature");
11 }
```

[1] `http://mads.uniud.it/`

4.3 Rappresentare i Bigrafi

JLibBig fornisce la classe `Bigraph`, che permette di definire i bigrafi aggiungendo i vari elementi ad un `BigraphBuilder`, ovvero una *classe factory*. Nella libreria i bigrafi sono oggetti immutabili: quindi, ogni volta che vengono modificati dall'applicazione delle regole di riscrittura, in realtà viene generato un nuovo oggetto.

Proprietà Come anticipato, la libreria permette anche di assegnare ai nodi dei bigrafi delle proprietà, cioè coppie nome-valore; per esempio, si può assegnare dei nomi ai nodi. Questo permette anche di rappresentare informazioni che altrimenti andrebbero codificate nei bigrafi stessi con dei controlli appositi. Tuttavia, quando si applicano le regole di riscrittura, tali proprietà non vengono propagate automaticamente.

Codice 4.2: Codifica in JLibBig del bigrafo 4.1 nella pagina seguente

```
1  BigraphBuilder bb = new BigraphBuilder(SIGNATURE);
2
3  Root root = bb.addRoot();
4
5  Node p0 = bb.addNode("place", root);
6  Node p1 = bb.addNode("place", root);
7  Node p2 = bb.addNode("place", root);
8
9  bb.addNode("road", p0, p1.getPort(0).getHandle());
10 bb.addNode("road", p1, p2.getPort(0).getHandle());
11 bb.addNode("road", p1, p0.getPort(0).getHandle());
12 bb.addNode("road", p2, p0.getPort(0).getHandle());
13
14
15 Node target = bb.addNode("target", p2);
16 Node car = bb.addNode("car",p0,target.getPort(0).getHandle());
17 for (int i = 0; i < 5; i++) {
18      bb.addNode("fuel", car);
19 }
20
21 Bigraph big = bb.makeBigraph(true);
```

4.4 Rappresentare le Regole di Riscrittura

Anche le regole di riscrittura sono già pronte all'uso in JLibBig, grazie alla classe `RewritingRule`: si definiscono *redex* e *reactum*, la mappa d'istanziazione e, successivamente, si può applicare la regola chiamando `apply(Bigraph)`. Così facendo, si ottiene un `Iterable<Bigraph>`, ovvero una sequenza di bigrafi creati dalle possibili applicazioni della regola.

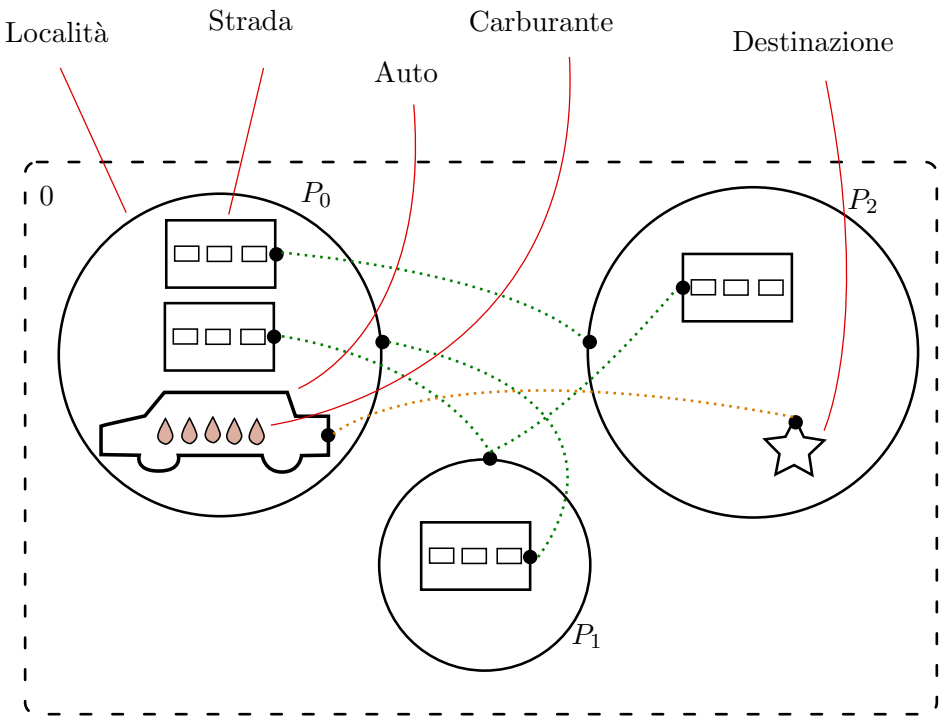


Figura 4.1: Esempio di bigrafo (si veda la sez. 6.1 a pagina 37).

Codice 4.3: Codifica in JLibBig della regola di riscrittura 4.2 a pagina 28

```
1  public class RR_Move extends RewritingRule {
2
3      private static final Bigraph redex = generateRedex(),
4              reactum = generateReactum();
5      private static final InstantiationMap map = new
           InstantiationMap(3, 0, 1, 2);
6      /* new InstantiationMap(numSitiReactum,[mappa siti reactum
           ->siti redex])*/
7
8      public RR_Move() {
9          super(redex, reactum, map);
10     }
11
12     private static Bigraph generateRedex() {
13         BigraphBuilder bb = new BigraphBuilder(Car.SIGNATURE);
14
15         Root root = bb.addRoot();
16
17         OuterName to1 = bb.addOuterName("to1");
18         // addNode(Controllo,Parent,[Link])
19         Node p1 = bb.addNode("place", root, to1);
20         bb.addSite(p1);
21
22         OuterName to0 = bb.addOuterName("to0");
```

```
23          Node to0 = bb.addNode("place", root, to0);
24          Node road = bb.addNode("road", to0, to1);
25          bb.addSite(to0);
26
27          OuterName tgt = bb.addOuterName("target");
28          Node car = bb.addNode("car", to0, tgt);
29          bb.addNode("fuel", car);
30          bb.addSite(car);
31
32          return bb.makeBigraph(true);
33      }
34
35      private static Bigraph generateReactum() {
36
37          BigraphBuilder bb = new BigraphBuilder(Car.SIGNATURE);
38
39          Root root = bb.addRoot();
40
41          OuterName to1 = bb.addOuterName("to1");
42          Node p1 = bb.addNode("place", root, to1);
43          bb.addSite(p1);
44
45          OuterName to0 = bb.addOuterName("to0");
46          Node to0 = bb.addNode("place", root, to0);
47          Node road = bb.addNode("road", to0, to1);
48          bb.addSite(to0);
49
50          OuterName tgt = bb.addOuterName("target");
51          Node car = bb.addNode("car", to1, tgt);
52          bb.addSite(car);
53
54          return bb.makeBigraph(true);
55      }
56
57  }
```

Matcher Per trovare le occorrenze del *redex* in un bigrafo si fa affidamento alla classe `Matcher`, che modella il problema in un *embedding* (un problema di flusso), e lo risolve con tecniche di programmazione a vincoli (appoggiandosi alla libreria *CHOCO Solver*).[2]

4.5 Modellare il BRS

Per modellare il BRS viene fornita l'omonima classe, che ha un semplice compito: essa incapsula le regole di riscrittura del sistema e la "strategia" con cui esse

[2] `http://choco-solver.org/`

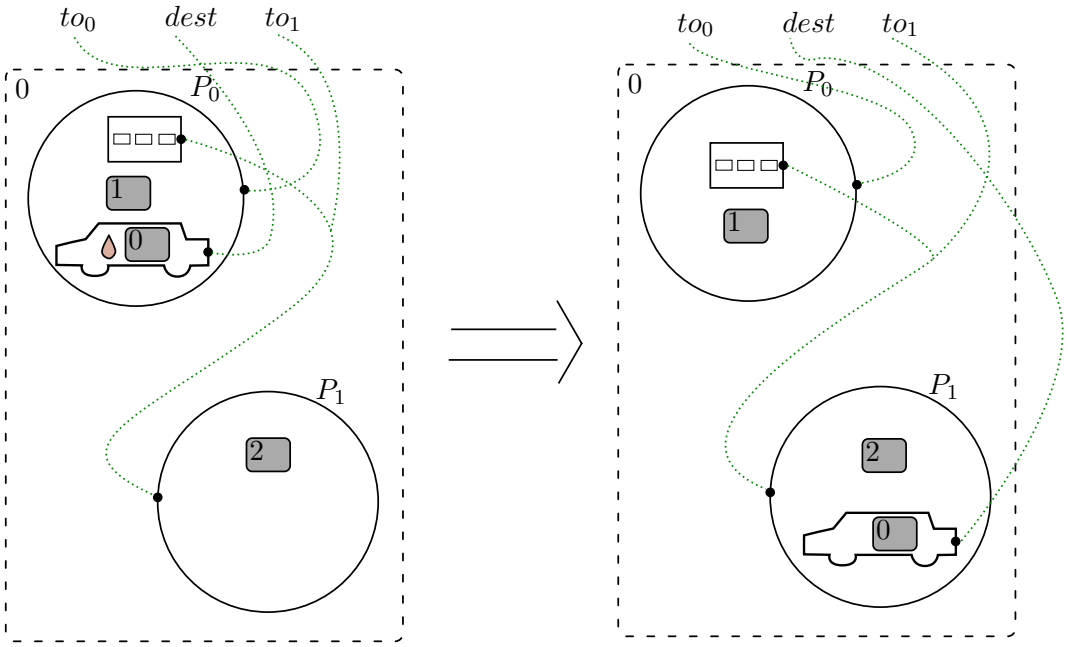
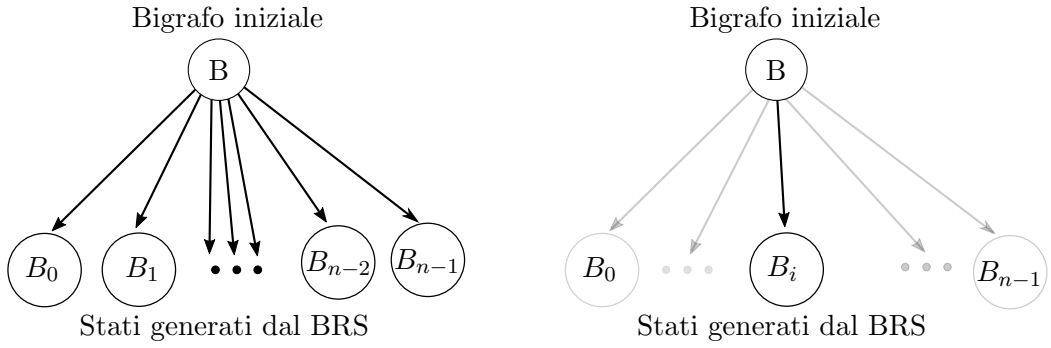


Figura 4.2: Regola di riscrittura dall'esempio 6.1 a pagina 37.

vengono applicate; passando un bigrafo al BRS, si ottiene la lista degli stati ottenuti applicandovi le regole.

Implementazione Nello specifico, BRS produce una lista di `Bigraph`(s) o, in alternativa, di coppie $\langle$`Bigraph`, `RewritingRule`$\rangle$ dette `RuleApplication`(s), in modo che gli altri moduli del simulatore possano capire che regola ha prodotto una data evoluzione del sistema. Ciò si ottiene chiamando, rispettivamente, i metodi `apply(Bigraph)` o `apply_RA(Bigraph)` (dove *"RA"* sta per `RuleApplication`).

Strategie Il modulo BRS è stato implementato in modo da poter definire con che criterio vengono applicate le regole di reazione: per esempio, è possibile definire delle regole di riscrittura con priorità, in modo da controllare l'ordine di applicazione, oppure usare un sistema probabilistico. L'implementazione standard `BreadthFirstBRS` usa una strategia stile *"breadth first"*, cioè applica le regole nell'ordine in cui sono date, esplorando gli stati successivi del sistema "in ampiezza". Esiste poi `RandomBRS` che ritorna una singola applicazione (in particolare la i-esima tra tutte le applicazioni, dove i è determinato *pseudo*casualmente).



(a) Applicazione di `BreadthFirstBRS`

(b) Applicazione di `RandomBRS`

Figura 4.3: Strategie di `BRS` a confronto.

4.6 Grafo degli stati

Com'è noto, dato uno stato del sistema (un bigrafo), in generale è possibile applicarvi più di una regola di riscrittura (non-deterministicamente); inoltre, una regola può essere applicata tante volte quanti sono i *match* del redex, generando evoluzioni potenzialmente diverse del sistema. Per questo motivo si propone di costruire un *grafo degli stati*, che è un grafo orientato in cui:

- i nodi rappresentano stati del sistema, cioè bigrafi;
- gli archi indicano le transizioni da uno stato al successivo, e la regola che l'ha prodotta.

Questa struttura dati permette di analizzare tutta la storia dell'evoluzione del sistema, e persino di generare tutti gli stati possibili (cosa che può essere sfruttata nel verificare proprietà del sistema, si veda il capitolo 5 a pagina 35).

4.6.1 Problematiche da affrontare

Ovviamente costruire, mantenere ed utilizzare una struttura del genere non è privo di criticità. In particolare:

1 *Se si vuole tener traccia di tutti gli stati possibili, come si fa a capire quando sono stati tutti computati?*

Capire se sono stati generati tutti gli stati possibili è un compito che verrà delegato al simulatore, e comunque non sarà sempre possibile ottenere una

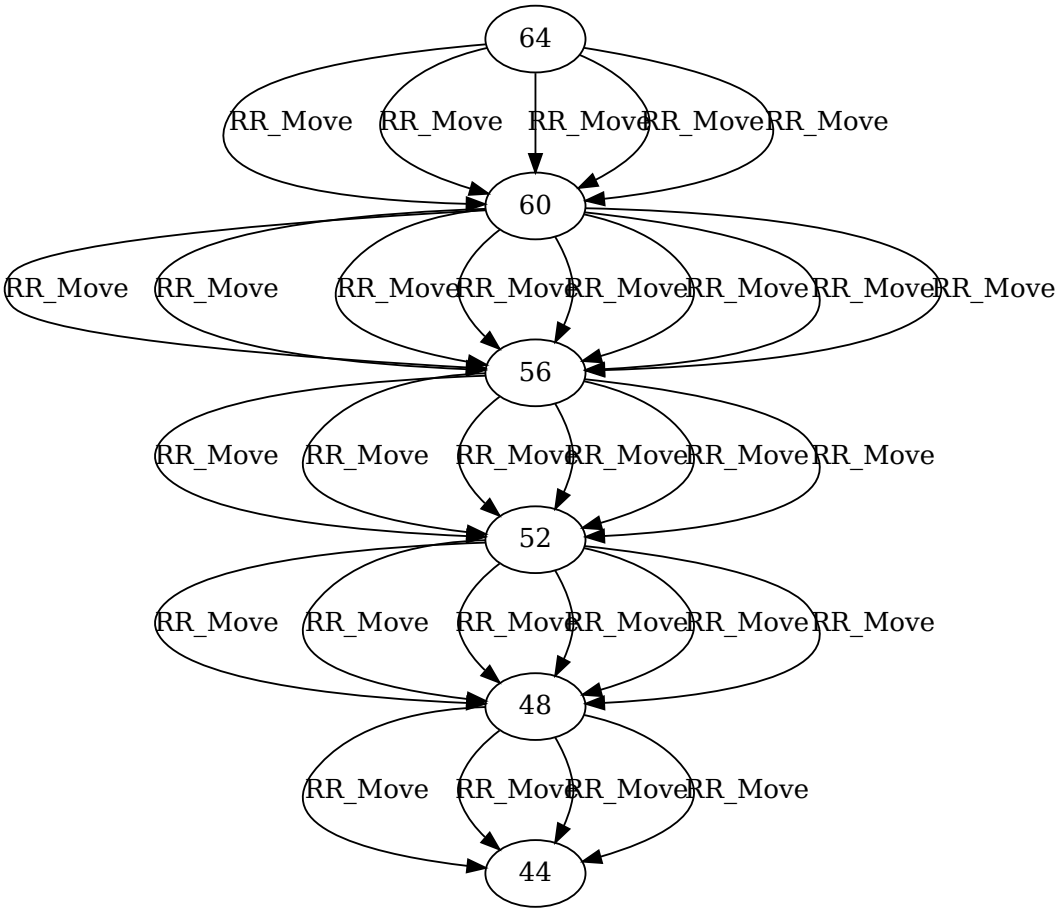


Figura 4.4: Esempio di grafo degli stati generati a partire dal bigrafo 4.1 a pagina 26.

risposta: per esempio, se si impiega `RandomBRS`, sarà impossibile dire quando sono stati ottenuti tutti gli stati.

2 *Se l'applicazione di alcune regole causa cicli nel grafo, come li si gestisce?*

Per quanto riguarda i cicli, prima di gestirli è necessario innanzitutto riconoscerli: per far ciò, ogni qualvolta viene computato un nuovo stato S a partire da uno T, prima di aggiungerlo al grafo si controllerà se ne è già presente uno *isomorfo* S'; in caso affermativo, invece di aggiungere S, si crea un arco da T a S', che può potenzialmente formare un ciclo. Nella figura 4.4 si può notare come gli stessi stati possano essere generati in più modi. Determinare se due bigrafi sono isomorfi è un altro problema non banale.

4.6.2 Implementazione

Il grafo degli stati è implementato dalla classe `BigStateGraph`, i nodi da `BSGNode` e gli archi da `BSGLink` (dove *"BSG"* sta per `BigStateGraph`). I `BSGNode`(s) sono coppie ⟨`Bigraph,[BSGLink]`⟩; I `BSGLink`(s) sono coppie ⟨`BSGNode,RewritingRule`⟩. Il grafo viene inizializzato con un singolo nodo, che rappresenta lo stato iniziale del sistema. I nodi sono mantenuti in una tabella di hash con chaining; la funzione di hash viene fornita dall'interfaccia `BigHashFunction`, la cui implementazione standard (`PlaceLinkHash`) genera un numero intero in base alla struttura del *place graph* e *link graph* del bigrafo. Per aggiungere un nodo si usa il metodo `applyRewritingRule(BSGNode node, RewritingRule rule, Bigraph state)`, che aggiunge un nuovo nodo figlio di `node`, con stato `state`, prodotto applicando la regola `rule` a `node`. Per verificare se ci sono nodi duplicati si svolge una ricerca in due fasi:

- Si verifica se c'è una collisione nella tabella di hash tra il nodo che si vuole aggiungere ed uno esistente;
- Se è così, si controlla se i due stati sono isomorfi sfruttando la classe `Isomorphism`, ed in particolare il suo metodo `areIsomorph(Bigraph,Bigraph)`.
- Se sono isomorfi il nuovo stato non viene aggiunto, ma viene inserito un arco "marchiato" come potenziale creatore di cicli; altrimenti viene aggiunto il nuovo nodo dove opportuno.

Isomorfismi tra bigrafi L'implementazione della classe `Isomorphism` è un argomento interessante ma corposo, che merita una discussione a parte; ci si limita a specificare che essa modella il problema dell'isomorfismo tra bigrafi come un'eguaglianza di flusso, risolto con tecniche di programmazione a vincoli. Per un'analisi dettagliata si rimanda il lettore a [1].

4.7 Comporre il Simulatore

A questo punto si dispone di tutti i moduli necessari a costruire un simulatore di sistemi bigrafici:

- JLibBig permette di rappresentare bigrafi e regole di riscrittura;
- `BRS` permette di gestire l'applicazione delle regole come si ritiene opportuno;

- `BigStateGraph` permette tenere traccia dell'evoluzione del sistema e di "ramificare" la simulazione in più percorsi;

Si definisce pertanto il modulo `Sim`, che sfrutta entrambi in maniera trasparente, fornendo un'interfaccia di più alto livello per simulare sistemi bigrafici. È definito come un'*interfaccia* che fornisce la API di base, e che va implementata in base alle necessità dell'utente (come `BRS`). In particolare tale API prevede la possibilità di fare un "passo" avanti nella simulazione (cioè applicare le regole ad uno o più stati raggiunti in base alle logiche definite dal programmatore) con `step()`, di sapere se sono stati esplorati tutti gli stati possibili del sistema con `simOver()`, o simulare l'evoluzione finché non si raggiunge un limite massimo di passi, o finché non si computano tutti gli stati, con `fullSim(int max)`. Di seguito vengono descritte alcune implementazioni già definite di `Sim`.

4.7.1 BreadthFirstSim

`BreadthFirstSim` è un simulatore che segue una logica *"breadth-first"*, ovvero ad ogni passo applica tutte le regole del BRS (anch'esso con strategia *breadth-first*) a tutti i nodi "foglia" (in senso lato) del grafo. Per far ciò mantiene una coda, in cui inizialmente è presente solo lo stato iniziale; una volta applicate le regole a tale stato, mette gli stati generati in fondo alla coda; ciò viene ripetuto per ogni nodo nella coda, finché essa rimane vuota: a questo punto tutti gli stati possibili sono stati generati.

4.7.2 RandomSim

`RandomSim` è un simulatore che segue un singolo "ramo" di evoluzione del sistema alla volta: opera in modo uguale a `BreadthFirstSim`, eccetto per il fatto che, ad ogni passo, sceglie pseudo-casualmente uno stato tra quelli generati e lo sposta al primo posto della coda. In tal modo, pur seguendo un percorso di evoluzione (pseudo)casuale, si tiene traccia degli altri percorsi possibili che si possono proseguire in seguito, permettendo di generare tutti gli stati.

4.7.3 TrueRandomSim

`TrueRandomSim` è un simulatore che segue un singolo "ramo" di evoluzione del sistema: per far ciò, ogni volta che vengono generati degli stati applicando le regole di riscrittura, ne viene scelto uno *pseudo*-casualmente e gli altri vengono scartati; in tal modo, il grafo degli stati si riduce ad una catena lineare di nodi con un solo figlio

(eccetto l'ultimo nodo). Si appoggia a `RandomBRS`. Non è possibile sapere quando si sono esplorati tutti gli stati possibili.

Esempi d'uso Alcuni esempi dell'impiego del simulatore sono esposti nel capitolo 6 a pagina 37.

5

Model Checking

Per *model checking* si intende la verifica di proprietà in sistemi reattivi bigrafici. Avere a disposizione un model checker vuol dire poter verificare se si presentano deteminate situazioni nel sistema, come stati interessanti o stati non validi. Potendosi appoggiare a un simulatore in grado di generare tutti gli stati possibili di un sistema, verificarne una proprietà si riduce a verificarla su tutti gli stati generati. Si può già, quindi, intuire che da simulatore a model checker il passo è (concettualmente) breve. In questo capitolo si descriverà sinteticamente l'implementazione di un model checker in grado di verificare alcune proprietà sui BRS. L'argomento è trattato più in dettaglio in [1].

5.1 Predicati

Innanzitutto è necessario definire un modo di rappresentare le proprietà da verificare. Per far ciò sono stati definiti dei predicati *ad-hoc*, basati sull'interfaccia `Predicate`, qui elencati:

- `TruePredicate` rappresenta la verità;
- `AndPredicate` rappresenta l'*and* logico fra due predicati;
- `NotPredicate` rappresenta la negazione di un predicato;
- `IsoPredicate` rappresenta la relazione "*è isomorfo a*" tra due bigrafi;
- `WarioPredicate` [1] è un predicato sui *match* che, dati due bigrafi F e G e tre predicati ϕ_i $(i = 0, 1, 2)$ è definito come

[1] Il nome è nato come un modo simpatico di chiamare quello che altrimenti sarebbe stato il `WPredicate`, dove la W in realtà è la M di "match" ruotata di 180°.

$$G \models \mathbb{W}_F \iff \exists C, D \text{ t.c.}$$
$$G = C \circ (F \otimes id_I) \circ D \wedge (C \models \phi_0 \wedge F \models \phi_1 \wedge D \models \phi_2)$$

Questo insieme di predicati risulta sufficiente a descrivere alcune proprietà interessanti.

5.2 Model Checker

La classe `ModelChecker` non fa altro che usare un simulatore (il default è `BreadthFirstSim`) per ottenere le evoluzioni di un sistema, verificando la soddisfazione di un predicato su ogni stato computato. Se esiste uno stato per cui il predicato è soddisfatto, la simulazione termina e la proprietà è verificata. Il model checker ha un tetto massimo (modificabile) di passi di simulazione, in modo da evitare la non-terminazione qualora si impieghino simulatori che sanno quando hanno generato tutti gli stati (come `TrueRandomSim`). Degli esempi d'uso verranno illustrati nel prossimo capitolo.

6
Esempi e applicazioni

In questo capitolo verranno proposti vari esempi per cui si definirà segnatura, stato iniziale, BRS ed eventualmente proprietà da verificare.

6.1 Viaggio in macchina

Si vuole rappresentare un'automobile con una scorta di carburante limitata, che deve raggiungere una locazione prestabilita.

6.1.1 Segnatura

La segnatura è così definita:

- Il controllo *car* rappresenta l'automobile, e ha una porta che si collega alla sua destinazione;
- *fuel* rappresenta un'unità di carburante;
- *place* rappresenta una località verso cui l'auto può spostarsi, se c'è una strada collegata alla sua porta;
- *road* rappresenta una strada che collega una località ad un'altra;
- *target* indica la destinazione dell'auto.

Codice 6.1: Segnatura del "Viaggio in macchina"

```
1 private static Signature generateSignature() {
2      SignatureBuilder sb = new SignatureBuilder();
3
4      sb.add("car", true, 1);
5      sb.add("fuel", true, 0);
6      sb.add("place", true, 1);
7      sb.add("road", true, 1);
```

```
8       sb.add("target", true, 1);
9
10      return sb.makeSignature("Car Signature");
11  }
```

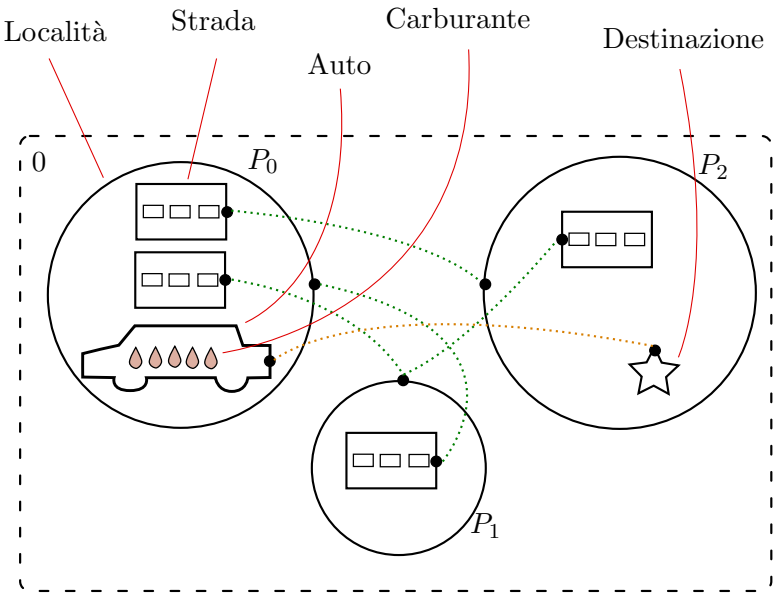


Figura 6.1: Esempio di stato iniziale.

6.1.2 Regole di riscrittura

Questo semplice esempio ha una sola regola di riscrittura, chiamata `RR_Move`, che permette all'auto di muoversi "alla cieca", ovvero senza necessariamente avvicinarsi alla sua destinazione (figura 6.2 a fronte).

Osservazione. Con questa sola regola le strade sono effettivamente a senso unico: per renderle a doppio senso di marcia si può aggiungere una seconda strada in senso opposto (per ogni strada), oppure aggiungere una seconda regola duale a `RR_Move`.

6.1.3 Verifica di proprietà

Una proprietà che si presta ad essere verificata è la raggiungibilità della destinazione: esiste uno stato in cui l'auto ha raggiunto il suo obiettivo? Un semplice test viene preparato nel codice 6.2 a pagina 40; viene definito il bigrafo B (in figura 6.3 a fronte) che rappresenta il raggiungimento della destinazione; al `ModelChecker` viene passato un predicato $\mathcal{W}_B$ che è soddisfatto se B ha un match nello stato del sistema.

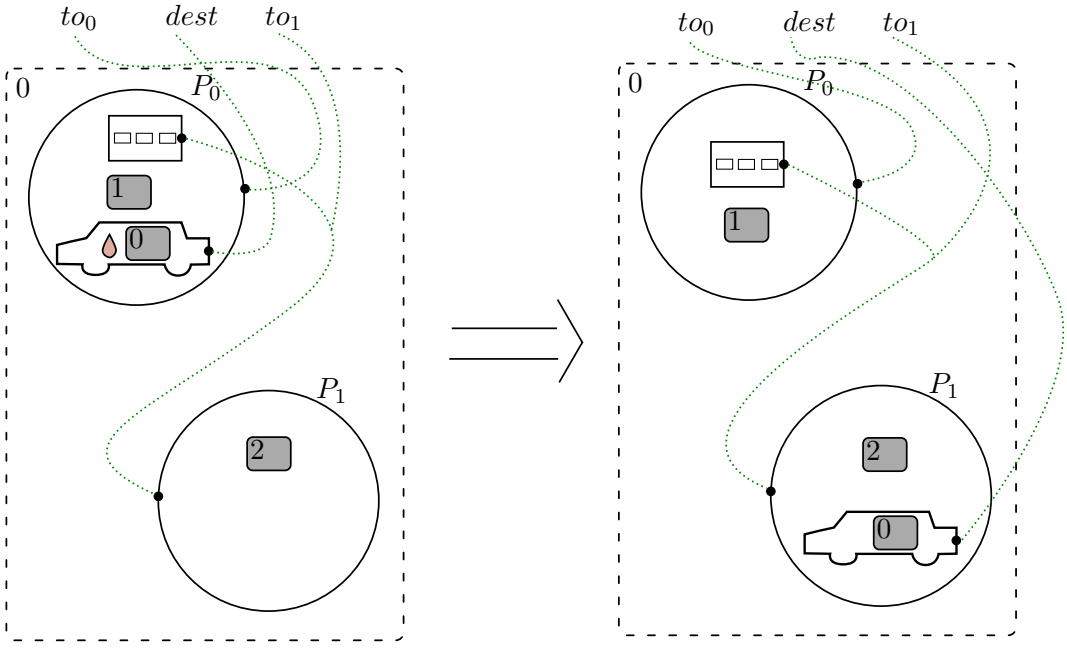


Figura 6.2: Regola `RR_Move`.

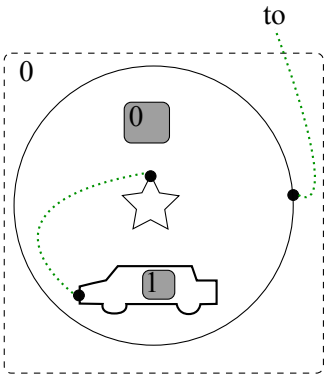


Figura 6.3: Bigrafo obiettivo per la verifica della raggiungibilità.

Codice 6.2: Verifica della raggiungibilità.

```
1     private void modelCheckerTest() {
2         Bigraph state = generateLevel(8);
3         Predicate p = new WarioPredicate(goalReached(), new
              TruePredicate(),
4                 new TruePredicate(), new TruePredicate());
5         ModelChecker mc = new ModelChecker(new BreadthFirstSim(
              state, RULES), p);
6         System.out.print("Is the destination reachable? ");
7         if (mc.modelCheck()) {
8             System.out.println("YES.");
9         } else {
10            System.out.println("NO");
11        }
12    }
```

Codice 6.3: Definizione del bigrafo obiettivo (figura 6.3 nella pagina precedente)

```
1     private static Bigraph goalReached() {
2         BigraphBuilder bb = new BigraphBuilder(SIGNATURE);
3
4         Root root = bb.addRoot();
5
6         OuterName to = bb.addOuterName("to");
7         Node place = bb.addNode("place", root, to);
8         bb.addSite(place);
9         Node tgt = bb.addNode("target", place);
10        Node car = bb.addNode("car", place, tgt.getPort(0).
              getHandle());
11        bb.addSite(car);
12
13        return bb.makeBigraph(true);
14    }
```

7

Conclusioni

Bibliografia

[1] Luca Geatti. ≪Verifica di proprietà locali su BRS≫. Tesi di Laurea Triennale in Tecnologie Web e Multimediali. Università degli Studi di Udine, 2015.

[2] Robin Milner. *The Space and Motion of Communicating Agents*. 1st. New York, NY, USA: Cambridge University Press, 2009. ISBN: 0521738334, 9780521738330.