

Linguaggi e Modelli Computazionali LM

Il set di tool GROOVE

Cialotti Andrea Giulia

27 Luglio 2012

Introduzione

Al giorno d'oggi, la maggior parte dei sistemi software sono concorrenti (e distribuiti) ed hanno un comportamento non deterministico perciò, in fase di verifica, il classico approccio del testing (dati alcuni input vedere se gli output ottenuti concordano con gli output desiderati) non è più sufficiente al fine di garantirne completamente la correttezza. I motivi alla base di ciò sono i seguenti:

- il numero dei test che coprono tutti i possibili casi è enorme (spesso infinito);
- ci sono errori che capitano in casi talmente eccezionali che non vengono identificati con i testcase tradizionali.

Quindi sono stati introdotti i metodi di verifica formali, i quali controllano se il sistema software rispetta i requisiti e ha certe proprietà sulla base del modello stesso del sistema. Uno di questi metodi è il model checking, il quale verifica se il sistema rispetta certe proprietà in ogni suo possibile scenario e stato. Questo approccio in teoria sarebbe applicabile per tutti i sistemi che hanno un numero di stati finito, però nella pratica tutti i tool sviluppati per fare model checking (tra i quali SPIN), si scontrano con il problema della crescita esponenziale dello spazio degli stati da esplorare, in letteratura "The State Explosion Problem". Per questo motivo solitamente si cerca di limitare la ricerca, o fare delle semplificazioni, o astrazioni sul modello del sistema da testare. Per fare model checking le proprietà del sistema da testare devono essere espresse in un linguaggio formale, tipicamente si usano linguaggi testuali come le logiche matematiche oppure linguaggi grafici come le reti di Petri, che comportano i seguenti svantaggi:

- le logiche non sono semplici da usare;
- le reti di Petri richiedono che il numero di stati di un sistema sia noto a priori, e non può cambiare a run-time, quindi non sono adatte per modellare sistemi di reti la cui struttura evolve dinamicamente.

Le **Graph Grammar** (Grammatiche per Grafi), introdotte nei primi anni 70' con lo scopo di generalizzare le grammatiche testuali di Chomsky, sono un potente formalismo per modellare sistemi concorrenti e distribuiti, e rispondono alle esigenze discusse precedentemente. Infatti forniscono un modo intuitivo e formale per specificare come evolve un sistema, senza necessariamente conoscere a priori il numero degli stati poichè la struttura del grafo può variare dinamicamente.

Definizione 1 (Graph Grammar). *Si definisce Graph Grammar o grammatica per grafi una coppia $GG = (G_0; P)$, dove*

- G_0 viene chiamato grafo di partenza ;
- P è l'insieme delle regole di produzione ;

Come vengono descritte e applicate le regole di produzione dipende dall'approccio adottato, nell'Appendice A viene discusso l'approccio algebrico in particolare Singolo Pushout (SPO) e Doppio Pushout (DPO). Negli ultimi anni sono stati sviluppati anche tool che consentono di automatizzare il model checking utilizzando le grammatiche per grafi, uno di questi tool è GROOVE sviluppato dall'università di Twente.

Il lavoro che segue è stato diviso in due parti. Una prima in cui si introducono alcune nozioni base del tool, una seconda in cui si argomenta il caso di studio, che riguarda la modellazione di un algoritmo di Spatial Computing su reti la cui tipologia evolve dinamicamente.

1 GROOVE: GRaphs for Object-Oriented VErification

Il set di tool Groove, implementato interamente in Java è composto da:

- **Editor.** Un editor con un interfaccia grafica per creare regole e grafi host.
- **Generator.** Un tool a linea di comando che genera lo spazio degli stati di un GPS (Graph Production System). Lo spazio degli stati viene archiviato come Labelled Transition System(LTS), dove ogni stato è un grafo e le transizioni sono etichettate dall'applicazione delle regole. La strategia con la quale viene esplorato lo spazio degli stati (depth-first, breadth-first, etc) può essere settata come parametro.
- **Model Checker.** Un tool a linea di comando che verifica se proprietà espresse in CTL sono valide in un LTS prodotto dal Generator. Se una proprietà non è valida, viene fornito un controesempio.
- **Simulator.** Un tool con interfaccia grafica che integra le funzionalità dell'Editor, del Generator, e del Model Checker. Inoltre il Simulator permette all'utente di esplorare interattivamente l'LTS, applicando manualmente le regole al grafo host .
- **Imager.** Un tool a linea di comando o con GUI che supporta varie conversioni da grafici e regole in GROOVE ad altri formati visuali, fra i quali jpeg, png, eps, o tikz(per Latex)

1.1 Rappresentazione e sintassi di grafi e regole con GROOVE

Grafi

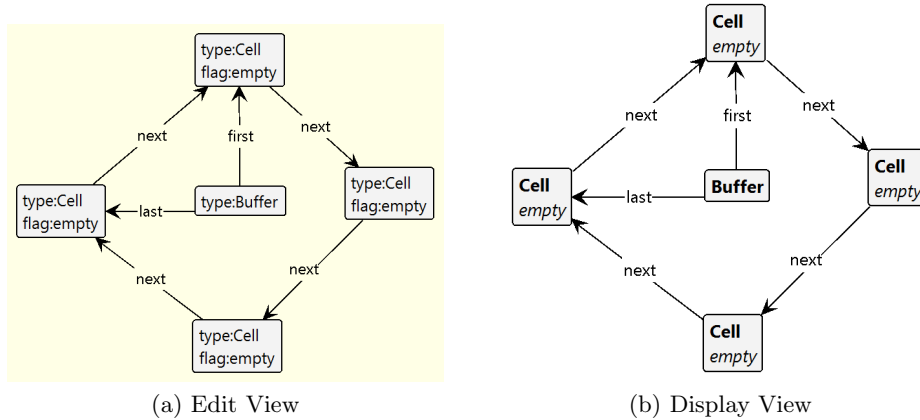


Figura 1: Grafo start dell'esempio del buffer circolare (uno degli esempi forniti all'interno del tool).

GROOVE utilizza grafi direzionati etichettati, i cui vertici vengono rappresentati con rettangoli arrotondati con iscritte le etichette, mentre gli archi con frecce tra i vertici con sopra le etichette. Queste sono espresse in formato testuale e sintatticamente sono composte da due elementi: il primo è un *prefisso*, di solito formato da un identificatore seguito dai due punti ":". Le etichette dei vertici possono avere due prefissi:

- **tipo** (prefisso "type:"). Le etichette tipo sono parzialmente ordinate dal subtyping (spesso chiamato ereditarietà). Questo influisce sul matching delle regole: un etichetta tipo in una regola fa matching con tutte le etichette di sottotipi (quelle che sono inferiori nell'ordinamento parziale) dell'host graph. Inoltre se viene utilizzato un grafo tipato, le etichette type devono essere uniche: ogni nodo deve avere esattamente una etichetta tipo.

- **flag** (prefisso "flag:"). Sono sorta di attributi del nodo che indicano che il nodo ha una determinata proprietà, possono essere anche aggiunti dinamicamente come verrà discusso in seguito, per esempio in fig. 1 nei nodi `Cell` vi è il flag *empty* che indica che la cella del Buffer è vuota. Si noti che un nodo può anche più di un flag contemporaneamente;

Le etichette degli archi sono semplicemente identificatori espressi con stringhe di caratteri che cominciano con una lettera o un underscore, e contengono solo lettere, cifre, underscore e simboli del dollaro.

Si noti che la sintassi visuale delle etichette in fase di editing (Edit View) è leggermente diversa da quella in fase di visualizzazione (Display View), ad esempio come si vede in fig.1 le etichette dei nodi con prefisso `type:` vengono visualizzate in grassetto, mentre quelle con prefisso `flag:` vengono visualizzate in corsivo e in entrambi i casi il prefisso viene omissso.

Regole

In GROOVE ogni regola ha un nome espresso da una sequenza di caratteri (lettere, cifre, underscore o simbolo del dollaro) che inizia con lettera minuscola. E' possibile avere una gerarchia delle regole, simile alla struttura dei package di Java, quindi usando nomi separati dal punto, per esempio "`a.b`" indica la regola "`b`" che si trova nel package "`a`". Questo meccanismo ha il solo scopo di permettere una gestione comoda di un grande insieme di regole ma non influenza minimamente il processo di applicazione delle regole.

Formalmente, le regole sono composte da tre parti: parte sinistra (LHS), parte destra (RHS), e condizioni di applicazione negative (NACs), ognuna delle quali viene rappresentata da un grafo e sono connesse tra loro tramite morfismi (per ulteriori dettagli in merito si faccia riferimento all'Appendice A).

GROOVE rappresenta le regole in un unico grafico, utilizzando i colori per caratterizzare i diversi ruoli che un elemento grafico può avere. Ora verranno descritti nel dettaglio i ruoli e loro sintassi visuale. GROOVE permette di distinguere se ci si trova in Edit View o Display View con il colore dello sfondo il quale nel primo caso è giallino mentre nel secondo è bianco.

Reader.

Sono elementi grafici che si trovano sia nella parte sinistra che nella parte destra della regola. I vertici sono visualizzati con rettangoli arrotondati e gli archi con frecce di colore nero. In fase di editing si può indicare esplicitamente che un arco è un reader usando il prefisso "`use:`" (opzionale) seguito dall'etichetta, mentre per le etichette dei nodi se non vi è nessun prefisso che indichi esplicitamente che il nodo ha uno degli altri ruoli, come verrà spiegato in seguito, allora il nodo di default avrà il ruolo di reader.

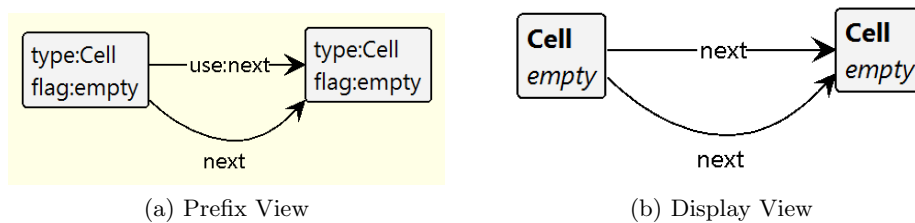


Figura 2: Nodo con ruolo Reader

Le differenze tra le etichette in fase di editing e visualizzazione sono le stesse discusse precedentemente, per il grafo che rappresenta lo stato. La cosa importante da notare nell'esempio riportato è che l'utilizzo o l'omissione di "`use:`" come prefisso nelle etichette degli archi porta ai medesimi effetti.

Eraser.

Sono elementi grafici presenti solo nella parte sinistra della regola, ciò significa che il grafo su cui si vuole applicare la regola deve fare matching con questi elementi, i quali verranno poi cancellati quando la regola sarà applicata. I vertici sono rappresentati con rettangoli arrotondati e gli archi con frecce tratteggiate di colore blu. In fase di editing si deve indicare che un elemento è un eraser con il prefisso "del:", il quale compare da solo nei nodi, mentre viene seguito dall'etichetta negli archi.



Figura 3: Nodo con ruolo Eraser

Si noti che nell'esempio in fig.3 nell'arco non è necessario specificare esplicitamente che è un eraser, poichè essendo un arco diretto verso un nodo di tipo eraser GROOVE lo interpreta automaticamente come eraser. Inoltre GROOVE per i nodi presenta un accorgimento in più, è possibile specificare non solo che un nodo sia un eraser, ma anche che solo un'etichetta all'interno del nodo sia un eraser e quindi all'applicazione della regola venga rimossa solo questa, ma non l'intero nodo. Nel primo caso sintatticamente si scrive nell'etichetta "del:", poi si va a capo (shift+invio), e si scrive il resto delle etichette del nodo, nell'esempio di figura "type:Object". Nel secondo caso si scrive l'etichetta "del:empty" tutta in un'unica linea e il risultato è quello mostrato nella figura che segue. Quando si specifica che un'etichetta del nodo è eraser, in fase di visualizzazione il prefisso dell'etichetta viene sostituito con il "-", come si vede in figura per l'etichetta "empty".

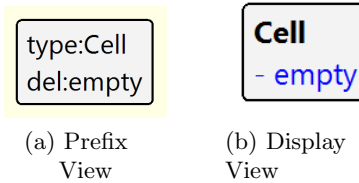


Figura 4: Flag con ruolo Eraser

Creator.

Sono elementi grafici presenti solo nella parte destra della regola, ciò significa che verranno creati quando la regola sarà applicata. I vertici sono visualizzati con rettangoli arrotondati e gli archi con frecce in grassetto di colore verde. In fase di editing si deve indicare che un elemento è un creator con il prefisso "new:", il quale compare allo stesso modo del "del:" visto sopra.



Figura 5: Nodo con ruolo Creator

In figura viene riportato l'esempio analogo a quello riportato per l'eraser, le considerazioni sono le stesse, tranne che questa volta nell'etichetta creator in fase di visualizzazione il prefisso diventa "+", come si vede nella figura che segue.

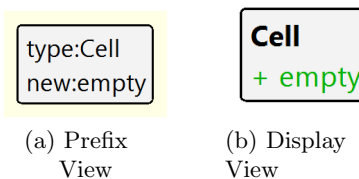


Figura 6: Flag con ruolo Creator

Embargo.

Sono elementi grafici che rappresentano le condizioni di applicazione negative della regola, e non si trovano nella parte sinistra della regola, ciò significa che se il grafo su cui si vuole applicare la regola fa match con questi elementi, la regola non può essere applicata. Nel caso in cui, il matching ha esito negativo, e la regola può essere applicata, questi elementi non influiscono ulteriormente. I vertici sono visualizzati con rettangoli arrotondati e gli archi con frecce in grassetto tratteggiate di colore rosso. In fase di editing si deve indicare che un elemento è un embargoes con il prefisso "not:", il quale compare allo stesso modo del "del:" visto sopra.



Figura 7: Nodo con ruolo Embargo

Anche in questo caso le considerazioni sono analoghe alle precedenti, ma questa volta nell'etichetta embargo in fase di visualizzazione il prefisso diventa "!", come si vede nella figura che segue.

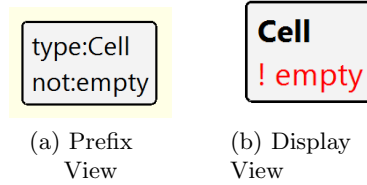


Figura 8: Flag con ruolo Embargo

Etichette Speciali utilizzabili nelle Regole

- **Negazioni.** Si possono usare solo sugli archi e funzionano allo stesso modo degli embargo, impedendo l'applicazione della regola qualora nel grafo sia presente l'arco, il flag o il tipo specificato. La sintassi dell'etichette è "!". L'aspetto più interessante di queste etichette è che si possono combinare con gli embargo per ottenere una doppia negazione. Per esempio, la regola che segue può essere applicata solo se il MainTask non ha già terminato e non c'è nessun SubTask che non ha finito, in pratica se tutti i SubTask sono terminati. Le negazioni possono essere usate solo all'interno di archi di tipo embargo o reader.

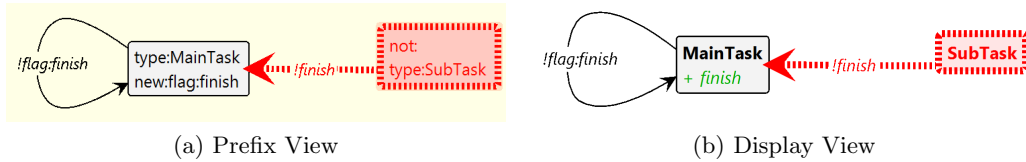


Figura 9: Esempio di Negazione

- **Ugualianza.** L'etichetta "=" usata da sola all'interno di un arco tra due nodi, indica semplicemente che quei due nodi rappresentano lo stesso nodo, anche se visualizzati separatamente. Mentre se usata in combinazione con archi creator, embargo da luogo a etichette molto interessanti discusse qui di seguito.
- **Merger.** L'etichetta "new:=" usata all'interno di un arco consente di unire i due nodi in un unico nodo risultante che avrà tutti gli archi e le etichette di entrambi i nodi, per questo motivo in qualunque direzione venga usato per gli stessi nodi, si otterrà sempre lo stesso risultato. In figura viene mostrato un esempio di applicazione di una regola con arco merger.

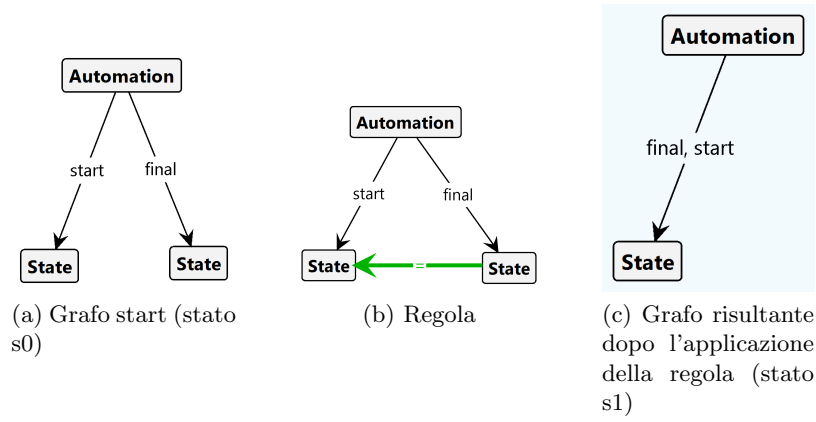


Figura 10: Esempio di applicazione di una regola con etichetta di tipo Merger.

Si noti nella fig.10(c) lo sfondo di colore azzurro, il quale indica caratterizza il pannello LTS, nel quale si può esplorare il Label Transition System che rappresenta lo spazio degli stati, ulteriori dettagli si possono trovare nella sezione 1.3.

- **Injectivity.** In generale il matching di una regola non è iniettivo, così LHS (parte sinistra di una regola, si veda l'appendice A.2 per approfondimenti) distinti possono fare match con lo stesso grafo. L'etichetta " $!=$ " o " $\text{not} :=$ " impone di far rispettare localmente l'injectività, e quindi i due nodi ai capi di questo arco (non direzionato) avranno sempre immagini diverse. Nella figura seguente viene mostrata una regola che dice che due persone si possono sposare solo se non hanno gli stessi genitori.

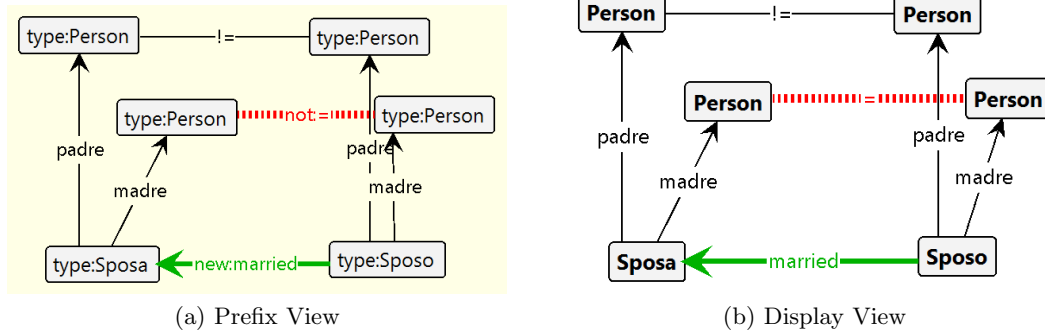


Figura 11: Esempio di Injectivity

Inoltre utilizzando questa etichetta insieme ai nodi embargo è possibile impostare dei veri e propri contatori. Nella figura che segue viene mostrata una regola che dice che il filosofo può mangiare solo se ha esattamente due forchette e un piatto. In particolare l'Injectivity garantisce che le forchette non siano meno di due, mentre l'embargo garantisce che non siano più, lo stesso discorso vale per il piatto.

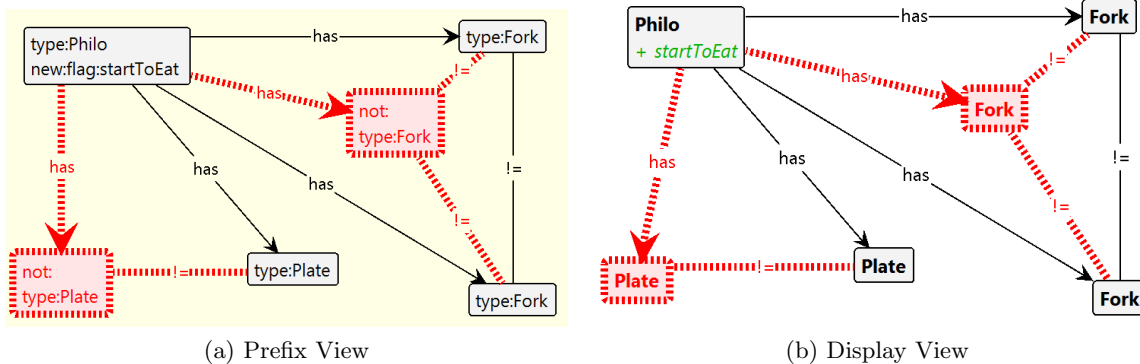


Figura 12: Esempio di uso dell'Injectivity per realizzare contatori.

1.2 Procedimento di applicazione di una regola

Date le premesse suddesse, per applicare una regola si procede così:

1. Si cerca almeno un match con i **reader** e gli **eraser**;
2. Si verifica che non vi sia neanche un oggetto (nodo, arco, flag) che fa match con gli **embargo**;
3. Si eliminano gli **eraser**;
4. Si aggiungono i **creator**.

Tenendo presente che:

- se non viene trovato nessun match nel passo 1, la regola non può essere applicata;
- se vengono trovati più di un match nel passo 1, ne viene scelto uno in modo non deterministico;
- se viene trovato almeno un match nel passo 2, la regola non può essere applicata.

Inoltre GROOVE permette di aggiungere ulteriori vincoli sull'applicazione delle regole utilizzando il linguaggio Control, per esempio è possibile specificare che due regole devono essere applicate in modo mutuamente esclusivo, oppure che una regola quando viene scelta deve continuare ad essere applicata fino a quando è possibile, per ulteriori dettagli in merito si rimanda al manuale.

1.3 Label Transition System

Il Label Transition System generato da GROOVE, rappresenta tutte le possibili evoluzioni del grafo iniziale in seguito all'applicazione delle regole.

Si parte dallo stato iniziale, che rappresenta il grafo di partenza, poi si genera uno stato per ogni grafo risultante dall'utilizzo delle regole applicabili sul grafo stesso. Le transizioni tra gli stati rappresentano le regole applicate. Si procede ricorsivamente per tutti i nuovi grafi generati, fino a quando vengono raggiunti gli stati finali, cioè quegli stati dove non vi è più nessuna regola applicabile. GROOVE permette di personalizzare la generazione dell'LTS scegliendo sia il tipo di esplorazione (per es. depth-first, breadth-first, linear), nel senso di ordine nel quale esaminare gli stati, sia la codizione di arresto della generazione, come per esempio se viene raggiunto uno o tutti gli stati finali.

1.4 File generati da GROOVE

In seguito alla modellazione di una grammatica quando l'utente salva, GROOVE genera una serie di file che hanno estensioni diverse per distinguere tra regole (.gtr) e grafi (.gts), contenenti codice GXL, un sottoinsieme del XML ed è lo standard per i grafi. Inoltre GROOVE permette di salvare anche l'LTS però questa volta l'estensione è .gxl. Di seguito vengono riportati alcune righe di un esempio di regola, uno di grafo e uno di LTS, presi dal caso di studio.

graphTest.gst

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd">
  <graph role="graph" edgeids="false" edgemode="directed" id="graphTest">
    <edge from="n0" to="n0">
      <attr name="label">
        <string>flag:start</string>
      </attr>
    </edge>
    <edge from="n0" to="n4">
      <attr name="label">
        <string>val</string>
      </attr>
    </edge>
    <edge from="n0" to="n1">
      <attr name="label">
        <string>link</string>
      </attr>
    </edge>
  </graph>
</gxl>
```

propagationVal.gtr

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd">
  <graph role="rule" edgeids="false" edgemode="directed"
    id="gradientCalculation.propagationVal">
    <attr name="priority">
      <string>1</string>
    </attr>
    <edge from="n7" to="n6">
      <attr name="label">
        <string>arg:1</string>
      </attr>
    </edge>
    <edge from="n7" to="n3">
      <attr name="label">
        <string>arg:0</string>
      </attr>
    </edge>
    <edge from="n7" to="n8">
      <attr name="label">
        <string>int:add</string>
      </attr>
    </edge>
    <edge from="n8" to="n8">
      <attr name="label">
        <string>int:</string>
      </attr>
    </edge>
  </graph>
</gxl>
```

lts.gxl

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd">
  <graph role="none" edgeids="false" edgemode="directed" id="lts">
    <node id="n0"/>
    <node id="n1"/>
    <node id="n2"/>
    <node id="n3"/>
    <node id="n4"/>
    <node id="n5"/>
    <node id="n6"/>
    <node id="n7"/>
    <node id="n8"/>
    <node id="n9"/>
    <edge from="n0" to="n1">
      <attr name="label">
        <string>createStartGraph.addCoupleNodeWithStart</string>
      </attr>
    </edge>
    <edge from="n1" to="n2">
      <attr name="label">
        <string>createStartGraph.addNodeWithRandVal</string>
      </attr>
    </edge>
  </graph>
</gxl>
```

2 Caso di studio: Modello di un algoritmo di Spatial Computing con GROOVE.

Il caso di studio ha l'obiettivo di capire se il tool GROOVE può essere utilizzato per modellare algoritmi nell'ambito della Spatial Computing, alcuni esempi sono:

- algoritmi per il calcolo del gradiente in una rete di nodi;
- algoritmi per il calcolo della distanza minima tra due nodi;
- algoritmi per il calcolo del cammino minimo tra due nodi;
- algoritmi per determinare quali nodi della rete si trovano ad una distanza inferiore ad una soglia dal cammino minimo tra due nodi.

In particolare si prenderanno in considerazione reti di nodi dinamiche nelle quali possono essere aggiunti nuovi nodi e/o collegamenti a run-time. Prima verrà discussa la modellazione di un algoritmo base per il calcolo del gradiente, in secondo luogo verrà discussa la modellazione per generare reti di partenza in modo casuale, infine viene discussa l'estensione dell'algoritmo al calcolo del cammino minimo dalla sorgente ad un nodo destinazione. In tutti i casi si prenderanno in considerazione reti con tutti gli archi di peso uguale a 1.

2.1 Algoritmo base per il calcolo del gradiente

Data una rete di nodi e un nodo sorgente i_s , per gradiente in un nodo i_i si intende il cammino minimo in termini di numero di passi/link che intercorre tra i_i e il nodo sorgente i_s . Nell'algoritmo inizialmente tutti i nodi della rete hanno un valore di gradiente molto alto (di solito infinito) eccetto il nodo sorgente il quale ha il valore 0. Ad ogni passo, se due vicini hanno valore che differisce per più di 1, allora il più grande dei due diventa il più piccolo più 1. Si procede in questo modo fino a quando tutti nodi pur ricevendo i valori dai vicini, non modificano più il proprio valore raggiungendo la configurazione finale, nella quale il valore di ogni nodo i_i è il gradiente del nodo sorgente.

Nella figura seguente viene riportata una rete nella configurazione iniziale e la rete risultante dopo la convergenza dell'algoritmo.

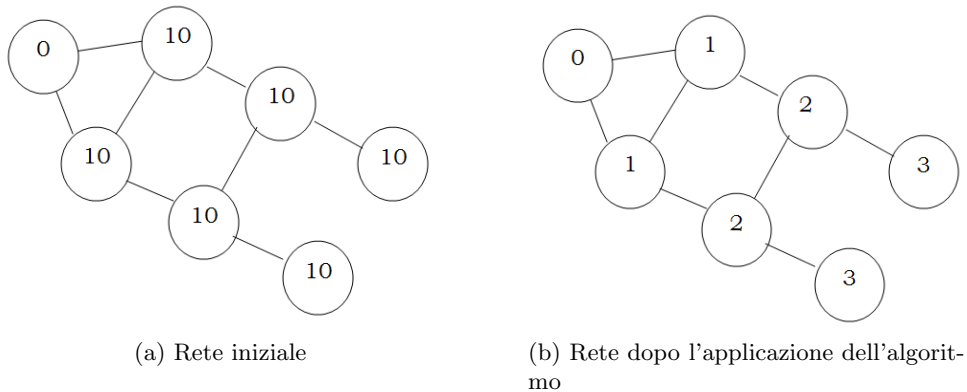


Figura 13: Esempio di applicazione dell'algoritmo del gradiente.

Modellazione delle Regole

La modellazione dell'algoritmo con GROOVE consiste nell'esprimere le regole con le quali evolverà la rete iniziale. Dati due nodi della rete i_i e i_j collegati direttamente tra loro mediante un arco che va da i_i a i_j , la regola di propagazione si esprime mediante due regole duali:

1. se $val_i > val_j + 1 \rightarrow val_i = val_j + 1$
2. se $val_j > val_i + 1 \rightarrow val_j = val_i + 1$

La regola 1 in GROOVE viene modellata dal grafico di fig.14a.

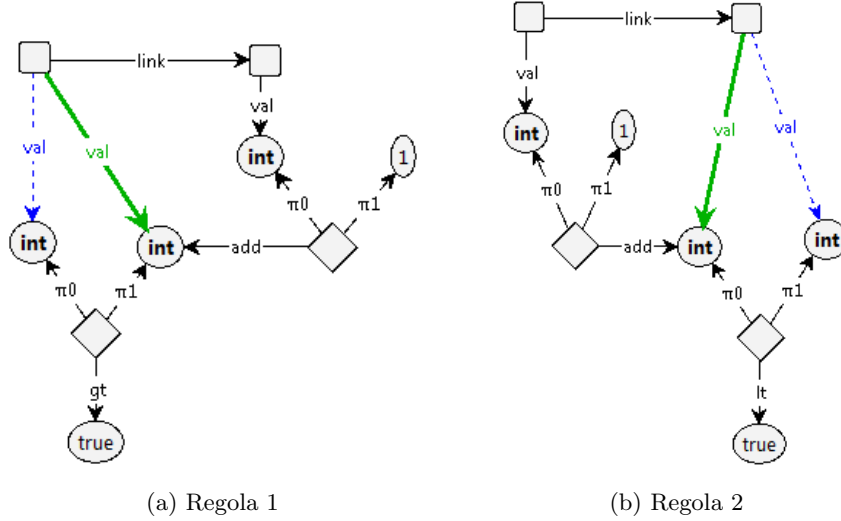


Figura 14: Modellazione regola 1 e regola 2 con GROOVE.

Gli elementi colorati di **nero** hanno il ruolo di **reader**, e sono presenti sia nella parte sinistra che nella parte destra della regola, mentre gli elementi in **blu** hanno il ruolo di **eraser** e sono presenti solo nella parte sinistra della regola e in seguito all'applicazione della stessa verranno eliminati. Infine gli elementi in **verde** sono i **creator** e sono presenti solo nella parte destra della regola e verranno creati in seguito all'applicazione della regola.

Applicando la regola prima di tutto si verifica se è presente un matching con il grafo, quindi se sono presenti nel grafo due nodi collegati mediante un arco `link` nel quale il nodo sorgente ha il valore maggiore del secondo aumentato di 1. Si noti che `gt` sta per "`int(i1, i2) : greater than`" cioè un operatore che confronta i due interi e restituisce `true` se i_1 è strettamente maggiore di i_2 , inoltre il nodo di forma romboidale è un nodo `prod`: che indica tuple di argomenti, mentre π_0 e π_1 sono gli archi che indicano gli argomenti che appartengono alla tupla, sui quali si andrà ad applicare l'operatore specificato. Se l'esito del confronto è uguale al booleano specificato, cioè `true` allora il matching è positivo, altrimenti il matching ha esito negativo e la regola non può essere applicata. L'altro operatore "`int(i1, i2) : add`", invece è l'operatore di somma tra due interi infatti, a differenza del precedente, è diretto verso un intero. Nel caso in cui venga trovato almeno un matching che rispetta i vincoli appena discussi, (se sono più di uno, ne viene scelto uno in modo non deterministico), nel nodo sorgente dell'arco `link` viene eliminato l'arco `val` verso il proprio valore. Infine viene creato un nuovo arco `val` come specificato dal **creator**.

Nella figura 14b viene riportata la regola 2 che è duale. I commenti sono simili, tranne che in questo caso l'operatore di confronto è `lt` cioè "`int(i1, i2) : less than`".

Se non vengono aggiunte altre regole, preso un qualunque grafo iniziale (sia con tutti gli archi birezionali o con archi tra due nodi con una sola direzione) applicando le regole suddette si converge sempre verso un'unica soluzione, ma dal momento che si vuole sfruttare la dinamicità di GROOVE sono state introdotte due regole che modificano la topologia della rete dello stato iniziale, una

aggiungendo un nuovo nodo, mentre l'altra aggiungendo un nuovo percorso.

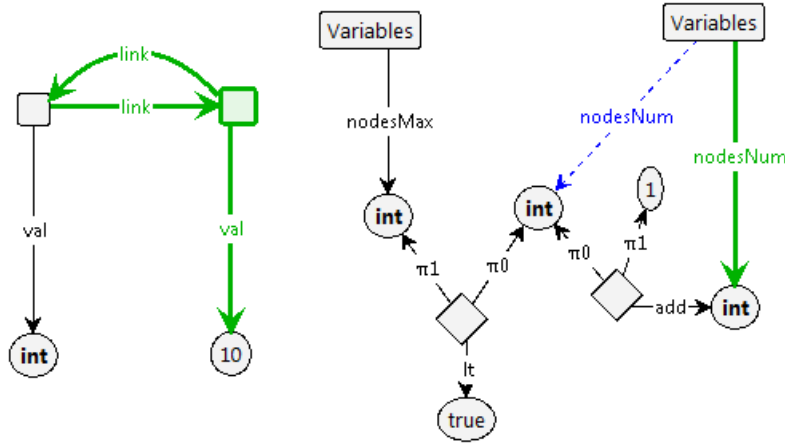


Figura 15: Rappresentazione in GROOVE della regola addNewNode.

Nella figura 15 è rappresentata la regola addNewNode, la quale se non è stato raggiunto il numero di nodi massimo, prende un qualsiasi nodo che ha un valore e aggiunge un nodo collegandolo direttamente a questo con due archi link uno in ogni direzione. Si noti che se non fosse stato specificato che il nodo a cui collegare il nuovo nodo è un qualsiasi nodo con un valore, la regola avrebbe fatto match con qualsiasi nodo, anche con il nodo Variables collegando il nuovo nodo a questo. Inoltre è stato scelto di assegnare un valore di default, perchè GROOVE non ha nessun meccanismo per generare un intero casuale, quindi l'alternativa sarebbe stata tenere un insieme di numeri da assegnare e specificare di prenderne uno. Questo avrebbe comportato la presenza di un match anche per ogni possibile valore aumentando notevolmente la cardinalità delle regole applicabili ad ogni stato. Una volta aggiunto il nuovo nodo viene incrementato il numero di nodi di 1.

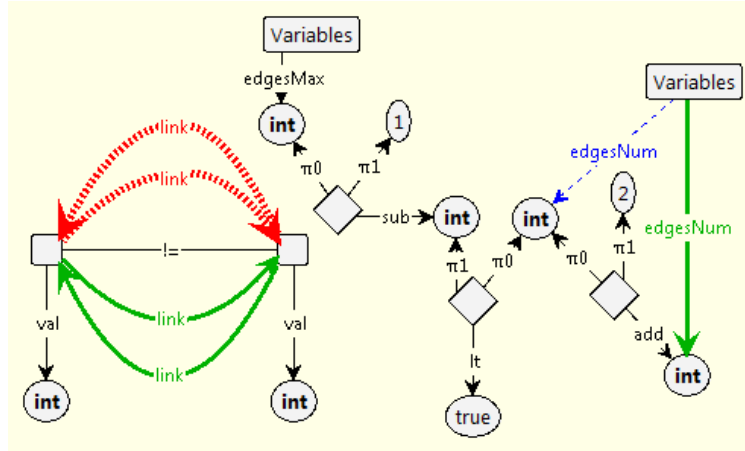


Figura 16: Rappresentazione in GROOVE della regola addNewPath.

Infine in figura 16 viene riportata la regola addNewPathB, la quale se non è stato raggiunto il numero massimo di percorsi, aggiunge un nuovo collegamento non direzionato (quindi due archi link) tra due nodi che non sono già direttamente collegati. In questa regola viene utilizzato anche un arco con ruolo **embargo**, che caratterizza gli elementi che fanno parte delle NACs della regola, ovvero condizioni di applicazione negative (per ulteriori dettagli si faccia riferimento all'Appendice A.2). Nel caso in cui nel grafo è presente almeno un match con l'elemento con ruolo **embargo**, la

regola non può essere applicata. Inoltre viene utilizzata la negazione $\neq$, che specifica che quei due nodi devono essere diversi, quindi non può essere aggiunto un nuovo arco `link` che parta da un nodo e termini sempre nello stesso nodo (per ulteriori dettagli si veda la sezione 1). Infine vanno fatte due precisazioni riguardo il conteggio del numero di archi:

- dal momento che ogni volta che vengono aggiunti 2 archi, viene incrementato il contatore di 2, questa volta il controllo viene fatto sul `edgesMax` decrementato di 1, per essere certi che alla fine il numero di archi aggiunti sarà effettivamente `edgesMax`;
- inoltre quando viene aggiunto un nuovo nodo, vengono aggiunti due nuovi archi i quali non vengono conteggiati nella variabile `edgesNum`, questo sia per evitare un controllo doppio (sul numero di nodi e sul numero di archi) ogni volta che si vuole aggiungere un nodo, sia perchè così è più semplice settare il numero di percorsi massimo da aggiungere (altrimenti $edgesMax = numeroDesirato + nodesMax \times 2$).

In questa sezione sono state commentate le regole che si riferiscono a grafi in cui tutti gli archi sono bidirezionali (in GROOVE tutti i nodi sono collegati sempre con una coppia di archi di verso opposto), è stato implementato anche il caso più generale che non verrà commentato in quanto le modifiche sono minime.

Le uniche regole che cambiano sono: `addNewNode` ogni volta che aggiunge un nodo aggiungerà un solo arco, e `addNewPath` ogni volta che aggiunge un nuovo collegamento aggiungerà un solo arco (naturalmente presi due nodi tra i quali non vi è nessun collegamento diretto, il verso in cui viene creato il nuovo collegamento è non deterministico), poi in entrambi i casi verranno incrementati i contatori opportunamente. Modifiche simili riguardano anche la generazione del grafo iniziale in modo casuale, di cui si argomenta subito dopo sempre facendo riferimento a grafi non direzionati. Le altre regole e il codice Prolog utilizzato per il model checking di cui si parlerà poi, sono assolutamente generali.

Tutti i file che riguardano il caso generale hanno il nome che termina con la lettera M al posto della lettera B.

Generazione di una rete di partenza in modo casuale

Nella generazione dell'albero in modo casuale le regole sono molto simili a quelle utilizzate per aggiungere un nodo o un percorso dinamicamente nel calcolo del gradiente. Anche in questo caso si utilizzano dei contatori per verificare che il numero di nodi e archi del grafo disegnato siano effettivamente quelli desiderati, a differenza di prima il numero degli archi viene contato meticolosamente anche quando viene aggiunto un nodo. I contatori per la generazione dell'albero alla fine verranno eliminati e rimarranno solo quelli che servono per l'algoritmo del gradiente.

Infine ci si deve assicurare che vi sia un nodo (e solo uno) con flag `start` e valore 0 e per questo è stato sfruttato il meccanismo delle priorità sulle regole di GROOVE. Assegnare alle regole diverse priorità consente di vincolare la scelta delle regole da applicare ad un grafo, che viene fatta in base a quelle, quindi verranno scelte sempre prima le regole più prioritarie.

Si noti che in GROOVE priorità 0 significa priorità minima, e poi si continua modo incrementale. In questo caso è utile in particolare per il nodo `start`, perchè se non viene scelta la regola che lo aggiunge tra i primi due nodi della rete, deve essere sicuramente scelta quella che lo aggiunge dopo, e questo è garantito settando la regola che si occupa di ciò più prioritaria delle altre.

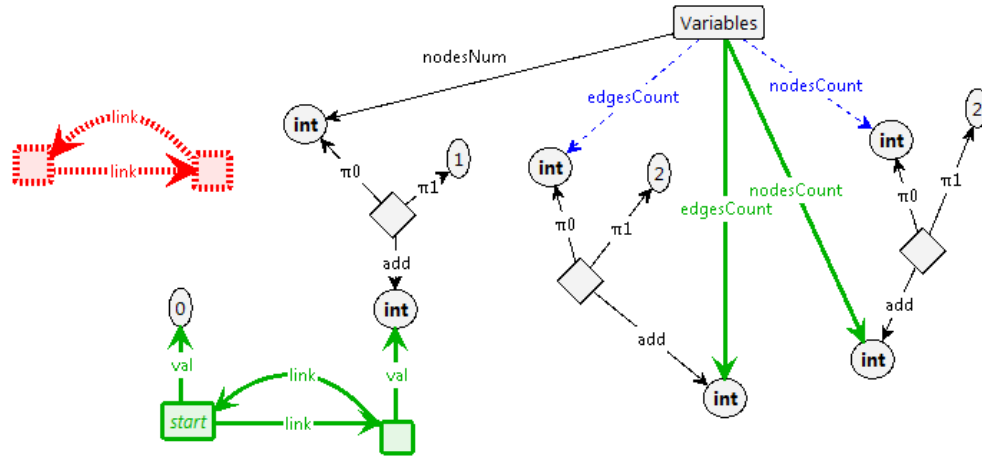


Figura 17: Regola addCoupleNodeWithStart: crea una coppia di nodi collegati tra loro di cui uno è il nodo start, se il grafo è vuoto.

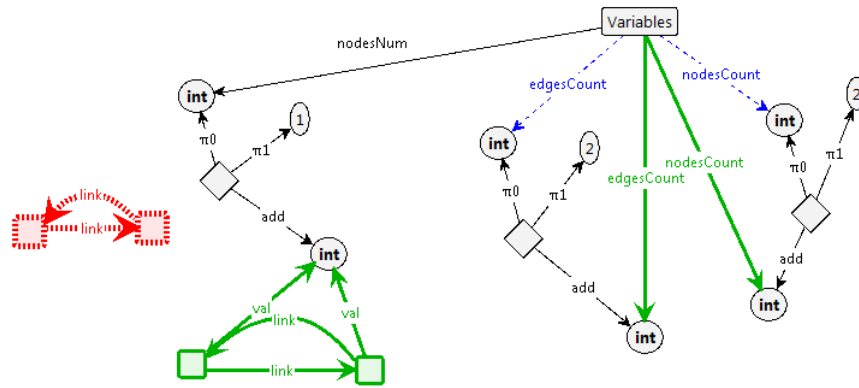


Figura 18: Regola addCoupleNodeWithoutStartB: crea una coppia di nodi collegati tra loro, se il grafo è vuoto.

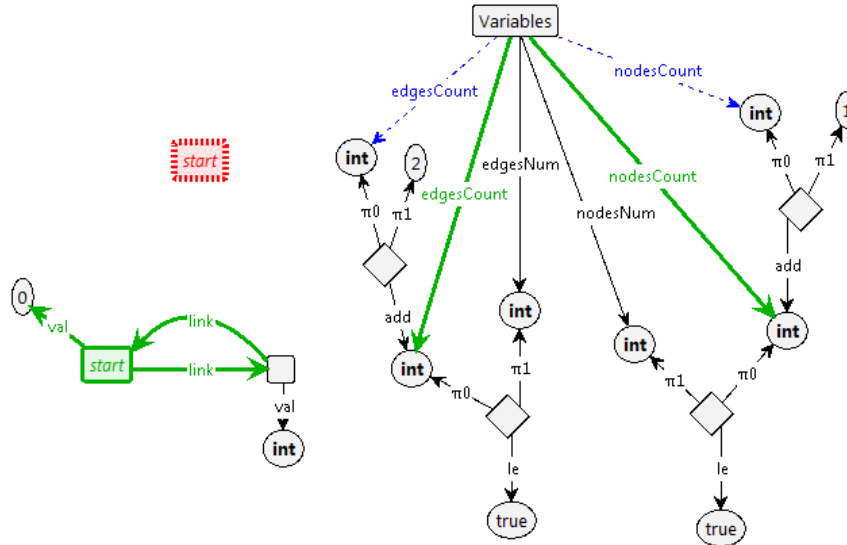


Figura 19: Regola addNodeStartB: aggiunge un nodo Start al grafo se non è ancora stato aggiunto e i contatori di nodi e archi risultano inferiori alle rispettive soglie.

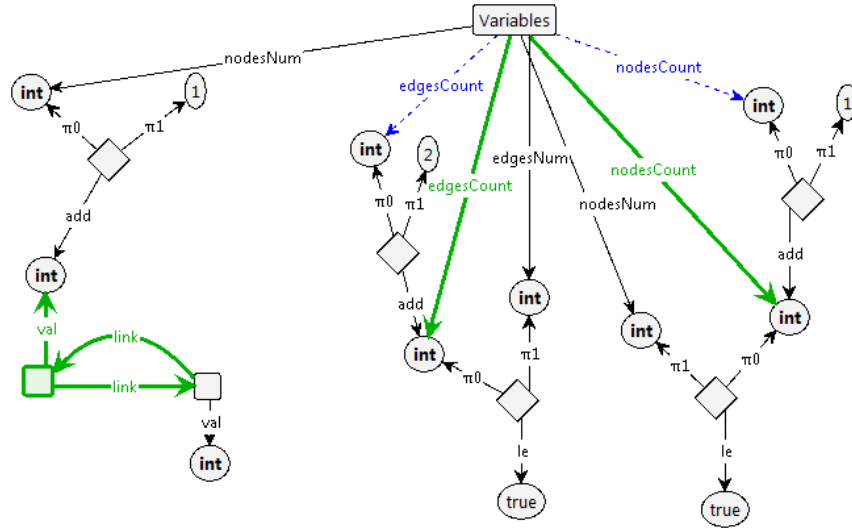


Figura 20: Regola `addNodeWithRandValB`: aggiunge un nodo con un valore uguale al numero di nodi aumentato di 1, e i contatori di nodi e archi risultano inferiori alle rispettive soglie.

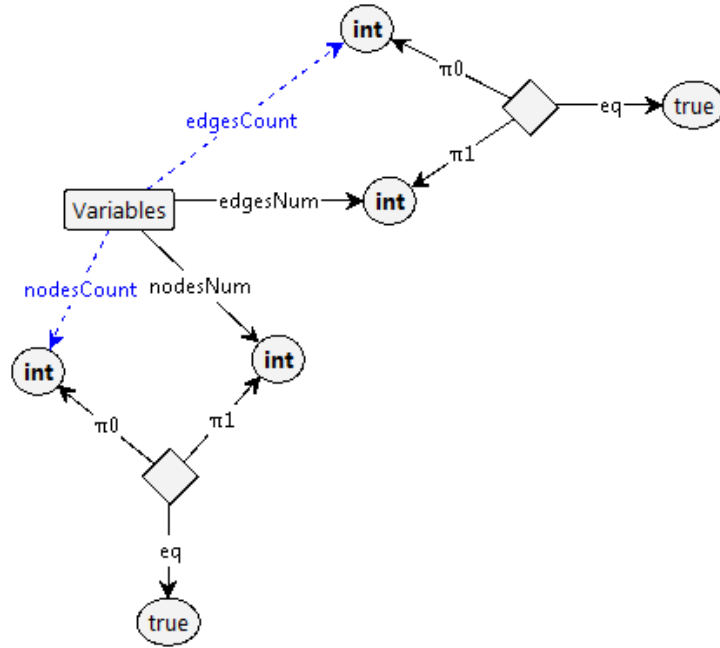


Figura 21: Regola `final`: elimina i contatori `edgesCount` e `nodesCount`, utili solo al fine del disegno della rete, quando è stato raggiunto il numero di archi e di nodi desiderato.

Variables
edgesCount = 0
edgesMax = 8
edgesNum = 8
nodesCount = 0
nodesMax = 6
nodesNum = 5

Figura 22: Grafo iniziale vuoto con parametri settati per la generazione di una rete con 5 nodi e 8 archi. Si noti che per la generazione del grafo verrà utilizzata la strategia di ricerca *RandomLinear* con condizione di terminazione *1 final*, cioè appena raggiunge uno stato finale.

Codice Prolog per il model checking

GROOVE una volta generato lo spazio degli stati offre tre possibilità:

1. esplorare le configurazioni finali (guardando ogni grafo finale) e controllare che siano corrette o presentino le proprietà di interesse;
2. verificare che formule espresse in CTL o LTL siano valide nello spazio generato;
3. verificare proprietà mediante programmi Prolog.

La prima quando il numero delle configurazioni finali è elevato è impraticabile, la seconda risulta molto utile se si devono testare le regole, per es. verificare se una regola viene sempre applicata per ogni possibile scenario, per questi motivi è stato scelto di utilizzare Prolog perchè grazie ai numerosi predicati forniti permette di interagire molto bene con lo spazio degli stati, con i grafi e con le regole. Quindi è stato implementato un predicato che ora verrà commentato nel dettaglio.

Si assume che in un grafo, il gradiente rispetto ad una sorgente sia corretto se è corretto il gradiente del nodo (o dei nodi) alla massima distanza dalla sorgente. Per il calcolo del cammino minimo è stata usata un'implementazione dell'algoritmo di Dijkstra trovata in rete (il link si trova nei Riferimenti Bibliografici), ma modificandola leggermente, come viene discusso in seguito. Inoltre sono stati usati vari predicati forniti da GROOVE indispensabili per interagire sia con i grafi, che con l'LTS.

I predicati prolog di cui si parla in seguito si trovano nei file `testGradient.pro` e `dijkstra.pro`, anche per quanto riguarda i file Prolog GROOVE permette di abilitarli e disabilitarli allo stesso modo delle regole, senza dover utilizzare i predicati `consult` e `reconsult`, quando si vogliono utilizzare predicati che si trovano in file diversi da quello dove risiede la teoria che si sta considerando.

L'idea generale è la seguente, si prendono tutti i possibili grafi finali, ovvero quelli in cui non è possibile applicare ulteriormente nessuna regola, e si controlla che siano corretti, e se ne viene trovato uno non corretto si restituisce no e si termina.

Il predicato principale è `test(?StateSet)` che prende la lista degli stati finali dell'LTS corrente, con il predicato `final_state_set(?StateSet)`, poi stampa la lunghezza della lista, e chiama il predicato `test_set(+StateSet,+CountState)` che è quello che effettivamente controlla stato per stato se i grafi sono corretti.

Il predicato `test_set` controlla ricorsivamente tutti gli stati finali dell'LTS, infine ogni volta che il predicato `test_graphG` da esito positivo fa una stampa, che è molto utile per capire quale è il grafico considerato non corretto se il test da esito negativo. Inoltre è utile anche in caso di esito positivo, per controllare che effettivamente tutti gli stati finali siano stati considerati corretti, infatti basta controllare che il numero dell'ultimo stato considerato corretto sia uguale al numero degli stati finali da testare. Nel predicato `test_set` è stato usato `state_graph(+State,?Graph)` il quale restituisce il grafo corrispondente allo stato `State`.

```
% test(?StateSet)
test(S) :- final_state_set(S), length(S,L),
           write(' numStati=_'), write(L), nl,
           test_set(S,1).
% test_set(+StateSet,+CountState).
test_set([],_).
test_set([H|T],C) :- state_graph(H,G), test_graphG(G),
                    C1 is C+1,
                    write(' STATE_'), write(C), write(' CORRECT'), nl,
                    test_set(T,C1).
```

Il predicato che si occupa di controllare se il gradiente di un grafo è corretto è `test_graphG(+Graph)` il quale prende tutti gli archi con flag presenti nel grafo con il predicato `graph_flag(+Graph, ?Edge)`, cioè solo il nodo sorgente che ha `flag:start`. Quindi si fa restituire dal predicato `edge_source` il nodo `start` (anche qui utilizzando `edge_target` non sarebbe cambiato nulla in quanto un flag viene visto come un arco che va da un nodo al nodo stesso), e in seguito prende il numero interno del nodo (assegnato da GROOVE per identificare il nodo) con il predicato `node_number(+Node, ?Number)`. Questi appena commentati e quelli che seguono sono una serie di predicati, che prendono i dati che servono e li mettono in una "forma" più semplice e comoda da manipolare. Continuando si prende l'insieme di archi con etichetta `link` e quello degli archi con etichetta `val`, con il predicato `label_edge_set(+Graph, +Label, ?EdgeSet)`, il quale prende la lista degli archi con etichetta `Label` presenti nel grafo `Graph`, ma siccome `Label` rappresenta un oggetto di tipo `Label`, prima di applicarlo si deve effettuare la conversione da stringa a oggetto `Label` con il predicato `label(+String, ?Label)`. Successivamente si chiama il predicato `dijkstraG(+Node, -MaxList)` che applica l'algoritmo di Dijkstra al grafo, restituisce la lista dei nodi del grafo che si trovano alla distanza massima dal nodo `Node` nella forma `[NumNodo, DistanzaMin]`. Il predicato `trasform_linkEdge(+ListLinkEdge)` serve per rappresentare il grafo formato dai soli nodi e archi di tipo `link` in un formato necessario per far funzionare il predicato `dijkstraG`, mentre il predicato `trasform_valEdge(+ListValEdgeIN, -ListValEdgeOUT)` trasforma gli elementi della lista degli archi `val` dal formato `[n2--val-->jint:3]` a `[n2, 3]`. Infine il predicato `test_longest(+MaxList, +ListVal)` controlla se il valore dei nodi a massima distanza trovati dall'algoritmo di Dijkstra è uguale al valore del gradiente che è stato assegnato ai nodi.

```
% test_graph(+Graph)
test_graphG(G):- graph_flag(G,ES), edge_source(ES,NS),
                 node_number(NS,NSN),
                 label(link,L1), label_edge_set(G,L1,LP),
                 trasform_linkEdge(LP),
                 label(val,L2), label_edge_set(G,L2,LV),
                 dijkstraG(NSN,Ss),
                 trasform_valEdge(LV,LV1),
                 test_longest(Ss,LV1).
```

Ora si passa a esaminare i predicati riguardanti l'algoritmo di Dijkstra, `trasform_linkEdge` e `dijkstraG`. Il primo prende in ingresso una lista di archi di tipo `link` e per ogni arco asserisce un predicato `dist(+NodeSourceNumber, +NodeTargetNumber, +Dist)`.

```
% trasform_linkEdge(+ListEdgeIN)
trasform_linkEdge([]).
trasform_linkEdge([H|T]):- edge_source(H,S),
                           node_number(S,S1),
                           edge_target(H,D),
                           node_number(D,D1),
                           assertz(dist(S1,D1,1)),
                           trasform_linkEdge(T).
```

Qui di seguito viene riportato prima il predicato originale trovato in rete `dijkstra(Vertex0, Ss)` seguito dal predicato modificato e utilizzato nel caso di studio. La differenza tra i due riguarda solamente l'output, il primo restituisce i cammini minimi dal nodo `Vertex0` verso tutti gli altri nodi del grafo con le rispettive distanze nella forma `s(Vertex, Dist, Path)` mentre il secondo restituisce la lista degli elementi con cammino minimo più lungo da `Vertex0`, nella forma `[NumVertice, Distanza]`. Entrambi i predicati partono dal presupposto che siano asseriti nella

teoria una serie di predicati

`dist(NodoSorgente,NodoDestinazione,Distanza)` che rappresentano il grafo, motivo per cui è stato implementato `trasform_linkEdge`. E' importante notare che, nella versione originale dati due nodi `i` e `j`, asserendo il predicato `dist(i,j,Value)` è implicitamente asserito anche `dist(j,i,Value)`. Nel caso di studio è stato scelto di evitare ciò, per rendere più generale il predicato sia in caso di grafi non direzionati (in cui sono presenti tutti collegamenti bidirezionali), sia nel caso direzionato. Quindi è necessario asserire esplicitamente il predicato `dist` per tutti gli archi del grafo.

Il primo dei due predicati aggiunti `trasform_resultV(+Si,-So)` si occupa di trasformare il risultato, mentre il secondo cioè `max(+List,-Max)` è una versione del massimo di una lista che tratta liste di elementi nella forma `[NumVertice,Distanza]` e restituisce l'elemento della lista che ha Distanza massima. Infine viene usato il predicato `findall` con il predicato `member` per prendere tutti gli elementi a distanza massima.

```
% dijkstra(Vertex0, Ss) is true if Ss is a list of structures s(Vertex, Dist,
% Path) containing the shortest Path from Vertex0 to Vertex, the distance of
% the path being Dist. The graph is defined by e/3.
% e.g. dijkstra(penzance, Ss)
dijkstra(Vertex, Ss):-
    create(Vertex, [Vertex], Ds),
    dijkstra_1(Ds, [s(Vertex,0,[])], Ss).

% dijkstraG(Vertex0,Ss)
dijkstraG(Vertex, Er):-
    create(Vertex, [Vertex], Ds),
    dijkstra_1(Ds, [s(Vertex,0,[])], Ss),
    trasform_resultV(Ss,Sr),
    max(Sr, [EN,EV]),
    findall([N,EV],member([N,EV],Sr),Er).

% trasform_resultV(+Si,-So)
% s(Vertex,Edge,Path) --> [Vertex,Edge]
trasform_resultV([], []).
trasform_resultV([H|T],[[H2,H3]|T1]):- H=..[H1,H2,H3|L],
    trasform_resultV(T,T1).

% max(List,Max)
% Max is the biggest element in List
max([X1,X2]|Xs,Max):- max(Xs,Max,[X1,X2]).
max([],TempMax,TempMax).
max([X1,X2]|Xs,Max,[XM,TempMax]):- X2 <= TempMax, max(Xs,Max,[XM,TempMax]).
max([X1,X2]|Xs,Max,[XM,TempMax]):- X2 > TempMax, max(Xs,Max,[X1,X2]).
```

Ritornando al predicato `test_graphG` ora verranno esaminati i predicati

`trasform_valEdge(+ListEdgeIN,-ListEdgeOUT)` e

`test_longest(+ListNodesWithDistanceMaxToStartNode,+ListValueNodes).`

Il primo, come già anticipato, si occupa di trasformare archi dalla forma `n2--val-->jint:0` alla forma `[n2,0]`, in particolare utilizzando il predicato `convert_valuenode(+Node,-Term)` che converte il nodo valore nel termine Prolog corrispondente, per esempio da `jint:0` all'intero corrispondente 0.

```

% trasform_valEdge(+ListEdgeIN,-ListEdgeOUT).
% trasform_valEdge([n2--val-->jint:0],[[n2,0]]).
trasform_valEdge([],[]).
trasform_valEdge([H|T],[[NS,DV]|T1]):- edge_source(H,S), edge_target(H,D),
                                         node_number(S,NS),
                                         convert_valuenode(D,DV),
                                         trasform_valEdge(T,T1).

```

Mentre `test_longest(+ListNodesWithDistanceMaxToStartNode,+ListValueNodes)` è il predicato che si occupa di confrontare i nodi a massima distanza dalla sorgente (calcolati con Dijkstra), con i valori assegnati dall'algoritmo del Gradiente, se sono uguali allora significa che il gradiente del nodo sorgente in quel grafo è stato calcolato correttamente.

Si procede in questo modo: si prendono i nodi con gradiente massimo, allo stesso modo in cui si faceva per quelli a distanza massima con Dijkstra, poi si passa a controllare che le due liste contengano elementi uguali, indipendentemente dall'ordine in cui compaiono nelle rispettive liste. Il predicato `removeX(+Elem,+List,-ListWithoutElem)` rimuove l'elemento dalla lista.

```

% test_longest(+ListNodesWithDistanceMaxToStartNode,+ListValueNodes)
test_longest(Ss,LV1):- max(LV1,[Nv,Vv]),
                      findall([N,Vv],member([N,Vv],LV1),LV2),
                      test_equal(Ss,LV2).
% test_equal(+ListNodesWithDistanceMaxToStartNode,+ListValueNodesWithMaxValue)
test_equal([],[]).
test_equal([N,V|T1],LV):- member([N,V],LV),
                          removeX([N,V],LV,LV1),
                          test_equal(T1,LV1).

```

Risultati ottenuti con l'Algoritmo base per il calcolo del gradiente

In questa parte verranno riportati i risultati ottenuti con l'Algoritmo base per il calcolo del gradiente, i quali sono stati ottenuti usando GROOVE su una macchina con sistema operativo Windows7, avviando il `Simulator.jar` con il seguente comando `java -Xmx1300m -Xss2m -jar Simulator.jar`.

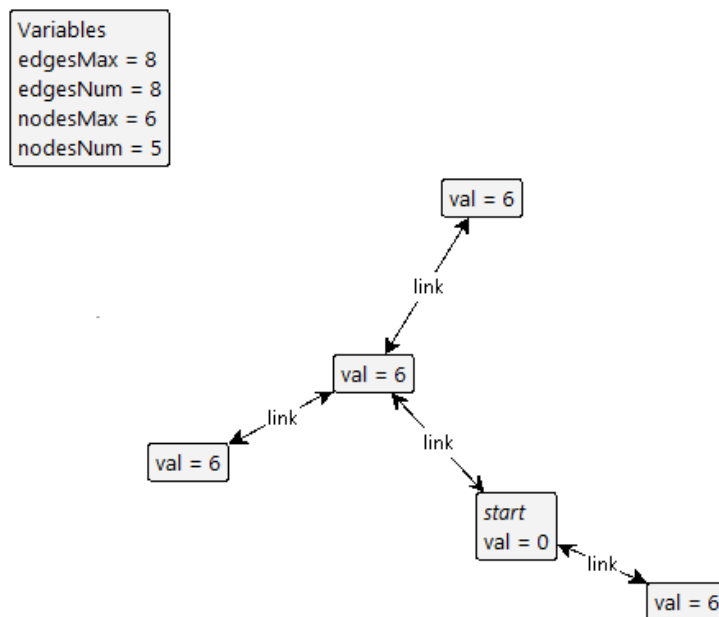


Figura 23: Grafo iniziale generato. Tale grafo è quello usato per le simulazioni. Le simulazioni sono state fatte utilizzando il grafo generato in modo casuale da GROOVE di fig. con 5 nodi e 8 archi, modificando ogni volta il numero di nodi e archi da aggiungere dinamicamente. Nella fig.24 viene mostrato come cresce lo spazio degli stati generato da GROOVE all'aumentare dei nodi e degli archi che possono essere aggiunti dinamicamente al grafo di partenza. Mentre nella fig.25 viene riportato un grafico del tempo impiegato per le simulazioni complessivo del tempo per generare l'LTS e per applicare il testing con Prolog, anche se quest'ultimo si può ritenere trascurabile dal momento che nel caso peggiore è 1 min e 30 sec. Per generare l'LTS è stata usata la strategia di ricerca *depthFirst* con condizione di terminazione *infinite*, cioè esplora tutti possibili stati. Sono state fatte delle simulazioni anche con la *breadthFirst*, di cui non sono riportati i risultati, ma non sono state riscontrate particolari differenze in termini di prestazioni.

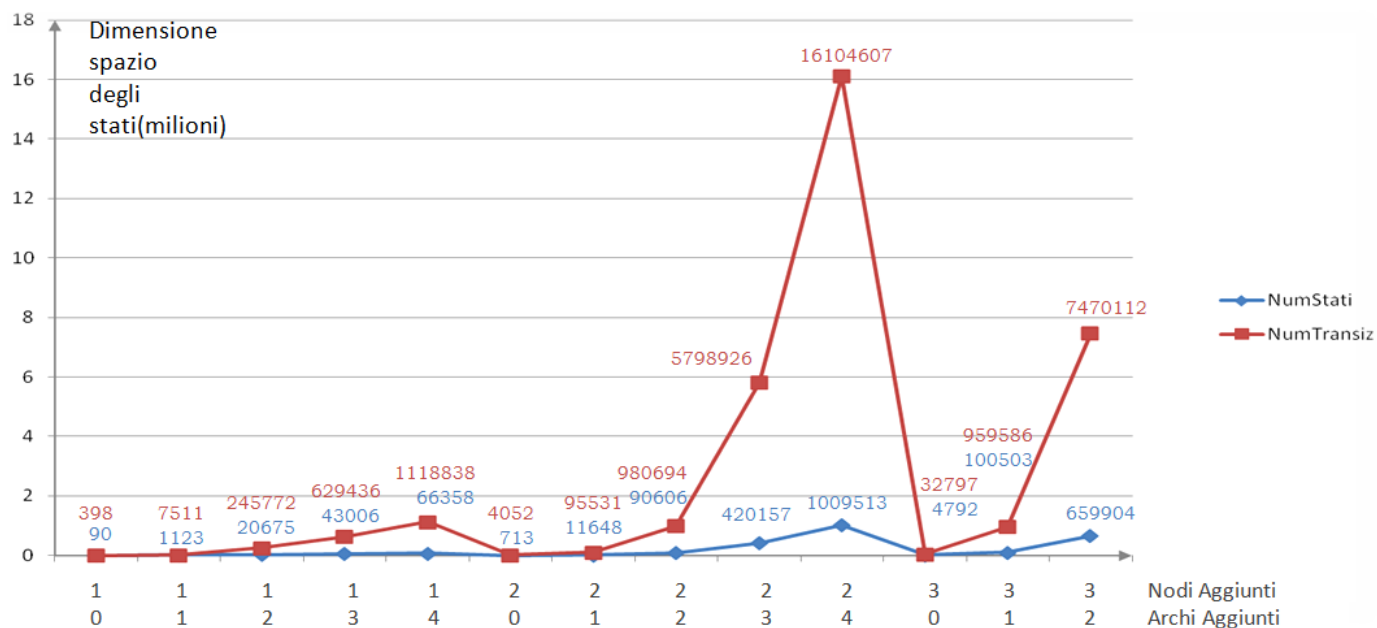


Figura 24: Diagramma che rappresenta l'evoluzione della dimensione dello spazio degli stati all'aumentare del numero di nodi e archi che si possono aggiungere dinamicamente alla rete di partenza.

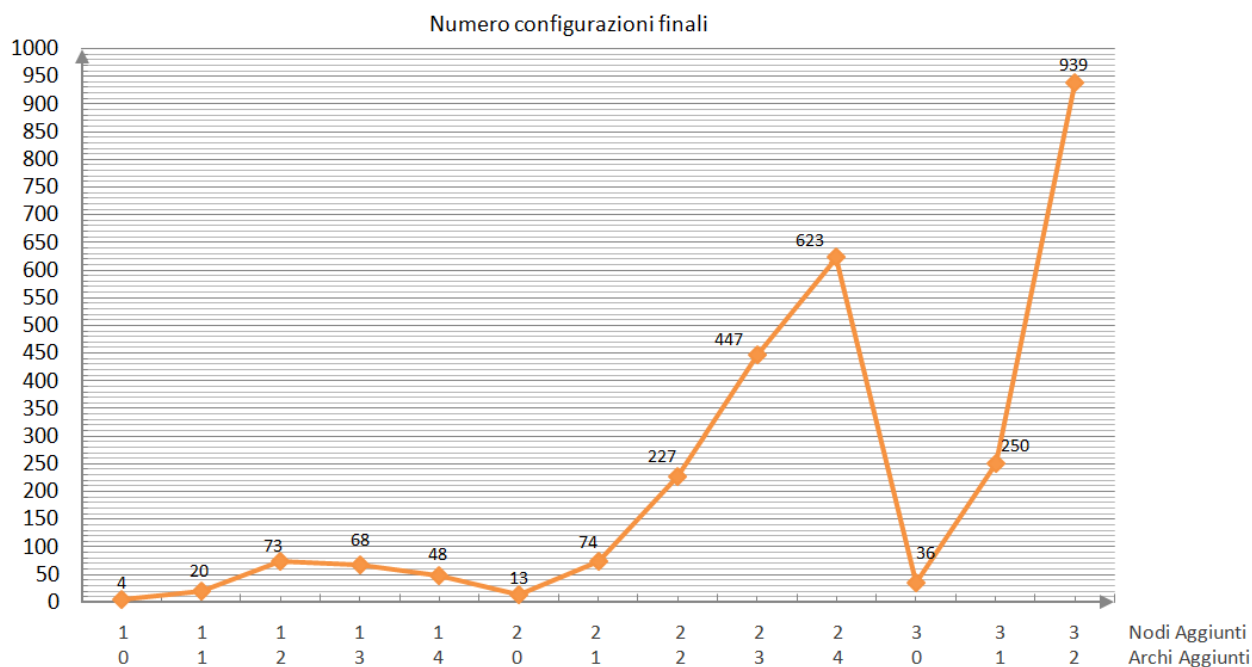


Figura 25: Diagramma che rappresenta l'evoluzione del numero delle configurazioni finali all'aumentare del numero di nodi e archi che si possono aggiungere dinamicamente alla rete di partenza.

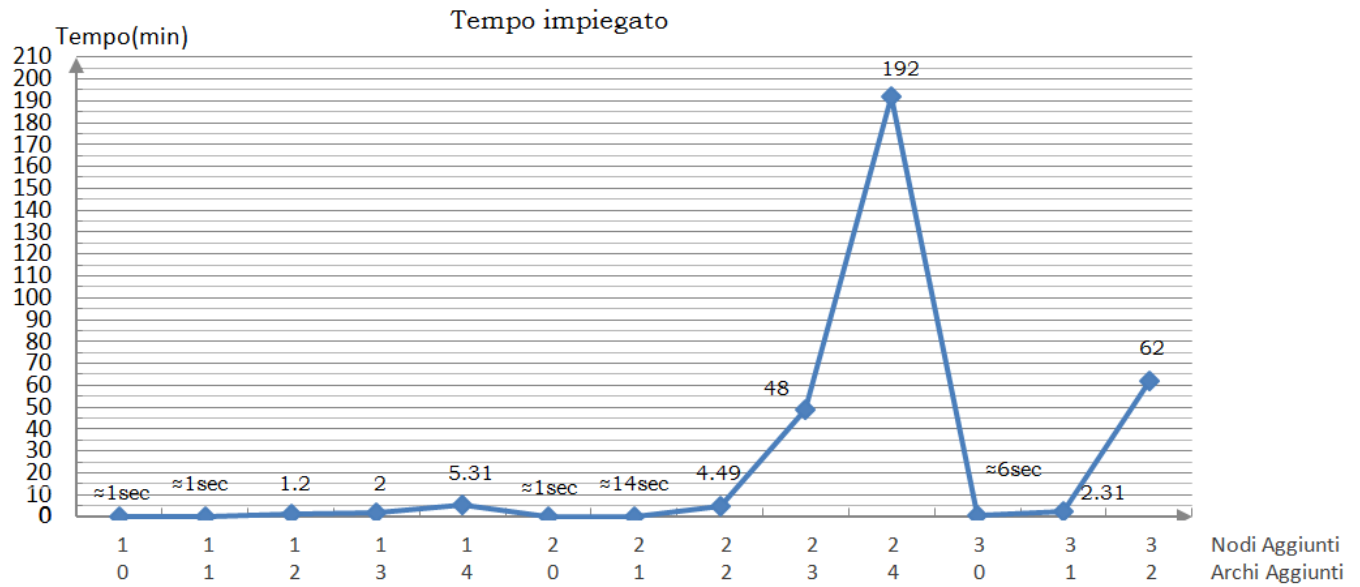


Figura 26: Diagramma che rappresenta l'evoluzione del tempo impiegato nelle simulazioni all'aumentare del numero di nodi e archi che si possono aggiungere dinamicamente alla rete di partenza.

I diagrammi hanno presentano, anche aggiungendo solo qualche nodo e arco, un andamento esponenziale.

La motivazione principale è la seguente: GROOVE partendo dallo stato iniziale prova ad applicare tutte le regole, ma non solo. Si supponga che sia abilitata solo la regola che aggiunge un nuovo nodo alla rete, questa non ha un unico match, ma ha tanti match quanti sono i nodi ai quali è possibile collegare il nuovo nodo, quindi nella rete che si sta considerando 5.

Nel caso si vogliano aggiungere due nodi si avranno per il primo nodo 5 matching e per il secondo nodo 6, perchè i nodi a cui è possibile collegare il nodo adesso sono diventati 6, quindi in totale 30. Per i collegamenti il meccanismo è lo stesso. In realtà le configurazioni risultanti sono inferiori rispetto quelle calcolate ragionando nel suddetto modo, la causa è il controllo sull'isomorfismo.

GROOVE durante l'applicazione delle regole fa un controllo sugli stati generati precedentemente e se una regola porta ad uno stato uguale ad uno già presente, non viene generato un nuovo stato ma si fa terminare la transizione nello stato già presente. Questo controllo diminuisce il numero degli stati generati notevolmente.

Per avere un'idea quantitativa prendendo la rete di fig.1 e abilitando le regole della propagazione del valore del gradiente e la regola che aggiunge un nuovo nodo, senza controllo sull'isomorfismo si ottengono 7 configurazioni finali, quelle corrispondenti al collegamento del nuovo nodo in ognuno dei nodi della rete. Mentre ripetendo lo stesso esperimento con il controllo sull'isomorfismo abilitato, si ottengono solo 4 configurazioni finali. Un esempio ulteriore si può fare rifacendo la stessa prova, questa volta abilitando la regola che aggiunge un nuovo percorso al posto della regola che aggiunge un nuovo nodo. Se non viene abilitato l'isomorfismo si ottengono 16 configurazioni diverse, mentre con l'isomorfismo solo 7.

Da ciò appena detto ci si aspetta che il diagramma quando vengono aggiunti 1 nodo e 3 archi e 1 nodo e 4 archi continui a crescere, invece decresce questo perchè quando il numero di archi diventa molto superiore al numero dei nodi, dal momento che non si possono aggiungere nuovi archi tra due nodi già direttamente collegati, le configurazioni possibili diminuiscono. Inoltre si deve tenere conto di un'ultima questione, GROOVE di default una volta generato l'LTS lo visualizza nella GUI, questo per grafi di una certa dimensione porta ad una penalizzazione delle prestazioni notevole, perciò nelle simulazioni dal momento che il model checking viene fatto tramite un predicato Prolog,

si è scelto di disabilitare la visualizzazione dell'LTS. Infine a parte il caso con l'aggiunta di 2 nodi e 3 archi, che come si vede nel diagramma del tempo di fig.26, ha richiesto circa 3 ore, nei casi successivi si è deciso di prendere come tempo massimo per una simulazione circa 1 ora e quindi nel caso non fosse terminata, di interromperla e controllare le configurazioni finali ottenute fino a quel momento.

2.2 Estensione dell'algoritmo base con il calcolo della distanza tra due punti

L'estensione consiste nell'aggiungere la possibilità di calcolare il cammino minimo tra il nodo sorgente etichettato come `start` e un nodo scelto dall'utente etichettato come `target`, in termini di sequenza di nodi attraversati. Per fare ciò è sufficiente aggiungere due regole all'algoritmo precedente, che una volta calcolato il gradiente andranno a "disegnare" il cammino minimo dal nodo `start` al nodo `target`, nel senso che verranno aggiunti una serie di archi `shortest` lungo il cammino minimo.

L'idea è la seguente: dopo aver calcolato il gradiente, si parte dal nodo `target`, si disegna l'arco `shortest` che ha come destinazione il nodo `target` stesso e sorgente il nodo (o uno dei nodi) che ha il valore del gradiente uguale a quello del nodo `target` diminuito di 1, e così via fino a raggiungere il nodo `shortest`. Naturalmente una volta che è stato aggiunto un arco `shortest` tra due nodi non può essere aggiunto una seconda volta. Dal momento che queste regole devono essere applicate una volta che il gradiente è stato calcolato, avranno priorità inferiore rispetto alle regole del calcolo del gradiente.

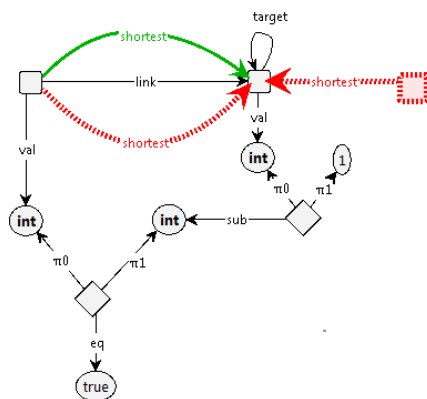


Figura 27: Regola drawShortestPathTarget

Inoltre per garantire che l'aggiunta degli archi venga fatta in ordine, a partire dal nodo `target` fino al nodo `start`, basta dire che un arco `shortest` tra due nodi qualsiasi, può essere aggiunto solo se la destinazione ha già un arco `shortest` verso un altro nodo, come si vede in figura 27.

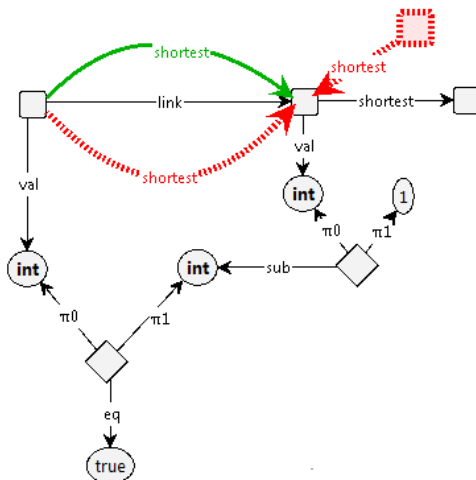


Figura 28: Regola drawShortestPath

Generazione di una rete di partenza in modo casuale

Per quanto concerne la generazione di una rete di partenza si utilizzano le stesse regole dell'algoritmo precedente; una volta generato si sceglie un nodo verso il quale si vuole calcolare il cammino minimo, poi si aggiunge un arco etichettato `target` sul nodo scelto diretto verso il nodo stesso.

Codice Prolog per il model checking

I predicati Prolog che verranno esaminati in questa sezione si trovano nel file `testShortestPath.pro`. Per verificare che il cammino minimo tra il nodo `start` e il nodo `target` l'idea è simile al test del gradiente: la lista contenente il cammino minimo calcolato mediante Dijkstra, in termini di elenco di nodi attraversati a partire dal nodo `start`, deve essere uguale alla lista contenente i nodi destinazione degli archi `shortest` del grafo a partire dal nodo `start`.

Inizialmente l'idea era di confrontare le due liste e controllare che contenessero gli stessi elementi, ma in ordine inverso, questo perchè GROOVE salva gli archi in ordine di creazione e come si evince da quanto detto sopra gli archi `shortest` vengono creati a partire dal nodo `target` al nodo `start`.

Però, a causa della topologia della rete, capita spesso che vi siano più di un percorso minimo equivalenti e quello scelto da Dijkstra non coincide con quello scelto dall'algoritmo, quindi il confronto sulle liste darebbe esito negativo anche se il cammino è stato calcolato correttamente, motivo per cui si è scelto di confrontare le lunghezze delle liste tenendo conto che in Dijkstra viene messo in lista anche il nodo `start`, mentre nel predicato si prendono solo i nodi destinazione degli archi `shortest`.

Il predicato principale è `test_graphP(+Graph)` il quale viene chiamato allo stesso modo di `test_graphG(+Graph)`, ottenuti i numeri dei nodi sorgente e `target`, si utilizza il predicato `dijkstraP(+Vertex,+Target,-ShortestPath)` che restituisce la lista dei numeri dei nodi attraversati nel percorso minimo tra il nodo `SourceNum` e il nodo `TargetNum`. Infine si confrontano la lunghezza di questo e della lista contenente i nodi destinazione degli archi `shortest` (aumentata di 1).

```
%test_graphP(+Graph)
test_graphP(G):- graph_flag(G,ES), edge_source(ES,NS),
    node_number(NS,NSN),
    label(link,L1), label_edge_set(G,L1,LP),
    trasform_linkEdge(LP),
    label(shortest,L2), label_edge_set(G,L2,LShort),
    label(target,L3), label_edge(G,L3,LTarget),
    edge_source(LTarget,Tar), node_number(Tar,TarNum),
    dijkstraP(NSN,TarNum,SP),
    length(SP,LeP), length(LShort,LeS),
    LeS1 is LeS +1,
    LeP==LeS1.

% dijkstraP(+Vertex0,+Target,-ShortestPath)
dijkstraP(Vertex, TarNum, P):- create(Vertex, [Vertex], Ds),
    dijkstra_1(Ds, [s(Vertex,0,[])], Ss),
    trasform_resultP(Ss,Sr),
    member([TarNum,P],Sr).

% trasform_resultP(+Si,-So) s(Vertex,Edge,Path) --> [Vertex,Path]
trasform_resultP([], []).
trasform_resultP([H|T],[[H2,H4]|T1]):- H=..[H1,H2,H3,H4|L],
    trasform_resultP(T,T1).
```

Risultati ottenuti con l'Algoritmo per il calcolo del cammino minimo tra due nodi

Le premesse sono le stesse dell'altro algoritmo, però questa volta la rete di partenza ha dimensioni maggiori (7 nodi e 16 archi) come si vede in fig.29 , perciò si è deciso di non far visualizzare mai l'LTS.

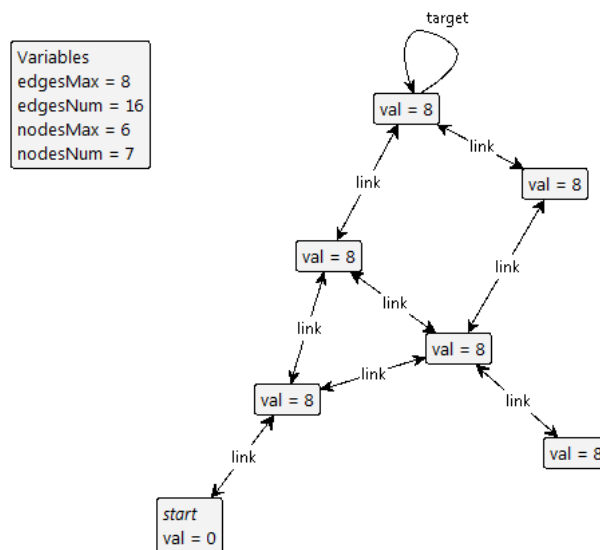


Figura 29: Grafo iniziale generato. Tale grafo è quello usato per le simulazioni

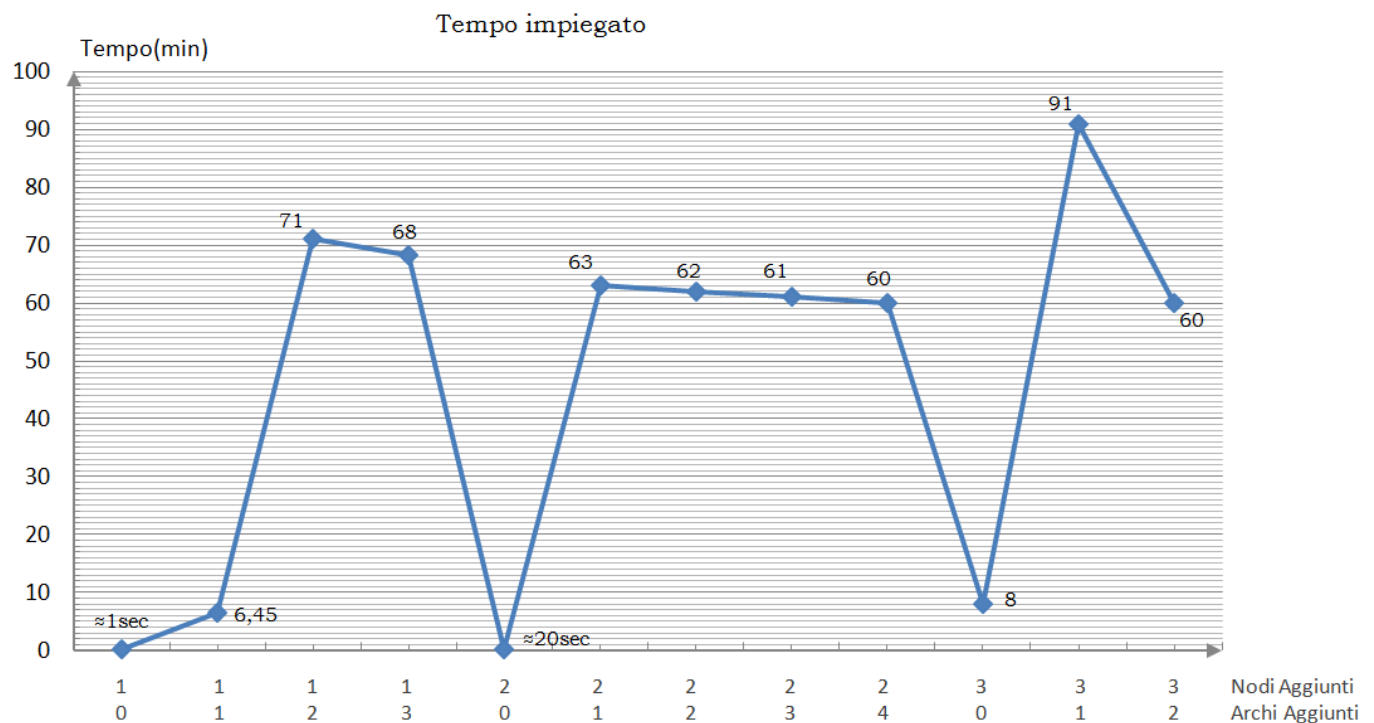


Figura 30: Diagramma che mostra il tempo impiegato nelle simulazioni

Questo primo diagramma non risulta particolarmente interessante poichè, per questioni di tempo, si è deciso di interrompere le simulazioni dopo circa 1 ora nonostante non fossero giunte a termine. Questa volta viene considerato solo il tempo impiegato nella generazione dell'LTS, per quanto riguarda la durata del model checking con il predicato Prolog se ne discute in seguito.

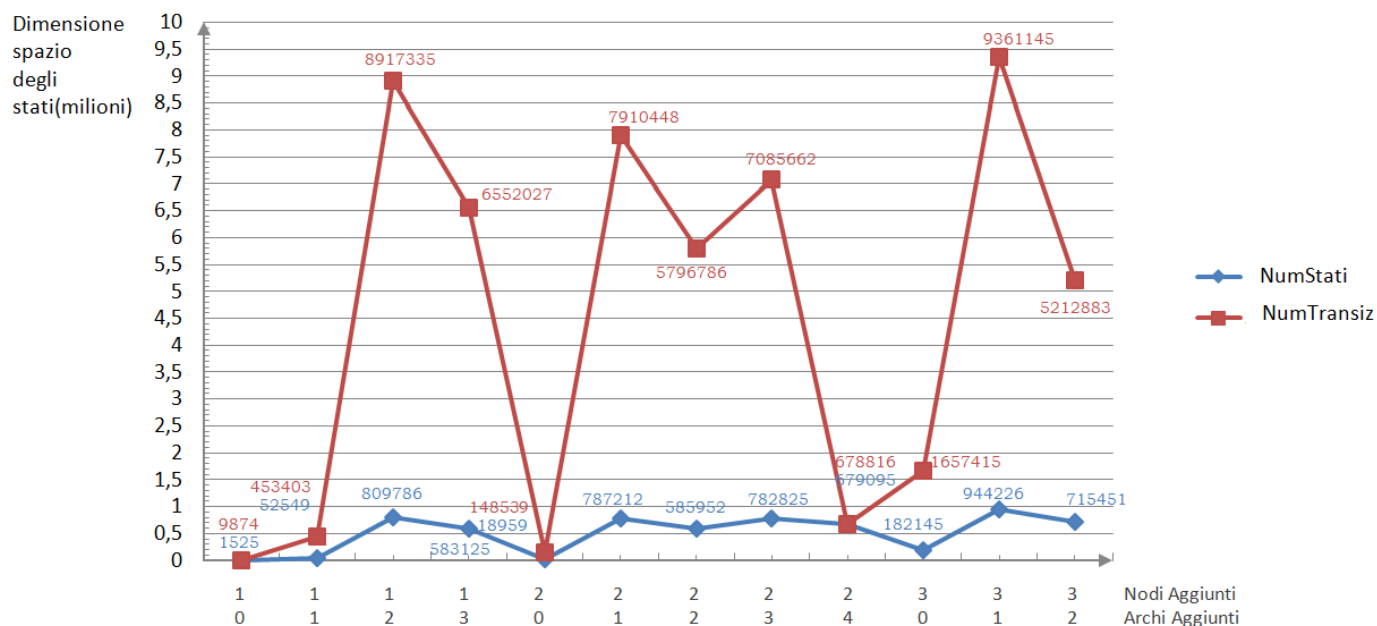


Figura 31: Diagramma che mostra come evolve dimensionalmente lo spazio degli stati all'aumentare del numero di nodi e archi aggiunti dinamicamente.

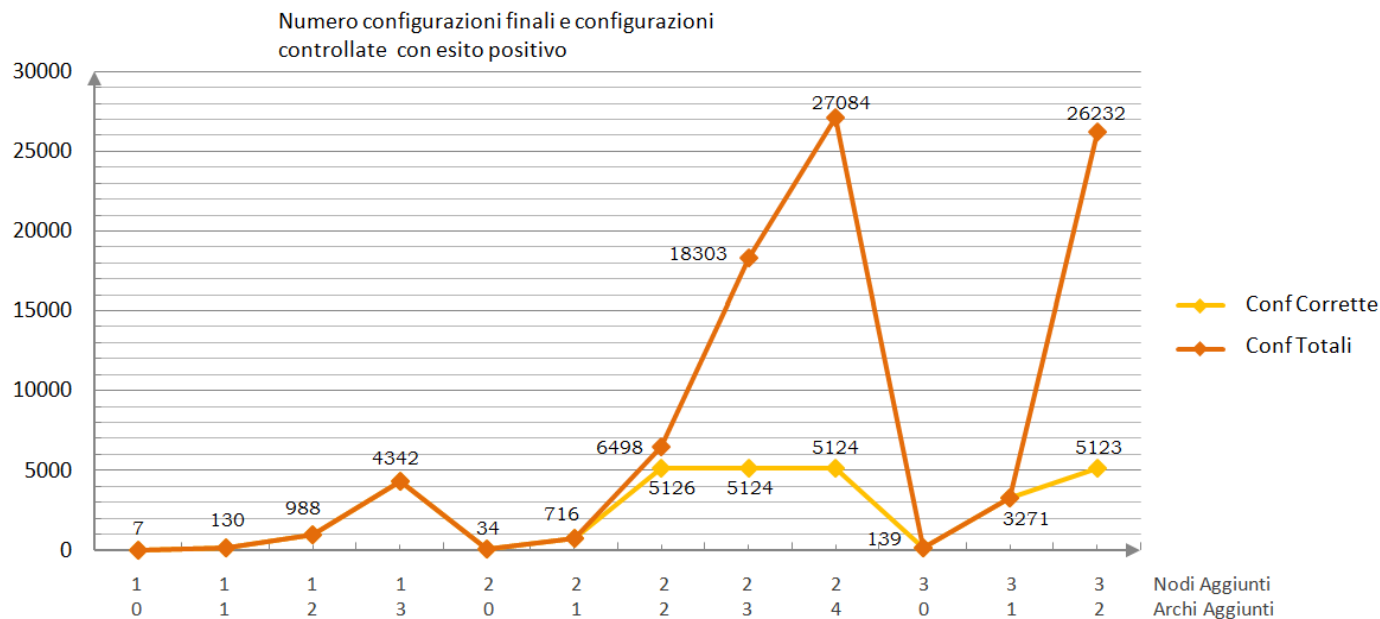


Figura 32: Diagramma che mostra come evolve dinamicamente il numero delle configurazioni finali (trovate prima che la simulazioni termini) e le configurazioni considerate corrette dal predicato Prolog.

Il secondo diagramma (fig.31) riporta l'evoluzione della dimensione dello spazio degli stati all'aumentare del numero di nodi e archi aggiunti dinamicamente. L'andamento è molto simile a quello del caso del gradiente, con un aumento delle dimensioni molto più accentuato; questo dimostra che aggiungendo anche solo pochi nodi e archi lo spazio degli stati cresce esponenzialmente. Sono presenti anche questa volta discese repentine, a causa del controllo sull'isomorfismo, che in questo caso particolarmente simmetrico permette di diminuire molto il numero degli stati, specialmente quando il numero dei nodi rimane costante e aumenta invece quello degli archi.

Nella lettura del diagramma si deve tener conto che in tutti i casi in cui la simulazione supera i 60 min significa che è stata interrotta prima del termine, perciò i risultati riportati nel diagramma sono molto inferiori rispetto a quelli che si otterrebbero se la simulazione giungesse a termine.

Il terzo diagramma (fig.32) riporta l'evoluzione del numero di configurazioni finali all'aumentare del numero di nodi e archi aggiunti dinamicamente, confrontandolo con il numero di grafi considerati corretti dal predicato Prolog.

Nel caso precedente tale confronto non era necessario poichè dato il numero di soluzioni relativamente contenuto il predicato Prolog riusciva sempre a verificare tutte le configurazioni finali (con esito positivo), in un tempo ragionevole.

Questa volta, a causa dell'elevato numero di grafi da considerare, spesso viene scatenata l'eccezione `java.overflow` prima di averli verificati tutti. In tali circostanze, quelli controllati con esito positivo si aggirano attorno ai 4000/5000 e il tempo impiegato è di circa 15 minuti, prima che venga lanciata l'eccezione.

Conclusioni

Visti i risultati ottenuti nel caso di studio si può concludere che, nonostante le numerose funzionalità che presenta GROOVE per fare model checking, attraverso l'uso delle grammatiche per grafi:

- un editor molto semplice e espressivo per consentire il disegno di grafi tipati e regole per interagire con gli stessi;
- un linguaggio `control` che permette di "guidare" l'applicazione delle regole;
- un simulatore che genera un Label Transition System, per rappresentare tutte le possibili evoluzioni del grafo iniziale in seguito all'applicazione delle regole, effettuando anche un controllo sull'isomorfismo dei grafi limitando il numero di stati generati;
- un framework Prolog dove è possibile scrivere programmi che interagiscono direttamente con i grafi stessi, grazie ai numerosi predicati forniti;
- un tool a linea di comando che consente di verificare formule in logica CTL e LTL sull'LTS generato

non è adatto per effettuare model checking di algoritmi di Spatial Computing, come ad esempio il calcolo del gradiente, su reti di dimensione elevata che evolvono dinamicamente in modo non deterministico, in quanto incorre nel tipico "State Explosion Problem".

Appendici

A Teoria della trasformazione dei grafi basata sulla teoria delle categorie

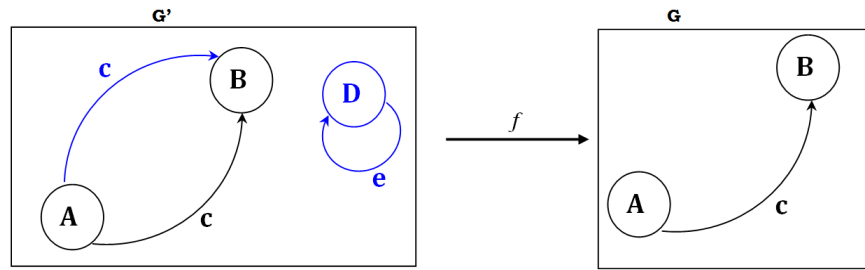
A.1 Definizioni base teoria dei Grafi

Definizione 2 (Grafo). Si definisce grafo una coppia ordinata $G=(V,E)$ dove

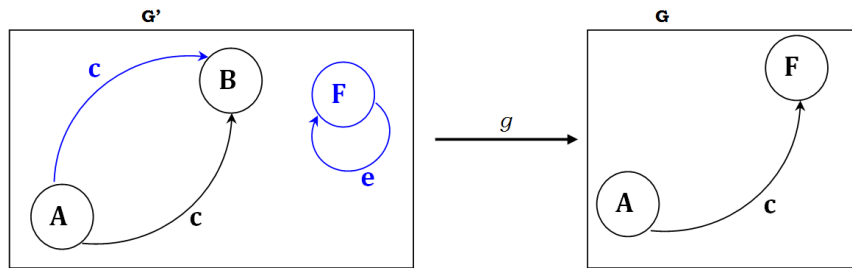
- V è un insieme finito di nodi chiamati vertici $V = \{v_1, v_2, \dots, v_n\}$;
- E è un insieme di archi, in cui ogni arco è una coppia di vertici $E = \{(v_1, v_2), \dots, (v_2, v_3)\}$;

Un grafo può essere di due tipi:

- **non direzionato:** Si intende che gli archi del grafo sono coppie non ordinate di nodi, quindi $(v_i, v_j) = (v_j, v_i)$;
- **direzionato:** Si intende che preso un arco tra due nodi, le due direzioni di quell'arco vengono viste come due archi distinti, per esempio l'arco (v_i, v_j) è diretto da v_i a v_j , ed è diverso dall'arco (v_j, v_i) che è diretto da v_j a v_i .



(a) Omomorfismo corretto.



(b) Omomorfismo non corretto.

Figura 33: **Esempi di omorfismi.**

Nella fig. (a) viene riportato l'omomorfismo grafico $f : G' \rightarrow G$. Si noti che f è un omomorfismo grafico corretto, in quanto i vertici A e B di G' possono essere mappati nei vertici A e B di G , poi l'arco c cioè (A, B) di G' può essere mappato nell'arco c corrispondente di G .

Mentre in figura (b) viene riportato l'omomorfismo grafico $g : G' \rightarrow G$. Si noti che g **non** è un omomorfismo grafico corretto, in quanto il vertice A di G' può essere mappato nel vertice A di G , il vertice F di G' può essere mappato nel vertice F di G , ma l'arco $c(A, F)$ di G non trova nessuna corrispondenza in G' .

Definizione 3 (Omomorfismo o morfismo). Si definisce omomorfismo o morfismo grafico a partire da un grafico $G = (V, E)$ ad un grafico $G' = (V', E')$, scritto $f : G \rightarrow G'$ un mapping $f : V \rightarrow V'$ dall'insieme dei vertici di G all'insieme dei vertici di G' tale che $\{u, v\} \in E$ implica $\{f(u), f(v)\} \in E'$. Inoltre, per quanto riguarda i grafi direzionati per un omomorfismo $f : G \rightarrow G'$, $(f(u), f(v))$ è un arco di G' se (u, v) è un arco di G .

I dati di cui sopra sono soggetti alle proprietà elencate nella seguente definizione.

Definizione 4 (Proprietà). Proprietà dell'omomorfismo:

1. Se esiste un omomorfismo $f : G \rightarrow H$ si può scrivere $G \rightarrow H$, e $G \rightarrowtail H$ altrimenti
2. Se $f : G \rightarrow H$, si dice che G è omomorfo di H o H -colorabile
3. Se l'omomorfismo $f : G \rightarrow G'$ è una biezione di cui la funzione inversa è anch'essa un omomorfismo grafico, allora f è un **isomorfismo grafico**
4. Due grafi G e G' sono **omorficamente equivalenti** se $G \rightarrow G'$ e $G' \rightarrow G$
5. Un sottografo H di un grafo G è un **retract** se vi è un omomorfismo $r : V(G) \rightarrow V(H)$, chiamato *retraction*, con $r(x) = x$, per ogni vertice $x \in V(H)$.
6. Un **core** è un grafo nel quale non esistono retract ad un sottografo proprio. Ogni grafo è omomorficamente equivalente ad un unico core
7. La composizione di omomorfismi sono omomorfismi
8. Gli omomorfismi grafici preservano la connettività

A.2 Approccio di trasformazione dei grafi Algebrico : Singolo Pushout e Doppio Pushout.

Il cuore dell'approccio algebrico è l'idea di esprimere un collage di grafi in termini categorici, come una costruzione di un pushout. Seguono la definizione delle regole di riscrittura dei grafi e la loro applicazione secondo i due principali approcci algebrici: singolo pushout e doppio pushout.

Definizione 5 (Categoria). Una categoria C è una collezione di oggetti, indicati $O_1, O_2, \dots$, insieme ad una collezione di morfismi, indicati $f, g, l, \dots$. Per ogni oggetto O , viene definito un morfismo speciale chiamato identità su O e indicato id . Per ogni morfismo f , una relazione chiamata *typing* è definita come $f : O_1 \rightarrow O_2$ dove $O_1 \rightarrow O_2$ è di tipo f e O_1 e O_2 sono chiamati sorgente e destinazione di f . Inoltre viene definita sull'insieme dei morfismi una operazione parziale chiamata composizione e indicata con $\circ$.

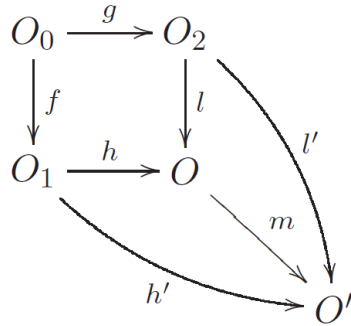
I dati di cui sopra sono soggetti agli assiomi elencati nella seguente definizione.

Definizione 6 (Assiomi). Assiomi del *typing*:

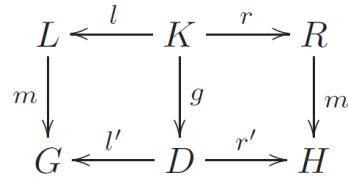
1. (Uncità del tipo) Se $f : O_1 \rightarrow O_2$ e $f : O'_1 \rightarrow O'_2$ allora $O_1 = O'_1$ e $O_2 = O'_2$;
2. (Composizione di tipi) Se $f : O_1 \rightarrow O_2$ e $g : O_2 \rightarrow O'_2$ allora $f \circ g : O_1 \rightarrow O'_2$;
3. (Tipo identità) $id : O \rightarrow O$.

Assiomi per la composizione di morfismi:

1. (Associatività) $(f \circ g) \circ l = f \circ (g \circ l)$
2. (Identità) $f \circ id = f = id \circ f$



(a) Pushout.



(b) Derivazione.

Figura 34: Pushout e derivazione (con approccio doppio pushout)

Definizione 7 (Pushout). Data una categoria C e una coppia di morfismi $f : O_0 \rightarrow O_1$ e $g : O_0 \rightarrow O_2$ di C , il pushout è una coppia di C -morfismi $(h : O_1 \rightarrow O, l : O_2 \rightarrow O)$ tali che siano soddisfatte le seguenti condizioni:

1. (Commutatività) $f \circ h = g \circ l$
2. (Universalità) Per ogni oggetto O' e morfismi $h' : O_1 \rightarrow O'$ e $l' : O_2 \rightarrow O'$ tali che $f \circ h' = g \circ l'$, esiste un unico morfismo $m : O \rightarrow O'$ tale che $h \circ m = h'$ e $l \circ m = l'$.

Definizione 8 (Doppio Pushout (DPO)). Secondo l'approccio doppio pushout (DPO) una regola $p = (L \leftarrow K \rightarrow R)$ è costituita da un grafo L nella parte sinistra, un grafo R nella parte destra e un grafo K di interfaccia, e una coppia di morfismi grafici $l : K \rightarrow L$, il quale deve essere iniettivo (elementi del dominio distinti portano sempre a immagini distinte) e $r : K \rightarrow R$.

Una regola può essere applicata, solo se tutti gli elementi di L sono presenti nel grafo G , cioè se esiste un morfismo totale $m : L \rightarrow G$ chiamato matching. Il grafo risultante viene costruito in due fasi, prima si costruisce il grafo D , ottenuto dal grafo dato G rimuovendo tutti gli elementi di G che hanno un'immagine in L ma non in K . In seguito si inseriscono in D tutti gli elementi di R che non hanno un'immagine in K , ottenendo H . Il grafo D deve rispettare le seguenti condizioni, in caso contrario la regola non può essere applicata.

Definizione 9 (Dangling Condition). *Se un nodo di G deve essere rimosso, allora anche ogni arco incidente ad esso, deve essere rimosso.*

Definizione 10 (Identification Condition (o precedenza della cancellazione rispetto alla preservazione)). *Non è possibile specificare in una regola che lo stesso oggetto deve essere sia preservato, che rimosso. Altrimenti verrà sempre rimosso.*

Definizione 11 (Singolo Pushout (SPO)). *Secondo l'approccio singolo pushout (SPO) una regola $p = (L \rightarrow R)$, è costituita da un grafo L nella parte sinistra, un grafo R nella parte destra e un morfismo p che fa un mapping dagli elementi di L agli elementi di R .*

Una regola può essere applicata, solo se tutti gli elementi di L sono presenti nel grafo G , cioè se esiste un morfismo totale $m : L \rightarrow G$ chiamato matching, e il grafo risultante H si ottiene rimuovendo gli elementi di G che si trovano in L e aggiungendo tutti gli elementi di R che non si trovano in L . In questo approccio una regola può essere sempre applicata, anche se nel matching con G , non viene rispettata la dangling condition discussa per il DPO. Infatti se succede questo allora vengono rimossi anche tutti gli archi incidenti al nodo rimosso.

In base a quanto detto sopra l'applicabilità delle regole di trasformazione del grafo dipende esclusivamente dall'esistenza di un matching m dalla parte sinistra della regola al grafo. In generale è possibile aggiungere alle regole delle condizioni di applicazione positive e negative, indipendenti dall'approccio algebrico. Ora si andranno a esaminare le regole di trasformazione con condizioni di applicazione negative, quelle utilizzate da GROOVE. Una regola di trasformazione equipaggiata con condizioni di applicazione negative (NACs Negative Application Conditions) richiede che siano soddisfatte *tutte* le condizioni per poter essere applicabile.

Quindi una regola di trasformazione p è estesa con un set $\mathcal{N}$ di condizioni di applicazione negative con un morfismo grafico $n : L \rightarrow N$ per ogni $NAC\ N \in \mathcal{N}$.

Definizione 12 (Soddisfazione NAC). *Data una regola di trasformazione p con una parte sinistra L_p e un set $\mathcal{N}_p$ di condizioni di applicazione negative, un grafo G , e un matching $m : L \rightarrow G$. $N \in \mathcal{N}_p$ sia una delle condizioni di applicazione negative di p . Allora, si dice che m soddisfa N se non esiste nessun morfismo grafico $m_N : N \rightarrow G$ tale che $m_N \circ n = m$, dove $n : L \rightarrow N$ è il morfismo grafico da L a N .*

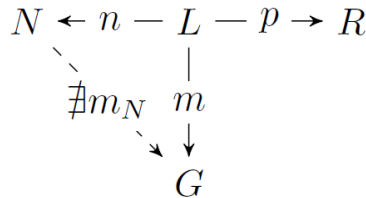


Figura 35: Applicazione di una regola con condizioni di applicazione negative.

Nella fig.3 la soddisfazione NAC è rappresentata schematicamente da una regola di trasformazione nell'approccio SPO. Una regola p con set $\mathcal{N}$ non vuoto si dice applicabile al grafo G se esiste un matching $m : L \rightarrow P$ che soddisfa tutte le condizioni di applicazione negative $N \in \mathcal{N}$.

Riferimenti bibliografici

- [1] <http://groove.sourceforge.net/groove-index.html>.
- [2] <http://groove.cs.utwente.nl>.
- [3] <http://colin.barker.pagesperso-orange.fr/lpa/dijkstra.html>.
- [4] Harmen Kastenberg Arend Rensink, Iovka Boneva and Tom Staijen. User manual for the groove tool set, 2011.
- [5] Clara Bertolissi. The graph rewriting calculus: properties and expressive capabilities, 2005.