UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Ingegneria - Dipartimento di Sistemi e Informatica

Tesi di Laurea in Ingegneria Informatica
Indirizzo Sistemi ed Applicazioni Informatici

Un Esecutore di Specifiche Logiche in Tempo Reale

Candidato
Andrea Giotti

Relatori
Prof. Ing. P. Nesi

Prof. Ing. G. Bucci

Correlatore
Ing. P. Bellini

Anno Accademico 1999–2000

Indice

Prefazione

Questa tesi è frutto di una lunga ricerca. Ringrazio la mia famiglia, Elena, gli amici per aver contribuito a renderla possibile con il loro sostegno. Altri ringraziamenti vanno a Paolo Nesi e Pierfrancesco Bellini per la fiducia e la pazienza dimostrate. Numerosi altri nomi meriterebbero di comparire in questo elenco, ma una tesi è troppo piccola per contenerli tutti.

Capitolo 1

Introduzione

Come il linguaggio naturale traccia i confini della conoscenza comunicabile da uomo a uomo, così i linguaggi formali delimitano i possibili contenuti della comunicazione tra uomo e macchina. Solitamente, questa comunicazione ha come oggetto più o meno esplicito un desiderio umano che la macchina dovrà soddisfare. Nonostante essa sia sempre fedele nella sua esecuzione, non tutti i desideri si realizzano. Come in una famosa favola, il genio esegue alla lettera quanto richiesto e quando il risultato non è quello gradito, al protagonista non resta che prendersela con se stesso. A ben guardare egli potrebbe invocare numerose attenuanti, perché per ogni formalismo esistono problemi abbastanza complessi da non ammettere soluzioni la cui formulazione sia di per sè evidentemente corretta. La soglia di complessità sopra cui il semplice esame della soluzione non è sufficiente a determinarne la correttezza può invece variare notevolmente con il livello del linguaggio in cui viene espressa. Tra i requisiti che ogni linguaggio formale dovrebbe soddisfare, i seguenti sono indici di un livello elevato:

- Espressività: deve essere possibile rappresentare i propri desideri
- Precisione: i desideri devono essere rappresentabili senza ambiguità
- Astrazione dal dettaglio: non deve essere necessario rappresentare dettagli ininfluenti
- Compattezza: la rappresentazione deve essere ragionevolmente breve
- Leggibilità: la rappresentazione deve essere comprensibile per altri lettori

Un modo abbastanza naturale di conseguire il risultato desiderato è quello di impartire alla macchina una successione di comandi elementari riconosciuti che conduce all'obiettivo. Questa è l'essenza della programmazione imperativa ed i linguaggi formali attraverso cui si esprime, dal codice macchina a quelli ad oggetti, sono detti semplicemente "di programmazione". Al crescere del loro livello si è però accompagnata una sempre maggiore rilevanza della sezione dichiarativa del programma, quella cioè in cui si descrivono gli enti coinvolti nell'esecuzione, rispetto a quella imperativa. Nel frattempo, da istruzioni modellate sulla struttura dell'hardware ci si è pian piano elevati verso una sintassi che ricalca le forme del pensiero, come avviene nei linguaggi di programmazione logica che si ispirano al calcolo dei predicati del primo ordine. Questi ultimi sono particolarmente importanti perché è possibile utilizzarli per scrivere programmi che ammettono una interpretazione completamente dichiarativa, cioè che possono essere letti come la definizione di un insieme di relazioni priva di qualunque comando esplicito. I linguaggi di programmazione logica permettono quindi un diverso stile di scrittura, detto appunto dichiarativo, che conferisce al programma proprietà rilevanti quali la possibilità di essere fuso con un diverso programma conservando la propria semantica.

Parallelamente ai linguaggi di programmazione si sviluppava una diversa categoria di linguaggi formali, detti "di specifica", il cui scopo è quello di consentire una definizione rigorosa dell'obiettivo da raggiungere, a prescindere dalla strada che verrà in effetti seguita. Le specifiche scritte facendo

uso di questi linguaggi sono poi utilizzate per verificare la consistenza dei vincoli di progetto e la loro successiva soddisfazione, possibilmente attraverso validatori automatici. In un certo senso, un programma logico è un particolare tipo di specifica eseguibile ed è ragionevole chiedersi se esistano algoritmi di esecuzione anche per altri linguaggi di specifica, con lo scopo di utilizzarli come linguaggi di programmazione dichiarativa. Questo approccio garantirebbe automaticamente la soddisfazione dei vincoli specificati, eliminando la fase di implementazione degli algoritmi in un linguaggio convenzionale.

L'affidabilità del software, cioè la garanzia del corretto funzionamento di un programma per tutti i possibili ingressi, è uno dei problemi aperti dell'ingegneria. Nel controllo di sistemi complessi, ad esempio, il punto debole solitamente non è costituito dall'hardware, per cui esistono metodologie di verifica affermate, quanto dal software preposto alla sua gestione. Per contribuire a risolvere questo problema si possono adottare linguaggi in cui i programmi fanno quello che dicono di fare, cioè il cui significato estensionale, l'insieme di tutte le possibili esecuzioni, è autoevidente dato il sorgente del programma stesso fino ad un grado ragionevole di complessità. Un linguaggio di specifica potrebbe essere un ottimo candidato a questo ruolo in quanto mette l'accento sull'obiettivo anziché sul metodo, pur di disporre dell'opportuno algoritmo di esecuzione.

Un comune pregiudizio nei confronti della programmazione dichiarativa è la sua presunta scarsa efficienza. In parte, questo è conseguenza di un suo uso troppo libero, oltre i limiti suggeriti dall'implementazione. In altre parole, non tutto ciò che si può scrivere in un linguaggio di specifica potrà essere eseguito con ragionevole efficienza, ma di solito esiste un modo per scrivere quelle stesse cose nel medesimo linguaggio tale da farle eseguire con elevata efficienza. Ad esempio, nella programmazione logica le prestazioni dipendono dal grado di determinismo che viene specificato durante la scrittura o semplicemente da un diverso ordinamento delle clausole. Non sfugge comunque il vantaggio insito nel disporre di un linguaggio in cui si possono scrivere programmi che fanno quello che dicono di fare, indipendentemente dall'efficienza con cui questo è in effetti possibile. Un passo ulteriore può essere infatti la riscrittura di un programma poco efficiente in uno migliore nello stesso linguaggio, magari condotta con ottimizzatori automatici per garantirne la correttezza.

Astrazione e prestazioni in esecuzione sono fattori generalmente costrastanti, e solo un accurato bilanciamento tra questi in vista di un certo ambito di applicazione può portare strumenti utili allo sviluppo di software più affidabile. In questa tesi si cercherà di sfatare il pregiudizio che vorrebbe i programmi dichiarativi poco prestanti scegliendo un ambito di applicazione particolarmente ostile, il controllo di sistemi in tempo reale. Al software di gestione di un microcontrollore è infatti richiesto di garantire:

- Produzione di uscite corrette per tutte le possibili storie degli ingressi
- Disponibilità delle uscite entro un tempo prefissato dall'istante di acquisizione degli ingressi

Inoltre, dato che tipicamente esso viene impiegato in apparecchiature di largo consumo e non è suscettibile di facile aggiornamento, è comprensibile che il mondo dell'industria non si accontenti della soddisfazione pratica dei due requisiti precedenti ma sempre più spesso ne richieda la dimostrabilità. Un programmatore capace può ottenere il risultato sperato anche attraverso un linguaggio convenzionale, ma difficilmente il codice prodotto si presterà ad alcuna forma di dimostrazione rigorosa.

L'approccio tradizionale allo sviluppo di un simile software è operazionale, nel senso che richiede una modellazione esplicita della successione dei passi di esecuzione, e si basa su formalismi quali macchine a stati, reti di Petri o loro varianti. Alternativamente sono utilizzati metodi descrittivi basati su linguaggi di specifica quali lo Z o svariate logiche temporali, come la TILCO da cui trae ispirazione la logica ridotta che verrà impiegata nel seguito. Evidentemente questi ultimi metodi consentono una maggiore astrazione, ma non assicurano l'effettiva eseguibilità del risultato. D'altra parte, neppure impongono una particolare strategia di esecuzione e questo apre la strada alla ricerca di algoritmi in grado di garantire il rispetto dei vincoli temporali assegnati.

L'obiettivo di questa tesi è appunto la realizzazione di uno strumento in grado di eseguire la specifica logica di un sistema causale in tempo reale, così da garantirne l'applicabilità ingegneri-

stica. Eseguire la specifica di un sistema significa a tutti gli effetti riprodurne il comportamento a partire dalla sua definizione in un certo linguaggio formale. Questo si realizza grazie ad un motore d'inferenza che, in ogni istante di tempo, produce conseguenze logiche implicate dalla specifica, assunta vera, data la conoscenza acquisita fino a quello stesso istante. Non tutte le specifiche risulteranno però eseguibili. Assumendo la correttezza dell'algoritmo di esecuzione, le possibili cause di ineseguibilità sono:

- Specifica del sistema non soddisfacibile
- Specifica del sistema incompleta
- Sistema specificato non causale
- Algoritmo di esecuzione incompleto

Agli errori di specifica si rimedia riscrivendo la stessa con maggiore attenzione, mentre la non causalità del sistema rende vano ogni tentativo di riprodurne il comportamento in tempo reale. Per quanto riguarda l'algoritmo di esecuzione, la sua incompletezza è solitamente il prezzo che si paga all'efficienza dato che qualunque algoritmo completo in grado di risolvere un generico problema di soddisfacibilità ha complessità esponenziale. Questo è il limite più evidente dei metodi descrittivi, poiché rende incerta la produzione di un risultato desiderato anche quando è conseguenza logica della specifica da eseguire. Tuttavia la programmazione dichiarativa è ampiamente utilizzata grazie ad un approccio misto che non perde di vista il significato operazionale che si nasconde dietro alla descrizione del sistema specificato e conduce alla scelta di una forma espressiva in grado di garantire l'eseguibilità. In questo caso, l'algoritmo di esecuzione deve essere stabilito precedentemente perché sia possibile aggirarne l'incompletezza durante la stesura della specifica. Nei comuni motori di inferenza per la programmazione logica, ad esempio, la ricerca in profondità sull'albero delle scelte costringe il programmatore ad assicurarsi che questo non contenga cammini infinitamente lunghi, frutto di definizioni ricorsive magari corrette ma certo eccessivamente contorte. Per questa ragione, il motore di inferenza prescelto è abbastanza semplice da poterne simulare manualmente il funzionamento nei casi di dubbia eseguibilità e consentire quindi una programmazione mirata.

In particolare si è preferito un tradizionale sistema di ragionamento in avanti, cioè basato su regole del tipo $\textit{premesse} \vdash \textit{conclusioni}$, ai metodi di ricerca legati all'obiettivo, accettando il prezzo di incompletezza derivante da questa scelta pur di poter enunciare una condizione che garantisse il funzionamento dell'esecutore in tempo reale. Ciò è reso possibile dalla strategia adottata per selezionare le clausole su cui fare inferenza, che è invece del tutto originale e può essere applicata con identici risultati a qualunque sistema di ragionamento in avanti, purché in ogni sua regola le *premesse* descrivano istanti di tempo non successivi a quelli descritti dalle *conclusioni*. La letteratura non presenta risultati simili e si è ritenuto opportuno formalizzarli, sviluppando una teoria assai generale che combina inferenza ed acquisizione di ingressi che variano nel tempo. Questa teoria prescinde dalla particolare modalità di rappresentazione della conoscenza e dalle regole di inferenza implementate.

Successivamente si è introdotto un formalismo innovativo per rappresentare i valori logici di un insieme di clausole attraverso l'orientazione degli archi di una speciale rete, abbastanza maneggevole da ammettere un utilizzo manuale ma anche adatta a costituire un formato di rappresentazione della conoscenza per il motore di inferenza precedentemente proposto. Inizialmente a questa rete è associata una semantica puramente descrittiva, legata alla sintassi della specifica, ma una volta fissate le regole di inferenza la sua interpretazione operazionale è immediata e ne consente una più libera e proficua manipolazione, agevolata dalla notazione grafica adottata. Questa duplice natura la rende uno strumento particolarmente adatto ad un approccio misto durante la stesura di specifiche eseguibili. Nonostante il formalismo adottato non presenti singoli aspetti in grado di assicurargli un vantaggio decisivo rispetto ad altri formalismi in uso, l'insieme delle sue caratteristiche può farne un valido concorrente.

La realizzazione di un esecutore di specifiche conforme a quanto teorizzato ha infine fornito il necessario riscontro sperimentale. Il minimo passo di discretizzazione sopra cui è dimostrato il suo funzionamento in tempo reale risulta lineare in alcuni dei parametri di interesse ed indipendente

dagli altri, superando così le aspettative più favorevoli. Dato che al software di gestione di un microcontrollore non è richiesta tanto la capacità di risolvere profondi problemi di ricerca, in cui anche una marginale incompletezza può essere inaccettabile, quanto quella di garantire risposte pronte a stimoli elementari, lo scopo di questa tesi appare sostanzialmente raggiunto.

Capitolo 2

Ragionamento in tempo reale

2.1 Definizioni generali

Si immagini di poter descrivere un sistema assegnato attraverso un insieme di asserzioni logiche, A, che comprende tutte le verità di interesse, affermate e negate, riguardo ad ogni possibile storia del sistema in questione. Rappresentare questa conoscenza significa associare ad un particolare sottoinsieme D di A una funzione *di rappresentazione* $\rho : D \subseteq A \rightarrow Z$ tale che:

1. $\forall\, a \in D \;\; \neg a \in D$
2. $\forall\, a, b \in D \;\; a \neq b \Rightarrow \rho(a) \neq \rho(b)$

L'insieme Z, o *di supporto*, è del tutto arbitrario e chiamando $Z_D = \rho(D)$ insieme *di rappresentazione* si può definire la funzione inversa come $\rho^{-1} : Z_D \rightarrow D$. Gli elementi di D e quelli di Z_D sono quindi in corrispondenza biunivoca ed entrambi questi insiemi si dicono *fondamentali* rispetto ad A se, indifferentemente:

- $\forall\, X \subset A \;\; X \not\models \bot \Rightarrow \exists\, Y \subset D \mid Y \not\models \bot, Y \models X$
- $\forall\, X \subset A \;\; X \not\models \bot \Rightarrow \exists\, S \subset Z_D \mid \rho^{-1}(S) \not\models \bot, \rho^{-1}(S) \models X$

In questo caso in Z_D è possibile rappresentare tutte le verità di interesse a in forma *esplicita*, se $a \in D$, o *implicita*, se $a \in A \setminus D$. Nel seguito Z_D verrà supposto fondamentale rispetto ad A.

È possibile definire l'operatore di negazione su Z_D come indotto dalla negazione logica su D:

- $\forall\, r \in Z_D \;\; \exists\, \neg r = \rho(\neg\rho^{-1}(r)) \in Z_D$
- $\forall\, S \subseteq Z_D \;\; \exists\, \overline{S} = \{r \mid \neg r \in S\} \subseteq Z_D$

Dalla definizione di negazione deriva che $Z_D = \overline{Z_D}$. Ad ogni possibile storia del sistema assegnato corrisponderà un particolare sottoinsieme dell'insieme di rappresentazione ed è conveniente definire la classe $\mathcal{Z}_D = \{S \mid S \subseteq Z_D\}$ che li contiene tutti. Un qualunque $S \in \mathcal{Z}_D$ viene classificato come:

- *Coerente* se $\forall\, r \in S \;\; \neg r \notin S$ ovvero $S \cap \overline{S} = \emptyset$
- *Corretto* se $\rho^{-1}(S) \not\models \bot$
- *Massimale* su Z_D se $\forall\, r \in Z_D \;\; r \in S \vee \neg r \in S$ ovvero $S \cup \overline{S} = Z_D$
- *Completo* rispetto ad A se $\forall\, a \in A \;\; \rho^{-1}(S) \models a \vee \rho^{-1}(S) \models \neg a$

La coerenza è una condizione necessaria, ma non sufficiente, per la correttezza. Z_D non gode di questa proprietà che sarà invece soddisfatta da tutti i suoi sottoinsiemi di interesse.

Una proprietà più forte della coerenza è la correttezza, cioè la garanzia di rappresentare conoscenza non contraddittoria. Questa proprietà non è immediatamente deducibile dalla struttura di S ma richiede l'interpretazione della conoscenza rappresentata in esso in termini di asserzioni logiche. La correttezza verrà assicurata postulandola per un insieme di assiomi e conservandola attraverso l'applicazione di regole d'inferenza corrette.

La massimalità indica che tutta la conoscenza rappresentabile sta in S, cioè che tutte le asserzioni in D trovano conferma o confutazione in $\rho^{-1}(S)$. Come la coerenza, è proprietà immediatamente deducibile dalla struttura di S ed è opportuno definire l'insieme degli elementi coerenti e massimali di $\mathcal{Z}_D$:

$$\mathcal{Z}_D^* = \{S \in \mathcal{Z}_D \mid S \cap \overline{S} = \emptyset, S \cup \overline{S} = Z_D\}$$

Un'ultima proprietà non deducibile dalla struttura è la completezza di S rispetto ad un insieme A di asserzioni logiche. Questa garantisce che la conoscenza rappresentata in S sia sufficiente a caratterizzare qualunque sistema descrivibile utilizzando gli elementi di A. È possibile assicurare la completezza facendo ipotesi sull'insieme di rappresentazione Z_D.

Teorema 2.1.1 *Se S è corretto e massimale su Z_D fondamentale rispetto ad A, allora S è completo rispetto ad A.*

Dimostrazione Si supponga che $\exists\, a \in A \mid \rho^{-1}(S) \not\models a, \rho^{-1}(S) \not\models \neg a$ e si considerino gli insiemi $\rho^{-1}(S) \cup \{a\}$ e $\rho^{-1}(S) \cup \{\neg a\}$: dato che si è supposto S corretto, almeno uno di questi due sottoinsiemi di A è non contraddittorio. Chiamando X questo insieme, per la definizione di Z_D fondamentale, $X \subset A, X \not\models \perp \Rightarrow \exists\, S_X \subset Z_D \mid \rho^{-1}(S_X) \not\models \perp, \rho^{-1}(S_X) \models X$. Dato che sia S che S_X sono inclusi in Z_D, si hanno due alternative:

1. $S_X \subseteq S$, quindi $\rho^{-1}(S_X) \subseteq \rho^{-1}(S)$ e $\rho^{-1}(S) \models X$. Ma questo comporta che $\rho^{-1}(S) \models a$ o $\rho^{-1}(S) \models \neg a$, in contraddizione con l'ipotesi iniziale.

2. $\exists\, r \in S_X \setminus S$, ma $r \notin S \Rightarrow \neg r \in S$ perché S è massimale su Z_D. Quindi $\rho^{-1}(S \cup S_X) = \rho^{-1}(S) \cup \rho^{-1}(S_X) \models \perp$, e dato che $\rho^{-1}(S) \subset X$ vale $\rho^{-1}(S_X) \models \rho^{-1}(S)$ ed è possibile concludere che $\rho^{-1}(S_X) \models \perp$, in contraddizione con la definizione di S_X.

L'utilità di questo teorema sta nel fatto che, disponendo di algoritmi corretti che operino su dati corretti, l'appartenenza a $\mathcal{Z}_D^*$ è condizione sufficiente per la completezza pur di aver scelto opportunamente D rispetto ad A. Visto che sugli elementi di $A \setminus D$ non è neppure definita una funzione di rappresentazione, l'unica possibilità operativa per verificare la completezza di S rispetto ad A resta controllare la massimalità di S su Z_D, operazione che può essere effettuata con relativa facilità. La scelta di un dominio fondamentale dovrebbe essere compiuta contestualmente alla definizione di A e resta comunque legata alla volontà di disporre di una rappresentazione in forma esplicita di talune asserzioni di interesse.

In questo quadro, una caratterizzazione completa del sistema assegnato sarà un particolare insieme $S^* \in \mathcal{Z}_D^* \mid \rho^{-1}(S^*) \not\models \perp$, mentre la conoscenza disponibile sul sistema sarà un qualunque $S \subseteq S^*$. All'insieme S^* scelto, detto *obiettivo*, si attribuisce verità a priori e da questo segue che $\rho^{-1}(S) \not\models \perp$, cioè la correttezza di ogni sottoinsieme dell'obiettivo è sempre garantita.

Incrementare la conoscenza disponibile $S_0 \subset S^*$ significa costruire un insieme $S_1 \mid S_0 \subset S_1 \subseteq S^*$. Questo può avvenire grazie a due procedimenti:

- Acquisendo nuova conoscenza sul sistema attraverso opportune funzioni d'ingresso
- Producendo nuova conoscenza a partire da quella disponibile attraverso l'applicazione di regole d'inferenza

Per il momento la funzione d'ingresso non verrà caratterizzata se non attraverso il suo nome e tipo, $\omega(\bullet) \in \mathcal{Z}_D$, e specificando che dovrà ritornare sottoinsiemi dell'obiettivo $\omega(\bullet) \subseteq S^*$, la cui unione è corretta per definizione. Questo comporta che un'asserzione acquisita dall'esterno come vera non potrà venir falsificata da nessuna successiva acquisizione. Se il sistema descritto sarà, come nel caso di studio, tempo-variante, un'asserzione relativa ad una variabile in due differenti istanti di tempo dovrà essere rappresentata da due distinti elementi di Z_D per garantire la coerenza della conoscenza disponibile al mutare della variabile.

2.2 Funzioni inferenza

Un processo di ragionamento deterministico su un insieme di rappresentazione Z_D può essere visto come la reiterata applicazione di una funzione *di inferenza* $\phi_D : \mathcal{Z}_D \to \mathcal{Z}_D$ tale che:

1. $\phi_D(\emptyset) = \emptyset$
2. $\forall\, S \in \mathcal{Z}_D \;\; S \subseteq \phi_D(S)$
3. $\forall\, R, S \in \mathcal{Z}_D \;\; R \subseteq S \Rightarrow \phi_D(R) \subseteq \phi_D(S)$
4. $\forall\, X \subseteq D \;\; S \in \mathcal{Z}_X \Rightarrow \phi_X(S) \subseteq \phi_D(S)$

La proprietà 1 esprime il fatto che in assenza di conoscenza iniziale nessuna inferenza è possibile. Il suo utilizzo non dichiarato è comune nel seguito.

La 2 stabilisce che la conoscenza cresce sempre durante l'inferenza ed il ragionamento procede in maniera monotona, senza mai rinnegare le verità acquisite. Se si rappresenta un sistema tempo-variante, verità relative ad una stessa variabile in istanti diversi dovranno ricevere rappresentazioni distinte perché questa proprietà possa valere. Inoltre, comunque sia scelta la funzione inferenza, questa non potrà processare in ingresso o rappresentare in uscita fatti ipotetici formulati ad uso di ulteriore inferenza, per la sua incapacità di rinnegarli quando questi avessero esaurito il loro compito. Se ipotesi di lavoro vengono fatte, ad esempio in dimostrazioni non costruttive, queste devono essere tutte interne alla funzione ϕ_D e vengono dimenticate tra un passo d'inferenza ed il successivo.

La 3 esprime il concetto che saperne di più è sempre meglio che saperne di meno, almeno quando lo scopo è quello di ragionare su quello che si sa per produrre ulteriore conoscenza, ed è essenziale nella stesura di algoritmi convergenti all'obiettivo.

La 4, di interesse più limitato, assicura che estendere l'insieme di rappresentazione non diminuisce la produzione di nuova conoscenza. Questo consente di unire due insiemi di conoscenza definiti come $S_X \subseteq Z_X \subseteq Z$ e $S_Y \subseteq Z_Y \subseteq Z$ a partire da domini diversi ma con supporto comune Z, in un unico insieme di conoscenza $S = S_X \cup S_Y \subseteq Z_D$, definito su un insieme di rappresentazione $Z_D = Z_X \cup Z_Y$.

Dato $Z_X \subseteq Z_D$ tale che $Z_X = \overline{Z_X}$ si definisce *restrizione* di ϕ_D a $\mathcal{Z}_X$ la funzione $\phi_X : \mathcal{Z}_X \to \mathcal{Z}_X$ soddisfacente:

$$\phi_X(S) = \phi_D(S) \cap Z_X$$

Tra le tante funzioni inferenza compatibili con la proprietà 4 che si possono definire su un sottoinsieme dell'insieme di rappresentazione, la restrizione è quella che le include tutte.

La funzione inferenza, d'ora in poi semplicemente ϕ, si dice inoltre *corretta* se:

$$\forall\, S \in \mathcal{Z}_D \;\; \rho^{-1}(S) \models \rho^{-1}(\phi(S))$$

La correttezza garantisce che la conoscenza prodotta sia inclusa nell'obiettivo S^* ed è sempre ipotizzata nel seguito.

Teorema 2.2.1 *Se $\phi : \mathcal{Z}_D \to \mathcal{Z}_D$ e $S^* \in \mathcal{Z}_D^*$ sono corretti, $\forall\, S \subseteq S^* \;\; \phi(S) \subseteq S^*$.*

Dimostrazione Si supponga che $\exists\, r \in \phi(S) \mid r \notin S^*$. Per la definizione di correttezza $\rho^{-1}(S) \models \rho^{-1}(r)$, e $S \subseteq S^*$ implica $\rho^{-1}(S^*) \models \rho^{-1}(r)$. D'altra parte, per la massimalità di S^*, $r \in Z_D = S^* \cup \overline{S^*}$ e l'ipotesi $r \notin S^*$ fa concludere che $r \in \overline{S^*}$. Dunque $\neg r \in S^*$ e $\rho^{-1}(\neg r) \in \rho^{-1}(S^*) \models \rho^{-1}(r)$, contraddicendo la correttezza di S^*.

Una conseguenza di questo risultato è che per ogni obiettivo S^*, per definizione corretto, $S^* \subseteq S^* \Rightarrow \phi(S^*) \subseteq S^*$ se ϕ è corretta. Ma per la proprietà 2 si può anche scrivere $S^* \subseteq S^* \Rightarrow S^* \subseteq \phi(S^*)$ ed il confronto delle due inclusioni fa concludere che $\forall\, S^* \in \mathcal{Z}_D^* \mid \rho^{-1}(S^*) \not\models \perp \;\; S^* = \phi(S^*)$, cioè tutti gli elementi corretti di $\mathcal{Z}_D^*$ sono insiemi invarianti per una ϕ corretta.

Dato che entrambe le funzioni che incrementano la conoscenza disponibile $S \subseteq S^*$, ω e ϕ, ritornano sottoinsiemi di S^* è ragionevole chiedersi se S tende a S^* iterando la loro applicazione per un numero sufficiente di passi. Il problema dell'effettivo raggiungimento dell'insieme obiettivo coinvolge la *completezza* della funzione inferenza e non verrà affrontato in questa sede. Garantirla ha infatti un costo computazionale intollerabile per applicazioni tempo reale e d'altra parte produrre una descrizione completa del sistema assegnato può essere superfluo se si è interessati a determinare solo il valore di alcune variabili.

Risulta invece essenziale assicurare la convergenza della conoscenza disponibile ad un insieme invariante $\widehat{S} \subseteq S^*$ a seguito di una reiterata applicazione della funzione ϕ. Questo si verifica sempre se, a fronte di un numero anche infinito di asserzioni logiche di interesse A, il dominio D su cui è definita la rappresentazione è stato scelto finito. L'ipotesi di operare su insiemi di rappresentazione finiti deve verificarsi perché si possa sperare in una effettiva implementazione della base di conoscenza e verrà quindi tacitamente postulata negli algoritmi che seguiranno.

Per dimostrare la tesi è necessario definire l'applicazione della funzione ϕ per k volte ad S, $\phi^k(S) = \phi_k(\phi_{k-1}(\cdots \phi_2(\phi_1(S)) \cdots))$, che in forma ricorsiva si scrive:

1. $\phi^0(S) = S$
2. $\forall\, k \geq 1 \;\; \phi^k(S) = \phi(\phi^{k-1}(S))$

La funzione ϕ^k gode delle proprietà indotte dalla definizione di ϕ:

1. $\forall\, k \geq 0 \;\; \phi^k(\emptyset) = \emptyset$
2. $\forall\, k \geq 0, \forall\, S \in \mathcal{Z}_D \;\; S \subseteq \phi^k(S)$
3. $\forall\, k \geq 0, \forall\, R, S \in \mathcal{Z}_D \;\; R \subseteq S \Rightarrow \phi^k(R) \subseteq \phi^k(S)$
4. $\forall\, k \geq 0, \forall\, X \subseteq D \;\; S \in \mathcal{Z}_X \Rightarrow \phi_X^k(S) \subseteq \phi_D^k(S)$

La proprietà 1 deriva immediatamente dal fatto che $\emptyset$ è insieme invariante per ϕ.

La 2 risulta evidente se scritta come $S = \phi^0(S) \subseteq \phi^1(S) \subseteq \phi^2(S) \subseteq \cdots \subseteq \phi^{k-1}(S) \subseteq \phi^k(S)$. Un modo alternativo di scriverla è $\forall\, k, p \geq 0 \;\; \phi^k(S) \subseteq \phi^{k+p}(S)$.

La 3 può essere dimostrata per induzione su k. Se $R \subseteq S$, certamente $R = \phi^0(R) \subseteq \phi^0(S) = S$. Supponendo che sia vera al passo k, vale $R^{(k)} = \phi^k(R) \subseteq \phi^k(S) = S^{(k)}$ e l'applicazione della funzione ϕ a questi due insiemi produce $\phi(R^{(k)}) \subseteq \phi(S^{(k)})$, cioè $\phi^{k+1}(R) \subseteq \phi^{k+1}(S)$.

Se la funzione inferenza è corretta ogni sua applicazione a un sottoinsieme dell'obiettivo S^*, per definizione corretto, restituisce un sottoinsieme di S^*. Una ulteriore proprietà è quindi che $\forall\, k \geq 0, \forall\, S \subseteq S^* \;\; \phi^k(S) \subseteq S^*$.

Definiamo *chiusura* di S rispetto a ϕ l'insieme:

$$\widehat{S} = \widehat{\phi}(S) = \lim_{k \to \infty} \phi^k(S)$$

e $\widehat{\phi} : \mathcal{Z}_D \to \mathcal{Z}_D$ *operatore di chiusura*. Dalla definizione di limite segue che $\phi(\widehat{S}) = \widehat{S}$, cioè la chiusura è un insieme invariante per ϕ.

Teorema 2.2.2 *Se $D \subseteq A$ è finito $\forall\, S \in \mathcal{Z}_D \;\; \exists\, \widehat{S} = \widehat{\phi}(S) = \lim_{k \to \infty} \phi^k(S)$.*

Dimostrazione La proprietà 2 stabilisce che $\forall\, k \geq 0 \;\; S^{(k)} = \phi^k(S) \subseteq \phi^{k+1}(S) \subseteq Z_D$. Ad ogni passo k di applicazione della ϕ si hanno due possibilità:

1. $S^{(k)} = \phi(S^{(k)})$. In questo caso $\widehat{S} = S^{(k)}$ è la chiusura a cui la successione converge e $\widehat{\phi} = \phi^k$.
2. $S^{(k)} \subset \phi(S^{(k)})$. In questo caso $\exists\, r \in \phi(S^{(k)}) \setminus S^{(k)}$, e la cardinalità dell'insieme $S^{(k+1)}$ viene aumentata almeno di uno rispetto al passo precedente. Finché si verifica questo caso, la successione delle cardinalità $\{\operatorname{card} S^{(k)}\}$ è strettamente crescente di una quantità ≥ 1 ad ogni passo.

Ma gli $S^{(k)}$ sono inclusi in Z_D che, essendo in corrispondenza biunivoca con D, è un insieme finito. La successione delle cardinalità è quindi limitata superiormente da $\operatorname{card} Z_D$ ed il secondo caso si potrà verificare un numero finito di volte $n \leq \operatorname{card} Z_D$, pari al numero di passi necessario per giungere alla convergenza.

Dalla definizione si ricava anche che $\forall\, k \geq 0 \;\; \phi^k(S) \subseteq \widehat{\phi}(S)$ e più precisamente:

- $\forall\, k \geq 0, k < n \;\; \phi^k(S) \subset \widehat{\phi}(S)$
- $\forall\, k \geq n \;\; \phi^k(S) = \widehat{\phi}(S)$

L'operatore di chiusura può dunque sempre essere scritto come $\widehat{\phi} = \phi^n$ e le quattro proprietà che definiscono la funzione inferenza ϕ continuano a valere per la $\widehat{\phi}$:

1. $\widehat{\emptyset} = \emptyset$
2. $\forall\, S \in \mathcal{Z}_D \;\; S \subseteq \widehat{S}$
3. $\forall\, R, S \in \mathcal{Z}_D \;\; R \subseteq S \Rightarrow \widehat{R} \subseteq \widehat{S}$
4. $\forall\, X \subseteq D \;\; S \in \mathcal{Z}_X \Rightarrow \widehat{\phi}_X(S) \subseteq \widehat{\phi}_D(S)$

Se inoltre ϕ è corretta è vero anche che $\forall\, S \subseteq S^* \;\; \widehat{S} \subseteq S^*$. In questo caso è possibile fornire una stima migliore del numero di passi necessari alla successione degli $S^{(k)}$ per convergere alla chiusura, cioè $n \leq \operatorname{card} S^* = \operatorname{card} Z_D/2$.

Dalla proprietà 3 si ricava che, dato un qualunque insieme invariante $\widehat{S} \in \mathcal{Z}_D$, $R \subseteq \widehat{S} \Rightarrow \phi(R) \subseteq \widehat{\phi}(R) \subseteq \widehat{\phi}(\widehat{S}) \Rightarrow \phi(R) \subseteq \widehat{R} \subseteq \widehat{S}$. Questo risultato evidenzia due ulteriori proprietà di interesse in prossime dimostrazioni:

- Per ogni insieme invariante, la funzione ϕ trasforma suoi sottoinsiemi in altri suoi sottoinsiemi.
- Ogni insieme invariante include la chiusura di ogni suo sottoinsieme.

Operare la chiusura dell'insieme di conoscenza disponibile S significa esplicitarne tutte le conseguenze logiche inferibili che ammettono una rappresentazione in Z_D. Simbolicamente questo verrà espresso scrivendo $\widehat{S} = \{r \in Z_D \mid \rho^{-1}(S) \vdash \cdots \vdash \rho^{-1}(r)\}$ e costituisce il massimo ottenibile con la funzione inferenza scelta. Infatti la proprietà 3 garantisce che, operando per incrementi successivi, tutto ciò che avrebbe potuto venir inferito a partire da sottinsiemi della conoscenza presente può ancora venire inferito. Questo sarà dimostrato in una forma leggermente diversa, utile nella scrittura di teoremi successivi.

Teorema 2.2.3 $\forall\, R, S \in \mathcal{Z}_D \;\; \widehat{\phi}(R \cup S) = \widehat{\phi}(R \cup \widehat{\phi}(S))$.

Dimostrazione Sia $\widehat{S} = \widehat{\phi}(S)$. La proprietà 2 assicura che $S \subseteq \widehat{S}$ mentre per la 3 vale $S \subseteq \widehat{S} \Rightarrow \widehat{\phi}(R \cup S) \subseteq \widehat{\phi}(R \cup \widehat{S})$. Per la proprietà 2, $R \subseteq \widehat{\phi}(R \cup S)$ mentre dalla 3 deriva $\widehat{S} = \widehat{\phi}(S) \subseteq \widehat{\phi}(R \cup S)$. Quindi $\exists Q \in \mathcal{Z}_D \mid \widehat{\phi}(R \cup S) = Q \cup R \cup \widehat{S}$. Ma per l'invarianza della chiusura $\widehat{\phi}(R \cup S) = \widehat{\phi}(\widehat{\phi}(R \cup S)) = \widehat{\phi}(Q \cup R \cup \widehat{S})$ e la proprietà 3 stabilisce che $\widehat{\phi}(R \cup \widehat{S}) \subseteq \widehat{\phi}(Q \cup R \cup \widehat{S}) = \widehat{\phi}(R \cup S)$. Da $\widehat{\phi}(R \cup S) \subseteq \widehat{\phi}(R \cup \widehat{S})$ e $\widehat{\phi}(R \cup \widehat{S}) \subseteq \widehat{\phi}(R \cup S)$ si può concludere che $\widehat{\phi}(R \cup S) = \widehat{\phi}(R \cup \widehat{\phi}(S))$.

Questo teorema permette di partizionare in modo arbitrario l'insieme della conoscenza disponibile, operando la chiusura dei sottoinsiemi scelti in qualunque ordine si preferisca purchè il risultato di ognuna di queste operazioni sia incluso nell'argomento della successiva. Ciò si rivela utile quando questo insieme non è dato a priori ma viene costruito incrementalmente attraverso lotti di dati restituiti dalla funzione d'ingresso in differenti istanti di tempo, come accade nei sistemi tempo reale. Cambiare l'ordine in cui i sottoinsiemi di S vengono elaborati dalla funzione inferenza non modifica infatti la convergenza verso l'unica chiusura $\widehat{S}$.

2.3 Inferenza condizionata e complementare

Per rendere praticabile la chiusura da un punto di vista computazionale è necessario esaminare più nel dettaglio le proprietà della ϕ. Si definisce *valore aggiunto* della coppia R, S la funzione $\xi(R,S) : \mathcal{Z}_D \times \mathcal{Z}_D \to \mathcal{Z}_D$ tale che:

$$\xi(R,S) = \phi(R \cup S) \setminus (\phi(R) \cup \phi(S))$$

Dalla definizione si ricavano alcune proprietà del valore aggiunto:

1. $\forall\, S \in \mathcal{Z}_D \;\; \xi(S,\emptyset) = \emptyset$
2. $\forall\, S \in \mathcal{Z}_D \;\; \xi(S,S) = \emptyset$
3. $\forall\, R,S \in \mathcal{Z}_D \;\; \xi(R,S) = \xi(S,R)$

La proprietà commutativa, 3, è implicata dalla commutatività dell'operatore unione. Quando $\xi(R,S) = \emptyset$ i due insiemi R, S si dicono *indipendenti* e si può quindi dire che l'insieme vuoto è indipendente da tutti gli altri e che ogni insieme è indipendente da se stesso.

Fare inferenza significa integrare informazioni diverse per giungere a nuove conclusioni ed in questo processo il tutto, per così dire, vale più della somma delle parti che lo compongono. Il valore aggiunto di una coppia di insiemi può essere interpretato come la conoscenza addizionale inferibile disponendo di entrambi rispetto alla semplice unione delle conoscenze inferibili operando sui singoli insiemi. Questo viene dimostrato nel seguente teorema:

Teorema 2.3.1 $\forall\, R,S \in \mathcal{Z}_D \;\; \phi(R \cup S) = \phi(R) \cup \phi(S) \cup \xi(R,S)$.

Dimostrazione La definizione di valore aggiunto, $\xi(R,S) = \phi(R \cup S) \setminus (\phi(R) \cup \phi(S))$, implica $\phi(R) \cup \phi(S) \cup \xi(R,S) = \phi(R) \cup \phi(S) \cup \phi(R \cup S)$. Per la proprietà 3, $R,S \subseteq R \cup S \Rightarrow \phi(R) \subseteq \phi(R \cup S), \phi(S) \subseteq \phi(R \cup S)$ e quindi $\phi(R) \cup \phi(S) \cup \xi(R,S) = \phi(R \cup S)$.

Teorema 2.3.2 $\forall\, R,S \in \mathcal{Z}_D \;\; \xi(R,S) = \emptyset \iff \phi(R \cup S) = \phi(R) \cup \phi(S)$.

Dimostrazione Per il teorema precedente vale $\phi(R \cup S) = \phi(R) \cup \phi(S) \cup \xi(R,S)$ e supponendo $\xi(R,S) = \emptyset$ si conclude che $\phi(R \cup S) = \phi(R) \cup \phi(S)$. Se invece si suppone $\phi(R \cup S) = \phi(R) \cup \phi(S)$ la definizione di ξ si riscrive come $\xi(R,S) = \phi(R \cup S) \setminus (\phi(R) \cup \phi(S)) = \phi(R \cup S) \setminus \phi(R \cup S) = \emptyset$.

Una diversa formalizzazione dello stesso concetto può essere ottenuta definendo la funzione *di inferenza condizionata*, $\phi(R \mid S) : \mathcal{Z}_D \times \mathcal{Z}_D \to \mathcal{Z}_D$, come:

$$\phi(R \mid S) = \phi(R \cup S) \setminus (\phi(S) \setminus \phi(R))$$

Il simbolo "|" tra R ed S si legge "dato" e la funzione rappresenta ciò che si può dedurre da R con un passo d'applicazione della ϕ se consideriamo già noto l'insieme S. La funzione inferenza condizionata gode di alcune proprietà notevoli:

1. $\forall\, S \in \mathcal{Z}_D \;\; \phi(\emptyset \mid S) = \emptyset$
2. $\forall\, R \in \mathcal{Z}_D \;\; \phi(R \mid \emptyset) = \phi(R)$

3. $\forall\, R \in \mathcal{Z}_D \;\; \phi(R \mid R) = \phi(R)$
4. $\forall\, R, S \in \mathcal{Z}_D \;\; R \subseteq \phi(R) \subseteq \phi(R \mid S)$
5. $\forall\, Q, R, S \in \mathcal{Z}_D \;\; Q \subseteq R \Rightarrow \phi(Q \mid S) \subseteq \phi(R \mid S)$

La proprietà 4 deriva dal fatto che $R \subseteq \phi(R) \subseteq \phi(R \cup S)$ e $\phi(R) \cap (\phi(S) \setminus \phi(R)) = \emptyset$, cioè $R \subseteq \phi(R) \subseteq \phi(R \cup S) \setminus (\phi(S) \setminus \phi(R)) = \phi(R \mid S)$.

La proprietà 5 merita una dimostrazione più articolata. Da $Q \subseteq R$ derivano $\phi(Q) \subseteq \phi(R)$ e $\phi(Q \cup S) \subseteq \phi(R \cup S)$. La prima implica $\phi(S) \setminus \phi(R) \subseteq \phi(S) \setminus \phi(Q)$ che, assieme alla seconda, produce $\phi(Q \cup S) \setminus (\phi(S) \setminus \phi(Q)) \subseteq \phi(R \cup S) \setminus (\phi(S) \setminus \phi(R))$ ovvero $\phi(Q \mid S) \subseteq \phi(R \mid S)$.

In generale, inoltre, $R \subseteq S \not\Rightarrow \phi(Q \mid R) \subseteq \phi(Q \mid S)$. Le condizioni sotto cui l'implicazione vale sono state esaminate ma i risultati non vengono acclusi perché di scarso interesse applicativo.

Controesempio Si considerino due insiemi invarianti $\widehat{R}$ e $\widehat{S}$ tali che $\phi(\widehat{R} \cup \widehat{S}) = \widehat{R} \cup \widehat{S} \cup \xi(\widehat{R}, \widehat{S})$ sia a sua volta insieme invariante. Si verifica facilmente che $\phi(\widehat{R} \mid \widehat{S}) = \widehat{R} \cup \xi(\widehat{R}, \widehat{S})$ mentre $\phi(\widehat{R} \mid \widehat{S} \cup \xi(\widehat{R}, \widehat{S})) = \widehat{R}$.

Le funzioni valore aggiunto e inferenza condizionata sono strettamente legate come viene dimostrato nei seguenti teoremi:

Teorema 2.3.3 $\forall\, R, S \in \mathcal{Z}_D \;\; \phi(R \mid S) \setminus \phi(R) = \xi(R, S)$.

Dimostrazione Dalla definizione di inferenza condizionata si ricava $\phi(R \mid S) \setminus \phi(R) = (\phi(R \cup S) \setminus (\phi(S) \setminus \phi(R))) \setminus \phi(R) = \phi(R \cup S) \setminus (\phi(R) \cup \phi(S)) = \xi(R, S)$.

Teorema 2.3.4 $\forall\, R, S \in \mathcal{Z}_D \;\; \phi(R \mid S) = \phi(R) \cup \xi(R, S)$.

Dimostrazione Il teorema precedente stabilisce che $\phi(R \mid S) \setminus \phi(R) = \xi(R, S)$, quindi $\phi(R) \cup \phi(R \mid S) = \phi(R) \cup \xi(R, S)$. Ma la proprietà 4 garantisce che $\phi(R) \subseteq \phi(R \mid S)$ e si può concludere che $\phi(R \mid S) = \phi(R) \cup \xi(R, S)$.

Due insiemi di conoscenza R, S si dicono *indipendenti* se $\phi(R \mid S) = \phi(R)$. Questa definizione è equivalente a quella precedentemente data come dimostrato di seguito.

Teorema 2.3.5 $\forall\, R, S \in \mathcal{Z}_D \;\; \xi(R, S) = \emptyset \iff \phi(R \mid S) = \phi(R)$.

Dimostrazione I teoremi precedenti assicurano che $\phi(R \mid S) = \phi(R) \cup \xi(R, S)$ e $\xi(R, S) = \phi(R \mid S) \setminus \phi(R)$. Supponendo $\xi(R, S) = \emptyset$ dalla prima si ottiene $\phi(R \mid S) = \phi(R)$. Se, al contrario, si suppone $\phi(R \mid S) = \phi(R)$ la seconda diviene $\xi(R, S) = \phi(R) \setminus \phi(R) = \emptyset$.

La funzione inferenza condizionata $\phi(R \mid S)$ non gode della proprietà commutativa, diversamente dal valore aggiunto, ma differenzia l'insieme su cui si sta effettivamente facendo inferenza, R, dal background di conoscenza disponibile S. Appare dunque più espressiva e verrà utilizzata nelle definizioni successive. Questa stessa caratteristica suggerisce inoltre che si possa realizzare questa funzione in modo che la sua complessità computazionale dipenda in modo diverso dalla dimensione di R ed S.

In realtà, la funzione inferenza condizionata non è la più adatta allo scopo. È necessario introdurre un'ultima funzione *di inferenza complementare*, $\psi(R \mid S) : \mathcal{Z}_D \times \mathcal{Z}_D \to \mathcal{Z}_D$, definita come:

$$\psi(R \mid S) = \phi(R \cup S) \setminus \phi(S)$$

La stretta parentela con la definizione precedente è palese ma la funzione inferenza complementare gode di un minor numero di proprietà notevoli:

1. $\forall\, S \in \mathcal{Z}_D \;\; \psi(\emptyset \mid S) = \emptyset$

2. $\forall\, R \in \mathcal{Z}_D \;\; \psi(R \mid \emptyset) = \phi(R)$

3. $\forall\, R \in \mathcal{Z}_D \;\; \psi(R \mid R) = \emptyset$

4. $\forall\, Q, R, S \in \mathcal{Z}_D \;\; Q \subseteq R \Rightarrow \psi(Q \mid S) \subseteq \psi(R \mid S)$

La proprietà 4 si dimostra facilmente ricordando che $Q \subseteq R$ implica $\phi(Q \cup S) \subseteq \phi(R \cup S)$ e quindi $\psi(Q \mid S) = \phi(Q \cup S) \setminus \phi(S) \subseteq \phi(R \cup S) \setminus \phi(S) = \psi(R \mid S)$.

La funzione inferenza complementare viene introdotta perché ammette una comoda formulazione ricorsiva. Prima di presentarla è però opportuno esaminare il rapporto che intercorre tra questa e le funzioni valore aggiunto ed inferenza condizionata.

Teorema 2.3.6 $\forall\, R, S \in \mathcal{Z}_D \;\; \psi(R \mid S) \setminus (\phi(R) \setminus \phi(S)) = \xi(R, S)$.

Dimostrazione Dalla definizione di inferenza complementare si ottiene $\psi(R \mid S) \setminus (\phi(R) \setminus \phi(S)) = (\phi(R \cup S) \setminus \phi(S)) \setminus (\phi(R) \setminus \phi(S)) = \phi(R \cup S) \setminus (\phi(R) \cup \phi(S)) = \xi(R, S)$.

Teorema 2.3.7 $\forall\, R, S \in \mathcal{Z}_D \;\; \psi(R \mid S) = \phi(R) \setminus \phi(S) \cup \xi(R, S)$.

Dimostrazione Il teorema precedente afferma che $\psi(R \mid S) \setminus (\phi(R) \setminus \phi(S)) = \xi(R, S)$, da cui deriva $\phi(R) \setminus \phi(S) \cup \psi(R \mid S) = \phi(R) \setminus \phi(S) \cup \xi(R, S)$. Ma $\phi(R) \setminus \phi(S) \subseteq \phi(R \cup S) \setminus \phi(S) = \psi(R \mid S)$ e la formula precedente diviene $\psi(R \mid S) = \phi(R) \setminus \phi(S) \cup \xi(R, S)$.

Teorema 2.3.8 $\forall\, R, S \in \mathcal{Z}_D \;\; \xi(R, S) = \emptyset \iff \psi(R \mid S) = \phi(R) \setminus \phi(S)$.

Dimostrazione I teoremi precedenti permettono di scrivere $\psi(R \mid S) = \phi(R) \setminus \phi(S) \cup \xi(R, S)$ e $\xi(R, S) = \psi(R \mid S) \setminus (\phi(R) \setminus \phi(S))$. Supponendo $\xi(R, S) = \emptyset$ dalla prima si ricava $\psi(R \mid S) = \phi(R) \setminus \phi(S)$. Se invece si suppone $\psi(R \mid S) = \phi(R) \setminus \phi(S)$ la seconda implica $\xi(R, S) = (\phi(R) \setminus \phi(S)) \setminus (\phi(R) \setminus \phi(S)) = \emptyset$.

Per l'equivalenza tra le due definizioni di indipendenza, un modo diverso per scrivere il precedente teorema è $\phi(R \mid S) = \phi(R) \iff \psi(R \mid S) = \phi(R) \setminus \phi(S)$. Confrontando inoltre le definizioni di ϕ e ψ e ricordando che $\phi(S) \setminus \phi(R) \subseteq \phi(S)$ si osserva che $\psi(R \mid S) \subseteq \phi(R \mid S)$. I seguenti teoremi approfondiscono questa osservazione.

Teorema 2.3.9 $\forall\, R, S \in \mathcal{Z}_D \;\; \phi(R \mid S) \setminus \psi(R \mid S) = \phi(R) \cap \phi(S)$.

Dimostrazione Per due teoremi precedenti, $\phi(R \mid S) = \phi(R) \cup \xi(R, S)$ e $\psi(R \mid S) = \phi(R) \setminus \phi(S) \cup \xi(R, S)$. La definizione di valore aggiunto garantisce che $\phi(R) \cap \xi(R, S) = \emptyset$, quindi $\phi(R \mid S) \setminus \psi(R \mid S) = (\phi(R) \cup \xi(R, S)) \setminus (\phi(R) \setminus \phi(S) \cup \xi(R, S)) = \phi(R) \setminus (\phi(R) \setminus \phi(S)) = \phi(R) \cap \phi(S)$.

Teorema 2.3.10 $\forall\, R, S \in \mathcal{Z}_D \;\; \phi(R \mid S) = \psi(R \mid S) \cup (\phi(R) \cap \phi(S))$.

Dimostrazione Dal teorema precedente, $\phi(R \mid S) \setminus \psi(R \mid S) = \phi(R) \cap \phi(S)$ e quindi $\psi(R \mid S) \cup \phi(R \mid S) = \psi(R \mid S) \cup (\phi(R) \cap \phi(S))$. Ma $\psi(R \mid S) \subseteq \phi(R \mid S)$ e si può concludere che $\phi(R \mid S) = \psi(R \mid S) \cup (\phi(R) \cap \phi(S))$.

Un primo vantaggio della scelta di rappresentare il ragionamento attraverso la funzione ψ è nella possibilità di esprimerla ricorsivamente sui sottoinsiemi di $\mathcal{Z}_D$.

Teorema 2.3.11 $\forall\, Q, R, S \in \mathcal{Z}_D \;\; \psi(Q \cup R \mid S) = \psi(Q \mid R \cup S) \cup \psi(R \mid S)$. *Inoltre* $\psi(Q \mid R \cup S) \cap \psi(R \mid S) = \emptyset$.

1. $\Psi \leftarrow \emptyset$
2. Finché $R \neq \emptyset$ ripeti...
   (a) $\exists\, r \in R$
   (b) $\Psi \leftarrow \Psi \cup \psi(\{r\} \mid S)$
   (c) $R \leftarrow R \setminus \{r\}$
   (d) $S \leftarrow S \cup \{r\}$

Figura 2.1. Calcolo di $\Psi = \psi(R \mid S)$

Dimostrazione Dalla definizione, $\psi(Q \cup R \mid S) = \phi(Q \cup R \cup S) \setminus \phi(S)$. Ma $\phi(S) \subseteq \phi(R \cup S) \subseteq \phi(Q \cup R \cup S)$ e la precedente espressione può riscriversi $\phi(Q \cup R \cup S) \setminus \phi(R \cup S) \cup \phi(R \cup S) \setminus \phi(S) = \psi(Q \mid R \cup S) \cup \psi(R \mid S)$. Peraltro, $\psi(R \mid S) \subseteq \phi(R \cup S)$ mentre $\psi(Q \mid R \cup S) \cap \phi(R \cup S) = \emptyset$ e da questo si può concludere che $\psi(Q \mid R \cup S) \cap \psi(R \mid S) = \emptyset$.

Se si dispone di un metodo per calcolare la $\psi(\{r\} \mid S)$, cioè l'inferenza su un singolo elemento r dato un sottoinsieme arbitrario S dell'insieme di rappresentazione, il problema del calcolo della generica $\psi(R \mid S)$ può essere risolto con il semplice algoritmo di figura 2.1.

Teorema 2.3.12 $\forall\, R, S \in \mathcal{Z}_D$, *l'algoritmo calcola* $\Psi = \psi(R \mid S)$.

Dimostrazione Sia $R = \{r_1, \ldots, r_n\}$, con gli r_k numerati nell'ordine in cui verranno scelti, uno per passo, per essere valutati e rimossi da R. L'algoritmo non specifica alcun criterio di scelta e se ne può assumere uno arbitrario in base al quale ordinare R. Si osserva che $R^{(0)} = R$, $S^{(0)} = S$ e $\Psi^{(0)} = \emptyset$. Inoltre, $\forall\, k \geq 1, k \leq n \;\; R^{(k)} = R^{(k-1)} \setminus \{r_k\} = \{r_{k+1}, \ldots, r_n\}$ e $S^{(k)} = S^{(k-1)} \cup \{r_k\} = S \cup \{r_1, \ldots, r_k\}$.

1. Per $n = 0$, $R = \emptyset$ e l'algoritmo produce $\Psi^{(0)} = \emptyset$ conformemente alla proprietà $\psi(\emptyset \mid S) = \emptyset$.
2. Si supponga la tesi vera per $n - 1$. Al passo $k = n - 1$ l'algoritmo produce una $\Psi^{(k)} = \psi(\{r_1, \ldots, r_k\} \mid S)$ per ipotesi d'induzione. Al passo $k + 1 = n$ ad essa viene aggiunta $\psi(\{r_{k+1}\} \mid S^{(k)}) = \psi(\{r_{k+1}\} \mid S \cup \{r_1, \ldots, r_k\})$ ed il risultato viene memorizzato in $\Psi^{(k+1)}$. Per il teorema precedente, $\psi(R \mid S) = \psi(\{r_1, \ldots, r_{n-1}\} \cup \{r_n\} \mid S) = \psi(\{r_n\} \mid S \cup \{r_1, \ldots, r_{n-1}\}) \cup \psi(\{r_1, \ldots, r_{n-1}\} \mid S)$. Quindi $\psi(R \mid S) = \psi(\{r_{k+1}\} \mid S^{(k)}) \cup \Psi^{(k)} = \Psi^{(k+1)} = \Psi^{(n)}$.

Si dimostra così che $\forall\, n, k \leq n \;\; \Psi^{(k)} = \psi(\{r_1, \ldots, r_k\} \mid S) = \psi(\{r_k\} \mid S \cup \{r_1, \ldots, r_{k-1}\}) \cup \cdots \cup \psi(\{r_2\} \mid S \cup \{r_1\}) \cup \psi(\{r_1\} \mid S)$. Per $k = n$ si verifica $R^{(n)} = \emptyset$ e l'algoritmo termina.

Supponendo di poter eseguire in tempo costante il controllo $R \neq \emptyset$, la scelta di un particolare elemento $r \in R$ e la sua cancellazione in $R \leftarrow R \setminus \{r\}$ si può dimostrare che l'algoritmo ha complessità lineare in $\operatorname{card} R$. Queste ipotesi si verificano se, ad esempio, R viene implementato come lista e la scelta di un nuovo elemento ricade ad ogni passo sulla testa della stessa. La complessità dell'algoritmo rispetto a $\operatorname{card} S$ dipende invece dalla realizzazione della funzione $\psi(\{r\} \mid S)$ e verrà studiata più avanti.

2.4 Algoritmi di chiusura

Un ulteriore vantaggio derivante dalla introduzione della funzione di inferenza complementare ψ si evidenzia definendo l'operatore *di incremento inferenziale* $\delta(S) : \mathcal{Z}_D \to \mathcal{Z}_D$ come:

$$\delta(S) = \phi(S) \setminus S$$

L'incremento inferenziale rappresenta la variazione della conoscenza disponibile a seguito dell'applicazione di un passo della funzione inferenza.

Dalla definizione deriva $S \cap \delta(S) = \emptyset$ e, dato che $S \subseteq \phi(S)$, anche $\phi(S) = S \cup \phi(S) \setminus S = S \cup \delta(S)$. La proprietà di invarianza di un generico insieme $S \in \mathcal{Z}_D$ si esprime in termini di incremento come $\delta(S) = \emptyset$ e si potrà quindi scrivere che $\delta(\emptyset) = \emptyset$ e $\delta(\widehat{S}) = \emptyset$.

Teorema 2.4.1 $\psi(\delta(S) \mid S) = \delta(\phi(S))$.

Dimostrazione Dalle definizioni di inferenza complementare e incremento inferenziale deriva $\psi(\delta(S) \mid S) = \phi(\delta(S) \cup S) \setminus \phi(S) = \phi(\phi(S) \setminus S \cup S) \setminus \phi(S)$. Dato che $S \subseteq \phi(S)$ l'espressione precedente si può riscrivere come $\phi(\phi(S)) \setminus \phi(S) = \delta(\phi(S))$.

Questo teorema consente di scrivere un algoritmo di chiusura che sfrutta la ψ per differenziare la conoscenza inferita durante l'ultimo passo da quella disponibile in precedenza, con il risultato riportato in figura 2.2.

1. $\Delta \leftarrow \delta(S)$
2. $\Phi \leftarrow S \cup \Delta$
3. Finché $\Delta \neq \emptyset$ ripeti...
   (a) $\Delta \leftarrow \psi(\Delta \mid \Phi \setminus \Delta)$
   (b) $\Phi \leftarrow \Phi \cup \Delta$

Figura 2.2. Calcolo di $\Phi = \widehat{\phi}(S)$

Teorema 2.4.2 $\forall\, S \in \mathcal{Z}_D$, *l'algoritmo calcola* $\Phi = \widehat{\phi}(S)$.

Dimostrazione Ad ogni passo k l'algoritmo calcola $\Delta^{(k)} = \delta(\phi^k(S))$ e $\Phi^{(k)} = \phi^{k+1}(S)$. Infatti:

1. L'inizializzazione produce $\Delta^{(0)} = \delta(S)$ e $\Phi^{(0)} = S \cup \delta(S) = \phi(S)$.
2. Si supponga che $\Delta^{(k-1)} = \delta(\phi^{k-1}(S))$ e $\Phi^{(k-1)} = \phi^k(S)$. Allora $\Phi^{(k-1)} \setminus \Delta^{(k-1)} = \phi^k(S) \setminus \delta(\phi^{k-1}(S)) = (\phi^{k-1}(S) \cup \delta(\phi^{k-1}(S))) \setminus \delta(\phi^{k-1}(S)) = \phi^{k-1}(S)$ perché $\phi^{k-1}(S) \cap \delta(\phi^{k-1}(S)) = \emptyset$. Per il precedente teorema, al passo k l'algoritmo calcola $\Delta^{(k)} = \psi(\Delta^{(k-1)} \mid \Phi^{(k-1)} \setminus \Delta^{(k-1)}) = \psi(\delta(\phi^{k-1}(S)) \mid \phi^{k-1}(S)) = \delta(\phi^k(S))$. La sua unione con $\Phi^{(k-1)}$ produce poi $\Phi^{(k)} = \phi^k(S) \cup \delta(\phi^k(S)) = \phi^{k+1}(S)$.

Quando $\Delta^{(n)} = \delta(\phi^n(S)) = \emptyset$ l'algoritmo si arresta perché $\Phi^{(n)} = \phi^{n+1}(S) = \phi^n(S)$ è insieme invariante. Questa condizione è appunto quella che definisce la chiusura $\widehat{S} = \widehat{\phi}(S) = \phi^n(S)$.

Una proprietà dell'algoritmo che avrà interessanti risvolti implementativi è che $\forall\, k \geq 0 \;\; \Delta^{(k)} \subseteq \Phi^{(k)}$ in quanto $\Delta^{(k)} = \delta(\phi^k(S)) \subseteq \phi^{k+1}(S) = \Phi^{(k)}$.

Sostituendo l'algoritmo 2.1 dentro il 2.2 ed operando alcune semplificazioni si può poi scrivere l'algoritmo 2.3. Le semplificazioni effettuate sono motivate di seguito:

- L'insieme Δ, parametro attuale della ψ, può essere utilizzato al posto di R, parametro formale, in quanto all'uscita del ciclo più interno ad esso viene assegnato un nuovo valore indipendente dal suo valore corrente. Il progressivo svuotamento di Δ non può quindi ripercuotersi all'esterno.
- Togliere un elemento da Δ produce l'effetto di aggiungere quello stesso elemento a $\Phi \setminus \Delta$. Chiamando $S^{(k)} = \Phi^{(k)} \setminus \Delta^{(k)}$ il secondo argomento della ψ ad un generico passo k del ciclo più interno, si osserva che $S^{(k+1)} = \Phi^{(k+1)} \setminus \Delta^{(k+1)} = \Phi^{(k)} \setminus (\Delta^{(k)} \setminus \{r^{(k)}\}) = \Phi^{(k)} \setminus \Delta^{(k)} \cup \{r^{(k)}\} = S^{(k)} \cup \{r^{(k)}\}$ perché $r^{(k)} \in \Delta^{(k)} \subseteq \Phi^{(k)}$. Il progressivo riempimento di S è quindi garantito dallo svuotamento di Δ e non necessita di istruzioni specifiche.

1. $\Delta \leftarrow \delta(S)$
2. $\Phi \leftarrow S \cup \Delta$
3. Finché $\Delta \neq \emptyset$ ripeti...
   (a) $\Psi \leftarrow \emptyset$
   (b) Finché $\Delta \neq \emptyset$ ripeti...
      i. $\exists\, r \in \Delta$
      ii. $\Psi \leftarrow \Psi \cup \psi(\{r\} \mid \Phi \setminus \Delta)$
      iii. $\Delta \leftarrow \Delta \setminus \{r\}$
   (c) $\Delta \leftarrow \Psi$
   (d) $\Phi \leftarrow \Phi \cup \Psi$

Figura 2.3. Calcolo di $\Phi = \widehat{\phi}(S)$

Si può scrivere un algoritmo simile al precedente ma con un solo ciclo, eliminando il ruolo di accumulatore della variabile Ψ come mostrato in figura 2.4. Quest'ultimo algoritmo calcola la chiusura in un modo che non permette di identificare i risultati dei singoli passi di applicazione della funzione inferenza, cioè per esso solitamente $\Phi^{(k)} \neq \phi^{k+1}(S)$. Viene quindi introdotto indipendentemente e se ne dimostra l'equivalenza in una seconda fase.

1. $\Delta \leftarrow \delta(S)$
2. $\Phi \leftarrow S \cup \Delta$
3. Finché $\Delta \neq \emptyset$ ripeti...
   (a) $\exists\, r \in \Delta$
   (b) $\Psi \leftarrow \psi(\{r\} \mid \Phi \setminus \Delta)$
   (c) $\Delta \leftarrow \Delta \setminus \{r\} \cup \Psi$
   (d) $\Phi \leftarrow \Phi \cup \Psi$

Figura 2.4. Calcolo di $\Phi = \widehat{\phi}(S)$

Per non appesantire l'esposizione si definiscono $S^{(k)} = \Phi^{(k)} \setminus \Delta^{(k)}$ e $X^{(k \geq n)} = X^{(n)}$ con $X \in \{S, \Delta, \Phi, \Psi\}$ e n numero dei passi in cui l'algoritmo termina. Quest'ultima convenzione consente affermazioni $\forall\, k \geq 0$ anche se il numero di passi è in effetti finito.

Una prima proprietà in comune con l'algoritmo precedente è che $\forall\, k \geq 0 \;\; \Delta^{(k)} \subseteq \Phi^{(k)}$. Questo è vero al momento dell'inizializzazione e resta tale nei passi successivi perché ogni elemento che viene aggiunto a Δ viene aggiunto anche a Φ.

Teorema 2.4.3 $\forall\, k \geq 0 \;\; \Phi^{(k)} = \phi(S^{(k)})$.

Dimostrazione Per induzione:

1. L'inizializzazione produce $\Delta^{(0)} = \delta(S)$, $\Phi^{(0)} = S \cup \delta(S) = \phi(S)$ e $S^{(0)} = \Phi^{(0)} \setminus \Delta^{(0)} = (S \cup \delta(S)) \setminus \delta(S) = S$ perché $S \cap \delta(S) = \emptyset$. Quindi $\Phi^{(0)} = \phi(S^{(0)})$.
2. Si supponga $\Phi^{(k)} = \phi(S^{(k)})$. Se $\Delta^{(k)} = \emptyset$ l'algoritmo si arresta e gli insiemi non vengono ulteriormente modificati. Ad ogni passo in cui $\Delta^{(k)} \neq \emptyset$, $\exists\, r^{(k)} \in \Delta^{(k)}$ e si può calcolare

$\Psi^{(k)} = \psi(\{r^{(k)}\} \mid S^{(k)}) = \phi(\{r^{(k)}\} \cup S^{(k)}) \setminus \phi(S^{(k)}) = \phi(\{r^{(k)}\} \cup S^{(k)}) \setminus \Phi^{(k)}$. Da questo deriva $\Psi^{(k)} \cap \Phi^{(k)} = \emptyset$ ed, essendo $r^{(k)} \in \Delta^{(k)} \subseteq \Phi^{(k)}$, si può scrivere che $S^{(k+1)} = \Phi^{(k+1)} \setminus \Delta^{(k+1)} = (\Phi^{(k)} \cup \Psi^{(k)}) \setminus (\Delta^{(k)} \setminus \{r^{(k)}\} \cup \Psi^{(k)}) = \Phi^{(k)} \setminus (\Delta^{(k)} \setminus \{r^{(k)}\}) = \Phi^{(k)} \setminus \Delta^{(k)} \cup \{r^{(k)}\} = S^{(k)} \cup \{r^{(k)}\}$. D'altronde, $\Phi^{(k+1)} = \Phi^{(k)} \cup \Psi^{(k)} = \Phi^{(k)} \cup \phi(\{r^{(k)}\} \cup S^{(k)}) \setminus \Phi^{(k)} = \phi(\{r^{(k)}\} \cup S^{(k)})$ perché $\Phi^{(k)} = \phi(S^{(k)}) \subseteq \phi(\{r^{(k)}\} \cup S^{(k)})$. Si può quindi concludere che $\Phi^{(k+1)} = \phi(S^{(k+1)})$.

Un modo equivalente per esprimere lo stesso concetto è $\forall\, k \geq 0 \quad \Delta^{(k)} = \delta(S^{(k)})$. Infatti $\delta(S^{(k)}) = \phi(S^{(k)}) \setminus S^{(k)} = \Phi^{(k)} \setminus (\Phi^{(k)} \setminus \Delta^{(k)}) = \Delta^{(k)}$ perché $\Delta^{(k)} \subseteq \Phi^{(k)}$.

La dimostrazione precedente mette in luce altre due proprietà rilevanti dell'algoritmo. La prima afferma che $\forall\, k \geq 0 \quad \Psi^{(k)} \cap \Phi^{(k)} = \emptyset$, cioè tutta la conoscenza inferita durante un passo è nuova rispetto a quella disponibile ed ogni elemento della rappresentazione viene prodotto al più una volta dalla funzione ψ. Anche questo fattore, evidentemente positivo, è comune con gli algoritmi di chiusura fin qui studiati.

La seconda proprietà è $\forall\, k \geq 0 \mid \Delta^{(k)} \neq \emptyset \quad S^{(k+1)} = S^{(k)} \cup \{r^{(k)}\}$ che, assieme a $r^{(k)} \in \Delta^{(k)} \Rightarrow r^{(k)} \notin \Phi^{(k)} \setminus \Delta^{(k)} = S^{(k)}$, produce $S^{(k+1)} \setminus S^{(k)} = \{r^{(k)}\}$, cioè S cresce di esattamente un elemento per passo finché non viene soddisfatta la condizione di uscita. Ma $\forall\, k \geq 0 \quad S^{(k)} \subseteq Z_D$ e Z_D è un insieme finito. La condizione $\Delta^{(k)} \neq \emptyset$ potrà dunque verificarsi un numero finito di volte $n \leq \operatorname{card} Z_D$ e questo assicura la terminazione dell'algoritmo.

I legami tra questi insiemi di conoscenza possono essere riassunti nell'espressione $\Psi^{(k)} = \phi(\{r^{(k)}\} \cup S^{(k)}) \setminus \phi(S^{(k)}) = \phi(S^{(k+1)}) \setminus \phi(S^{(k)}) = \Phi^{(k+1)} \setminus \Phi^{(k)}$.

Inoltre $S^{(k)} \subseteq S^{(k+1)} \Rightarrow \phi(S^{(k)}) \subseteq \phi(S^{(k+1)})$ cioè $\Phi^{(k)} \subseteq \Phi^{(k+1)}$, cosa peraltro evidente se si osserva che Φ non viene mai decrementato durante le iterazioni. Dato che per ogni passo k ad esso si uniscono gli elementi di $\Psi^{(k)}$ che è disgiunto da $\Phi^{(k)}$, quando $\Psi^{(k)} \neq \emptyset$ vale $\Phi^{(k)} \neq \Phi^{(k+1)} \Rightarrow \Phi^{(k)} \subset \Phi^{(k+1)}$. Per le stesse ragioni che implicano la convergenza, $\exists\, k_0 \mid \forall\, k \geq k_0 \quad \Psi^{(k)} = \emptyset$. Da quel punto in poi l'insieme Φ non varierà mentre $\Delta \subseteq \Phi$ sarà decrementato di un elemento per passo fino a svuotarlo completamente, soddisfacendo così la condizione di uscita.

L'insieme prodotto dall'algoritmo in uscita è invariante. Infatti $\Delta^{(n)} = \emptyset$ implica per il precedente teorema $\Phi^{(n)} = \phi(S^{(n)}) = \phi(\Phi^{(n)} \setminus \Delta^{(n)}) = \phi(\Phi^{(n)})$. Come dimostrato tra le proprietà della chiusura, ogni insieme invariante contiene la chiusura di un qualunque suo sottoinsieme e $S \subseteq \Phi^{(0)} \subseteq \cdots \subseteq \Phi^{(n)}$. Si può dunque affermare che $\widehat{S} \subseteq \Phi^{(n)}$.

D'altra parte, l'algoritmo viene inizializzato con un insieme di conoscenza disponibile $S \cup \delta(S) = \phi(S) \subseteq \widehat{S}$. Ogni volta che la funzione inferenza viene applicata ad un sottinsieme di $\widehat{S}$ produce un sottinsieme di $\widehat{S}$ perché questo insieme è invariante per ϕ, come precedentemente dimostrato. Le operazioni di unione e complementare che vengono compiute sui sottoinsiemi di $\widehat{S}$ durante le iterazioni conservano questa relazione di inclusione ed alla terminazione dovrà ancora valere che $\Phi^{(n)} \subseteq \widehat{S}$. Confrontando questo risultato con il suo simmetrico $\widehat{S} \subseteq \Phi^{(n)}$ si può concludere che $\Phi^{(n)} = \widehat{S}$.

Gli algoritmi 2.3 e 2.4 risultano quindi equivalenti. Entrambi non prescrivono alcun criterio particolare per la scelta di $r \in \Delta$, ma se Δ è implementata come una coda si osserva che la successione degli $r^{(k)}$ è la stessa nei due casi pur di aver ordinato in ugual modo gli elementi di $\Delta^{(0)}$.

Per formalizzare questo concetto si definisce una relazione di ordinamento $r \preceq s$ su Z_D:

1. $\forall\, s \in Z_D \quad s \preceq s$
2. $\forall\, r, s \in Z_D \quad r \preceq s \vee s \preceq r$. Inoltre si definisce $r \simeq s$ come $r \preceq s \wedge s \preceq r$.
3. $\forall\, q, r, s \in Z_D \quad q \preceq r \preceq s \Rightarrow q \preceq s$

Il simbolo "$\preceq$" si legge "precede o equivale" mentre "$\simeq$" si legge "equivale" ed in generale $r \simeq s \not\Rightarrow r = s$. Per completezza notazionale si definiscono infine la relazione di precedenza stretta $r \prec s$ come $r \preceq s \wedge \neg(s \preceq r)$ e gli operatori simmetrici $r \succeq s$ come $s \preceq r$ e $r \succ s$ come $s \prec r$.

Un esempio di ordinamento soddisfacente i requisiti è $r \preceq s \iff \forall\, k \mid s \in \Delta^{(k)} \quad \exists\, h \leq k \mid r \in \Delta^{(h)}$. La verifica può essere fatta osservando che $\forall\, z \in Z_D \mid \forall\, k \geq 0 \quad z \notin \Delta^{(k)}$ vale $\forall\, s \in Z_D \quad s \preceq z$.

Assumendo questo particolare ordinamento e richiedendo che l'elemento $r \in \Delta$ selezionato dall'algoritmo 2.4 soddisfi il vincolo $\forall\, s \in \Delta \;\; r \preceq s$ si può ancora scegliere r in modo da riprodurre la stessa successione di $r^{(k)}$ che gli algoritmi generano quando Δ è implementata come una coda. In altre parole, l'implementazione costituisce un vincolo compatibile con il precedente ma più stringente. Si può inoltre dimostrare che per ogni criterio impiegato nell'algoritmo 2.3 ne esiste uno per il 2.4 soddisfacente il vincolo $\forall\, s \in \Delta \;\; r \preceq s$ che genera la medesima successione di $r^{(k)}$.

Da questo punto di vista, il secondo algoritmo privo di vincoli risulta più generale del primo e verrà preferito proprio grazie a questa sua caratteristica. Infatti nei sistemi tempo reale non è importante solo che tutti i risultati richiesti vengano prodotti, ma anche che la loro produzione avvenga in un ordine tale da consentirne un utilizzo entro precisi limiti temporali. Un vincolo sulle scelte simile al precedente ma indotto da una diversa relazione di ordinamento sugli elementi di Δ risolverà questo problema.

Resta da affrontare il problema del calcolo della $\psi(\{r\} \mid S)$. Prima di farlo verranno presentati alcuni risultati che, oltre a renderne più agevole lo studio, hanno un interesse generale e torneranno assai utili nella messa a punto di algoritmi efficienti.

2.5 Funzioni locali e teorema di sostituzione

Durante un processo d'inferenza reale, un qualunque fatto sarà direttamente correlato ad un numero limitato di fatti diversi e le eventuali relazioni con tutti gli altri fatti rappresentati potranno emergere solo per via transitiva, applicando ripetutamente le regole d'inferenza. Dato $r \in Z_D$, se l'insieme degli $s \in Z_D$ direttamente correlati con r è ragionevolmente piccolo rispetto a Z_D risulta conveniente individuarlo attraverso una formalizzazione esplicita.

Assegnata una particolare funzione inferenza ϕ, si definiscono gli insiemi di funzioni:

$$\mathcal{M}_\phi = \{\mu : Z_D \to \mathcal{Z}_D \mid \forall\, r \in Z_D, S \in \mathcal{Z}_D \;\; \phi(\{r\} \mid S) \subseteq \phi(\{r\} \mid \mu(r) \cap S)\}$$

$$\mathcal{N}_\phi = \{\nu : Z_D \to \mathcal{Z}_D \mid \forall\, r \in Z_D, S \in \mathcal{Z}_D \;\; \phi(\{r\} \mid S) \subseteq \{r\} \cup \nu(r)\}$$

Indicando con $\mu_S(r)$ l'insieme $\mu(r) \cap S$ ed osservando che la seconda definizione deve valere per ogni scelta di S, si può scrivere che in generale $\forall\, \mu \in \mathcal{M}_\phi, \nu \in \mathcal{N}_\phi, r \in Z_D, S \in \mathcal{Z}_D \;\; \phi(\{r\} \mid S) \subseteq \phi(\{r\} \mid \mu_S(r)) \subseteq \{r\} \cup \nu(r)$.

Ricordando poi che $\phi(\{r\} \mid S) = \phi(\{r\}) \cup \xi(\{r\}, S) = \{r\} \cup \delta(\{r\}) \cup \xi(\{r\}, S)$ si ricavano le due definizioni del tutto equivalenti $\mathcal{M}_\phi = \{\mu : Z_D \to \mathcal{Z}_D \mid \forall\, r \in Z_D, S \in \mathcal{Z}_D \;\; \xi(\{r\}, S) \subseteq \xi(\{r\}, \mu_S(r))\}$ e $\mathcal{N}_\phi = \{\nu : Z_D \to \mathcal{Z}_D \mid \forall\, r \in Z_D, S \in \mathcal{Z}_D \;\; \delta(\{r\}) \cup \xi(\{r\}, S) \subseteq \nu(r)\}$. Entrambi questi insiemi sono non vuoti, poiché contengono certamente la funzione $\forall\, r \in Z_D \;\; \mu(r) = \nu(r) = Z_D$.

Teorema 2.5.1 *Assegnata ϕ, $\forall\, \mu, \mu' \in \mathcal{M}_\phi$ la funzione $\forall\, r \in Z_D \;\; \mu''(r) = \mu(r) \cap \mu'(r)$ appartiene a $\mathcal{M}_\phi$.*

Dimostrazione Chiamando $U(r) = \mu'(r) \setminus \mu''(r)$ e $V(r) = S \setminus U(r)$, si osserva che per ogni $r \in Z_D$ vale $\mu(r) \cap U(r) = \mu(r) \cap (\mu'(r) \setminus \mu''(r)) = (\mu(r) \cap \mu'(r)) \setminus \mu''(r) = \emptyset$ per la definizione di μ''. Di conseguenza $\mu(r) \cap V(r) = \mu(r) \cap (S \setminus U(r)) = (\mu(r) \setminus U(r)) \cap S = \mu(r) \cap S$, ovvero $\mu_S(r) = \mu_{V(r)}(r)$. L'appartenenza di μ a $\mathcal{M}_\phi$ consente allora di scrivere $\phi(\{r\} \mid S) \subseteq \phi(\{r\} \mid \mu_S(r)) = \phi(\{r\} \mid \mu_{V(r)}(r)) = \phi(\{r\} \cup \mu_{V(r)}(r)) \setminus (\phi(\mu_{V(r)}(r)) \setminus \phi(\{r\})) \subseteq \phi(\{r\} \cup \mu_{V(r)}(r)) \subseteq \phi(\{r\} \cup V(r))$ perché $\mu_{V(r)}(r) \subseteq V(r)$. Da $V(r) \subseteq S$ si ricava $\phi(V(r)) \setminus \phi(\{r\}) \subseteq \phi(S) \setminus \phi(\{r\})$ e dunque $\phi(\{r\} \mid S) = \phi(\{r\} \cup S) \setminus (\phi(S) \setminus \phi(\{r\})) = \phi(\{r\} \cup S) \setminus (\phi(S) \setminus \phi(\{r\})) \setminus (\phi(V(r)) \setminus \phi(\{r\})) = \phi(\{r\} \mid S) \setminus (\phi(V(r)) \setminus \phi(\{r\})) \subseteq \phi(\{r\} \cup V(r)) \setminus (\phi(V(r)) \setminus \phi(\{r\})) = \phi(\{r\} \mid V(r))$. D'altra parte, $\mu'(r) \cap V(r) = \mu'(r) \cap (S \setminus (\mu'(r) \setminus \mu''(r))) = (\mu'(r) \setminus (\mu'(r) \setminus \mu''(r))) \cap S = \mu''(r) \cap S$ perché $\mu''(r) \subseteq \mu'(r)$, ovvero $\mu''_S(r) = \mu'_{V(r)}(r)$. In questo caso l'appartenenza di μ' a $\mathcal{M}_\phi$ comporta $\phi(\{r\} \mid V(r)) \subseteq \phi(\{r\} \mid \mu'_{V(r)}(r)) = \phi(\{r\} \mid \mu''_S(r))$, che assieme al risultato precedente implica $\phi(\{r\} \mid S) \subseteq \phi(\{r\} \mid V(r)) \subseteq \phi(\{r\} \mid \mu''_S(r))$, cioè $\mu'' \in \mathcal{M}_\phi$.

Non è difficile convincersi del fatto che anche gli elementi dell'insieme $\mathcal{N}_\phi$ godono di una proprietà analoga, cioè $\forall\, \nu, \nu' \in \mathcal{N}_\phi \;\; \phi(\{r\} \mid S) \subseteq \{r\} \cup \nu(r) \cap \nu'(r) = \{r\} \cup \nu''(r)$.

A questo punto è possibile definire le funzioni *frontiera di ingresso* $\mu^* : Z_D \to \mathcal{Z}_D$ e *frontiera di uscita* $\nu^* : Z_D \to \mathcal{Z}_D$ associate ad una particolare funzione inferenza come:

$$\mu^*(r) = \bigcap_{\mu \in \mathcal{M}_\phi} \mu(r)$$

$$\nu^*(r) = \bigcap_{\nu \in \mathcal{N}_\phi} \nu(r)$$

Queste due funzioni esistono sempre perché $\mathcal{M}_\phi, \mathcal{N}_\phi \neq \emptyset$ e per quanto appena dimostrato vale $\mu^* \in \mathcal{M}_\phi$, $\nu^* \in \mathcal{N}_\phi$.

Inoltre $\forall\, \mu \in \mathcal{M}_\phi, \nu \in \mathcal{N}_\phi$ le funzioni $\forall\, r \in Z_D \;\; \mu'(r) = \mu(r) \setminus \{r\}$ e $\nu'(r) = \nu(r) \setminus \{r\}$ appartengono rispettivamente a $\mathcal{M}_\phi$ e $\mathcal{N}_\phi$ perché $\phi(\{r\} \mid S) \subseteq \phi(\{r\} \mid \mu_S(r)) = \phi(\{r\} \cup \mu_S(r)) \setminus (\phi(\mu_S(r)) \setminus \phi(\{r\})) \subseteq \phi(\{r\} \cup \mu_S(r) \setminus \{r\}) \setminus (\phi(\mu_S(r) \setminus \{r\}) \setminus \phi(\{r\})) = \phi(\{r\} \cup \mu'_S(r)) \setminus (\phi(\mu'_S(r)) \setminus \phi(\{r\})) = \phi(\{r\} \mid \mu'_S(r))$ e $\phi(\{r\} \mid S) \subseteq \{r\} \cup \nu(r) = \{r\} \cup \nu(r) \setminus \{r\} = \{r\} \cup \nu'(r)$. La definizione delle frontiere implica $\mu^*(r) \subseteq \mu'(r), \nu^*(r) \subseteq \nu'(r)$ e si può concludere che $\forall\, r \in Z_D \;\; r \notin \mu^*(r), r \notin \nu^*(r)$. La scelta della ϕ determina quindi univocamente una particolare coppia di funzioni frontiera comprese tra i casi limite $\forall\, r \in Z_D \;\; \mu^*(r) = \nu^*(r) = \emptyset$ e $\forall\, r \in Z_D \;\; \mu^*(r) = \nu^*(r) = Z_D \setminus \{r\}$, anche se i casi di interesse sono solo quelli in cui si verifica $\operatorname{card} \mu^*(r), \operatorname{card} \nu^*(r) \ll \operatorname{card} Z_D$.

Si definisce poi la funzione *intorno inferenziale* $\upsilon : Z_D \to \mathcal{Z}_D$ come:

$$\upsilon(r) = \{r\} \cup \mu^*(r) \cup \nu^*(r)$$

Dati $Z_X, Z_Y \subseteq Z_D \mid Z_X = \overline{Z_X}, Z_Y = \overline{Z_Y}, Z_X \cup Z_Y = Z_D$, se è vero che:

$$\forall\, r \in Z_D \;\; \upsilon(r) \subseteq Z_X \vee \upsilon(r) \subseteq Z_Y$$

allora $Z_\cap = Z_X \cap Z_Y$ si dice *interfaccia* tra Z_X e Z_Y. Se due insiemi ammettono una interfaccia è possibile localizzare in essa il percorso seguito dalla conoscenza durante la propagazione da un insieme all'altro. Ad esempio, se l'interfaccia è vuota gli elementi di un insieme sono indipendenti da quelli dell'altro, mentre se non è vuota si può dimostrare che la parte di chiusura appartenente ad uno degli insiemi non varia sostituendo la parte dell'altro esterna all'interfaccia con un nuovo insieme che presenta uguale chiusura all'interfaccia.

Per farlo occorre introdurre la funzione *di chiusura parziale* $\gamma_X : \mathcal{Z}_D \to \mathcal{Z}_D$ tale che:

$$\gamma_X(S) = S \cup \widehat{\phi}_X(S \cap Z_X)$$

dove $Z_X = \overline{Z_X} \subseteq Z_D$. Con $\phi_X(S) = \phi_D(S) \cap Z_X$ si è indicata la restrizione di ϕ_D a $\mathcal{Z}_X$, definita solo su quest'ultimo insieme. Da ϕ, γ eredita le seguenti proprietà notevoli:

1. $\gamma_X(\emptyset) = \emptyset$
2. $\forall\, S \in \mathcal{Z}_D \;\; S \subseteq \gamma_X(S)$
3. $\forall\, R, S \in \mathcal{Z}_D \;\; R \subseteq S \Rightarrow \gamma_X(R) \subseteq \gamma_X(S)$

Come suggerisce il nome di questa funzione, l'insieme $\Gamma = \gamma_X(S)$ è invariante rispetto a γ_X. Infatti $\gamma_X(\Gamma) = \gamma_X(\gamma_X(S)) = \gamma_X(S) \cup \widehat{\phi}_X(\gamma_X(S) \cap Z_X) = \gamma_X(S) \cup \widehat{\phi}_X(S \cap Z_X \cup \widehat{\phi}_X(S \cap Z_X)) = S \cup \widehat{\phi}_X(S \cap Z_X) \cup \widehat{\phi}_X(\widehat{\phi}_X(S \cap Z_X)) = S \cup \widehat{\phi}_X(S \cap Z_X) = \gamma_X(S) = \Gamma$.

Ricordando che $S \subseteq \widehat{\phi}_D(S)$ e $\widehat{\phi}_X(S \cap Z_X) \subseteq \widehat{\phi}_D(S \cap Z_X) \subseteq \widehat{\phi}_D(S)$, anche la loro unione $\gamma_X(S)$ è inclusa in $\widehat{\phi}_D(S)$. Dati $Z_X = \overline{Z_X} \subseteq Z_D$, $Z_Y = \overline{Z_Y} \subseteq Z_D$ si definisce $\gamma_{X,Y} : \mathcal{Z}_D \to \mathcal{Z}_\mathcal{D}$ come:

$$\gamma_{X,Y}(S) = \gamma_Y(\gamma_X(S))$$

Si verifica immediatamente come le proprietà fondamentali di γ_X siano mantenute da $\gamma_{X,Y}$, ed anche per questa funzione vale $\gamma_{X,Y}(S) \subseteq \widehat{\phi}_D(S)$, mentre non è più vero che $\Gamma = \gamma_{X,Y}(S)$ è un insieme invariante rispetto a $\gamma_{X,Y}$. Ha quindi interesse definire la chiusura rispetto a questo nuovo operatore come:

$$\widehat{\Gamma} = \widehat{\gamma}_{X,Y}(S) = \lim_{k\to\infty} \gamma^k_{X,Y}(S)$$

che esiste certamente poichè $\forall\, S \in \mathcal{Z}_D \;\; S \subseteq \gamma_{X,Y}(S) \subseteq Z_D$ con Z_D finito. La chiusura $\widehat{\Gamma}$ è evidentemente invariante rispetto a $\gamma_{X,Y}$.

Da $\widehat{\Gamma} = \gamma_{X,Y}(\widehat{\Gamma}) = \gamma_Y(\gamma_X(\widehat{\Gamma}))$ si ricava $\widehat{\Gamma} \subseteq \gamma_X(\widehat{\Gamma}) \subseteq \gamma_Y(\gamma_X(\widehat{\Gamma})) = \widehat{\Gamma}$, da cui $\widehat{\Gamma} = \gamma_X(\widehat{\Gamma}) = \gamma_Y(\widehat{\Gamma})$, cioè la chiusura $\widehat{\Gamma}$ è invariante sia rispetto a γ_X che a γ_Y. Inoltre $S \subseteq \gamma_{X,Y}(S) \subseteq \widehat{\phi}_D(S)$ implica $S \subseteq \gamma^k_{X,Y}(S) \subseteq \widehat{\phi}_D(S)$ e quindi $S \subseteq \widehat{\Gamma} \subseteq \widehat{\phi}_D(S)$.

Se due sottoinsiemi dell'insieme di rappresentazione ammettono una interfaccia, è possibile esprimere la chiusura $\widehat{\phi}$ attraverso l'operatore $\widehat{\gamma}$.

Teorema 2.5.2 *Siano $Z_X, Z_Y \subseteq Z_D \mid Z_X = \overline{Z_X}, Z_Y = \overline{Z_Y}, Z_X \cup Z_Y = Z_D$ una coppia di insiemi tra cui è definita una interfaccia. Allora $\forall\, S \in \mathcal{Z}_D \;\; \widehat{\gamma}_{X,Y}(S) = \widehat{\phi}_D(S)$.*

Dimostrazione Si supponga $\widehat{\Gamma} = \widehat{\gamma}_{X,Y}(S) \neq \widehat{\phi}_D(S)$. Ricordando il funzionamento dell'algoritmo di chiusura, si ha $\Phi^{(0)} = S \subseteq \widehat{\Gamma} \subseteq \widehat{\phi}_D(S) = \Phi^{(n)}$ ed avendo ipotizzato $\Phi^{(n)} \setminus \widehat{\Gamma} \neq \emptyset$, certamente $\exists\, k \in [0, n-1] \mid \Phi^{(k)} \subseteq \widehat{\Gamma}, \Phi^{(k+1)} \setminus \widehat{\Gamma} \neq \emptyset$. Perché un qualunque elemento $s \in \Phi^{(k+1)} \setminus \widehat{\Gamma}$ possa essere inferito al passo k devono esistere $r \in \Phi^{(k)}, Q \subseteq \Phi^{(k)}$ tali che $s \in \psi(\{r\} \mid Q) \subseteq \phi(\{r\} \mid Q)$. Ma $\phi(\{r\} \mid Q) \subseteq \phi(\{r\} \mid \mu^*_Q(r)) \subseteq \phi(\{r\} \cup \mu^*(r) \cap Q) \subseteq \phi(\{r\} \cup \upsilon(r) \cap Q) = \phi((\{r\} \cup Q) \cap \upsilon(r))$ e $\phi(\{r\} \mid Q) \subseteq \{r\} \cup \nu^*(r) \subseteq \upsilon(r)$. Quindi $\psi(\{r\} \mid Q) \subseteq \phi((\{r\} \cup Q) \cap \upsilon(r)) \cap \upsilon(r)$, con $\{r\} \cup Q \subseteq \Phi^{(k)} \subseteq \widehat{\Gamma}$ e dunque $\psi(\{r\} \mid Q) \subseteq \phi(\widehat{\Gamma} \cap \upsilon(r)) \cap \upsilon(r)$. Dato che tra Z_X e Z_Y è definita una interfaccia, vale $\forall\, r \in Z_D \;\; \upsilon(r) \subseteq Z_X \vee \upsilon(r) \subseteq Z_Y$ e di conseguenza $\phi(\widehat{\Gamma} \cap \upsilon(r)) \cap \upsilon(r) \subseteq \phi(\widehat{\Gamma} \cap Z_X) \cap Z_X = \phi_X(\widehat{\Gamma} \cap Z_X) \subseteq \widehat{\phi}_X(\widehat{\Gamma} \cap Z_X) = \gamma_X(\widehat{\Gamma} \cap Z_X) \subseteq \gamma_X(\widehat{\Gamma}) = \widehat{\Gamma}$ oppure $\phi(\widehat{\Gamma} \cap \upsilon(r)) \cap \upsilon(r) \subseteq \phi(\widehat{\Gamma} \cap Z_Y) \cap Z_Y = \phi_Y(\widehat{\Gamma} \cap Z_Y) \subseteq \widehat{\phi}_Y(\widehat{\Gamma} \cap Z_Y) = \gamma_Y(\widehat{\Gamma} \cap Z_Y) \subseteq \gamma_Y(\widehat{\Gamma}) = \widehat{\Gamma}$. Ricapitolando, $\forall\, s \in \Phi^{(k+1)} \setminus \widehat{\Gamma} \;\; s \in \widehat{\Gamma}$, ovvero $\Phi^{(k+1)} \setminus \widehat{\Gamma} = \emptyset$, il che contraddice l'ipotesi.

Teorema 2.5.3 *Siano $Z_X, Z_Y \subseteq Z_D \mid Z_X = \overline{Z_X}, Z_Y = \overline{Z_Y}$. Allora $\forall\, S_X \in \mathcal{Z}_X, S_Y \in \mathcal{Z}_Y, k \geq 0 \;\; \widehat{\phi}_X(\gamma^k_{X,Y}(S_X \cup S_Y) \cap Z_X) = \widehat{\phi}_X(S_X \cup \gamma^k_{X,Y}(S_X \cup S_Y) \cap Z_\cap)$, con $Z_\cap = Z_X \cap Z_Y$.*

Dimostrazione Chiamando $\Gamma_k = \gamma^k_{X,Y}(S_X \cup S_Y)$, si procede per induzione su k:

1. Per $k = 0$ si verifica $\Gamma_0 = S_X \cup S_Y$ e $\Gamma_0 \cap Z_X = (S_X \cup S_Y) \cap Z_X = S_X \cup S_Y \cap Z_\cap = S_X \cup (S_X \cup S_Y) \cap Z_\cap = S_X \cup \Gamma_0 \cap Z_\cap$ per definizione di S_X e S_Y, quindi $\widehat{\phi}_X(\Gamma_0 \cap Z_X) = \widehat{\phi}_X(S_X \cup \Gamma_0 \cap Z_\cap)$.

2. Si supponga che al passo k valga $\widehat{\phi}_X(\Gamma_k \cap Z_X) = \widehat{\phi}_X(S_X \cup \Gamma_k \cap Z_\cap)$. Al passo successivo, $\Gamma_{k+1} \cap Z_X = \gamma_{X,Y}(\Gamma_k) \cap Z_X = \gamma_X(\Gamma_k) \cap Z_X \cup \widehat{\phi}_Y(\gamma_X(\Gamma_k) \cap Z_Y) \cap Z_X = \Gamma_k \cap Z_X \cup \widehat{\phi}_X(\Gamma_k \cap Z_X) \cup \widehat{\phi}_Y(\gamma_X(\Gamma_k) \cap Z_Y) \cap Z_\cap = \widehat{\phi}_X(\Gamma_k \cap Z_X) \cup \widehat{\phi}_Y(\gamma_X(\Gamma_k) \cap Z_Y) \cap Z_\cap$, mentre $\Gamma_{k+1} \cap Z_\cap = \gamma_{X,Y}(\Gamma_k) \cap Z_\cap = \gamma_X(\Gamma_k) \cap Z_\cap \cup \widehat{\phi}_Y(\gamma_X(\Gamma_k) \cap Z_Y) \cap Z_\cap = \widehat{\phi}_Y(\gamma_X(\Gamma_k) \cap Z_Y) \cap Z_\cap$ perché $Z_\cap \subseteq Z_Y$, e dunque nel complesso $\Gamma_{k+1} \cap Z_X = \widehat{\phi}_X(\Gamma_k \cap Z_X) \cup \Gamma_{k+1} \cap Z_\cap$. Usando l'ipotesi di induzione questo risultato può essere riscritto come $\Gamma_{k+1} \cap Z_X = \widehat{\phi}_X(S_X \cup \Gamma_k \cap Z_\cap) \cup \Gamma_{k+1} \cap Z_\cap = \widehat{\phi}_X(S_X \cup \Gamma_k \cap Z_\cap) \cup S_X \cup \Gamma_{k+1} \cap Z_\cap$ poiché $S_X \subseteq \widehat{\phi}_X(S_X \cup \Gamma_k \cap Z_\cap)$ e ricordando le proprietà della chiusura si può concludere $\widehat{\phi}_X(\Gamma_{k+1} \cap Z_X) = \widehat{\phi}_X(\widehat{\phi}_X(S_X \cup \Gamma_k \cap Z_\cap) \cup S_X \cup \Gamma_{k+1} \cap Z_\cap) = \widehat{\phi}_X(\widehat{\phi}_X(S_X \cup \Gamma_k \cap Z_\cap) \cup \widehat{\phi}_X(S_X \cup \Gamma_{k+1} \cap Z_\cap)) = \widehat{\phi}_X(\widehat{\phi}_X(S_X \cup \Gamma_{k+1} \cap Z_\cap)) = \widehat{\phi}_X(S_X \cup \Gamma_{k+1} \cap Z_\cap)$ perché $\Gamma_k \subseteq \Gamma_{k+1}$.

Sia $Z_{D'}$ un sottoinsieme dello stesso insieme di supporto Z di cui è sottoinsieme Z_D e si supponga di aver esteso il dominio di ϕ a tutti i sottoinsiemi di $Z_D \cup Z_{D'}$. Vale il seguente:

Teorema 2.5.4 *Siano $Z_X \subseteq Z_D, Z_{X'} \subseteq Z_{D'}, Z_Y \subseteq Z_D \cap Z_{D'} \mid Z_X = \overline{Z_X}, Z_{X'} = \overline{Z_{X'}}, Z_Y = \overline{Z_Y}, Z_\cap = Z_X \cap Z_Y = Z_{X'} \cap Z_Y$. Se esistono $\Omega_X \subseteq Z_X \setminus Z_Y, \Omega_{X'} \subseteq Z_{X'} \setminus Z_Y$ tali che $\forall\, S_\cap \in \mathcal{Z}_\cap \;\; \widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_\cap \subseteq \widehat{\phi}_{X'}(\Omega_{X'} \cup S_\cap) \cap Z_\cap$ allora $\forall\, S_Y \in \mathcal{Z}_Y, k \geq 0 \;\; \gamma^k_{X,Y}(\Omega_X \cup S_Y) \cap Z_Y \subseteq \gamma^k_{X',Y}(\Omega_{X'} \cup S_Y) \cap Z_Y$.*

Dimostrazione Indicando con $\Gamma_k = \gamma^k_{X,Y}(\Omega_X \cup S_Y)$, $\Gamma'_k = \gamma^k_{X',Y}(\Omega_{X'} \cup S_Y)$, si procede per induzione su k:

1. Per $k = 0$ si verifica $\Gamma_0 \cap Z_Y = (\Omega_X \cup S_Y) \cap Z_Y = S_Y = (\Omega_{X'} \cup S_Y) \cap Z_Y = \Gamma'_0 \cap Z_Y$ e quindi $\Gamma_0 \cap Z_Y \subseteq \Gamma'_0 \cap Z_Y$.
2. Si supponga che al passo k valga $\Gamma_k \cap Z_Y \subseteq \Gamma'_k \cap Z_Y$. Al passo successivo, $\Gamma_{k+1} \cap Z_Y = \gamma_{X,Y}(\Gamma_k) \cap Z_Y = \gamma_X(\Gamma_k) \cap Z_Y \cup \widehat{\phi}_Y(\gamma_X(\Gamma_k) \cap Z_Y) = \widehat{\phi}_Y(\gamma_X(\Gamma_k) \cap Z_Y) = \widehat{\phi}_Y(\Gamma_k \cap Z_Y \cup \widehat{\phi}_X(\Gamma_k \cap Z_X) \cap Z_Y) = \widehat{\phi}_Y(\Gamma_k \cap Z_Y \cup \widehat{\phi}_X(\Gamma_k \cap Z_X) \cap Z_\cap) = \widehat{\phi}_Y(\Gamma_k \cap Z_Y \cup \widehat{\phi}_X(\Omega_X \cup \Gamma_k \cap Z_\cap) \cap Z_\cap)$ per il teorema precedente. Analogamente, $\Gamma'_{k+1} \cap Z_Y = \widehat{\phi}_Y(\Gamma'_k \cap Z_Y \cup \widehat{\phi}_{X'}(\Omega_{X'} \cup \Gamma'_k \cap Z_\cap) \cap Z_\cap)$ e confrontando queste due relazioni si ottiene $\Gamma_{k+1} \cap Z_Y = \widehat{\phi}_Y(\Gamma_k \cap Z_Y \cup \widehat{\phi}_X(\Omega_X \cup \Gamma_k \cap Z_\cap) \cap Z_\cap) \subseteq \widehat{\phi}_Y(\Gamma'_k \cap Z_Y \cup \widehat{\phi}_X(\Omega_X \cup \Gamma'_k \cap Z_\cap) \cap Z_\cap) \subseteq \widehat{\phi}_Y(\Gamma'_k \cap Z_Y \cup \widehat{\phi}_{X'}(\Omega_{X'} \cup \Gamma'_k \cap Z_\cap) \cap Z_\cap) = \Gamma'_{k+1} \cap Z_Y$ perché $\Gamma_k \cap Z_Y \subseteq \Gamma'_k \cap Z_Y$, $\Gamma_k \cap Z_\cap \subseteq \Gamma'_k \cap Z_\cap$ e $\forall\, S_\cap \in \mathcal{Z}_\cap \;\; \widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_\cap \subseteq \widehat{\phi}_{X'}(\Omega_{X'} \cup S_\cap) \cap Z_\cap$.

Finalmente è possibile dimostrare un teorema di notevole importanza che consente di modificare l'insieme di rappresentazione senza ridurre la conoscenza inferibile:

Teorema 2.5.5 (di sostituzione) *Siano* $Z_X \subseteq Z_D, Z_{X'} \subseteq Z_{D'}, Z_Y \subseteq Z_D \cap Z_{D'} \mid Z_X = \overline{Z_X}, Z_{X'} = \overline{Z_{X'}}, Z_Y = \overline{Z_Y}, Z_X \cup Z_Y = Z_D, Z_{X'} \cup Z_Y = Z_{D'}$ *e* $Z_\cap = Z_X \cap Z_Y = Z_{X'} \cap Z_Y$ *interfaccia comune ad entrambe le coppie. Se esistono* $\Omega_X \subseteq Z_X \setminus Z_Y, \Omega_{X'} \subseteq Z_{X'} \setminus Z_Y$ *tali che* $\forall\, S_\cap \in \mathcal{Z}_\cap \;\; \widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_\cap \subseteq \widehat{\phi}_{X'}(\Omega_{X'} \cup S_\cap) \cap Z_\cap$ *allora* $\forall\, S_Y \in \mathcal{Z}_Y \;\; \widehat{\phi}_D(\Omega_X \cup S_Y) \cap Z_Y \subseteq \widehat{\phi}_{D'}(\Omega_{X'} \cup S_Y) \cap Z_Y$.

Dimostrazione L'ipotesi di interfaccia implica $\widehat{\gamma}_{X,Y}(\Omega_X \cup S_Y) = \widehat{\phi}_D(\Omega_X \cup S_Y)$ e $\widehat{\gamma}_{X',Y}(\Omega_{X'} \cup S_Y) = \widehat{\phi}_{D'}(\Omega_{X'} \cup S_Y)$. Per il teorema precedente $\forall\, S_Y \in \mathcal{Z}_Y, k \geq 0 \;\; \gamma^k_{X,Y}(\Omega_X \cup S_Y) \cap Z_Y \subseteq \gamma^k_{X',Y}(\Omega_{X'} \cup S_Y) \cap Z_Y$ e visto che $\exists\, n, n' \mid \gamma^n_{X,Y}(\Omega_X \cup S_Y) = \widehat{\gamma}_{X,Y}(\Omega_X \cup S_Y), \gamma^{n'}_{X',Y}(\Omega_{X'} \cup S_Y) = \widehat{\gamma}_{X',Y}(\Omega_{X'} \cup S_Y)$ si può concludere che $\widehat{\phi}_D(\Omega_X \cup S_Y) \cap Z_Y = \gamma^n_{X,Y}(\Omega_X \cup S_Y) \cap Z_Y = \gamma^{n''}_{X,Y}(\Omega_X \cup S_Y) \cap Z_Y \subseteq \gamma^{n''}_{X',Y}(\Omega_{X'} \cup S_Y) \cap Z_Y = \gamma^{n'}_{X',Y}(\Omega_{X'} \cup S_Y) \cap Z_Y = \widehat{\phi}_{D'}(\Omega_{X'} \cup S_Y) \cap Z_Y$, con $n'' = \max\{n, n'\}$.

Scambiando $X \subseteq D$ con $X' \subseteq D'$ nelle ipotesi del teorema precedente e confrontando i risultati si ottiene che se è vero che $\forall\, S_\cap \in \mathcal{Z}_\cap \;\; \widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_\cap = \widehat{\phi}_{X'}(\Omega_{X'} \cup S_\cap) \cap Z_\cap$ allora $\forall\, S_Y \in \mathcal{Z}_Y \;\; \widehat{\phi}_D(\Omega_X \cup S_Y) \cap Z_Y = \widehat{\phi}_{D'}(\Omega_{X'} \cup S_Y) \cap Z_Y$. Questo teorema può essere applicato nel caso in cui gli ingressi e le uscite d'interesse siano contenuti in Z_Y, permettendo la sostituzione della coppia (Z_X, Ω_X) con una nuova coppia $(Z_{X'}, \Omega_{X'})$ che alla chiusura presenti un uguale insieme di conoscenza sull'interfaccia comune $Z_\cap$. Da un punto di vista pratico, il teorema consente di semplificare strutture di rappresentazione sostituendole con altre di più comodo utilizzo, purché queste mostrino un analogo comportamento attraverso l'interfaccia con tutte le altre strutture di rappresentazione.

Un caso particolare di applicazione del teorema si ha quando $Z_\cap = \emptyset$. Dati $Z_X \subseteq Z_D, Z_{X'} \subseteq Z_{D'}, Z_Y \subseteq Z_D \cap Z_{D'} \mid Z_X = \overline{Z_X}, Z_{X'} = \overline{Z_{X'}}, Z_Y = \overline{Z_Y}, Z_X \cup Z_Y = Z_D, Z_{X'} \cup Z_Y = Z_{D'}, Z_X \cap Z_Y = Z_{X'} \cap Z_Y = \emptyset$ soddisfacenti l'ipotesi di interfaccia, vale $\forall\, \Omega_X \in \mathcal{Z}_X, \Omega_{X'} \in \mathcal{Z}_{X'}, S_Y \in \mathcal{Z}_Y \;\; \widehat{\phi}_D(\Omega_X \cup S_Y) \cap Z_Y = \widehat{\phi}_{D'}(\Omega_{X'} \cup S_Y) \cap Z_Y$. Se i sottoinsiemi di rappresentazione ammettono interfaccia e sono disgiunti, non è possibile alcuna forma di interazione tra la conoscenza contenuta nell'uno e quella contenuta nell'altro.

Non sfuggirà un possibile problema formale in questo approccio: nessuna estensione è stata fatta a dominio ed immagine della funzione di rappresentazione ρ e l'interpretazione degli elementi di $Z_{D'} \setminus Z_D$ resta dunque indefinita. Questo significa che non è più possibile garantire la correttezza della funzione inferenza ϕ quando questa opera su elementi di quest'ultimo insieme, in quanto mancanti di una corrispondente lettura in termini di asserzioni logiche. Tuttavia, il teorema nella sua versione più stringente assicura che tutti gli elementi inferiti in Z_Y sono gli stessi che sarebbero stati inferiti da una funzione inferenza corretta ed il problema si risolve banalmente astenendosi

dal sostituire elementi che rappresentano fatti di interesse. Se invece si considera il teorema nella sua forma più generale, sarà necessario porsi di volta in volta il problema della correttezza degli elementi di $\widehat{\phi}_{D'}(\Omega_{X'} \cup S_Y) \cap Z_Y \setminus \widehat{\phi}_D(\Omega_X \cup S_Y) \cap Z_Y$, cioè di tutta la nuova conoscenza generata a seguito della sostituzione.

La scelta di non estendere la funzione di rappresentazione può costituire un vantaggio, in quanto garantisce una maggiore libertà nel manipolare l'insieme di supporto da un punto di vista operazionale senza curarsi delle formule logiche corrispondenti. In un esempio che verrà studiato successivamente vengono definite strutture di rappresentazione tali da consentire la scrittura di una efficiente funzione inferenza corretta, dopodiché quella stessa funzione viene applicata ad una maggior generalità di strutture ottenute manipolando le precedenti in un modo che ne impedisce l'interpretazione in termini logici ma che garantisce equivalenza da un punto di vista operazionale.

Per semplificare una prossima applicazione del teorema di sostituzione, è opportuno introdurre due teoremi ausiliari di dimostrazione abbastanza immediata:

Teorema 2.5.6 *Se* $Z_0 \subseteq Z_D$ *soddisfa* $\forall r \in Z_D \;\; \mu^*(r) \cap Z_0 = \emptyset$ *e* $\forall r \in Z_0 \;\; \nu^*(r) = \emptyset$*, allora* $\forall S \in \mathcal{Z}_D \;\; \widehat{\phi}(S) = S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0)$

Dimostrazione Si supponga $\widehat{\phi}(S) \neq S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0)$. Dato che $S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0) \subseteq \widehat{\phi}(S)$, la precedente ipotesi può essere riscritta come $\widehat{\phi}(S) \setminus (S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0)) \neq \emptyset$. Ricordando il funzionamento dell'algoritmo di chiusura, si ha $\Phi^{(0)} = S = S \cap Z_0 \cup S \setminus Z_0 \subseteq S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0) \subseteq \widehat{\phi}(S) = \Phi^{(n)}$ e per quanto ipotizzato $\exists k \in [0, n-1] \mid \Phi^{(k)} \subseteq S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0), \Phi^{(k+1)} \setminus (S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0)) \neq \emptyset$. Perché un qualunque elemento $s \in \Phi^{(k+1)} \setminus (S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0))$ possa essere inferito al passo k devono esistere $r \in \Phi^{(k)}, Q \subseteq \Phi^{(k)}$ tali che $s \in \psi(\{r\} \mid Q) \subseteq \phi(\{r\} \mid Q)$. Ma $\phi(\{r\} \mid Q) \subseteq \phi(\{r\} \mid \mu^*_Q(r)) \subseteq \phi(\{r\} \cup \mu^*(r) \cap Q) = \phi(\{r\} \cup \mu^*(r) \setminus Z_0 \cap Q) = \phi(\{r\} \cup \mu^*(r) \cap Q \setminus Z_0) \subseteq \phi(\{r\} \cup Q \setminus Z_0)$ e $\phi(\{r\} \mid Q) \subseteq \{r\} \cup \nu^*(r)$. Se $r \notin Z_0$, la prima relazione implica $\psi(\{r\} \mid Q) \subseteq \phi(\{r\} \cup Q \setminus Z_0) = \phi((\{r\} \cup Q) \setminus Z_0)$ con $\{r\} \cup Q \subseteq \Phi^{(k)} \subseteq S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0)$ e dunque $(\{r\} \cup Q) \setminus Z_0 \subseteq \widehat{\phi}(S \setminus Z_0) \setminus Z_0 \subseteq \widehat{\phi}(S \setminus Z_0)$, da cui $\phi((\{r\} \cup Q) \setminus Z_0) \subseteq \widehat{\phi}(S \setminus Z_0)$ ed infine $\psi(\{r\} \mid Q) \subseteq \widehat{\phi}(S \setminus Z_0) \subseteq S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0)$. Quando invece $r \in Z_0$, dalla seconda relazione si deduce $\psi(\{r\} \mid Q) \subseteq \{r\} \cup \nu^*(r) = \{r\} \subseteq \Phi^{(k)} \subseteq S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0)$, per cui nel complesso $\forall r \in \Phi^{(k)}, Q \subseteq \Phi^{(k)} \;\; \psi(\{r\} \mid Q) \subseteq S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0)$. Ricapitolando, $\forall s \in \Phi^{(k+1)} \setminus (S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0)) \;\; s \in S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0)$, ovvero $\Phi^{(k+1)} \setminus (S \cap Z_0 \cup \widehat{\phi}(S \setminus Z_0)) = \emptyset$, il che contraddice l'ipotesi.

Teorema 2.5.7 *Se* $Z_0 \subseteq Z_D$ *soddisfa* $\forall r \in Z_D \;\; \nu^*(r) \cap Z_0 = \emptyset$*, allora* $\forall S \in \mathcal{Z}_D \;\; \widehat{\phi}(S) = S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0$

Dimostrazione Si supponga $\widehat{\phi}(S) \neq S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0$. Dato che $S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0 \subseteq \widehat{\phi}(S)$, la precedente ipotesi può essere riscritta come $\widehat{\phi}(S) \setminus (S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0) \neq \emptyset$. Ricordando il funzionamento dell'algoritmo di chiusura, si ha $\Phi^{(0)} = S = S \cap Z_0 \cup S \setminus Z_0 \subseteq S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0 \subseteq \widehat{\phi}(S) = \Phi^{(n)}$ e per quanto ipotizzato $\exists k \in [0, n-1] \mid \Phi^{(k)} \subseteq S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0, \Phi^{(k+1)} \setminus (S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0) \neq \emptyset$. Perché un qualunque elemento $s \in \Phi^{(k+1)} \setminus (S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0)$ possa essere inferito al passo k devono esistere $r \in \Phi^{(k)}, Q \subseteq \Phi^{(k)}$ tali che $s \in \psi(\{r\} \mid Q) \subseteq \phi(\{r\} \mid Q)$. Ma $\{r\} \cup Q \subseteq \Phi^{(k)} \subseteq \widehat{\phi}(S)$ e quindi $\phi(\{r\} \mid Q) \subseteq \phi(\{r\} \cup Q) \subseteq \widehat{\phi}(S)$, che assieme alla relazione $\phi(\{r\} \mid Q) \subseteq \{r\} \cup \nu^*(r) = \{r\} \cup \nu^*(r) \setminus Z_0$ porta a concludere $\psi(\{r\} \mid Q) \subseteq \widehat{\phi}(S) \cap (\{r\} \cup \nu^*(r) \setminus Z_0) \subseteq \{r\} \cup \widehat{\phi}(S) \cap \nu^*(r) \setminus Z_0 = \{r\} \cup \widehat{\phi}(S) \setminus Z_0 \cap \nu^*(r) \subseteq \{r\} \cup \widehat{\phi}(S) \setminus Z_0 \subseteq S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0$ poiché $r \in \Phi^{(k)} \subseteq S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0$. Ricapitolando, $\forall s \in \Phi^{(k+1)} \setminus (S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0) \;\; s \in S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0$, ovvero $\Phi^{(k+1)} \setminus (S \cap Z_0 \cup \widehat{\phi}(S) \setminus Z_0) = \emptyset$, il che contraddice l'ipotesi.

A partire dal concetto di frontiera si può poi definire la funzione *di inferenza locale* $\lambda_S : Z_D \to \mathcal{Z}_D$ come:

$$\lambda_S(r) = \delta(\{r\}) \cup \xi(\{r\}, \mu^*_S(r))$$

Dalle precedenti osservazioni si ricava che $\forall r \in Z_D, S \in \mathcal{Z}_D \quad \lambda_S(r) \subseteq \nu^*(r)$. Inoltre vale $\phi(\{r\} \mid \mu_S^*(r)) = \phi(\{r\}) \cup \xi(\{r\}, \mu_S^*(r)) = \{r\} \cup \delta(\{r\}) \cup \xi(\{r\}, \mu_S^*(r)) = \{r\} \cup \lambda_S(r)$. Questa funzione non gode di altre proprietà rilevanti ma viene impiegata nell'algoritmo definitivo per la sua comodità di calcolo quando $\operatorname{card} \mu^*(r), \operatorname{card} \nu^*(r) \ll \operatorname{card} Z_D$.

Simbolo	Definizione
$\phi^k(S)$	$\phi_k(\phi_{k-1}(\cdots\phi_2(\phi_1(S))\cdots))$
$\widehat{\phi}(S)$	$\lim_{k\to\infty} \phi^k(S)$
$\phi_X(S)$	$\phi(S) \cap Z_X$
$\delta(S)$	$\phi(S) \setminus S$
$\xi(R, S)$	$\phi(R \cup S) \setminus (\phi(R) \cup \phi(S))$
$\phi(R \mid S)$	$\phi(R \cup S) \setminus (\phi(S) \setminus \phi(R))$
$\psi(R \mid S)$	$\phi(R \cup S) \setminus \phi(S)$
$\lambda_S(r)$	$\delta(\{r\}) \cup \xi(\{r\}, \mu^*(r) \cap S)$
$\gamma_X(S)$	$S \cup \widehat{\phi}_X(S \cap Z_X)$
$\gamma_{X,Y}(S)$	$\gamma_Y(\gamma_X(S))$
$\widehat{\gamma}_{X,Y}(S)$	$\lim_{k\to\infty} \gamma_{X,Y}^k(S)$
$\mu^*(r)$	$\bigcap_{\mu\in\mathcal{M}_\phi} \mu(r)$
$\nu^*(r)$	$\bigcap_{\nu\in\mathcal{N}_\phi} \nu(r)$
$\upsilon(r)$	$\{r\} \cup \mu^*(r) \cup \nu^*(r)$

Tabella 2.1. Tabella riassuntiva delle funzioni su $\mathcal{Z}_D$

Come si può intuire, le funzioni frontiera hanno lo scopo di limitare la porzione dell'insieme S coinvolta nel processo di inferenza condizionata su r a fini computazionali. La funzione inferenza ϕ e quella locale λ si dicono *localmente complete* se:

$$\forall r \in Z_D, S \subseteq \mu^*(r), s \in \nu^*(r) \quad \rho^{-1}(\{r\} \cup S) \models \rho^{-1}(s) \Rightarrow s \in \lambda_S(r)$$

Se la completezza è un obiettivo troppo ambizioso per un algoritmo tempo reale, si garantirà quanto meno che la funzione inferenza produca tutto ciò che una funzione corretta può produrre, una volta assegnati gli insiemi frontiera.

2.6 Procedura di chiusura progressiva

La funzione di inferenza locale può essere utilizzata al posto di quella complementare negli algoritmi di chiusura fin qui esaminati. Infatti vale $\psi(\{r\} \mid S) \subseteq \phi(\{r\} \mid S) \subseteq \phi(\{r\} \mid \mu_S^*(r)) = \{r\} \cup \lambda_S(r) \subseteq \phi(\{r\} \cup S)$ e facendo il complementare di $\phi(S)$ in questi insiemi si ottiene $\psi(\{r\} \mid S) \subseteq (\{r\} \cup \lambda_S(r)) \setminus \phi(S) \subseteq \phi(\{r\} \cup S) \setminus \phi(S) = \psi(\{r\} \mid S)$, cioè $\psi(\{r\} \mid S) = (\{r\} \cup \lambda_S(r)) \setminus \phi(S)$. Nel caso in cui $S = \Phi \setminus \Delta$ e $r \in \Delta \subseteq \Phi = \phi(S)$, si ricava $\psi(\{r\} \mid S) = (\{r\} \cup \lambda_S(r)) \setminus \phi(S) = \lambda_{\Phi\setminus\Delta}(r) \setminus \Phi$. Queste condizioni si verificano nell'algoritmo di chiusura 2.4 e sostituendo l'espressione $\psi(\{r\} \mid \Phi \setminus \Delta)$ con $\lambda_{\Phi\setminus\Delta}(r) \setminus \Phi$ nel calcolo di Ψ si ottiene l'algoritmo definitivo di chiusura, 2.5.

Nell'algoritmo 2.5 si inizializzano diversamente gli insiemi Δ e Φ e questo richiede una giustificazione. Nell'ipotesi di inizializzarli come nell'algoritmo 2.4 si è dimostrato che la sostituzione effettuata nel calcolo di Ψ è del tutto trasparente ma la nuova formulazione consente all'algoritmo di convergere alla chiusura anche a partire da condizioni iniziali diverse senza perdere in prestazioni. Se quelle stesse condizioni iniziali venissero applicate all'algoritmo 2.4 lo priverebbero di alcune importanti proprietà come $\Psi \cap \Phi = \emptyset$ e per questo non sono state introdotte precedentemente. La nuova inizializzazione è preferibile perché non richiede il calcolo di $\delta(S)$, ma quando questo è noto a priori la vecchia risulta ancora conveniente e verrà impiegata se $\operatorname{card} \delta(S) \ll \operatorname{card} S$.

1. $\Delta \leftarrow \Omega$
2. $\Phi \leftarrow \Omega$
3. Finché $\Delta \neq \emptyset$ ripeti...
   (a) $\exists\, r \in \Delta$
   (b) $\Psi \leftarrow \lambda_{\Phi \setminus \Delta}(r) \setminus \Phi$
   (c) $\Delta \leftarrow \Delta \setminus \{r\} \cup \Psi$
   (d) $\Phi \leftarrow \Phi \cup \Psi$

Figura 2.5. Calcolo di $\Phi = \widehat{\phi}(\Omega)$

Si può innanzitutto osservare che l'algoritmo 2.5 gode della proprietà $\forall\, k \geq 0 \;\; \Psi^{(k)} \cap \Phi^{(k)} = \emptyset$ per il modo in cui è calcolata Ψ, indipendentemente dalle condizioni iniziali. Inoltre continua a valere $\forall\, k \geq 0 \;\; \Delta^{(k)} \subseteq \Phi^{(k)}$ perché questo è vero all'inizializzazione e l'algoritmo conserva questa proprietà, come già visto.

Teorema 2.6.1 $\forall\, k \geq 0 \;\; \phi(S^{(k)}) \subseteq \Phi^{(k)}$.

Dimostrazione Per induzione:

1. L'inizializzazione produce $\Delta^{(0)} = \Phi^{(0)} = \Omega$ e $S^{(0)} = \Phi^{(0)} \setminus \Delta^{(0)} = \emptyset$. Dunque $\phi(S^{(0)}) = \phi(\emptyset) = \emptyset \subseteq \Phi^{(0)}$.
2. Si supponga $\phi(S^{(k)}) \subseteq \Phi^{(k)}$. Se $\Delta^{(k)} = \emptyset$ l'algoritmo si arresta e gli insiemi non vengono ulteriormente modificati. Ad ogni passo in cui $\Delta^{(k)} \neq \emptyset$, $\exists\, r^{(k)} \in \Delta^{(k)}$ e si può calcolare $\Psi^{(k)} = \lambda_{S^{(k)}}(r^{(k)}) \setminus \Phi^{(k)} = (\{r^{(k)}\} \cup \lambda_{S^{(k)}}(r^{(k)})) \setminus \Phi^{(k)} = (\{r^{(k)}\} \cup \lambda_{S^{(k)}}(r^{(k)})) \setminus \phi(S^{(k)}) \setminus \Phi^{(k)} = \psi(\{r^{(k)}\} \mid S^{(k)}) \setminus \Phi^{(k)} = \phi(\{r^{(k)}\} \cup S^{(k)}) \setminus \phi(S^{(k)}) \setminus \Phi^{(k)} = \phi(\{r^{(k)}\} \cup S^{(k)}) \setminus \Phi^{(k)}$ perché $r^{(k)} \in \Delta^{(k)} \subseteq \Phi^{(k)}$ e $\phi(S^{(k)}) \subseteq \Phi^{(k)}$. Ma $\Psi^{(k)} \cap \Phi^{(k)} = \emptyset$ per costruzione e si può scrivere che $S^{(k+1)} = \Phi^{(k+1)} \setminus \Delta^{(k+1)} = (\Phi^{(k)} \cup \Psi^{(k)}) \setminus (\Delta^{(k)} \setminus \{r^{(k)}\} \cup \Psi^{(k)}) = \Phi^{(k)} \setminus (\Delta^{(k)} \setminus \{r^{(k)}\}) = \Phi^{(k)} \setminus \Delta^{(k)} \cup \{r^{(k)}\} = S^{(k)} \cup \{r^{(k)}\}$. Quindi $\Phi^{(k+1)} = \Phi^{(k)} \cup \Psi^{(k)} = \Phi^{(k)} \cup \phi(\{r^{(k)}\} \cup S^{(k)}) \setminus \Phi^{(k)} = \Phi^{(k)} \cup \phi(\{r^{(k)}\} \cup S^{(k)}) = \Phi^{(k)} \cup \phi(S^{(k+1)})$ e ne consegue che $\phi(S^{(k+1)}) \subseteq \Phi^{(k+1)}$.

Dato che $\forall\, k \geq 0 \;\; \phi(S^{(k)}) \subseteq \Phi^{(k)}$, certamente anche $\delta(S^{(k)}) \subseteq \Phi^{(k)}$. Ma $\delta(S^{(k)}) \cap S^{(k)} = \emptyset$ con $S^{(k)} = \Phi^{(k)} \setminus \Delta^{(k)}$. Dal confronto di queste due relazioni si ricava $\delta(S^{(k)}) \subseteq \Delta^{(k)}$.

Anche per questo algoritmo vale $\forall\, k \geq 0 \mid \Delta^{(k)} \neq \emptyset \;\; S^{(k+1)} = S^{(k)} \cup \{r^{(k)}\}$, da cui si ricava $S^{(k+1)} \setminus S^{(k)} = \{r^{(k)}\}$ e la terminazione viene dimostrata in maniera analoga a quanto fatto per l'algoritmo precedente. Dalla precedente dimostrazione si ricava anche l'espressione esatta di $\Phi^{(k+1)} = \Phi^{(k)} \cup \phi(\{r^{(k)}\} \cup \Phi^{(k)} \setminus \Delta^{(k)})$.

Inoltre $\delta(S^{(n)}) \subseteq \Delta^{(n)} = \emptyset$. Quindi $S^{(n)} = \Phi^{(n)} \setminus \Delta^{(n)} = \Phi^{(n)}$ è un insieme invariante e $\Omega = \Phi^{(0)} \subseteq \cdots \subseteq \Phi^{(n)}$ dato che Φ non viene mai decrementato durante le iterazioni. Ogni insieme invariante contiene la chiusura di un qualunque suo sottoinsieme e quindi $\widehat{\Omega} \subseteq \Phi^{(n)}$. D'altronde, l'algoritmo viene inizializzato con un insieme di conoscenza disponibile $\Omega \subseteq \widehat{\Omega}$ e ragionando come per l'algoritmo precedente si conclude che $\Phi^{(n)} \subseteq \widehat{\Omega}$, da cui $\Phi^{(n)} = \widehat{\Omega}$.

Per quanto riguarda la complessità di un singolo passo dell'algoritmo, la definizione di frontiera di uscita implica $\lambda_S(r) \subseteq \nu^*(r)$ e chiamando $M_\mu = \max_{r \in Z_D} \operatorname{card} \mu^*(r)$, $M_\nu = \max_{r \in Z_D} \operatorname{card} \nu^*(r)$ si osserva che $\operatorname{card} \Psi \leq M_\nu$ indipendentemente dall'argomento di λ. La complessità computazionale sarà quindi legata a M_ν ma anche alla complessità di λ e quindi a M_μ. Stime più precise verranno fornite nel seguito a partire da questa semplice considerazione.

La dimostrazione di $\phi(S^{(k)}) \subseteq \Phi^{(k)}$ si regge su una particolare scelta delle condizioni iniziali, ma è facile verificare che qualunque inizializzazione è ammissibile purchè soddisfacente le seguenti proprietà:

1. $\phi(S^{(0)}) \subseteq \Phi^{(0)}$ ovvero $\delta(S^{(0)}) \subseteq \Delta^{(0)}$
2. $\Delta^{(0)} \subseteq \Phi^{(0)}$, da cui $\forall\, k \geq 0 \;\; \Delta^{(k)} \subseteq \Phi^{(k)}$

Da $\phi(S^{(k)}) \subseteq \Phi^{(k)}$ segue la convergenza alla chiusura e sinteticamente si può scrivere che $\delta(\Phi^{(0)} \setminus \Delta^{(0)}) \subseteq \Delta^{(0)} \subseteq \Phi^{(0)} \Rightarrow \Phi^{(n)} = \widehat{\phi}(\Phi^{(0)})$. L'implicazione è vera nei due casi limite $\delta(S) = \Delta^{(0)} \subseteq \Phi^{(0)} = \phi(S)$ e $\delta(\emptyset) \subseteq \Delta^{(0)} = \Phi^{(0)} = \Omega$ ma ha interesse perché si applica anche a tutti i casi intermedi.

Per approfondire questo concetto si definiscono:

$$\mathcal{X}_\Omega = \{(R, S) \in \mathcal{Z}_D \times \mathcal{Z}_D \mid \delta(S \setminus R) \subseteq R \subseteq S, \Omega \subseteq S \subseteq \widehat{\Omega}\}$$

e $\zeta : \mathcal{Z}_D \times \mathcal{Z}_D \to \mathcal{Z}_D \times \mathcal{Z}_D$ la trasformazione operata sulla coppia $\Xi = (\Delta, \Phi)$ da un singolo passo dell'algoritmo.

1. $(\Delta, \Phi) \leftarrow \Xi$
2. Se $\Delta \neq \emptyset$ allora...
   (a) $\exists\, r \in \Delta$
   (b) $\Psi \leftarrow \lambda_{\Phi \setminus \Delta}(r) \setminus \Phi$
   (c) $\Delta \leftarrow \Delta \setminus \{r\} \cup \Psi$
   (d) $\Phi \leftarrow \Phi \cup \Psi$
3. $\Upsilon \leftarrow (\Delta, \Phi)$

Figura 2.6. Calcolo di $\Upsilon = \zeta(\Xi)$

Definendo poi $\zeta^k(\Xi) = \zeta_k(\zeta_{k-1}(\cdots \zeta_2(\zeta_1(\Xi)) \cdots))$ e:

$$\widehat{\zeta}(\Xi) = \lim_{k \to \infty} \zeta^k(\Xi)$$

si possono dimostrare le seguenti proprietà:

1. $\forall\, \Omega \in \mathcal{Z}_D \;\; (\Omega, \Omega) \in \mathcal{X}_\Omega$
2. $\forall\, \Omega \in \mathcal{Z}_D \;\; \Xi \in \mathcal{X}_\Omega \Rightarrow \zeta(\Xi) \in \mathcal{X}_\Omega$
3. $\forall\, \Omega \in \mathcal{Z}_D \;\; \Xi \in \mathcal{X}_\Omega \Rightarrow \exists \lim_{k \to \infty} \zeta^k(\Xi) = \widehat{\zeta}(\Xi) = (\emptyset, \widehat{\Omega}) \in \mathcal{X}_\Omega$

Da $\delta(\Omega \setminus \Omega) = \emptyset \subseteq \Omega$ e $\Omega \subseteq \widehat{\Omega}$ deriva che $(\Omega, \Omega) \in \mathcal{X}_\Omega$. Inoltre tutte le coppie $(R, S) \in \mathcal{X}_\Omega$ soddisfano la condizione $\delta(S \setminus R) \subseteq R \subseteq S$ che l'algoritmo conserva durante le iterazioni, come già dimostrato, e che quindi saranno conservate anche dall'applicazione della funzione ζ associata al singolo passo.

Sotto questa condizione si possono applicare tutti i precedenti risultati e chiamando $(P, Q) = \zeta(R, S)$ si verifica che $Q = S \cup \phi(\{r\} \cup S \setminus R)$, da cui $\Omega \subseteq S \Rightarrow \Omega \subseteq Q$ e $r \in R \subseteq S \subseteq \widehat{\Omega} \Rightarrow \{r\} \cup S \setminus R \subseteq \widehat{\Omega} \Rightarrow \phi(\{r\} \cup S \setminus R) \subseteq \widehat{\Omega} \Rightarrow Q \subseteq \widehat{\Omega}$. Quindi $\Omega \subseteq Q \subseteq \widehat{\Omega}$ e $\zeta(R, S) \in \mathcal{X}_\Omega$.

La convergenza dell'algoritmo alla chiusura di S garantisce che $\exists\, \widehat{\zeta}(R, S) = \zeta^n(R, S) = (\emptyset, \widehat{S})$. Inoltre $\Omega \subseteq S \subseteq \widehat{\Omega}$ implica $\widehat{\phi}(\Omega) \subseteq \widehat{\phi}(S) \subseteq \widehat{\phi}(\widehat{\Omega})$, cioè $\widehat{\Omega} \subseteq \widehat{S} \subseteq \widehat{\Omega}$ e si può concludere che $\widehat{S} = \widehat{\Omega}$.

Dati $(P, Q) \in \mathcal{X}_\mathrm{O}$, $(R, S) \in \mathcal{X}_\Omega$ in generale $(P \cup R, Q \cup S) \notin \mathcal{X}_{\mathrm{O} \cup \Omega}$ per il decadere della condizione $\delta(S \setminus R) \subseteq R$. Al suo posto vale una proprietà più debole:

Teorema 2.6.2 $\forall\, \mathrm{O}, \Omega \in \mathcal{Z}_D \;\; (Q, Q) \in \mathcal{X}_\mathrm{O}, (R, S) \in \mathcal{X}_\Omega \Rightarrow (Q \cup R, Q \cup S) \in \mathcal{X}_{\mathrm{O} \cup \Omega}$.

Dimostrazione Da $R \subseteq S$ deriva $Q \cup R \subseteq Q \cup S$. Inoltre $\delta((Q \cup S) \setminus (Q \cup R)) = \delta(S \setminus (Q \cup R)) = \phi(S \setminus (Q \cup R)) \setminus (S \setminus (Q \cup R))$. Ma $S \setminus (Q \cup R) \subseteq S \setminus R \Rightarrow \phi(S \setminus (Q \cup R)) \subseteq \phi(S \setminus R) = S \setminus R \cup \delta(S \setminus R) \subseteq S \setminus R \cup R = S$. Quindi $\delta((Q \cup S) \setminus (Q \cup R)) \subseteq S \setminus (S \setminus (Q \cup R)) = (Q \cup R) \cap S \subseteq Q \cup R$. Infine da $\mathrm{O} \subseteq Q \subseteq \widehat{\mathrm{O}}$ e $\Omega \subseteq S \subseteq \widehat{\Omega}$ si ricava $\mathrm{O} \cup \Omega \subseteq Q \cup S \subseteq \widehat{\mathrm{O}} \cup \widehat{\Omega} \subseteq \widehat{\phi}(\mathrm{O} \cup \Omega)$ e questo, assieme a $\delta((Q \cup S) \setminus (Q \cup R)) \subseteq Q \cup R \subseteq Q \cup S$, consente di concludere che $(Q \cup R, Q \cup S) \in \mathcal{X}_{\mathrm{O} \cup \Omega}$.

Data una coppia $(R, S) \in \mathcal{X}_\Omega$ si consideri la coppia $(\mathrm{O} \setminus S, \mathrm{O} \setminus S) \in \mathcal{X}_{\mathrm{O} \setminus \mathcal{S}}$. Per il precedente teorema la coppia $(\mathrm{O} \setminus S \cup R, \mathrm{O} \cup S) \in \mathcal{X}_{\mathrm{O} \setminus \mathcal{S} \cup \Omega}$. Ma $\Omega \subseteq S \subseteq \widehat{\Omega}$ e per il secondo termine della coppia valgono $\mathrm{O} \cup \Omega \subseteq \mathrm{O} \cup S$ e $\mathrm{O} \cup S \subseteq \widehat{\phi}(\mathrm{O} \cup S) = \widehat{\phi}(\mathrm{O} \cup \widehat{\phi}(S)) = \widehat{\phi}(\mathrm{O} \cup \widehat{\Omega}) = \widehat{\phi}(\mathrm{O} \cup \widehat{\phi}(\Omega)) = \widehat{\phi}(\mathrm{O} \cup \Omega)$. Si può quindi concludere che $(\mathrm{O} \setminus S \cup R, \mathrm{O} \cup S) \in \mathcal{X}_{\mathrm{O} \cup \Omega}$.

Questo ha interesse per introdurre un nuova procedura, di *chiusura progressiva*, che combina inferenza con acquisizione di conoscenza dall'esterno allo scopo di costruire un insieme invariante quanto più grande possibile. La terminazione di questa procedura è garantita solo sotto ipotesi che tipicamente non si verificano, ma si può in compenso assicurare la produzione di un particolare risultato d'interesse in un numero di passi finito. Come sarà chiaro nel seguito, non si desidera affatto che questa procedura termini ma solo che renda disponibili all'esterno i risultati parziali che calcola a mano a mano che la sua esecuzione procede.

Sia $\Omega^* \in \mathcal{Z}_D^*$ un obiettivo e $\omega(\bullet) \subseteq \Omega^*$ una funzione di ingresso che per il momento può essere pensata come variabile aleatoria. Indicando con $\Omega_0 \subseteq \Omega^*$ la conoscenza disponibile all'inizializzazione e $\{\Omega_1, \Omega_2, \ldots\}$ i campioni di $\omega(\bullet)$ raccolti dalla procedura durante l'esecuzione, si definiscono le loro unioni:

$$\Omega[a, b] = \bigcup_{k=a}^{b} \Omega_k \subseteq \Omega^*$$

$$\Omega_\infty = \Omega[0, \infty] = \bigcup_{k=0}^{\infty} \Omega_k \subseteq \Omega^*$$

Dato che si è supposto l'insieme di rappresentazione finito, anche Ω_∞ lo sarà e quindi $\exists\, h \geq 0 \mid \Omega_\infty = \Omega[0, h]$ e $\forall\, k \geq h \;\; \Omega_k \subseteq \Omega[0, h]$.

Si definisce poi la trasformazione $\eta(\bullet) : \mathcal{Z}_D \times \mathcal{Z}_D \to \mathcal{Z}_D \times \mathcal{Z}_D$ che incrementa la conoscenza disponibile campionando l'ingresso. Una proprietà precedente assicura che $\forall (R, S) \in \mathcal{X}_{\Omega[0,n-1]} \;\; (R \cup \Omega_n \setminus S, S \cup \Omega_n) \in \mathcal{X}_{\Omega[0,n-1] \cup \Omega_n} = \mathcal{X}_{\Omega[0,n]}$. Data $\omega(\bullet)$, $\eta(\bullet, (R, S))$ può dunque essere calcolata con la funzione 2.7.

1. $(\Delta, \Phi) \leftarrow \Xi$
2. $\Psi \leftarrow \omega(\bullet) \setminus \Phi$
3. $\Delta \leftarrow \Delta \cup \Psi$
4. $\Phi \leftarrow \Phi \cup \Psi$
5. $\Upsilon \leftarrow (\Delta, \Phi)$

Figura 2.7. Calcolo di $\Upsilon = \eta(\bullet, \Xi)$

La composizione di ζ con η, $\theta(\bullet) = \eta(\bullet) \circ \zeta : \mathcal{Z}_D \times \mathcal{Z}_D \to \mathcal{Z}_D \times \mathcal{Z}_D$, è alla base della procedura di chiusura progressiva. θ rappresenta infatti la trasformazione compiuta da un singolo passo della procedura e per essa vale $\forall\, \Xi \in \mathcal{X}_{\Omega[0,n-1]}$ certamente anche $\zeta(\Xi) \in \mathcal{X}_{\Omega[0,n-1]}$ e $\theta(\bullet, \Xi) = \eta(\bullet, \zeta(\Xi)) \in \mathcal{X}_{\Omega[0,n]}$.

Dato che $\Xi^{(0)} = (\Omega_0, \Omega_0) \in \mathcal{X}_{\Omega_0} = \mathcal{X}_{\Omega[0,0]}$, applicando θ per k volte a $\Xi^{(0)}$ si otterrà una coppia $\Xi^{(k)} = (\Delta^{(k)}, \Phi^{(k)}) \in \mathcal{X}_{\Omega[0,k]}$. D'altronde, $\forall\, k \geq h \mid \Omega_\infty = \Omega[0, h]$ vale $\Xi^{(k)} \in \mathcal{X}_{\Omega_\infty}$ e questo implica

1. $\Delta \leftarrow \Omega_0$
2. $\Phi \leftarrow \Omega_0$
3. Finché $\Phi \neq \Omega^*$ ripeti...
   (a) $(\Delta, \Phi) \leftarrow \eta(\bullet, \zeta(\Delta, \Phi))$

Figura 2.8. Calcolo di $\Phi = \widehat{\phi}(\bigcup_{k=0}^{\infty} \Omega_k)$

$\Omega_\infty \subseteq \Phi^{(k)} \Rightarrow \Omega_k \setminus \Phi^{(k)} = \emptyset \Rightarrow \eta(\bullet, \Upsilon^{(k)}) = \Upsilon^{(k)}$ con $\Upsilon^{(k)} = \zeta(\Xi^{(k)})$, ovvero $\theta(\bullet, \Xi^{(k)}) = \zeta(\Xi^{(k)})$. Da un certo passo h in poi, la procedura di chiusura progressiva degenera dunque in semplice chiusura di $\Xi \in \mathcal{X}_{\Omega_\infty}$ che converge a $\widehat{\zeta}(\Xi) = (\emptyset, \widehat{\Omega}_\infty)$, come già dimostrato.

La relazione $\Omega_\infty \subseteq \Omega^*$ implica $\widehat{\Omega}_\infty \subseteq \Omega^*$. Se $\widehat{\Omega}_\infty = \Omega^*$ la procedura termina e merita il nome di algoritmo, altrimenti essa continua ad applicare θ su $(\emptyset, \widehat{\Omega}_\infty)$ per sempre, in attesa di nuovi campioni di $\omega(\bullet) \notin \Omega_\infty$. Questo perché Ω_∞ non è dato a priori e nessuna ipotesi è stata fatta sulla funzione $\omega(\bullet)$.

A rigore, neppure Ω^* è dato ma $\Phi \subseteq \Omega^*$ e si può sostituire il confronto $\Phi \neq \Omega^*$ con un più semplice $\operatorname{card} \Phi < \operatorname{card} \Omega^*$, con $\operatorname{card} \Omega^* = \operatorname{card} Z_D/2$ noto. Ricordando poi che $r \notin \mu^*(r)$, vale $\mu^*_{\Phi\setminus\Delta}(r) = \mu^*_{\Phi\setminus(\Delta\setminus\{r\})}(r)$ e dunque $\lambda_{\Phi\setminus\Delta}(r) = \phi(\{r\} \mid \mu^*_{\Phi\setminus\Delta}(r)) = \phi(\{r\} \mid \mu^*_{\Phi\setminus(\Delta\setminus\{r\})}(r)) = \lambda_{\Phi\setminus(\Delta\setminus\{r\})}(r)$. Risulta dunque lecito rimuovere r da Δ prima del calcolo di λ, anzichè dopo, ed esplicitando le funzioni ζ ed η si giunge a riscrivere la procedura di chiusura progressiva come mostrato in 2.9.

1. $\Delta \leftarrow \Omega_0$
2. $\Phi \leftarrow \Omega_0$
3. Finché $\operatorname{card} \Phi < \operatorname{card} \Omega^*$ ripeti...
   (a) $\Psi \leftarrow \omega(\bullet) \setminus \Phi$
   (b) Se $\Delta \neq \emptyset$ allora...
      i. $\exists\, r \in \Delta \mid \forall\, s \in \Delta \;\; r \preceq s$
      ii. $\Delta \leftarrow \Delta \setminus \{r\}$
      iii. $\Psi \leftarrow \Psi \cup \lambda_{\Phi\setminus\Delta}(r) \setminus \Phi$
   (c) Se $\Psi \neq \emptyset$ allora...
      i. $\Delta \leftarrow \Delta \cup \Psi$
      ii. $\Phi \leftarrow \Phi \cup \Psi$

Figura 2.9. Calcolo di $\Phi = \widehat{\phi}(\bigcup_{k=0}^{\infty} \Omega_k)$

Evidentemente la proprietà $\forall\, k \geq 0 \;\; \Delta^{(k)} \subseteq \Phi^{(k)}$ continua a valere e così pure $\forall\, k \geq 0 \;\; \Psi^{(k)} \cap \Phi^{(k)} = \emptyset$ per il modo in cui è stata scritta η. Di conseguenza si potrà ancora affermare che $\forall\, k \geq 0 \mid \Delta^{(k)} \neq \emptyset \;\; S^{(k+1)} = \Phi^{(k+1)} \setminus \Delta^{(k+1)} = (\Phi^{(k)} \cup \Psi^{(k)}) \setminus (\Delta^{(k)} \setminus \{r^{(k)}\} \cup \Psi^{(k)}) = \Phi^{(k)} \setminus (\Delta^{(k)} \setminus \{r^{(k)}\}) = \Phi^{(k)} \setminus \Delta^{(k)} \cup \{r^{(k)}\} = S^{(k)} \cup \{r^{(k)}\}$ perché $r^{(k)} \in \Delta^{(k)}$. Tutte le proprietà positive dei precedenti algoritmi di chiusura sono mantenute ma in questa procedura il criterio di scelta di $r \in \Delta$ acquisisce una importanza molto maggiore ed è stato specificato attraverso un vincolo di precedenza, $\forall\, s \in \Delta \;\; r \preceq s$, con $\preceq$ un qualunque operatore di ordinamento soddisfacente i requisiti già esposti.

L'importanza del criterio di selezione di r è legata alla non terminazione della procedura quando

$\widehat{\Omega}_\infty \subset \Omega^*$. Si è anticipato che questo non costituisce un problema pur di poter restituire all'esterno gli elementi di Φ a mano a mano che vengono calcolati ed in effetti, nel quadro di un'applicazione tempo reale, quello che ci si aspetta da una simile procedura è che reagisca ad un ingresso Ω_k producendo una opportuna successione di uscite. La terminazione si verifica solo nel caso estremo in cui $\Phi \in \mathcal{Z}_D^*$, cioè tutto ciò che poteva essere prodotto lo è stato fino al raggiungimento di un insieme di conoscenza massimale.

L'ordine di produzione delle conseguenze logiche di r indotto dalla relazione di ordinamento $r \preceq s$ determina la progressiva disponibilità delle uscite. Per formalizzare questo concetto è opportuno definire gli insiemi:

$$N_{h,k} = \{i \in [h,k] \mid \Delta^{(i)} \neq \emptyset\} \subset \mathbf{N}$$

$$N_{h,\infty} = \{i \in [h,\infty) \mid \Delta^{(i)} \neq \emptyset\} \subseteq \mathbf{N}$$

$$R_{h,k} = \bigcup_{i \in N_{h,k}} \{r^{(i)}\} \subseteq Z_D$$

Anche a prescindere da un particolare criterio di selezione, è possibile dimostrare che per ogni $s^* \in \widehat{\Omega}_\infty$, se $\exists\, k \geq 0 \mid \forall\, i \in N_{k+1,\infty} \;\; s^* \neq r^{(i)}$ allora $s^* \in \Phi^{(k)}$. La dimostrazione di questa proprietà può però essere notevolmente semplificata se si opera con vincolo di precedenza come quello già discusso. Inoltre, sebbene a rigore superflua in questa fase, la scelta di un simile criterio risulterà essenziale in teoremi successivi e non si perde quindi in generalità supponendola fin da questo punto. Ricordando che $r \prec s \Rightarrow r \neq s$ si preferisce il seguente enunciato:

Teorema 2.6.3 *Dato $s^* \in \widehat{\Omega}_\infty$, se $\exists\, k \geq 0 \mid \forall\, i \in N_{k+1,\infty} \;\; s^* \prec r^{(i)}$ allora $s^* \in \Phi^{(k)}$.*

Dimostrazione Supponendo che $s^* \notin \Phi^{(k)}$, se $s^* \in \widehat{\Omega}_\infty$ certamente $\exists\, n \mid \Phi^{(n)} = \widehat{\Omega}_\infty$ e quindi $s^* \in \Phi^{(n)} \setminus \Phi^{(k)} = \bigcup_{i=k}^{n-1} \Psi^{(i)} \subseteq \bigcup_{i=k+1}^{n} \Delta^{(i)} = \bigcup_{i \in N_{k+1,n}} \Delta^{(i)}$ perché $\forall\, i \geq 1 \;\; \Psi^{(i-1)} \subseteq \Delta^{(i)}$. Il criterio di selezione garantisce che se $\Delta^{(i)} \neq \emptyset$, $\forall\, s \in \Delta^{(i)} \;\; r^{(i)} \preceq s$ e se $i \in N_{k+1,\infty}$ vale $\forall\, s \in \Delta^{(i)} \;\; s^* \prec r^{(i)} \preceq s$, cioè $\forall\, s \in \bigcup_{i \in N_{k+1,n}} \Delta^{(i)} \;\; s^* \prec s$. Ma $s^* \in \bigcup_{i \in N_{k+1,n}} \Delta^{(i)}$ e questo implica $s^* \prec s^*$, contraddicendo l'ipotesi $s^* \notin \Phi^{(k)}$.

Se le uscite sono soggette a vincoli temporali la scelta di un particolare ordinamento può garantire o meno il rispetto degli stessi. La procedura di chiusura progressiva che verrà implementata nell'esecutore di specifiche riuscirà a farlo sotto opportune ipotesi. Per analizzarla è però necessario caratterizzare maggiormente l'insieme di rappresentazione.

2.7 Inferenza causale e teorema di tempo reale

Dato che le asserzioni logiche rappresentate in Z_D descrivono storie di un sistema dinamico, asserzioni relative ad una stessa variabile in istanti diversi dovranno ricevere rappresentazioni distinte. Una possibile scelta è quella di definire l'insieme di rappresentazione come prodotto cartesiano di un insieme E in cui siano rappresentate asserzioni logiche istantanee con un orizzonte temporale discreto $T = [t_m, t_M] \subseteq \mathbf{Z}$. La coppia $(e,t) \in E \times T = Z_D$, con $e \in E$ e $t \in T$, sarà indicata anche come $e \,@\, t$ e prenderà il nome di *evento*.

Si definisce inoltre la negazione di $e \,@\, t \in Z_D$ come $\neg(e \,@\, t) = (\neg e) \,@\, t$. Dato che $\forall\, r \in Z_D \;\; \neg r \in Z_D$, varrà anche $\forall\, e \in E \;\; \neg e \in E$ ovvero $E = \overline{E}$.

Infine, si sceglie come ordinamento la relazione:

$$e \,@\, t_1 \preceq f \,@\, t_2 \iff t_1 \leq t_2$$

Le sue proprietà fondamentali sono verificate di seguito:

1. $\forall\, e \,@\, t \in Z_D \;\; e \,@\, t \preceq e \,@\, t$ perché $t \leq t$.

2. $\forall e @ t_1, f @ t_2 \in Z_D \quad e @ t_1 \preceq f @ t_2 \vee f @ t_2 \preceq e @ t_1$ perché $t_1 \leq t_2$ oppure $t_2 \leq t_1$. Entrambe queste condizioni si verificano solo nel caso $e @ t_1 \simeq f @ t_2 \iff t_1 = t_2$.

3. $\forall e @ t_1, f @ t_2, g @ t_3 \in Z_D \;\; e @ t_1 \preceq f @ t_2 \preceq g @ t_3 \Rightarrow e @ t_1 \preceq g @ t_3$ perché $t_1 \leq t_2 \leq t_3 \Rightarrow t_1 \leq t_3$.

Se gli elementi $r^{(i)}$ selezionati dalla procedura sono ordinati la finitezza di E, implicata da quella di Z_D, consente una stima della loro distribuzione su T.

Teorema 2.7.1 *Se* $h, k \in N_{h,k}$, $\forall i, j \in N_{h,k} \;\; i \leq j \Rightarrow r^{(i)} \preceq r^{(j)}$ *e* $\operatorname{card} N_{h,k} > \operatorname{card} E$ *allora* $r^{(h)} \prec r^{(k)}$.

Dimostrazione Siano $i, j \in N_{h,k} \Rightarrow \Delta^{(i)} \neq \emptyset, \Delta^{(j)} \neq \emptyset \Rightarrow \exists\, r^{(i)}, r^{(j)} \in R_{h,k}$. Al passo i vale $S^{(i+1)} = S^{(i)} \cup \{r^{(i)}\} \Rightarrow r^{(i)} \in S^{(i+1)} \subseteq S^{(j)} = \Phi^{(j)} \setminus \Delta^{(j)}$ per ogni $j \in N_{i+1,k}$. Dunque $r^{(i)} \notin \Delta^{(j)}, r^{(j)} \in \Delta^{(j)}$ e da questo si ricava che $\forall i, j \in N_{h,k} \;\; i < j \Rightarrow r^{(i)} \neq r^{(j)}$. Gli elementi $r^{(i)} \in R_{h,k}$ sono quindi tutti distinti e vale $\operatorname{card} R_{h,k} = \operatorname{card} N_{h,k} > \operatorname{card} E$ per ipotesi. Se $r^{(h)} \simeq e @ t$ l'ordinamento $r^{(i)} \preceq r^{(j)}$ consente di scrivere che $R_{h,k} \subseteq E \times [t, t_M]$ e, dato che $\operatorname{card} R_{h,k} > \operatorname{card} E = \operatorname{card} E \times \{t\}$, certamente $R_{h,k} \not\subseteq E \times \{t\} \Rightarrow \exists\, i_0 \in N_{h,k} \mid r^{(i_0)} \in E \times [t+1, t_M]$. Ma $\forall j \in N_{i_0,k} \;\; r^{(i_0)} \preceq r^{(j)}$ e vale $R_{i_0,k} \subseteq E \times [t+1, t_M]$, da cui $r^{(h)} \simeq e @ t \prec e @ (t+1) \preceq r^{(i_0)} \preceq r^{(k)}$ ovvero $r^{(h)} \prec r^{(k)}$.

Teorema 2.7.2 *Se* $h, k \in N_{h,k}$, $\forall i, j \in N_{h,k} \;\; i \leq j \Rightarrow r^{(i)} \preceq r^{(j)}$ *e* $\operatorname{card} N_{h,k} > \tau \cdot \operatorname{card} E > 0$ *allora* $e @ t \preceq r^{(h)} \Rightarrow e @ (t + \tau) \preceq r^{(k)}$.

Dimostrazione L'ipotesi $\tau \cdot \operatorname{card} E > 0$ assicura $\tau \geq 1$ e $\operatorname{card} N_{h,k} > \tau \cdot \operatorname{card} E \geq 1 \Rightarrow \operatorname{card} N_{h,k} \geq 2$. Da $k \in N_{h,k}$ si ricava $\operatorname{card} N_{h,k-1} = \operatorname{card} N_{h,k} - 1$ che, assieme a $\operatorname{card} N_{h,k} > \tau \cdot \operatorname{card} E$, implica $\operatorname{card} N_{h,k-1} = \operatorname{card} N_{h,k} - 1 \geq \tau \cdot \operatorname{card} E$. Allora esiste una successione $\{i_l\}_{l=1}^{\tau+1}$ a valori in $N_{h,k}$, con $i_1 = h$, $i_{\tau+1} = k$ e $\forall l, m \in [1, \tau+1] \;\; l < m \Rightarrow i_l < i_m$, tale che $N_{h,k-1} = \bigcup_{l=1}^{\tau} N_{i_l, i_{l+1}-1}$ e $\forall l \in [1, \tau] \;\; \operatorname{card} N_{i_l, i_{l+1}-1} \geq \operatorname{card} E$. Ma $\forall l \in [1, \tau+1] \;\; i_l \in N_{h,k} \Rightarrow \forall l \in [1, \tau] \;\; i_{l+1} \in N_{i_l, i_{l+1}}$ e quindi $\operatorname{card} N_{i_l, i_{l+1}} = \operatorname{card} N_{i_l, i_{l+1}-1} + 1 \geq \operatorname{card} E + 1 > \operatorname{card} E$. Dato che $\forall i, j \in N_{h,k} \;\; i \leq j \Rightarrow r^{(i)} \preceq r^{(j)}$ ad ognuno degli insiemi $N_{i_l, i_{l+1}}$ si può applicare il teorema precedente ottenendo $\forall l \in [1, \tau] \;\; r^{(i_l)} \prec r^{(i_{l+1})}$. Se $r^{(i_l)} \simeq e @ t_l$ si può scrivere $t \leq t_1$ e $\forall l \in [1, \tau] \;\; t_l < t_{l+1}$ ovvero $t_l \leq t_{l+1} - 1$. Nel complesso vale dunque $t \leq t_1 \leq t_2 - 1 \leq \cdots \leq t_{l+1} - l \leq \cdots \leq t_{\tau+1} - \tau$, da cui $t + \tau \leq t_{\tau+1} \Rightarrow e @ (t + \tau) \preceq e @ t_{\tau+1} \simeq r^{(k)}$.

Le ipotesi, abbastanza artificiose, del seguente teorema hanno interesse tecnico in una dimostrazione successiva ma non sono in effetti più forti di quelle già incontrate.

Teorema 2.7.3 *Se* $h, k \in N_{h,k}$, $r^{(h)} \simeq e @ t$, $\forall i, j \in N_{h,\infty} \;\; i \leq j \Rightarrow r^{(i)} \preceq r^{(j)} \vee e @ t^* \preceq r^{(j)}$ *e* $\operatorname{card} N_{h,k} > \max\{0, t^* - t\} \cdot \operatorname{card} E$ *allora* $\forall j \in N_{k,\infty} \;\; e @ t^* \preceq r^{(j)}$.

Dimostrazione Sia $\tau = t^* - t$. Se $\tau \leq 0$ vale $t^* \leq t$, cioè $e @ t^* \preceq e @ t \simeq r^{(h)}$. Supponendo invece $\tau > 0$, si ha $\operatorname{card} N_{h,k} > \tau \cdot \operatorname{card} E > 0$ e se $\forall i, j \in N_{h,k} \;\; i \leq j \Rightarrow r^{(i)} \preceq r^{(j)}$ si può applicare il teorema precedente a $N_{h,k}$ ottenendo $e @ t \simeq r^{(h)} \Rightarrow e @ (t + \tau) = e @ t^* \preceq r^{(k)}$. In caso contrario $\exists\, i, j_0 \in N_{h,k}$ tali che $i \leq j_0$ e $r^{(i)} \succ r^{(j_0)}$, quindi deve valere $e @ t^* \preceq r^{(j_0)}$. In tutti e tre i casi $\exists\, i_0 \in \{h, j_0, k\} \subseteq N_{h,k} \mid e @ t^* \preceq r^{(i_0)}$ e $\forall j \in N_{i_0,\infty} \;\; r^{(i_0)} \preceq r^{(j)} \vee e @ t^* \preceq r^{(j)}$, dunque $\forall j \in N_{k,\infty} \subseteq N_{i_0,\infty} \;\; e @ t^* \preceq r^{(j)}$.

Per esprimere in forma più compatta il criterio di selezione degli $r^{(i)}$ si definisce l'operatore *estremo sinistro* $\sigma : \mathcal{Z}_D \to T \cup \{+\infty\}$ come:

1. $\sigma(\emptyset) = +\infty$
2. $\forall\, S \in \mathcal{Z}_D \setminus \{\emptyset\} \;\; \sigma(S) = \min \prod_T(S)$

dove con $\prod_T(S)$ si è indicata la proiezione di S su T. L'operatore σ gode di una proprietà notevole, $\forall\, R, S \in \mathcal{Z}_D \;\; \sigma(R \cup S) = \min\{\sigma(R), \sigma(S)\}$. Da questa deriva che $\forall\, R, S \in \mathcal{Z}_D \;\; \sigma(S) \leq \sigma(S \setminus R)$.

In questi termini, il criterio di selezione $\forall\, s \in \Delta \;\; r \preceq s$ può essere espresso da $r = e \,@\, \sigma(\Delta)$. Infatti la scelta cade ogni volta su un elemento con t minimo, cioè un estremo sinistro di Δ. La generica relazione di precedenza $r^{(i)} \preceq r^{(j)}$ si riscrive come $\sigma(\Delta^{(i)}) \leq \sigma(\Delta^{(j)})$ e questo consente di riformulare il precedente teorema in modo più sintetico:

Teorema 2.7.4 *Se* $\forall\, i \geq h \;\; \sigma(\Delta^{(i+1)}) \geq \min\{\sigma(\Delta^{(i)}), t^*\}$ *allora* $\forall\, j \geq h + \max\{0, t^* - \sigma(\Delta^{(h)})\} \cdot \operatorname{card} E \;\; t^* \leq \sigma(\Delta^{(j)})$.

Dimostrazione Da $\forall\, i \geq h \;\; \sigma(\Delta^{(i+1)}) \geq \min\{\sigma(\Delta^{(i)}), t^*\}$ si ricava che finché $\sigma(\Delta^{(i)}) < t^*$ vale $\sigma(\Delta^{(i)}) \leq \sigma(\Delta^{(i+1)})$, cioè la successione dei $\sigma(\Delta)$ è monotona non decrescente. Se poi si verifica $\sigma(\Delta^{(i)}) \geq t^*$ anche $\sigma(\Delta^{(i+1)}) \geq t^*$, proprietà che si conserva da quel punto in poi. In altre parole, $\forall\, i, j \in [h, \infty) \;\; i \leq j \Rightarrow \sigma(\Delta^{(i)}) \leq \sigma(\Delta^{(j)}) \vee t^* \leq \sigma(\Delta^{(j)})$, da cui $\forall\, i, j \in N_{h,\infty} \;\; i \leq j \Rightarrow r^{(i)} \preceq r^{(j)} \vee e \,@\, t^* \preceq r^{(j)}$. Chiamando $k = h + \max\{0, t^* - \sigma(\Delta^{(h)})\} \cdot \operatorname{card} E$, se si suppone $N_{h,k} = [h, k]$ valgono $h, k \in N_{h,k}$ e $\operatorname{card} N_{h,k} = \operatorname{card}[h, k] = k - h + 1 > k - h = \max\{0, t^* - \sigma(\Delta^{(h)})\} \cdot \operatorname{card} E$. Si può dunque applicare il precedente teorema ottenendo $e \,@\, t^* \preceq r^{(k)}$, cioè $t^* \leq \sigma(\Delta^{(k)})$. Se invece $N_{h,k} \neq [h, k]$ certamente $\exists\, j_0 \in [h, k] \setminus N_{h,k}$ per cui vale $\Delta^{(j_0)} = \emptyset$ ovvero $\sigma(\Delta^{(j_0)}) = +\infty \geq t^*$. In entrambi i casi $\exists\, i_0 \in \{j_0, k\} \subseteq [h, k] \mid t^* \leq \sigma(\Delta^{(i_0)})$ e $\forall\, j \geq i_0 \;\; \sigma(\Delta^{(i_0)}) \leq \sigma(\Delta^{(j)}) \vee t^* \leq \sigma(\Delta^{(j)})$, quindi $\forall\, j \geq k \geq i_0 \;\; t^* \leq \sigma(\Delta^{(j)})$.

La funzione inferenza ϕ e quella locale λ si dicono *causali* se:

$$\forall\, r \in Z_D \;\; s \in \nu^*(r) \Rightarrow r \preceq s$$

dove ν^* è la funzione frontiera di uscita ad esse associata. Un diverso modo di esprimere lo stesso concetto è $\forall\, e \,@\, t \in Z_D \;\; \nu^*(e \,@\, t) \subseteq E \times [t, +\infty)$. Intuitivamente, questo significa che il processo d'inferenza si basa sulla storia passata delle variabili logiche per produrne i valori futuri. Sebbene questo non sembri un assunto troppo limitante, si sottolinea che nel ragionamento comune può accadere che da dati presenti si inferisca conoscenza sul passato e questo è un esempio di processo non causale secondo questa definizione.

Per giustificare questo assunto si postula che, tra tutte le possibili funzioni inferenza, ne esista una causale capace di produrre le uscite future del sistema specificato, data la storia passata degli ingressi e lo stato iniziale, se il sistema specificato è causale nel senso comune del termine.

In ipotesi di inferenza causale si può scrivere che $\forall\, e \,@\, t \in Z_D, S \in \mathcal{Z}_D \;\; t \leq \sigma(\lambda_S(e \,@\, t))$. Infatti se $\lambda_S(e \,@\, t) = \emptyset$ vale $t < +\infty = \sigma(\lambda_S(e \,@\, t))$, mentre se $\exists\, f \,@\, t' \in \lambda_S(e \,@\, t) \subseteq \nu^*(e \,@\, t)$ per esso si verifica $e \,@\, t \preceq f \,@\, t' \Rightarrow t \leq t'$ e quindi $t \leq \min \prod_T(\lambda_S(e \,@\, t)) = \sigma(\lambda_S(e \,@\, t))$.

Teorema 2.7.5 *Se* ϕ *è causale allora* $\forall\, i \geq 0 \;\; \sigma(\Delta^{(i+1)}) \geq \min\{\sigma(\Delta^{(i)}), \sigma(\Omega_{i+1} \setminus \Phi^{(i)})\}$.

Dimostrazione Ad ogni passo $i \in N_{0,\infty}$, l'algoritmo di chiusura progressiva calcola $\Delta^{(i+1)} = \Delta^{(i)} \setminus \{r^{(i)}\} \cup \lambda_{S^{(i)}}(r^{(i)}) \setminus \Phi^{(i)} \cup \Omega_{i+1} \setminus \Phi^{(i)}$, da cui $\sigma(\Delta^{(i+1)}) = \min\{\sigma(\Delta^{(i)} \setminus \{r^{(i)}\}), \sigma(\lambda_{S^{(i)}}(r^{(i)}) \setminus \Phi^{(i)}), \sigma(\Omega_{i+1} \setminus \Phi^{(i)})\}$. Il criterio di scelta assicura che $r^{(i)} \simeq e \,@\, \sigma(\Delta^{(i)})$ e se la funzione inferenza è causale si può scrivere che $\sigma(\Delta^{(i)}) \leq \sigma(\lambda_{S^{(i)}}(r^{(i)}))$. Le proprietà dell'estremo sinistro implicano $\sigma(\Delta^{(i)}) \leq \sigma(\Delta^{(i)} \setminus \{r^{(i)}\})$ e $\sigma(\lambda_{S^{(i)}}(r^{(i)})) \leq \sigma(\lambda_{S^{(i)}}(r^{(i)}) \setminus \Phi^{(i)})$, quindi $\min\{\sigma(\Delta^{(i)} \setminus \{r^{(i)}\}), \sigma(\lambda_{S^{(i)}}(r^{(i)}) \setminus \Phi^{(i)})\} \geq \min\{\sigma(\Delta^{(i)}), \sigma(\lambda_{S^{(i)}}(r^{(i)}))\} = \sigma(\Delta^{(i)})$ e nel complesso $\sigma(\Delta^{(i+1)}) \geq \min\{\sigma(\Delta^{(i)}), \sigma(\Omega_{i+1} \setminus \Phi^{(i)})\}$. Per quanto riguarda i restanti passi $i \in [0, \infty) \setminus N_{0,\infty}$, valgono $\Delta^{(i)} = \emptyset$ e $\Delta^{(i+1)} = \Omega_{i+1} \setminus \Phi^{(i)}$. Si può dunque scrivere $\sigma(\Delta^{(i+1)}) = \sigma(\Omega_{i+1} \setminus \Phi^{(i)})$, per cui $\sigma(\Delta^{(i+1)}) = \min\{+\infty, \sigma(\Omega_{i+1} \setminus \Phi^{(i)})\} = \min\{\sigma(\Delta^{(i)}), \sigma(\Omega_{i+1} \setminus \Phi^{(i)})\}$ e la tesi viene così dimostrata sull'intero intervallo $[0, \infty)$.

Se si realizza la funzione d'ingresso campionando un insieme costante di variabili logiche $I \subseteq E$ tale che $I = \overline{I}$ in una successione di istanti $\{t_k\}_{k=1}^{\infty}$ a valori in T e monotona non decrescente, si può scrivere che:

$$\forall\, k \geq 1 \;\; \Omega_k = \omega(t_k) = \Omega^* \cap (I \times \{t_k\})$$

Si noti che $\forall\, k \geq 1 \;\; \Omega_k \subseteq I \times \{t_k\} \subseteq I \times [t_1, t_M]$ per definizione di $\omega(t)$ ma questo non è richiesto all'insieme di conoscenza iniziale $\Omega_0 \subseteq E \times T$, quindi $\prod_E(\Omega[1,\infty]) \subseteq I$ e $\prod_T(\Omega[1,\infty]) \subseteq [t_1, t_M]$ mentre non si può dire altrettanto di Ω_∞. L'ipotesi $I = \overline{I}$ è necessaria per poter associare ad ogni variabile di ingresso valori istantanei sia veri che falsi, mentre quella di successione non decrescente, $\forall\, k \geq 1 \;\; t_k \leq t_{k+1}$, assume un significato particolare se si immagina che la rappresentazione del tempo interna agli algoritmi sia discreta ma le variabili di ingresso siano assegnate campionando un vettore di segnali continui.

Se i segnali sono funzioni costanti a tratti di lunghezza fissa $\epsilon > 0$ sull'intervallo di tempo continuo $[\epsilon \cdot t_m, \epsilon \cdot (t_M + 1)) \subseteq \mathbf{R}$, una scelta opportuna della funzione di rappresentazione conduce ad associare ad ogni valore discreto $t \in T$ un intervallo continuo $[\epsilon \cdot t, \epsilon \cdot (t+1))$ su cui il valore delle variabili d'ingresso rimane costante. Se due campioni consecutivi Ω_k e Ω_{k+1} vengono raccolti ad istanti di tempo reale compresi in $[\epsilon \cdot t, \epsilon \cdot (t+1))$ per essi vale $\Omega_k = \Omega_{k+1} = \Omega^* \cap (I \times \{t\})$ perché $t_k = t_{k+1} = t$, se invece il secondo campione viene raccolto in $[\epsilon \cdot (t+1), \epsilon \cdot (t_M + 1))$ vale $t_k < t_{k+1}$. Per evitare facili confusioni, tutti i valori di tempo reale che seguiranno avranno la forma $\epsilon \cdot \tau$ mentre gli altri riferimenti temporali saranno espressi in unità compatibili con la rappresentazione interna della procedura. Questa convenzione ha lo scopo di rendere $\tau \in \mathbf{R}$ confrontabile con $t \in T \subseteq \mathbf{Z}$ a prescindere da ϵ.

Analogamente ad I, si può individuare un insieme di variabili di interesse $O \subseteq E$ di cui si desidera conoscere la storia su T e che verranno rese disponibili in uscita durante l'elaborazione sotto forma di segnali costanti a tratti. Come E ed I, anche l'insieme di uscita deve soddisfare la condizione $O = \overline{O}$ per consentire la produzione di segnali con valori istantanei sia veri che falsi.

Se la procedura non termina entro $\epsilon \cdot (t_M - t_1)$ unità di tempo reale si assume che da quel punto in poi valga $\Omega_\infty = \Omega[0,k]$ ovvero $\Omega_{k+1} \subseteq \Omega[0,k]$, cioè che nessuna nuova conoscenza venga più acquisita dall'esterno. Chiamando $t_\infty = \max \prod_T(\Omega[1,\infty])$, per il precedente assunto vale $\Omega[1,\infty] \subseteq \Omega[0,\infty] = \Omega_\infty = \Omega[0,k]$ e si può fissare convenzionalmente $t_{k+1} = t_\infty$ in modo da verificare $\Omega_{k+1} = \omega(t_\infty) = \Omega^* \cap (I \times \{t_\infty\}) \subseteq \Omega[1,\infty] \subseteq \Omega[0,k]$. La scelta di t_∞ massimo su $\prod_T(\Omega[1,\infty]) = \bigcup_{k=1}^{\infty}\{t_k\}$ conserva la monotonia della successione dei t_k e definendo *orizzonte di esecuzione* l'intervallo $T_\infty = [t_1, t_\infty] \subseteq T$ si può scrivere che $\forall\, k \geq 1 \;\; t_k \in T_\infty$ ovvero $\prod_T(\Omega[1,\infty]) \subseteq T_\infty$.

Inoltre $t_i, t_j \in T \Rightarrow t_j - t_i \leq t_M - t_m = \operatorname{card} T - 1$ e quindi esiste $\tau_M \in \mathbf{R}$ non maggiore di $\operatorname{card} T - 1$ tale che:

$$\forall\, i, j \in [1,\infty) \;\; i \leq j \Rightarrow t_j - t_i \leq \tau_M \cdot (j - i)$$

Per $j = i + 1$ si ha $t_{k+1} - t_k \leq \tau_M$ ed è possibile interpretare questo risultato come la garanzia che se $\tau_M < t_M - t_1$ le prime $\lfloor (t_M - t_1)/\tau_M \rfloor$ iterazioni della procedura non durano più di $\epsilon \cdot \tau_M$ unità di tempo reale ciascuna. Per quanto riguarda le eventuali iterazioni rimanenti, la loro durata in termini di tempo reale non è definita perché fuori dall'intervallo $[\epsilon \cdot t_m, \epsilon \cdot (t_M + 1))$.

Se poi $\tau_M \leq 1$ vale $t_{k+1} - t_k \leq 1$ e l'unione di tutti i t_k costituisce un intervallo discreto, cioè $\prod_T(\Omega[1,\infty]) = \bigcup_{k=1}^{\infty}\{t_k\} = T_\infty$ perché gli estremi t_1, t_∞ appartengono alla successione $\{t_k\}_{k=1}^{\infty}$ con $t_k \in T_\infty$. Dato che ad ogni iterazione viene raccolto un campione degli ingressi, sotto questa condizione il passo di campionamento risulta non maggiore di ϵ e per ogni $t \in T_\infty$ ciascuno degli intervalli $[\epsilon \cdot t, \epsilon \cdot (t+1))$ viene campionato almeno una volta. L'intera storia degli ingressi successiva all'istante di lancio della procedura, $\epsilon \cdot t_1$, viene quindi acquisita progressivamente fino alla terminazione o al raggiungimento del limite $\epsilon \cdot (t_M + 1)$ e si può scrivere $\Omega_\infty = \Omega_0 \cup \Omega^* \cap (I \times T_\infty)$.

Infine, dalla definizione di Ω_k si ricava $\forall\, k \geq 1 \;\; \prod_T(\Omega_k) = \prod_T(\Omega^* \cap (I \times \{t_k\})) = \{t_k\}$, per cui $\forall\, k \geq 1 \;\; \sigma(\Omega_k) = t_k$. Risulta allora conveniente definire $t_0 = \sigma(\Omega_0)$ ed assumere che $t_0 \leq t_1$ in modo da rendere la successione $\{t_k\}_{k=0}^{\infty} = \{\sigma(\Omega_k)\}_{k=0}^{\infty}$ monotona fin dall'inizio. L'ipotesi $\sigma(\Omega_0) \leq t_1$ potrebbe essere eliminata senza pregiudicare i risultati successivi a prezzo di una certa complicazione formale, ma questo non è stato fatto anche perché nelle situazioni di interesse pratico si può sempre operare in modo da soddisfarla.

In questo quadro si osserva che se $\Omega_{k+1} \subseteq \Phi^{(k)}$ vale $\sigma(\Omega_{k+1} \setminus \Phi^{(k)}) = +\infty > t_{k+1}$, altrimenti $\sigma(\Omega_{k+1} \setminus \Phi^{(k)}) = t_{k+1}$. In entrambi i casi $\sigma(\Omega_{k+1} \setminus \Phi^{(k)}) \geq t_{k+1}$ e, visto che la successione di istanti a cui corrispondono i campioni di ingresso è non decrescente, $\forall\, k \geq 0 \;\; \sigma(\Omega_{k+1} \setminus \Phi^{(k)}) \geq t_{k+1} \geq t_k$.

Teorema 2.7.6 $\forall\, k \geq 1 \;\; \sigma(\Omega_{k+1} \setminus \Phi^{(k)}) > t_k$.

Dimostrazione Ad ogni passo vale $\Omega_k \subseteq \Phi^{(k)}$ e se $k \geq 1$ ciò determina $\Omega_{k+1} \setminus \Phi^{(k)} \subseteq \Omega_{k+1} \setminus \Omega_k = \Omega^* \cap (I \times (\{t_{k+1}\} \setminus \{t_k\}))$. Ne deriva che $\prod_T(\Omega_{k+1} \setminus \Phi^{(k)}) \subseteq \prod_T(\Omega_{k+1} \setminus \Omega_k) = \{t_{k+1}\} \setminus \{t_k\}$, per cui $t_k \notin \prod_T(\Omega_{k+1} \setminus \Phi^{(k)})$ e $\sigma(\Omega_{k+1} \setminus \Phi^{(k)}) \neq t_k$. Ma $\sigma(\Omega_{k+1} \setminus \Phi^{(k)}) \geq t_k$ e questa disuguaglianza, assieme alla precedente, fa concludere che $\sigma(\Omega_{k+1} \setminus \Phi^{(k)}) > t_k$.

Teorema 2.7.7 *Se ϕ è causale e $t^* \leq t_h + 1$ con $h \geq 1$ allora* $\forall\, j \geq h + \max\{0, t^* - \sigma(\Delta^{(h)})\} \cdot \operatorname{card} E \;\; t^* \leq \sigma(\Delta^{(j)})$.

Dimostrazione Se ϕ è causale, $\forall\, i \geq 0 \;\; \sigma(\Delta^{(i+1)}) \geq \min\{\sigma(\Delta^{(i)}), \sigma(\Omega_{i+1} \setminus \Phi^{(i)})\}$ e per $i \geq h \geq 1$ il teorema precedente assicura che $\sigma(\Omega_{i+1} \setminus \Phi^{(i)}) > t_i$, cioè $\sigma(\Omega_{i+1} \setminus \Phi^{(i)}) \geq t_i + 1 \geq t_h + 1 \geq t^*$. Quindi $\forall\, i \geq h \;\; \sigma(\Delta^{(i+1)}) \geq \min\{\sigma(\Delta^{(i)}), t^*\}$ e per un precedente teorema questo implica $\forall\, j \geq h + \max\{0, t^* - \sigma(\Delta^{(h)})\} \cdot \operatorname{card} E \;\; t^* \leq \sigma(\Delta^{(j)})$.

Si possono così tirare le somme di queste ultime dimostrazioni enunciando un criterio generale per garantire la produzione di un qualsiasi elemento $e \,@\, t_h \in \widehat{\Omega}_\infty$ entro un certo numero di passi.

Teorema 2.7.8 *Dato $e \,@\, t_h \in \widehat{\Omega}_\infty$ con $h \geq 1$, se ϕ è causale allora $e \,@\, t_h \in \Phi^{(h + \max\{0, t_h - \sigma(\Delta^{(h)}) + 1\} \cdot \operatorname{card} E - 1)}$.*

Dimostrazione La causalità di ϕ e la scelta $t^* = t_h + 1$ soddisfano le ipotesi del teorema precedente, quindi se $k = h + \max\{0, t_h - \sigma(\Delta^{(h)}) + 1\} \cdot \operatorname{card} E$ vale $\forall\, j \geq k \;\; t_h < t_h + 1 \leq \sigma(\Delta^{(j)})$. Di conseguenza $\forall\, j \in N_{k,\infty} \;\; e \,@\, t_h \prec r^{(j)}$ e per un precedente teorema questo, assieme a $e \,@\, t_h \in \widehat{\Omega}_\infty$, implica $e \,@\, t_h \in \Phi^{(k-1)}$, che esiste certamente perché $k \geq h \geq 1$.

Questo criterio ammette un interessante caso particolare. Se $t_h \leq \sigma(\Delta^{(h)})$ si ottiene $h + \max\{0, t_h - \sigma(\Delta^{(h)}) + 1\} \cdot \operatorname{card} E - 1 \leq h + \operatorname{card} E - 1$ ed ogni uscita ad un certo istante è quindi resa disponibile entro $\operatorname{card} E$ passi dal primo passo in cui si acquisiscono gli ingressi per quello stesso istante. La condizione $t_h \leq \sigma(\Delta^{(h)})$ si dice di *decongestione* e rappresenta uno stato desiderabile poiché quando si verifica la prontezza della procedura nel reagire agli ingressi è massima. Ricordando l'interpretazione di $\epsilon \cdot \tau_M$ come limite superiore alla durata di una singola iterazione della procedura si può infatti affermare che il ritardo che può intercorrere tra l'istante di campionamento degli ingressi e la produzione delle uscite per quello stesso istante, in condizione di decongestione, è non maggiore di $\epsilon \cdot \tau_{io}$ con $\tau_{io} = \tau_M \cdot \operatorname{card} E$.

Sotto opportune ipotesi la procedura raggiunge e mantiene uno stato di decongestione. Per esaminarle è necessario definire la successione $\{k_i\}_{i=0}^{\infty}$ a valori in $\mathbf{N}$ come:

1. $k_0 = 0$
2. $k_1 = 1$
3. $\forall\, i > 1 \;\; k_i = k_{i-1} + (t_{k_{i-1}} - t_{k_{i-2}}) \cdot \operatorname{card} E$

La successione è monotona non decrescente per $i \geq 0$. Infatti $k_0 = 0 \leq 1 = k_1$ e se $k_{i-1} \leq k_i$ vale $t_{k_{i-1}} \leq t_{k_i}$ e quindi $k_i \leq k_i + (t_{k_i} - t_{k_{i-1}}) \cdot \operatorname{card} E = k_{i+1}$.

Teorema 2.7.9 *Se ϕ è causale allora* $\forall\, i \geq 0 \;\; t_{k_i} \leq \sigma(\Delta^{(k_{i+1})})$.

Dimostrazione Per induzione su i:

1. Se $i = 0$ valgono $k_0 = 0$, $k_1 = 1$ e $t_0 \leq \sigma(\Delta^{(1)})$. Infatti $\sigma(\Delta^{(0)}) = \sigma(\Omega_0) = t_0$ e $\sigma(\Omega_1 \setminus \Phi^{(0)}) \geq t_0$, come precedentemente osservato. La causalità di ϕ consente allora di concludere che $\sigma(\Delta^{(1)}) \geq \min\{\sigma(\Delta^{(0)}), \sigma(\Omega_1 \setminus \Phi^{(0)})\} = t_0$.

2. Se $i \geq 1$ anche $k_i \geq 1$ e supponendo $t_{k_{i-1}} \leq \sigma(\Delta^{(k_i)})$, al passo successivo vale $k_{i+1} = k_i + (t_{k_i} - t_{k_{i-1}}) \cdot \mathrm{card}\, E \geq k_i + (t_{k_i} - \sigma(\Delta^{(k_i)})) \cdot \mathrm{card}\, E$, cioè $k_{i+1} - k_i \geq (t_{k_i} - \sigma(\Delta^{(k_i)})) \cdot \mathrm{card}\, E$. Dato che la successione dei k_i è non decrescente si può scrivere $k_{i+1} - k_i \geq 0$ e confrontando questa disuguaglianza con la precedente si ottiene $k_{i+1} - k_i \geq \max\{0, t_{k_i} - \sigma(\Delta^{(k_i)})\} \cdot \mathrm{card}\, E$. In ipotesi di causalità, per ogni $t^* \leq t_h + 1$ vale $\forall\, j \geq h + \max\{0, t^* - \sigma(\Delta^{(h)})\} \cdot \mathrm{card}\, E \;\; t^* \leq \sigma(\Delta^{(j)})$ e scegliendo $h = k_i \geq 1$, $t^* = t_{k_i}$ e $j = k_{i+1} \geq k_i + \max\{0, t_{k_i} - \sigma(\Delta^{(k_i)})\} \cdot \mathrm{card}\, E$ si ottiene $t_{k_i} \leq \sigma(\Delta^{(k_{i+1})})$.

Se $\tau_{io} = \tau_M \cdot \mathrm{card}\, E < 1$ si dimostra che la successione converge a $\lim_{i \to \infty} k_i = k^*$.

Teorema 2.7.10 *Se ϕ è causale e $\tau_{io} < 1$ allora $t_{k^*} \leq \sigma(\Delta^{(k^*)})$ con $1 \leq k^* \leq (t_1 - t_0) \cdot \mathrm{card}\, E \cdot ((t_1 - t_0) \cdot \mathrm{card}\, E + 1)/2 + 1$.*

Dimostrazione Per $i \geq 1$ valgono $k_i \geq 1$ e $k_{i+1} = k_i + (t_{k_i} - t_{k_{i-1}}) \cdot \mathrm{card}\, E$, quindi chiamando $d_i = k_{i+1} - k_i$ si può scrivere che $d_{i+1} = k_{i+2} - k_{i+1} = (t_{k_{i+1}} - t_{k_i}) \cdot \mathrm{card}\, E \leq \tau_M \cdot (k_{i+1} - k_i) \cdot \mathrm{card}\, E = \tau_{io} \cdot d_i$. La successione $\{d_i\}_{i=1}^{\infty}$ ha valori in $\mathbf{N}$ perché $k_{i+1} \geq k_i \Rightarrow d_i \geq 0$ e finché $d_i \neq 0$ vale $d_{i+1} \leq \tau_{io} \cdot d_i < d_i$ per l'ipotesi $\tau_{io} < 1$, quando invece $d_i = 0$ anche $d_{i+1} = 0$. Dato che $d_{i+1} < d_i \Rightarrow d_{i+1} \leq d_i - 1$, si può scrivere $d_i \leq d_{i-1} - 1 \leq d_{i-2} - 2 \leq \cdots \leq d_1 - (i-1)$ e concludere che per $i^* = d_1 + 1$ si verifica $d_{i^*} \leq d_1 - (i^* - 1) = 0$, cioè la successione raggiunge lo zero in al più i^* passi e si mantiene nulla da quel punto in poi. Da $d_{i^*} = k_{i^*+1} - k_{i^*} = 0$ si ricava che $k_{i^*+1} = k_{i^*} = k^*$ è il limite a cui la successione dei k_i converge e la causalità di ϕ implica per il teorema precedente $t_{k_{i^*}} \leq \sigma(\Delta^{(k_{i^*+1})}) \Rightarrow t_{k^*} \leq \sigma(\Delta^{(k^*)})$. Inoltre $k^* = k_{i^*} - k_0 = (k_{d_1+1} - k_1) + (k_1 - k_0) = \sum_{i=1}^{d_1} (k_{i+1} - k_i) + 1 = \sum_{i=1}^{d_1} d_i + 1$ e da questo deriva $k^* \leq \sum_{i=1}^{d_1} (d_1 - (i-1)) + 1 = d_1 \cdot (d_1 + 1) - \sum_{i=1}^{d_1} i + 1$. Essendo $\sum_{i=1}^{d_1} i = d_1 \cdot (d_1 + 1)/2$ si ricava infine $k^* \leq d_1 \cdot (d_1 + 1)/2 + 1$ con $d_1 = k_2 - k_1 = (t_{k_1} - t_{k_0}) \cdot \mathrm{card}\, E = (t_1 - t_0) \cdot \mathrm{card}\, E$.

Si è così dimostrato che la decongestione è raggiunta entro un numero di passi che è $O(((t_1 - \sigma(\Omega_0)) \cdot \mathrm{card}\, E)^2)$. Ipotesi leggermente più deboli sono sufficienti a garantirne la conservazione:

Teorema 2.7.11 *Se ϕ è causale, $\tau_{io} \leq 1$ e $t_h \leq \sigma(\Delta^{(h)})$ con $h \geq 1$ allora $\forall\, k \geq h \;\; t_k \leq \sigma(\Delta^{(k)})$.*

Dimostrazione Sia $\{h_p\}_{p=0}^{\infty}$ una successione a valori in $\mathbf{N}$ con $h_p = h + p \cdot \mathrm{card}\, E \geq 1$ e si consideri la tesi $t_{h_p} \leq \sigma(\Delta^{(h_p)})$. Questa è vera per ipotesi per $p = 0$ mentre supponendola vera per p, la causalità di ϕ implica che per ogni $t^* \leq t_{h_p} + 1$ vale $\forall\, j \geq h_p + \max\{0, t^* - \sigma(\Delta^{(h_p)})\} \cdot \mathrm{card}\, E \;\; t^* \leq \sigma(\Delta^{(j)})$ e scegliendo $t^* = t_{h_p} + 1$ e $j = h_{p+1}$ si verifica $\max\{0, t_{h_p} - \sigma(\Delta^{(h_p)}) + 1\} \leq 1$ per ipotesi induttiva. Ne consegue che $h_{p+1} = h_p + \mathrm{card}\, E \geq h_p + \max\{0, t_{h_p} - \sigma(\Delta^{(h_p)}) + 1\} \cdot \mathrm{card}\, E$ e dunque $t_{h_p} + 1 \leq \sigma(\Delta^{(h_{p+1})})$. Ma $t_{h_{p+1}} - t_{h_p} \leq \tau_M \cdot (h_{p+1} - h_p) = \tau_M \cdot \mathrm{card}\, E = \tau_{io} \leq 1$ per ipotesi e quindi $t_{h_{p+1}} - t_{h_p} \leq 1 \Rightarrow t_{h_{p+1}} \leq t_{h_p} + 1 \leq \sigma(\Delta^{(h_{p+1})})$. Si dimostra così che $\forall\, p \geq 0 \;\; t_{h_p} \leq \sigma(\Delta^{(h_p)})$, mentre per quanto riguarda i rimanenti passi $k \in [h_p, h_{p+1})$ vale $t_k - t_{h_p} \leq \tau_M \cdot (k - h_p) < \tau_M \cdot \mathrm{card}\, E = \tau_{io} \leq 1$, da cui $t_k - t_{h_p} < 1 \Rightarrow t_k = t_{h_p}$. In questo secondo caso caso la scelta $t^* = t_{h_p}$ e $j = k$ conduce a $\max\{0, t_{h_p} - \sigma(\Delta^{(h_p)})\} = 0$ perché $t_{h_p} \leq \sigma(\Delta^{(h_p)})$, per cui $k \geq h_p = h_p + \max\{0, t_{h_p} - \sigma(\Delta^{(h_p)})\} \cdot \mathrm{card}\, E$ e di conseguenza $t_k = t_{h_p} \leq \sigma(\Delta^{(k)})$.

Finalmente è possibile scrivere un teorema che assicura il funzionamento della procedura in tempo reale da un certo passo in poi:

Teorema 2.7.12 (di tempo reale) *Se ϕ è causale e $\tau_{io} < 1$ allora $\forall\, k \geq (t_1 - t_0) \cdot \mathrm{card}\, E \cdot ((t_1 - t_0) \cdot \mathrm{card}\, E + 1)/2 + 1 \;\; e \,@\, t_k \in \widehat{\Omega}_\infty \Rightarrow e \,@\, t_k \in \Phi^{(k + \mathrm{card}\, E - 1)}$.*

Dimostrazione Chiamando $l = (t_1 - t_0) \cdot \mathrm{card}\, E \cdot ((t_1 - t_0) \cdot \mathrm{card}\, E + 1)/2 + 1$, per i due teoremi precedenti esiste $k^* \geq 1$, $k^* \leq l$ tale che $t_{k^*} \leq \sigma(\Delta^{(k^*)})$ e per ogni $k \geq l \geq k^*$ vale $t_k \leq \sigma(\Delta^{(k)})$. Quindi $\max\{0, t_k - \sigma(\Delta^{(k)}) + 1\} \leq 1$ ovvero $k + \max\{0, t_k - \sigma(\Delta^{(k)}) + 1\} \cdot \mathrm{card}\, E - 1 \leq k + \mathrm{card}\, E - 1$ e se $e \,@\, t_k \in \widehat{\Omega}_\infty$ con $k \geq k^* \geq 1$ un precedente teorema assicura che $e \,@\, t_k \in \Phi^{(k + \max\{0, t_k - \sigma(\Delta^{(k)}) + 1\} \cdot \mathrm{card}\, E - 1)} \subseteq \Phi^{(k + \mathrm{card}\, E - 1)}$.

Questo risultato ha una notevole importanza e può essere facilmente interpretato in termini di tempo reale. Per una funzione inferenza causale, se il passo di discretizzazione ϵ è maggiore del tempo $\epsilon \cdot \tau_{io}$ necessario per eseguire $\operatorname{card} E$ iterazioni della procedura, da un certo punto in poi il ritardo ingresso-uscita introdotto dal processo d'inferenza non eccede $\epsilon \cdot \tau_{io}$ e risulta quindi minore di ϵ. Il numero di iterazioni necessario a raggiungere il punto di decongestione è al più quadratico in $\operatorname{card} E$ e nella differenza tra l'istante di lancio della procedura ed il più antico istante cui si riferisce l'insieme di conoscenza iniziale.

2.8 Complessità computazionale

Per valutare la complessità computazionale di una singola iterazione della procedura è necessario esprimere gli insiemi in gioco attraverso strutture dati appropriate. Una possibile scelta è quella di implementare Δ attraverso una lista ordinata di coppie:

$$D = [F_i \mathbin{@} t_i]_{i=1}^{l} = [(F_i, t_i) \mid i \in [1, l] \subseteq \mathbf{N}, F_i \subseteq E, t_i \in T]$$

con $l = \operatorname{card} \prod_T(\Delta)$, per cui valgano le seguenti proprietà:

1. $\forall\, i \in [1, l] \;\; F_i \neq \emptyset$
2. $\forall\, i, j \in [1, l] \;\; i < j \Rightarrow t_i < t_j$
3. $\Delta = \bigcup_{i \in [1,l]} F_i \times \{t_i\}$

Su questa struttura, la selezione un un $e \mathbin{@} \sigma(\Delta)$ può avvenire prelevando un qualunque elemento $e \in F_1$, con $F_1 \mathbin{@} t_1$ testa di D, visto che $t_1 = \min \prod_T(\Delta) = \sigma(\Delta)$ a causa dell'ordinamento. Inoltre, la seconda proprietà può essere applicata con $j = i + 1$ ottenendo $t_i < t_{i+1} \Rightarrow t_i + 1 \leq t_{i+1} \Rightarrow t_{i+1} - t_i \geq 1$, da cui $t_p - \sigma(\Delta) = t_p - t_1 = \sum_{i=1}^{p-1} t_{i+1} - t_i \geq p - 1$ con $p \in [1, l]$. In altre parole, data una qualunque coppia $F_p \mathbin{@} t_p$ appartenente a D, la distanza $p - 1$ che intercorre tra questa e la testa della lista non supera la differenza tra il secondo elemento della coppia, t_p, e l'estremo sinistro σ di Δ.

A loro volta, gli insiemi F_i possono essere implementati attraverso liste non ordinate di elementi $e \in E$ ed indicando con $[]$ la lista vuota e con $\&$ l'operatore di concatenazione tra liste si arriva a riscrivere la procedura di chiusura progressiva come mostrato nelle figure 2.10 e 2.11. La procedura è stata divisa in più moduli e si è adottata la notazione $b \leftarrow \mathit{Function}_A(a)$ come sinonimo di $(A, b) \leftarrow \mathit{Function}(A, a)$ per rendere più leggibile il risultato.

Si osserva innanzitutto come il criterio di arresto sia stato intenzionalmente lasciato nel vago. Si adotteranno infatti criteri diversi a seconda dello scopo per cui si utilizza la procedura. Se si desidera esplicitare tutto ciò che è inferibile ma con il rischio della non terminazione si può ancora impiegare il precedente criterio $\operatorname{card} \Phi < \operatorname{card} \Omega^*$, implementabile efficientemente tramite un contatore da incrementare dopo ogni aggiunta di un nuovo elemento a Φ. Altri criteri ragionevoli sono $\prod_T(\omega(\bullet)) \neq \{t_M\}$, se si desidera interrompere l'inferenza quando il tempo reale raggiunge la fine dell'orizzonte, o un controllo interattivo da parte dell'utente. Il costo computazionale della valutazione di tutti questi criteri risulta trascurabile rispetto alle successive stime per il ciclo principale.

La procedura utilizza come unica struttura dati di supporto una terna $\Sigma = (\Phi, S, D)$ dove all'insieme Φ ed alla lista D si affianca una rappresentazione esplicita dell'insieme $S = \Phi \setminus \Delta$, ad uso della funzione di inferenza locale λ. La gestione di D è affidata alle funzioni *State* e *Select*, che rispettivamente aggiungono nuovi elementi nella posizione appropriata o li rimuovono dalla testa restituendoli in uscita. All'inizializzazione, la lista vuota gode di tutte le proprietà richieste dalla definizione di D e supponendo di applicare queste due funzioni ad una D che le soddisfa si può dimostrare che la D restituita in uscita conserva questa caratteristica.

Se D è composta da coppie $F \mathbin{@} t$ con $F \neq []$ ed è ordinata su t, anche le sue sottoliste D_L e D_R lo sono. Per quanto riguarda la funzione $\mathit{State}_\Sigma(e \mathbin{@} t)$, nel caso in cui in D sia presente una coppia ad ugual t in cui inserire il nuovo elemento e, la conservazione della proprietà $F \neq []$ è garantita dal

$Main(\Omega_0, \omega)$:

1. $\Sigma.(\Phi, S, D) \leftarrow (\emptyset, \emptyset, [])$
2. $\forall\, s \in \Omega_0$ esegui $State_\Sigma(s)$
3. Finché è opportuno ripeti...
   (a) $r \leftarrow Select_\Sigma()$
   (b) Se $r \neq \perp$ allora $Process_\Sigma(r)$
   (c) $\forall\, s \in \omega(\bullet)$ esegui...
      i. Se $\neg Stated?_\Sigma(s)$ allora $State_\Sigma(s)$

$State_\Sigma(e \,@\, t)$:

1. $\Sigma.\Phi \leftarrow \Sigma.\Phi \cup \{e \,@\, t\}$
2. Se $\exists\, F, D_L, D_R \mid \Sigma.D = D_L \,\&[F \,@\, t]\,\& D_R$ allora $\Sigma.D \leftarrow D_L \,\&[([e] \,\&\, F) \,@\, t]\,\& D_R$
3. ...altrimenti...
   (a) $\exists\, D_L, D_R \mid \Sigma.D = D_L \,\&\, D_R$,
   $D_L = C_L \,\&[F_L \,@\, t_L] \Rightarrow t_L < t, D_R = [F_R \,@\, t_R]\,\& C_R \Rightarrow t < t_R$
   (b) $\Sigma.D \leftarrow D_L \,\&[[e] \,@\, t]\,\& D_R$

$Select_\Sigma()$:

1. Se $\Sigma.D \neq []$ allora...
   (a) $\exists\, e, F, t, D_R \mid \Sigma.D = [([e] \,\&\, F) \,@\, t]\,\& D_R$
   (b) $\Sigma.S \leftarrow \Sigma.S \cup \{e \,@\, t\}$
   (c) Se $F \neq []$ allora $\Sigma.D \leftarrow [F \,@\, t]\,\& D_R$, altrimenti $\Sigma.D \leftarrow D_R$
   (d) $Return(e \,@\, t)$
2. ...altrimenti $Return(\perp)$

Figura 2.10. Procedura di chiusura progressiva (1/2)

$Process_\Sigma(r)$:

1. $\forall\, s \in \lambda_{\Sigma.S}(r)$ esegui...
   (a) Se $\neg Stated?_\Sigma(s)$ allora $State_\Sigma(s)$

$Stated?_\Sigma(s)$:

1. Se $s \in \Sigma.\Phi$ allora $Return(\top)$, altrimenti $Return(\bot)$

$Selected?_\Sigma(s)$:

1. Se $s \in \Sigma.S$ allora $Return(\top)$, altrimenti $Return(\bot)$

Figura 2.11. Procedura di chiusura progressiva (2/2)

fatto che la coppia modificata $([e] \,\&\, F) \,@\, t$ ha come primo elemento una lista $[e] \,\&\, F \neq []$ mentre l'ordinamento non viene alterato perché t è fissato e la struttura di D non cambia. In mancanza di una coppia con t appropriato viene creata una nuova coppia $[e] \,@\, t$, con primo elemento $[e] \neq []$, ed inserita tra le coppie a t minore e quelle a t maggiore così da garantire la conservazione dell'ordinamento. Questo secondo caso si verifica anche quando $D = [] \Rightarrow D_L = D_R = []$. Per quanto riguarda la $Select_\Sigma()$, un controllo esplicito su F impedisce che si ottenga una lista vuota a seguito della rimozione di un elemento e da $([e] \,\&\, F) \,@\, t$, eliminando la coppia $[] \,@\, t$ ed assegnando D_R, che gode della proprietà desiderata per ipotesi, a D. Questa funzione non può alterare l'ordinamento di D in quanto si limita a modificare il primo elemento della coppia di testa, irrilevante al fine della sua collocazione lungo la lista, o a rimuovere la coppia stessa restituendo al posto di D la coda ordinata D_R. Infine, la conservazione della proprietà $\Delta = \bigcup_{i \in [1,l]} F_i \times \{t_i\}$ è assicurata dal fatto che esiste una corrispondenza biunivoca tra le chiamate a *State* o *Select* e l'aggiunta o rimozione di elementi da Δ effettuata nella versione precedente della procedura.

Si osserva poi come la sola funzione che modifica S sia la *Select*, da cui la scelta di interpretare questo insieme come quello degli elementi già selezionati dalla procedura. Dato che λ si basa solo su r ed S per l'inferenza, si può ancora evitare di dettagliarne l'algoritmo di calcolo pur di garantire che all'interno della *Process* non compaiano chiamate alla *Select* o modifiche esplicite di S. Risulta invece possibile chiamare la *State* sugli elementi inferiti a mano a mano che procede la valutazione di λ senza timore di modificarne i risultati successivi.

Per procedere nel calcolo della complessità sono necessarie alcune ipotesi sulle strutture utilizzate:

- Su un insieme deve essere possibile effettuare:
  - Verifica di appartenenza di un qualunque elemento in tempo costante.
  - Inserimento di un qualunque elemento in tempo costante.
- Su una lista di lunghezza l deve essere possibile effettuare:
  - Verifica di lista vuota in tempo costante
  - Accesso ad un elemento in posizione $p \in [1, l]$ in tempo $O(p)$
  - Scansione dell'intervallo $[1, p] \subseteq [1, l]$ in tempo $O(p)$
  - Inserimento di un elemento in posizione $p \in [1, l + 1]$ in tempo $O(p)$
  - Cancellazione della testa della lista in tempo costante

Da un punto di vista pratico, le ipotesi sugli insiemi possono essere soddisfatte immaginando che gli elementi $e \in E$ siano riferimenti a strutture contenenti vettori booleani che indicano, per ogni $t \in T$, l'appartenenza di $e \,@\, t$ ad ognuno degli insiemi in questione. Le ipotesi sulle liste sono invece verificate da una qualunque implementazione dinamica delle stesse.

In queste ipotesi, le funzioni *Stated?* e *Selected?* sono eseguite in tempo costante, e così pure la *Select* poiché tutte le operazioni che questa compie sulle liste coinvolgono soltanto la testa delle stesse. Per la *State* vale invece la stima di $O(\operatorname{card} \prod_T(\Delta))$, conseguenza della scansione necessaria ad individuare il punto di inserimento del nuovo elemento in D. Supponendo poi che lettura degli ingressi $\omega(\bullet)$ possa avvenire in un tempo $O(\operatorname{card} I)$, la loro memorizzazione ha complessità $O(\operatorname{card} I \cdot \operatorname{card} \prod_T(\Delta))$, mentre quella di una singola iterazione della procedura principale è pari alla somma di quest'ultimo fattore con la complessità della funzione *Process*.

Indicando con:

$$M_\nu = \max_{r \in Z_D} \operatorname{card} \nu^*(r)$$

$$B_\nu = \max_{e \,@\, t \in Z_D, e' \,@\, t' \in \nu^*(e \,@\, t)} |t' - t|$$

si osserva che $\operatorname{card} \lambda_S(r) \leq M_\nu$ e $\forall\, e' \,@\, t' \in \lambda_S(e \,@\, \sigma(\Delta)) \subseteq \nu^*(e \,@\, \sigma(\Delta))\ \ t' - \sigma(\Delta) \leq |t' - \sigma(\Delta)| \leq B_\nu$. Gli elementi $e' \,@\, t'$ inferiti durante l'esecuzione della *Process* vengono memorizzati in D tramite coppie $F_p \,@\, t \mid e' \in F_p$ che soddisfano la relazione $p - 1 \leq t - \sigma(\Delta) \leq B_\nu$, cioè si trovano in posizioni la cui distanza dalla testa della lista non supera i B_ν elementi. Immaginando di scandire D da sinistra a destra è allora possibile maggiorare la complessità di ogni chiamata alla *State* interna alla *Process* con $O(B_\nu)$ e, dato che il numero di queste chiamate non può superare il massimo numero di elementi inferibili localmente M_ν, il costo complessivo della loro memorizzazione risulta $O(M_\nu \cdot B_\nu)$.

Racchiudendo in un generico termine M_λ il contributo dovuto alla valutazione di λ, si può scrivere che la complessità della funzione *Process* è $O(M_\lambda + M_\nu \cdot B_\nu)$ mentre per una singola iterazione della procedura principale vale la stima di $O(M_\lambda + M_\nu \cdot B_\nu + \operatorname{card} I \cdot \operatorname{card} \prod_T(\Delta))$. Nel caso generale questa espressione è maggiorabile con $O(M_\lambda + M_\nu \cdot B_\nu + \operatorname{card} I \cdot \operatorname{card} T)$ poiché $\prod_T(\Delta) \subseteq T$, ma in condizione di decongestione gli elementi $e \,@\, t$ restituiti dalla funzione d'ingresso avranno $t \leq \sigma(\Delta)$ e verranno memorizzati nella testa di D riducendo la complessità della singola iterazione a $O(M_\lambda + M_\nu \cdot B_\nu + \operatorname{card} I)$.

Una tecnica che garantisce la decongestione all'istante del lancio della procedura è quella di modificare la scelta della coppia $(\Omega_0, \Omega_0) \in \mathcal{X}_{\Omega_0}$ che inizializza lo stato $\Xi^{(0)} = (\Delta^{(0)}, \Phi^{(0)})$ sostituendola con $(\emptyset, \widehat{\Omega}_0) \in \mathcal{X}_{\Omega_0}$. Questa seconda coppia soddisfa le condizioni di convergenza ed assicura contemporaneamente che $t_1 = \sigma(\Omega_1) = \sigma(\Delta^{(1)})$. Qualora non fosse possibile disporre di $\widehat{\Omega}_0$ all'istante di lancio $\epsilon \cdot t_1$ si possono escogitare altri metodi, come la memorizzazione di un riferimento alla coppia contenente l'ultimo ingresso che è stato inserito in D, per rendere la complessità computazionale delle successive iterazioni indipendente da $\operatorname{card} T$ anche in assenza di decongestione[1]. Inoltre, nonostante la lettura degli ingressi avvenga ad ogni iterazione del ciclo principale, la memorizzazione di nuovi ingressi si verifica solo una volta ogni ϵ istanti di tempo reale ed in un qualunque sistema pratico non sottodimensionato questo fattore risulta trascurabile rispetto al costo computazionale dell'inferenza che viene fatta in ogni passo di esecuzione. Qualunque sia la strada seguita per poter assumere come corretta la stima di $O(M_\lambda + M_\nu \cdot B_\nu + \operatorname{card} I)$, si pone il problema di determinare una soglia minima ϵ_m al passo di discretizzazione ϵ perché sia verificata la condizione di tempo reale $\tau_{io} = \tau_M \cdot \operatorname{card} E < 1$.

In termini di tempo reale, la durata di una singola iterazione della procedura è maggiorabile con $\epsilon \cdot \tau_M = O(M_\lambda + M_\nu \cdot B_\nu + \operatorname{card} I)$, quindi la condizione si può scrivere come $\tau_{io} = O(M_\lambda + M_\nu \cdot B_\nu + \operatorname{card} I) \cdot \operatorname{card} E / \epsilon < 1$, ovvero:

[1] In questo caso è necessario che la lista D consenta accesso in tempo costante ad un qualunque elemento, dato un riferimento allo stesso. Il ciclo principale dovrebbe essere modificato in modo da utilizzare una nuova funzione di memorizzazione per gli ingressi che acceda direttamente alla coppia successiva a quella puntata dal riferimento, aggiornandolo. Numerose altre ottimizzazioni sono possibili in conseguenza del fatto che tutti gli ingressi campionati in un certo istante finiscono nella medesima coppia di D. Pur praticabile, questa scelta non viene analizzata perché è un esempio di pura tecnica informatica.

$$\epsilon_m = \epsilon \cdot \tau_{io} = O(M_\lambda + M_\nu \cdot B_\nu + \operatorname{card} I) \cdot \operatorname{card} E < \epsilon$$

Se i fattori M_λ, M_ν, B_ν e $\operatorname{card} I$ sono indipendenti da $\operatorname{card} Z_D$, la soglia minima risulta quindi una funzione lineare in $\operatorname{card} E$ ed indipendente da $\operatorname{card} T$.

2.9 Funzioni binarie

Prima di proseguire è opportuno ripercorrere alcuni dei risultati fin qui presentati, sia per renderli più intuitivi che per introdurre la particolare forma di funzione inferenza esaminata nel seguito. Si immagina che questa sia infatti scrivibile come unione di funzioni al più binarie, cioè:

$$\forall\, S \in \mathcal{Z}_D \;\; \phi(S) = S \cup \bigcup_{r \in S} f(r) \cup \bigcup_{r,s \in S} g(r,s)$$

con $f : Z_D \to \mathcal{Z}_D$ e $g : Z_D \times Z_D \to \mathcal{Z}_D$ soddisfacenti le seguenti proprietà:

1. $\forall\, r \in Z_D \;\; g(r,r) = \emptyset$
2. $\forall\, r,s \in Z_D \;\; g(r,s) = g(s,r)$
3. $\forall\, r,s \in Z_D \;\; r \notin f(r) \cup g(r,s)$
4. $\forall\, r,s \in Z_D \;\; f(r) \cap g(r,s) = \emptyset$
5. $\forall\, r \in Z_D \;\; \rho^{-1}(\{r\}) \models \rho^{-1}(f(r))$
6. $\forall\, r,s \in Z_D \;\; \rho^{-1}(\{r,s\}) \models \rho^{-1}(g(r,s))$

Una volta presa la decisione di esprimere ϕ attraverso f e g, l'assunzione delle prime quattro ipotesi non ne limita ulteriormente la generalità ma rende assai più comprensibili i risultati. Le ultime due ipotesi garantiscono la correttezza.

Si verifica immediatamente che $\phi(\emptyset) = \emptyset$ e $\forall\, S \in \mathcal{Z}_D \;\; S \subseteq \phi(S)$. Inoltre $\phi(R \cup S) = R \cup S \cup \bigcup_{r \in R} f(r) \cup \bigcup_{s \in S} f(s) \cup \bigcup_{p,r \in R} g(p,r) \cup \bigcup_{q,s \in S} g(q,s) \cup \bigcup_{r \in R, s \in S} g(r,s) \cup \bigcup_{r \in R, s \in S} g(s,r) = \phi(R) \cup \phi(S) \cup \bigcup_{r \in R, s \in S} g(r,s)$ per la proprietà 2. Questa espressione mostra come per un qualunque $S \subseteq Q = R \cup S$ valga $\phi(S) \subseteq \phi(Q) = \phi(R \cup S)$ e tutte le proprietà caratteristiche delle funzioni inferenza sono così soddisfatte.

La stessa espressione consente di esplicitare la funzione valore aggiunto come $\xi(R,S) = \phi(R \cup S) \setminus (\phi(R) \cup \phi(S)) = \bigcup_{r \in R, s \in S} g(r,s) \setminus (\phi(R) \cup \phi(S))$. Per quanto riguarda l'incremento inferenziale di un singolo elemento $r \in Z_D$ si osserva che $\delta(\{r\}) = \phi(\{r\}) \setminus \{r\} = (\{r\} \cup f(r) \cup g(r,r)) \setminus \{r\} = f(r)$ per le proprietà 1 e 3. Il valore aggiunto di un singolo elemento rispetto ad un insieme diviene invece $\xi(\{r\}, S) = \bigcup_{s \in S} g(r,s) \setminus (\phi(\{r\}) \cup \phi(S)) = \bigcup_{s \in S} g(r,s) \setminus (\{r\} \cup f(r) \cup \phi(S)) = \bigcup_{s \in S} g(r,s) \setminus \phi(S)$ per le proprietà 3 e 4. Infine, se $S = \{s\}$ si possono nuovamente applicare le proprietà 2, 3 e 4 ottenendo $\xi(\{r\}, \{s\}) = g(r,s) \setminus \phi(\{s\}) = g(r,s) \setminus (\{s\} \cup f(s)) = g(r,s)$, analogamente a quanto fatto nel caso precedente. Le funzioni f e g possono essere quindi interpretate rispettivamente come l'incremento inferenziale di un singolo elemento ed il valore aggiunto di una coppia di elementi.

Teorema 2.9.1 $\forall\, r \in Z_D \;\; \mu^*(r) = \{s \in Z_D \mid g(r,s) \neq \emptyset\}$.

Dimostrazione Si consideri la funzione $\mu : Z_D \to \mathcal{Z}_D$ definita come $\forall\, r \in Z_D \;\; \mu(r) = \{s \in Z_D \mid g(r,s) \neq \emptyset\}$. Per essa vale $\xi(\{r\}, S) = \bigcup_{s \in S} g(r,s) \setminus \phi(S) = \bigcup_{s \in \mu_S(r)} g(r,s) \setminus \phi(S) \subseteq \bigcup_{s \in \mu_S(r)} g(r,s) \setminus \phi(\mu_S(r)) = \xi(\{r\}, \mu_S(r))$ e quindi $\mu \in \mathcal{M}_\phi$. Inoltre, supponendo che esista una funzione $\mu' \in \mathcal{M}_\phi$ ed un elemento $r_0 \in Z_D$ tale che $\exists\, s_0 \in \mu(r_0) \setminus \mu'(r_0)$, si verifica $g(r_0, s_0) = \xi(\{r_0\}, \{s_0\}) \subseteq \xi(\{r_0\}, \mu'_{\{s_0\}}(r_0)) = \xi(\{r_0\}, \emptyset) = \emptyset$ perché $s_0 \notin \mu'(r_0)$. Ma $s_0 \in \mu(r_0) \Rightarrow g(r_0, s_0) \neq \emptyset$ e la contraddizione conduce a negare l'esistenza di s_0, dimostrando così che $\forall\, \mu' \in \mathcal{M}_\phi, r \in Z_D \;\; \mu(r) \subseteq \mu'(r)$ e quindi $\forall\, r \in Z_D \;\; \mu(r) = \mu^*(r)$, ovvero la funzione μ proposta coincide con la frontiera di ingresso.

La funzione inferenza locale può essere scritta come:

$$\lambda_S(r) = \delta(\{r\}) \cup \xi(\{r\}, \mu_S^*(r)) = f(r) \cup \bigcup_{s \in \mu_S^*(r)} g(r,s) \setminus \phi(\mu_S^*(r))$$

e, dato che nell'algoritmo ad ogni passo vale $\phi(\mu_S^*(r)) \subseteq \phi(S) \subseteq \Phi$ con $S = \Phi \setminus \Delta$, l'espressione $\lambda_{\Phi\setminus\Delta}(r) \setminus \Phi$ si semplifica in $(f(r) \cup \bigcup_{s \in \mu^*_{\Phi\setminus\Delta}(r)} g(r,s)) \setminus \Phi$, con il risultato mostrato in figura 2.12.

1. $\Delta \leftarrow \Omega_0$
2. $\Phi \leftarrow \Omega_0$
3. Finché $\operatorname{card}\Phi < \operatorname{card}\Omega^*$ ripeti...
   (a) $\Psi \leftarrow \omega(\bullet) \setminus \Phi$
   (b) Se $\Delta \neq \emptyset$ allora...
      i. $\exists\, r \in \Delta \mid \forall\, s \in \Delta \;\; r \preceq s$
      ii. $\Delta \leftarrow \Delta \setminus \{r\}$
      iii. $\Psi \leftarrow \Psi \cup f(r) \setminus \Phi \cup \bigcup_{s \in \mu^*_{\Phi\setminus\Delta}(r)} g(r,s) \setminus \Phi$
   (c) Se $\Psi \neq \emptyset$ allora...
      i. $\Delta \leftarrow \Delta \cup \Psi$
      ii. $\Phi \leftarrow \Phi \cup \Psi$

Figura 2.12. Calcolo di $\Phi = \widehat{\phi}(\bigcup_{k=0}^{\infty} \Omega_k)$ per funzioni binarie

Per quanto riguarda la versione modulare della procedura, questa esplicitazione di λ porta alla riscrittura della sola funzione *Process* come in figura 2.13.

Process$_\Sigma(r)$:

1. $\forall\, s \in f(r)$ esegui...
   (a) Se $\neg$*Stated*?$_\Sigma(s)$ allora *State*$_\Sigma(s)$
2. $\forall\, s \in \mu^*(r)$ esegui...
   (a) Se *Selected*?$_\Sigma(s)$ allora...
      i. $\forall\, q \in g(r,s)$ esegui...
         A. Se $\neg$*Stated*?$_\Sigma(q)$ allora *State*$_\Sigma(q)$

Figura 2.13. Variante della *Process* per funzioni binarie

L'ipotesi di inferenza causale per funzioni binarie si riscrive come $\forall\, r, s \in Z_D \;\; z \in f(r) \cup g(r,s) \Rightarrow r \preceq z$. Se soddisfatta, consente di applicare il teorema di tempo reale per giungere ad una stima della frequenza operativa del motore d'inferenza.

Ipotizzando che la complessità di valutazione delle funzioni $f(r)$, $g(r,s)$ e $\mu^*(r)$ sia rispettivamente $O(1 + \operatorname{card} f(r))$, $O(1 + \operatorname{card} g(r,s))$ e $O(1 + \operatorname{card} \mu^*(r))$ e definendo:

$$M_\mu = \max_{r \in Z_D} \operatorname{card} \mu^*(r)$$

si osserva che $\operatorname{card} f(r) + \sum_{s \in \mu^*(r)} \operatorname{card} g(r,s) \leq M_\nu$ mentre $\operatorname{card} \mu^*(r) \leq M_\mu$. La somma delle complessità dell'unica chiamata di f con tutte le chiamate di g ritornanti insiemi non vuoti è quindi trascurabile rispetto alla complessità di memorizzazione dei loro valori di ritorno, precedentemente

stimata in $O(M_\nu \cdot B_\nu)$. Per quanto riguarda le chiamate di g che non restituiscono risultati, la loro complessità è per ipotesi costante ed il loro numero non supera $\operatorname{card} \mu^*(r) \leq M_\mu$. Si può così stimare la complessità di una chiamata alla *Process* in $O(M_\mu + M_\nu \cdot B_\nu)$, da cui la stima di $O(M_\mu + M_\nu \cdot B_\nu + \operatorname{card} I \cdot \operatorname{card} T)$ per una singola iterazione della procedura principale, che si riduce a $O(M_\mu + M_\nu \cdot B_\nu + \operatorname{card} I)$ sotto decongestione o una delle altre ipotesi equivalenti. In quest'ultimo caso, la soglia minima del passo di discretizzazione che garantisce la condizione di tempo reale è di $\epsilon_m = O(M_\mu + M_\nu \cdot B_\nu + \operatorname{card} I) \cdot \operatorname{card} E$.

I fattori M_μ, M_ν e B_ν dipendono dall'estensione degli intorni $\upsilon(r)$ in termini di cardinalità o di proiezione sull'asse dei tempi e nei casi di interesse sono limitati dalla struttura locale della funzione inferenza secondo $M_\mu, M_\nu \ll \operatorname{card} E$ e $B_\nu \ll t_M - t_m \simeq \operatorname{card} T$. Da un punto di vista ingegneristico, questo rende praticabile una estensione dell'insieme di rappresentazione proporzionale al processo tecnologico, al contrario di ciò che avviene con altri algoritmi computazionalmente più onerosi.

Si fa infine osservare che, facendo solo ipotesi sulla complessità computazionale di f, g e μ^* ed assumendo prudentemente $M_\mu = M_\nu = \operatorname{card} Z_D$, $B_\nu = \operatorname{card} T$ e $\operatorname{card} I \cdot \operatorname{card} \prod_T(\Delta) = \operatorname{card} E \cdot \operatorname{card} T$, si ottiene la stima di $O(\operatorname{card} Z_D \cdot \operatorname{card} T)$ per un singolo passo del ciclo principale, mentre la condizione di tempo reale determina $\epsilon_m = O(\operatorname{card} Z_D \cdot \operatorname{card} T) \cdot \operatorname{card} E = O((\operatorname{card} Z_D)^2)$. La soglia minima del passo di discretizzazione che assicura il tempo reale nella produzione delle uscite è quindi lineare nella maggior parte dei casi ragionevoli e comunque nella peggiore delle ipotesi quadratica rispetto alla cardinalità del dominio di rappresentazione, pur di limitare la complessità di f, g e μ^* come descritto.

Per completare la formalizzazione si introduce una tupla costante K di *conoscenza strutturale* sul sistema che è data a priori ed individua una particolare funzione di inferenza locale λ all'interno di un più vasto insieme di funzioni locali. Si può pensare che la funzione λ sia parametrizzata in K e che Z_D contenga solo la conoscenza soggetta a variare con le possibili storie. Questo ha lo scopo di mantenere Z_D quanto più piccolo possibile, in quanto la sua cardinalità è fattore limitante per la frequenza operativa di un qualunque esecutore di specifiche. Nessuna ipotesi è fatta sulle componenti di K ma intuitivamente queste saranno in qualche modo legate a $Z_D = E \times T$. Data la specifica logica di un sistema ed un orizzonte temporale, la configurazione di un generico esecutore di specifiche capace di riprodurre il comportamento dinamico del sistema sarà individuata da una sestupla $(K, E, I, O, T, \Omega_0)$.

Capitolo 3

Rappresentazione di specifiche logiche

3.1 Logica BTL

Per applicare i risultati ottenuti nel precedente capitolo è necessario formalizzare nel dettaglio la struttura dell'insieme di rappresentazione. Dato che la conoscenza manipolata dalle macchine trova spesso espressione tramite linguaggi, in questa sezione viene presentato un opportuno linguaggio di specifica logica temporale. Come caso di studio è stata scelta una logica definita per l'occasione, BTL (Basic Temporal Logic), essenziale nei suoi costrutti ma sufficientemente potente da descrivere il comportamento dinamico di sistemi complessi su orizzonte limitato. In seguito verrà mostrato come costrutti di linguaggi più sofisticati possano essere riscritti in BTL o, quando questo non è computazionalmente conveniente, come si possano rappresentare costrutti equivalenti applicando il teorema di sostituzione sul particolare insieme di supporto che verrà scelto per la BTL.

Assegnato un insieme $\mathcal{L}$ di funzioni $\ell : \mathbf{Z} \to \{\bot, \top\}$, si definisce l'insieme dei letterali:

$$\tilde{\mathcal{L}} = \{\tilde{\ell} ::= \ell^i \parallel \neg\ell^i \mid \ell \in \mathcal{L}, i \in \mathbf{N}\}$$

dove l'indice i ha lo scopo di permettere l'esistenza di letterali sintatticamente distinti ma associati alla medesima funzione ℓ. Combinando gli elementi di $\tilde{\mathcal{L}}$ tramite gli operatori di congiunzione ($\wedge$), disgiunzione ($\vee$) e ritardo intero (∂^k) si costruisce l'insieme delle clausole:

$$\mathcal{G} = \{C ::= \tilde{\ell} \parallel A \wedge B \parallel A \vee B \parallel \partial^k A \parallel (A) \mid \tilde{\ell} \in \tilde{\mathcal{L}}, A, B \in \mathcal{G}, k \in \mathbf{Z}\}$$

e si assumono le convenzioni $\ell = \ell^1$, $\neg\ell = \neg\ell^1$ e $\partial A = \partial^1 A$. Come avviene per i letterali, ogni elemento $C \in \mathcal{G}$ rappresenta una funzione del tempo che può essere valutata in un particolare istante $t \in \mathbf{Z}$ e produrre un risultato $C|_t$ vero ($\top$) o falso ($\bot$). A partire da questa interpretazione, si può caratterizzare l'insieme delle formule ben formate:

$$\mathcal{F} = \{F ::= C \,@\, t \parallel \neg C \,@\, t \parallel \bowtie C \mid C \in \mathcal{G}, t \in \mathbf{Z}\}$$

Con $C \,@\, t$ e $\neg C \,@\, t$ si sono rappresentati i due possibili esiti della valutazione della funzione definita da C, cioè rispettivamente $C|_t = \top$ e $C|_t = \bot$, mentre $\bowtie C$ esprime sinteticamente il concetto che C è $\top$ in ogni istante discreto. Quest'ultimo operatore serve a postulare la specifica logica del sistema dinamico assegnato indipendentemente dall'orizzonte di interesse $T \subseteq \mathbf{Z}$, attraverso la clausola C che ne descrive il comportamento. Le prime due formule esprimono *fatti* appartenenti all'insieme di conoscenza disponibile mentre la terza esprime una *regola* che caratterizza la struttura del sistema specificato e concorre all'individuazione di una particolare funzione inferenza.

La semantica degli operatori che compaiono in una formula ben formata può essere così riassunta:

1. $\forall \ell \in \mathcal{L}, i, j \in \mathbf{N}, t \in \mathbf{Z} \;\; \ell^i \,@\, t \iff \ell^j \,@\, t$
2. $\forall \ell \in \mathcal{L}, i, j \in \mathbf{N}, t \in \mathbf{Z} \;\; (\neg \ell^i) \,@\, t \iff \neg(\ell^j) \,@\, t$
3. $\forall A, B \in \mathcal{G}, t \in \mathbf{Z} \;\; (A \wedge B) \,@\, t \iff A \,@\, t \wedge B \,@\, t$
4. $\forall A, B \in \mathcal{G}, t \in \mathbf{Z} \;\; (A \vee B) \,@\, t \iff A \,@\, t \vee B \,@\, t$
5. $\forall A \in \mathcal{G}, k, t \in \mathbf{Z} \;\; (\partial^k A) \,@\, t \iff A \,@(t - k)$
6. $\forall A \in \mathcal{G}, t \in \mathbf{Z} \;\; (A) \,@\, t \iff A \,@\, t$
7. $\forall C \in \mathcal{G} \;\; \bowtie C \iff \forall t \in \mathbf{Z} \;\; C \,@\, t$

Tutte le negazioni che compaiono nelle clausole di $\mathcal{G}$ sono interne ai letterali, ma alla luce di questa semantica è facile verificare come un qualunque operatore negato possa essere riscritto distribuendo la negazione sui suoi argomenti secondo le seguenti regole:

1. $\forall \ell^i \in \tilde{\mathcal{L}} \;\; \neg(\ell^i) \rightarrow \neg \ell^i$
2. $\forall \neg \ell^i \in \tilde{\mathcal{L}} \;\; \neg(\neg \ell^i) \rightarrow \ell^i$
3. $\forall A \in \mathcal{G} \;\; \neg(\neg A) \rightarrow A$
4. $\forall A, B \in \mathcal{G} \;\; \neg(A \wedge B) \rightarrow \neg A \vee \neg B$
5. $\forall A, B \in \mathcal{G} \;\; \neg(A \vee B) \rightarrow \neg A \wedge \neg B$
6. $\forall A \in \mathcal{G}, k \in \mathbf{Z} \;\; \neg(\partial^k A) \rightarrow \partial^k(\neg A)$

Le parentesi tonde hanno lo scopo di modificare la precedenza degli operatori e la loro presenza sarà ignorata nelle successive analisi sintattiche, potendosi applicare all'occorrenza la regola di riscrittura $\forall A \in \mathcal{G} \;\; (A) \rightarrow A$. Altri operatori logici di uso comune vengono così riscritti:

- $\forall A, B \in \mathcal{G} \;\; A \Rightarrow B \rightarrow \neg A \vee B$
- $\forall A, B \in \mathcal{G} \;\; A \Longleftrightarrow B \rightarrow (A \Rightarrow B) \wedge (B \Rightarrow A) \rightarrow (\neg A \vee B) \wedge (\neg B \vee A)$

Esempio Un esempio di clausola BTL è $\partial(\neg \ell) \wedge \ell$, con $\ell \in \mathcal{L}$. Essa descrive il passaggio dal valore logico falso al vero della funzione $\ell(t)$. Delle tre formule ben formate che si possono costruire con questa clausola, solo $(\partial(\neg \ell) \wedge \ell) \,@\, t$, che significa che ℓ diviene vera al tempo t, e $\neg(\partial(\neg \ell) \wedge \ell) \,@\, t$, sua negazione, sono soddisfacibili. Una specifica che associa ad una funzione $up_\ell \in \mathcal{L}$ il valore logico istantaneo della precedente clausola è $\forall t \in \mathbf{Z} \;\; up_\ell \,@\, t \iff (\partial(\neg \ell) \wedge \ell) \,@\, t$, la cui traduzione in BTL è la formula ben formata $\bowtie((\neg up_\ell \vee (\partial(\neg \ell) \wedge \ell)) \wedge (up_\ell \vee (\partial\, \ell \vee \neg \ell)))$. La figura 3.1 ne mostra l'albero sintattico.

Esempio Nella logica temporale TILCO sono presenti gli operatori $@[x, y]$ e $?[x, y]$, con $x, y \in \mathbf{Z}$, definiti come $A \,@[x, y]|_t \iff \bigwedge_{\tau=x}^{y} A|_{t+\tau}$ e $A \,?[x, y]|_t \iff \bigvee_{\tau=x}^{y} A|_{t+\tau}$. Se $A \in \mathcal{G}$, i termini destri delle due precedenti implicazioni si traducono in BTL con le formule ben formate $(\bigwedge_{\tau=x}^{y} \partial^{-\tau} A) \,@\, t$ e $(\bigvee_{\tau=x}^{y} \partial^{-\tau} A) \,@\, t$. Per associare alla funzione $at_{a,[x,y]} \in \mathcal{L}$ il valore istantaneo di $a \,@[x, y]$, con $a \in \mathcal{L}$, si può scrivere la specifica BTL equivalente alla sua definizione come $\bowtie((\neg at_{a,[x,y]} \vee (\bigwedge_{\tau=x}^{y} \partial^{-\tau} a)) \wedge (at_{a,[x,y]} \vee (\bigvee_{\tau=x}^{y} \partial^{-\tau}(\neg a))))$. Analogamente, la specifica $\bowtie((\neg qm_{a,[x,y]} \vee (\bigvee_{\tau=x}^{y} \partial^{-\tau} a)) \wedge (qm_{a,[x,y]} \vee (\bigwedge_{\tau=x}^{y} \partial^{-\tau}(\neg a))))$ descrive il comportamento dell'operatore $a \,?[x, y]$ attraverso la funzione $qm_{a,[x,y]} \in \mathcal{L}$.

Esempio In numerose logiche temporali sono presenti gli operatori until e since, che ammettono una definizione ricorsiva nel tempo secondo $\text{until}(A, B)|_t \iff A|_t \vee (B|_t \wedge \text{until}(A, B)|_{t+1})$ e $\text{since}(A, B)|_t \iff A|_t \vee (B|_t \wedge \text{since}(A, B)|_{t-1})$. Rappresentando il valore istantaneo di $\text{until}(A, B)$ attraverso la funzione $until_{A,B} \in \mathcal{L}$ ed assumendo $A, B \in \mathcal{G}$, il termine destro della prima delle

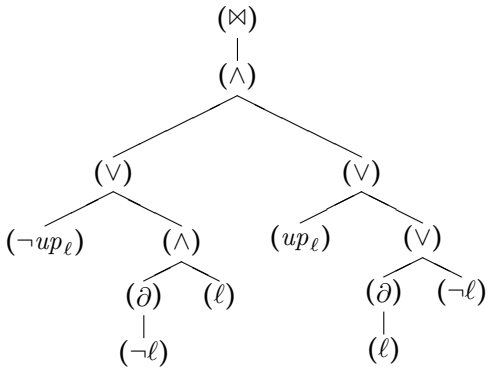

Figura 3.1. Albero sintattico di $\bowtie((\neg up_\ell \vee (\partial(\neg\ell) \wedge \ell)) \wedge (up_\ell \vee (\partial\,\ell \vee \neg\ell)))$

precedenti implicazioni si traduce in BTL con la formula ben formata $(A \vee (B \wedge \partial^{-1}\, until_{A,B}))\,@\,t$. Una specifica BTL equivalente alla definizione di $until_{a,b}$, con $a, b \in \mathcal{L}$, è $\bowtie((\neg until_{a,b} \vee (a \vee (b \wedge \partial^{-1}\, until_{a,b}))) \wedge (until_{a,b} \vee (\neg a \wedge (\neg b \vee \partial^{-1}(\neg until_{a,b})))))$, il cui albero sintattico è riportato in figura 3.2. Analoghe considerazioni valgono per l'operatore since, pur di sostituire le occorrenze di ∂^{-1} con ∂^1.

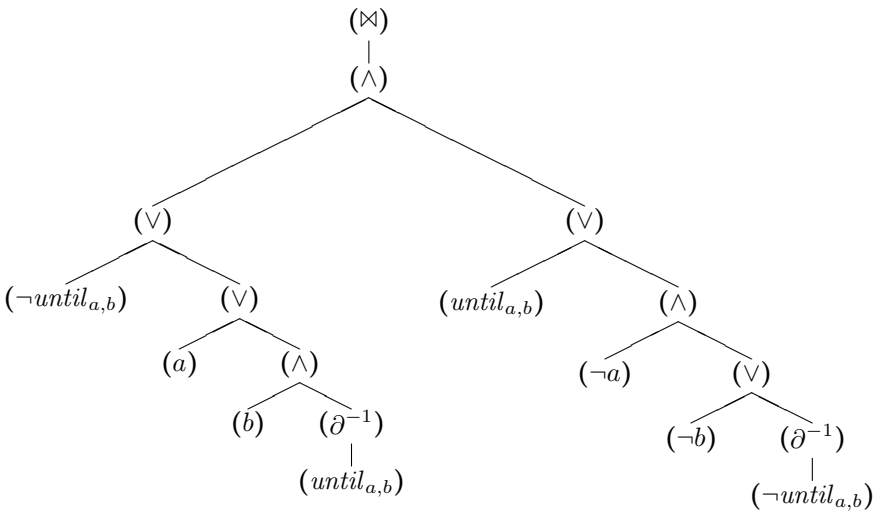

Figura 3.2. Albero sintattico di definizione della funzione $until_{a,b}$

Scelto un qualunque simbolo $\star \notin \mathcal{L}$ che non può comparire in alcun elemento di $\mathcal{G}$, si definiscono $\tilde{\mathcal{L}}^\star = \tilde{\mathcal{L}} \cup \{\star\}$ e:

$$\mathcal{G}^\star = \{C ::= \tilde{\ell} \parallel A \wedge B \parallel A \vee B \parallel \partial^k A \parallel (A) \mid \tilde{\ell} \in \tilde{\mathcal{L}}^\star, A, B \in \mathcal{G}^\star, k \in \mathbf{Z}\}$$

Al simbolo $\star$ ed alle formule che lo contengono non viene associata alcuna semantica ma sono introdotti per scopi esclusivamente sintattici. Dati $C, C', G \in \mathcal{G}^\star$, si indica con $[G]_{C \to C'} \in \mathcal{G}^\star$ l'operatore sintattico che sostituisce la sottoespressione C' ad ogni occorrenza della sottoespressione C in G. La sua definizione formale è riassunta nelle seguenti regole:

1. $\forall\, C, C' \in \mathcal{G}^\star \;\; [C]_{C \to C'} = C'$

2. $\forall\, C, C' \in \mathcal{G}^\star, \tilde{\ell} \in \tilde{\mathcal{L}}^\star \;\; C \neq \tilde{\ell} \Rightarrow [\tilde{\ell}]_{C \to C'} = \tilde{\ell}$
3. $\forall\, A, B, C, C' \in \mathcal{G}^\star \;\; C \neq (A \wedge B) \Rightarrow [A \wedge B]_{C \to C'} = ([A]_{C \to C'} \wedge [B]_{C \to C'})$
4. $\forall\, A, B, C, C' \in \mathcal{G}^\star \;\; C \neq (A \vee B) \Rightarrow [A \vee B]_{C \to C'} = ([A]_{C \to C'} \vee [B]_{C \to C'})$
5. $\forall\, A, C, C' \in \mathcal{G}^\star, k \in \mathbf{Z} \;\; C \neq \partial^k A \Rightarrow [\partial^k A]_{C \to C'} = \partial^k [A]_{C \to C'}$

L'utilizzo del simbolo $\star$ viene evidenziato nella scrittura $[G]_{C \to \star} = G$, con $C, G \in \mathcal{G}$, vera quando la sottoespressione C non compare in G, o $[G]_{C \to \star} \neq G$ quando vi compare almeno una volta. Se poi tutte le sottoespressioni di G sono distinte, scrivere che $[G]_{C \to \star} \neq G$ equivale ad affermare che C compare in G una sola volta.

Si osserva inoltre che affermare che tutte le sottoespressioni di G sono distinte equivale ad affermare che tutti i letterali che compaiono in G sono distinti. Se tutte le sottoespressioni sono distinte, infatti, anche i letterali lo sono in quanto particolari sottoespressioni mentre due sottoespressioni possono coincidere solo a condizione che anche i letterali che vi compaiono coincidano.

Assegnata $G \in \mathcal{G}$, se $\forall\, \tilde{\ell}, \tilde{\ell}' \in \tilde{\mathcal{L}}, G' \in \mathcal{G} \;\; [G']_{\tilde{\ell}' \to \tilde{\ell}} = G \Rightarrow [G]_{\tilde{\ell} \to \tilde{\ell}'} = G'$ si può affermare che tutti i letterali che compaiono in G sono distinti. In caso contrario, infatti, se esistesse un letterale $\tilde{\ell}$ presente in G in più di una occorrenza si potrebbe costruire a partire da G una clausola H in cui una singola occorrenza di $\tilde{\ell}$ viene sostituita da un letterale $\tilde{\ell}'$ soddisfacente $[G]_{\tilde{\ell}' \to \star} = G$, lasciando tutte le altre occorrenze di $\tilde{\ell}$ presenti in H. La H prescelta verifica $[H]_{\tilde{\ell}' \to \tilde{\ell}} = G$ perché $[G]_{\tilde{\ell}' \to \star} = G$, ma non $[G]_{\tilde{\ell} \to \tilde{\ell}'} = H$. Infatti $[G]_{\tilde{\ell} \to \star} \neq G, [G]_{\tilde{\ell}' \to \star} = G \Rightarrow \tilde{\ell} \neq \tilde{\ell}'$ e chiamando $G' = [G]_{\tilde{\ell} \to \tilde{\ell}'}$, si ha $[G']_{\tilde{\ell} \to \star} = G'$ mentre $[H]_{\tilde{\ell} \to \star} \neq H$ per costruzione e dunque $G' \neq H$.

Gli insiemi di clausole e formule ben formate contenenti solo sottoespressioni distinte sono dunque così definiti:

$$\dot{\mathcal{G}} = \{G \in \mathcal{G} \mid \forall\, \tilde{\ell}, \tilde{\ell}' \in \tilde{\mathcal{L}}, G' \in \mathcal{G} \;\; [G']_{\tilde{\ell}' \to \tilde{\ell}} = G \Rightarrow [G]_{\tilde{\ell} \to \tilde{\ell}'} = G'\}$$

$$\dot{\mathcal{F}} = \{F ::= C \mathbin{@} t \parallel \neg C \mathbin{@} t \parallel \bowtie C \mid C \in \dot{\mathcal{G}}, t \in \mathbf{Z}\}$$

Visto che $\dot{\mathcal{G}} \subseteq \mathcal{G}$ e $\dot{\mathcal{F}} \subseteq \mathcal{F}$, gli elementi degli insiemi appena introdotti godranno di tutte le proprietà di cui godono gli elementi di $\mathcal{G}$ e $\mathcal{F}$. Inoltre, sia la specifica logica del sistema che i fatti rappresentanti la sua evoluzione dinamica saranno scelti tra gli elementi di $\dot{\mathcal{F}}$ con semantica appropriata. Questo accorgimento è una necessità tecnica che garantisce l'unicità della rappresentazione tramite la funzione ρ che verrà definita nel seguito e non limita la generalità espressiva.

Infatti, dato che le regole di riscrittura $[G]_{\ell^i \to \ell^j}$ e $[G]_{\neg \ell^i \to \neg \ell^j}$ non modificano la funzione individuata da G, si possono applicare al fine di rendere tutti i letterali che compaiono in G distinti. Data una qualunque $G \in \mathcal{G}$, finché esistono $\tilde{\ell} = \ell^i, \tilde{\ell}' = \ell^j \in \tilde{\mathcal{L}}$ oppure $\tilde{\ell} = \neg \ell^i, \tilde{\ell}' = \neg \ell^j \in \tilde{\mathcal{L}}$ e $G' \in \mathcal{G}$ tali che $G \neq G'$, $[G']_{\tilde{\ell}' \to \tilde{\ell}} = G$ e $[G]_{\tilde{\ell} \to \tilde{\ell}'} \neq G'$ si può applicare la regola di riscrittura $G \to G'$, producendo alla fine una clausola semanticamente equivalente ma contenente solo letterali distinti.

Assegnata specifica logica $\bowtie G \in \dot{\mathcal{F}}$ e l'orizzonte d'interesse $T \subseteq \mathbf{Z}$, si definisce l'insieme delle formule $D \subseteq \dot{\mathcal{F}}$ prescelte per descrivere l'evoluzione del sistema specificato come:

$$D = \{C \mathbin{@} t, \neg C \mathbin{@} t \in \dot{\mathcal{F}} \mid [G]_{C \to \star} \neq G, t \in T\}$$

In altre parole, il dominio della funzione di rappresentazione ρ è costituito dal prodotto cartesiano dell'insieme delle sottoespressioni di G, affermate e negate, con l'orizzonte di interesse. Una volta definita ρ ed introdotta una opportuna funzione inferenza ϕ su $\rho(D)$ si potranno applicare tutti i risultati ottenuti nel precedente capitolo ed in particolare si arriverà ad implementare un efficiente algoritmo di chiusura progressiva.

La conoscenza a priori che verrà fornita a questo algoritmo sarà ristretta senza perdita di generalità alla rappresentazione di fatti della forma $C \mathbin{@} t$, ed in particolare la specifica $\bowtie G \in \dot{\mathcal{F}}$ sarà tradotta nell'insieme di fatti:

$$\mathcal{H}_G = \{G \mathbin{@} t \in D\}$$

Analogamente, la storia di ogni funzione di ingresso $in \in \mathcal{L}$ sarà descritta da un insieme di fatti:

$$\mathcal{H}_{in} = \{in^1 \,@\, t, (\neg in^1) \,@\, \bar{t} \in D \mid in|_t = \top, in|_{\bar{t}} = \bot\}$$

Quest'ultima definizione privilegia la forma $(\neg in) \,@\, t$ a quella $\neg(in) \,@\, t$, semanticamente equivalente, ed assieme alla precedente definizione consente di escludere la presenza di fatti della forma $\neg C \,@\, t$ dall'insieme di conoscenza nota a priori.

Per ragioni che risulteranno più chiare nella prossima sezione, ad ogni funzione di uscita $out \in \mathcal{L}$ sarà invece associato un insieme di fatti della forma $\neg C \,@\, t$:

$$\mathcal{K}_{out} = \{\neg(\neg out^1) \,@\, t, \neg(out^1) \,@\, \bar{t} \in D \mid out|_t = \top, out|_{\bar{t}} = \bot\}$$

La definizione di questi insiemi di conoscenza a priori vale solo come prima approssimazione, la soluzione effettivamente adottata è più sofisticata e verrà discussa nel seguito. Prima di procedere nella definizione di ρ e ϕ è però opportuno esaminare alcune possibili regole d'inferenza sugli elementi di D.

3.2 Inferenza su alberi sintattici

Si possono scrivere numerose regole di inferenza che operano sul dominio D e risultano corrette secondo la semantica presentata. Nel seguente elenco di produzioni della forma $a_1, \ldots, a_n \vdash b_1, \ldots, b_m$ è sottinteso che queste valgono $\forall\, a_1, \ldots, a_n, b_1, \ldots, b_m \in D$:

1. Letterali:
   (a) $\ell^i \,@\, t \vdash \neg(\neg\ell^j) \,@\, t$
   (b) $\neg(\ell^i) \,@\, t \vdash (\neg\ell^j) \,@\, t$
   (c) $(\neg\ell^i) \,@\, t \vdash \neg(\ell^j) \,@\, t$
   (d) $\neg(\neg\ell^i) \,@\, t \vdash \ell^j \,@\, t$
2. Congiunzioni:
   (a) $(A \wedge B) \,@\, t \vdash A \,@\, t, B \,@\, t$
   (b) $\neg(A \wedge B) \,@\, t, A \,@\, t \vdash \neg B \,@\, t$
   (c) $\neg(A \wedge B) \,@\, t, B \,@\, t \vdash \neg A \,@\, t$
   (d) $\neg A \,@\, t \vdash \neg(A \wedge B) \,@\, t$
   (e) $\neg B \,@\, t \vdash \neg(A \wedge B) \,@\, t$
   (f) $A \,@\, t, B \,@\, t \vdash (A \wedge B) \,@\, t$
3. Disgiunzioni:
   (a) $\neg(A \vee B) \,@\, t \vdash \neg A \,@\, t, \neg B \,@\, t$
   (b) $(A \vee B) \,@\, t, \neg A \,@\, t \vdash B \,@\, t$
   (c) $(A \vee B) \,@\, t, \neg B \,@\, t \vdash A \,@\, t$
   (d) $A \,@\, t \vdash (A \vee B) \,@\, t$
   (e) $B \,@\, t \vdash (A \vee B) \,@\, t$
   (f) $\neg A \,@\, t, \neg B \,@\, t \vdash \neg(A \vee B) \,@\, t$
4. Ritardi:
   (a) $(\partial^k A) \,@\, t \vdash A \,@(t - k)$
   (b) $\neg(\partial^k A) \,@\, t \vdash \neg A \,@(t - k)$

(c) $A \mathbin{@} t \vdash (\partial^k A) \mathbin{@} (t+k)$

(d) $\neg A \mathbin{@} t \vdash \neg(\partial^k A) \mathbin{@} (t+k)$

Le seguenti regole di inferenza su letterali non compaiono nel precedente elenco in quanto scrivibili come combinazioni delle regole sopra elencate. La sequenza di produzioni equivalenti ad ognuna di esse è racchiusa tra $\{\cdots\}$:

1. $\ell^i \mathbin{@} t \vdash \ell^j \mathbin{@} t \quad \{\ell^i \mathbin{@} t \vdash \neg(\neg\ell^k) \mathbin{@} t \vdash \ell^j \mathbin{@} t\}$
2. $(\neg\ell^i) \mathbin{@} t \vdash (\neg\ell^j) \mathbin{@} t \quad \{(\neg\ell^i) \mathbin{@} t \vdash \neg(\ell^k) \mathbin{@} t \vdash (\neg\ell^j) \mathbin{@} t\}$
3. $\neg(\ell^i) \mathbin{@} t \vdash \neg(\ell^j) \mathbin{@} t \quad \{\neg(\ell^i) \mathbin{@} t \vdash (\neg\ell^k) \mathbin{@} t \vdash \neg(\ell^j) \mathbin{@} t\}$
4. $\neg(\neg\ell^i) \mathbin{@} t \vdash \neg(\neg\ell^j) \mathbin{@} t \quad \{\neg(\neg\ell^i) \mathbin{@} t \vdash \ell^k \mathbin{@} t \vdash \neg(\neg\ell^j) \mathbin{@} t\}$

Le regole proposte non sono complete, nel senso che per ogni insieme di conoscenza non vuoto esistono formule ben formate che ne sono conseguenza logica ma non appartengono alla sua chiusura rispetto a queste regole.

Controesempi La semantica definita garantisce che $\forall\, t \in T \;\; (\ell^i \vee \ell^j) \mathbin{@} t \iff \ell^i \mathbin{@} t$, ma la regola $(\ell^i \vee \ell^j) \mathbin{@} t \vdash \ell^i \mathbin{@} t$ non è scrivibile come combinazione delle precedenti. Analoga osservazione può valere per $\forall\, t \in T \;\; (\partial^{-k}(\partial^k A)) \mathbin{@} t \iff A \mathbin{@} t$, infatti i passi necessari a realizzare la produzione $(\partial^{-k}(\partial^k A)) \mathbin{@} t \vdash (\partial^k A) \mathbin{@} (t+k) \vdash A \mathbin{@} t$ sono applicabili solo a condizione che $(\partial^k A) \mathbin{@} (t+k) \in D$, cioè $t + k \in T$. In pratica, questi semplici controesempi verranno neutralizzati tramite opportune semplificazioni ma questo non potrà essere fatto nella generalità dei casi.

Di seguito viene discusso come ridurre il numero di queste regole senza perdere in conoscenza utile inferibile. Dato che esse sono incomplete e frutto di scelta arbitraria, questa garanzia non è particolarmente rilevante sul piano formale ma risulta psicologicamente assai rassicurante. Nel percorrere questa strada si adotterà quindi uno stile più discorsivo, per tornare al necessario rigore al momento della definizione dell'insieme di rappresentazione. Niente avrebbe impedito di scegliere, al posto delle precedenti, le regole ridotte elencate alla fine della sezione.

Ricordando che il risultato della valutazione dell'espressione $C \in \mathcal{G}$ all'istante $t \in T$ è rappresentato da uno dei due fatti $C \mathbin{@} t$ o $\neg C \mathbin{@} t$, corrispondenti ai valori logici istantanei $\top$ o $\bot$, queste regole possono essere classificate come:

- *Ascendenti* se consentono di inferire il valore di una espressione a partire da quello di una o entrambe le sottoespressioni che la compongono
- *Discendenti* se consentono di inferire il valore di una sottoespressione a partire da quello dell'espressione che la contiene, unito eventualmente al valore dell'altra sottoespressione da cui la precedente espressione è composta
- *Trasversali* se consentono di inferire il valore di un letterale a partire da quello di un altro letterale

A partire da questa classificazione, si osserva che:

- In tutte le regole ascendenti il valore inferito per l'espressione coincide con quello delle sue sottoespressioni di cui è necessaria la conoscenza
- In tutte le regole discendenti il valore inferito per la sottoespressione coincide con quello dell'espressione che la contiene
- In tutte le regole discendenti che richiedono la conoscenza del valore di altre sottoespressioni, questo è la negazione di quello inferito
- In tutte le regole trasversali il valore inferito per un letterale è la negazione di quello dell'altro letterale coinvolto

Queste osservazioni consentono di assegnare un tipo alle regole di inferenza, che può essere $\uparrow \top$ o $\uparrow \bot$ per le regole ascendenti che producono rispettivamente valori logici $\top$ o $\bot$, $\downarrow \top$ o $\downarrow \bot$ per le corrispondenti regole discendenti e $\hookrightarrow \top$ o $\hookrightarrow \bot$ per quelle trasversali, anch'esse suddivise in base al valore prodotto. In tabella 3.1 sono riportati tutti i casi in cui le precedenti regole possono essere applicate ed il loro tipo.

$\ell^i \,@\, t$	$(\neg \ell^j) \,@\, t$		Tipo
$\top$	$\cdots \leftarrow \bot$		$\hookrightarrow \bot$
$\bot$	$\cdots \leftarrow \top$		$\hookrightarrow \top$
$\cdots \leftarrow \bot$	$\top$		$\hookrightarrow \bot$
$\cdots \leftarrow \top$	$\bot$		$\hookrightarrow \top$
$A \,@\, t$	$B \,@\, t$	$(A \wedge B) \,@\, t$	Tipo
$\cdots \leftarrow \top$	$\cdots \leftarrow \top$	$\top$	$\downarrow \top$
$\top$	$\cdots \leftarrow \bot$	$\bot$	$\downarrow \bot$
$\cdots \leftarrow \bot$	$\top$	$\bot$	$\downarrow \bot$
$\bot$	$\cdots$	$\cdots \leftarrow \bot$	$\uparrow \bot$
$\cdots$	$\bot$	$\cdots \leftarrow \bot$	$\uparrow \bot$
$\top$	$\top$	$\cdots \leftarrow \top$	$\uparrow \top$
$A \,@\, t$	$B \,@\, t$	$(A \vee B) \,@\, t$	Tipo
$\cdots \leftarrow \bot$	$\cdots \leftarrow \bot$	$\bot$	$\downarrow \bot$
$\bot$	$\cdots \leftarrow \top$	$\top$	$\downarrow \top$
$\cdots \leftarrow \top$	$\bot$	$\top$	$\downarrow \top$
$\top$	$\cdots$	$\cdots \leftarrow \top$	$\uparrow \top$
$\cdots$	$\top$	$\cdots \leftarrow \top$	$\uparrow \top$
$\bot$	$\bot$	$\cdots \leftarrow \bot$	$\uparrow \bot$
$A \,@\, (t - k)$		$(\partial^k A) \,@\, t$	Tipo
$\cdots \leftarrow \top$		$\top$	$\downarrow \top$
$\cdots \leftarrow \bot$		$\bot$	$\downarrow \bot$
$\top$		$\cdots \leftarrow \top$	$\uparrow \top$
$\bot$		$\cdots \leftarrow \bot$	$\uparrow \bot$

Tabella 3.1. Regole d'inferenza per la BTL

Confrontando le varie possibilità si osserva che se il valore logico di un'espressione è stato inferito attraverso una regola ascendente, l'applicazione alla stessa espressione di una regola discendente non può produrre nuova conoscenza. Ad esempio, per l'operatore $\wedge$ le due regole $\uparrow \bot$ richiedono la conoscenza di una sottoespressione $\bot$ mentre le regole $\downarrow \bot$, uniche discendenti applicabili al risultato delle prime, richiedono una sottoespressione $\top$. Per potere applicare una regola $\downarrow \bot$ al risultato di una $\uparrow \bot$ è necessario che entrambe le sottoespressioni rette da $\wedge$ siano note a valori logici opposti e niente di nuovo può quindi essere inferito. Per quanto riguarda la regola $\uparrow \top$, questa presuppone da sola la conoscenza di entrambe le sottoespressioni e per essa vale la precedente considerazione. Casi duali si verificano per l'operatore $\vee$, mentre per ∂^k la presenza

di una corrispondenza biunivoca tra i valori dell'espressione e quella della sottoespressione che ne costituisce l'unico argomento verifica banalmente la tesi.

Considerando l'insieme delle espressioni note $\perp$ in un qualche istante, si può dimostrare che i suoi elementi non sono stati determinati attraverso regole di tipo $\downarrow \perp$. Si osserva innanzitutto che queste regole possono essere applicate solo a partire da espressioni $\perp$ ma nessuna espressione è nota $\perp$ a priori per la struttura degli insiemi di conoscenza $\mathcal{H}_G$ e $\mathcal{H}_{in}$, composti rispettivamente da fatti della forma $G \,@\, t$ e $\tilde{\ell} \,@\, t$. Per dimostrare la tesi ad un passo d'inferenza generico la si supponga vera a quello precedente, cioè si assuma che tutte le espressioni $\perp$ determinate fino a quel passo siano state prodotte attraverso regole di tipo $\uparrow \perp$ e $\hookrightarrow \perp$. Per quanto precedentemente dimostrato, l'applicazione di una regola $\downarrow \perp$ al risultato di una $\uparrow \perp$ non produce nuova conoscenza, mentre la sua applicazione al risultato di una $\hookrightarrow \perp$ è resa impossibile dal fatto che le regole traversali operano solo tra letterali ed a questi non può essere applicata alcuna regola discendente.

In queste ipotesi, le regole di tipo $\downarrow \perp$ possono essere rimosse senza ridurre l'insieme di conoscenza inferibile con il seguente risultato:

1. Letterali:
   (a) $\ell^i \,@\, t \vdash \neg(\neg\ell^j) \,@\, t$
   (b) $\neg(\ell^i) \,@\, t \vdash (\neg\ell^j) \,@\, t$
   (c) $(\neg\ell^i) \,@\, t \vdash \neg(\ell^j) \,@\, t$
   (d) $\neg(\neg\ell^i) \,@\, t \vdash \ell^j \,@\, t$
2. Congiunzioni:
   (a) $(A \wedge B) \,@\, t \vdash A \,@\, t, B \,@\, t$
   (b) $\neg A \,@\, t \vdash \neg(A \wedge B) \,@\, t$
   (c) $\neg B \,@\, t \vdash \neg(A \wedge B) \,@\, t$
   (d) $A \,@\, t, B \,@\, t \vdash (A \wedge B) \,@\, t$
3. Disgiunzioni:
   (a) $(A \vee B) \,@\, t, \neg A \,@\, t \vdash B \,@\, t$
   (b) $(A \vee B) \,@\, t, \neg B \,@\, t \vdash A \,@\, t$
   (c) $A \,@\, t \vdash (A \vee B) \,@\, t$
   (d) $B \,@\, t \vdash (A \vee B) \,@\, t$
   (e) $\neg A \,@\, t, \neg B \,@\, t \vdash \neg(A \vee B) \,@\, t$
4. Ritardi:
   (a) $(\partial^k A) \,@\, t \vdash A \,@(t - k)$
   (b) $A \,@\, t \vdash (\partial^k A) \,@(t + k)$
   (c) $\neg A \,@\, t \vdash \neg(\partial^k A) \,@(t + k)$

Se poi l'obiettivo dell'inferenza è la produzione delle storie delle uscite, si possono operare ulteriori semplificazioni. Ricordando che gli insiemi di conoscenza $\mathcal{K}_{out}$ sono composti da fatti della forma $\neg\tilde{\ell} \,@\, t$, si osserva che tutti i loro elementi noti devono essere stati determinati attraverso regole di tipo $\hookrightarrow \perp$. Infatti le regole $\uparrow \perp$ sono ascendenti e non possono determinare il valore logico di un letterale, tutte le altre regole producono valori logici $\top$ e nessun fatto è noto $\perp$ a priori per definizione di $\mathcal{H}_G$ e $\mathcal{H}_{in}$.

Per quanto riguarda le regole di tipo $\uparrow \top$, il risultato della loro applicazione non è utilizzabile per ulteriore inferenza dalle regole $\uparrow \perp$, in cui compaiono solo valori logici $\perp$, nè da quelle $\downarrow \top$, che come precedentemente dimostrato non producono nuova conoscenza se applicate al risultato delle prime, nè da alcuna regola trasversale in quanto queste operano solamente su letterali. Di

$\ell^i \,@\, t$	$(\neg \ell^j) \,@\, t$		Tipo
$\top$	$\cdots \leftarrow \bot$		$\hookrightarrow \bot$
$\bot$	$\cdots \leftarrow \top$		$\hookrightarrow \top$
$\cdots \leftarrow \bot$	$\top$		$\hookrightarrow \bot$
$\cdots \leftarrow \top$	$\bot$		$\hookrightarrow \top$
$A \,@\, t$	$B \,@\, t$	$(A \wedge B) \,@\, t$	Tipo
$\cdots \leftarrow \top$	$\cdots \leftarrow \top$	$\top$	$\downarrow \top$
$\bot$	$\cdots$	$\cdots \leftarrow \bot$	$\uparrow \bot$
$\cdots$	$\bot$	$\cdots \leftarrow \bot$	$\uparrow \bot$
$\top$	$\top$	$\cdots \leftarrow \top$	$\uparrow \top$
$A \,@\, t$	$B \,@\, t$	$(A \vee B) \,@\, t$	Tipo
$\bot$	$\cdots \leftarrow \top$	$\top$	$\downarrow \top$
$\cdots \leftarrow \top$	$\bot$	$\top$	$\downarrow \top$
$\top$	$\cdots$	$\cdots \leftarrow \top$	$\uparrow \top$
$\cdots$	$\top$	$\cdots \leftarrow \top$	$\uparrow \top$
$\bot$	$\bot$	$\cdots \leftarrow \bot$	$\uparrow \bot$
$A \,@\, (t - k)$		$(\partial^k A) \,@\, t$	Tipo
$\cdots \leftarrow \top$		$\top$	$\downarrow \top$
$\top$		$\cdots \leftarrow \top$	$\uparrow \top$
$\bot$		$\cdots \leftarrow \bot$	$\uparrow \bot$

Tabella 3.2. Regole con chiusura equivalente su conoscenza $\top$ a priori

conseguenza, le regole di tipo $\uparrow \top$ non concorrono alla determinazione delle storie delle uscite direttamente o per via transitiva e possono venire rimosse.

Si è già osservato come neppure le regole di tipo $\hookrightarrow \top$ ($\neg\tilde{\ell}_1 \,@\, t \vdash \tilde{\ell}_2 \,@\, t$) producano elementi di $\mathcal{K}_{out}$. Inoltre, il risultato $\tilde{\ell}_2 \,@\, t$ della loro applicazione non è utilizzabile dalle regole $\uparrow \bot$, che richiedono valori logici $\bot$, nè da quelle $\downarrow \top$, in quanto non si possono applicare regole discendenti a valori logici di letterali, ma soltanto dalle $\hookrightarrow \bot$ ($\tilde{\ell}_2 \,@\, t \vdash \neg\tilde{\ell}_3 \,@\, t$). In altre parole, tutte le applicazioni di regole $\hookrightarrow \top$ utili alla determinazione delle storie delle uscite compaiono all'interno di produzioni della forma $\neg\tilde{\ell}_1 \,@\, t \vdash \tilde{\ell}_2 \,@\, t \vdash \neg\tilde{\ell}_3 \,@\, t$. Ricordando poi che tutti i letterali noti $\bot$ in un qualche istante sono stati determinati attraverso regole $\hookrightarrow \bot$, per ogni fatto $\neg\tilde{\ell}_1 \,@\, t$ noto deve esistere un secondo fatto $\tilde{\ell}_0 \,@\, t$ noto, da cui il primo è stato inferito. Applicando a quest'ultimo fatto $\tilde{\ell}_0 \,@\, t$ una regola $\hookrightarrow \bot$ si possono inferire quegli stessi fatti $\neg\tilde{\ell}_3 \,@\, t$ che costituiscono l'unico prodotto d'interesse della combinazione tra regole $\hookrightarrow \top$ e $\hookrightarrow \bot$. Si osserva infatti che le due possibili forme in cui la produzione $\tilde{\ell}_0 \,@\, t \vdash \neg\tilde{\ell}_1 \,@\, t \vdash \tilde{\ell}_2 \,@\, t \vdash \neg\tilde{\ell}_3 \,@\, t$ può essere esplicitata, $\ell^h \,@\, t \vdash \neg(\neg\ell^i) \,@\, t \vdash \ell^j \,@\, t \vdash \neg(\neg\ell^k) \,@\, t$ e $(\neg\ell^h) \,@\, t \vdash \neg(\ell^i) \,@\, t \vdash (\neg\ell^j) \,@\, t \vdash \neg(\ell^k) \,@\, t$, sono rispettivamente semplificabili in $\ell^h \,@\, t \vdash \neg(\neg\ell^k) \,@\, t$ e $(\neg\ell^h) \,@\, t \vdash \neg(\ell^k) \,@\, t$, ovvero $\tilde{\ell}_0 \,@\, t \vdash \neg\tilde{\ell}_3 \,@\, t$. In conclusione, ogni contributo che le regole $\hookrightarrow \top$ possono portare alla determinazione delle storie delle uscite è ottenibile anche attraverso l'applicazione delle sole regole $\hookrightarrow \bot$.

Anche le regole di tipo $\hookrightarrow \top$ possono dunque venire rimosse lasciando le seguenti:

1. Letterali:
   (a) $\ell^i \,@\, t \vdash \neg(\neg\ell^j) \,@\, t$
   (b) $(\neg\ell^i) \,@\, t \vdash \neg(\ell^j) \,@\, t$
2. Congiunzioni:
   (a) $(A \wedge B) \,@\, t \vdash A \,@\, t, B \,@\, t$
   (b) $\neg A \,@\, t \vdash \neg(A \wedge B) \,@\, t$
   (c) $\neg B \,@\, t \vdash \neg(A \wedge B) \,@\, t$
3. Disgiunzioni:
   (a) $(A \vee B) \,@\, t, \neg A \,@\, t \vdash B \,@\, t$
   (b) $(A \vee B) \,@\, t, \neg B \,@\, t \vdash A \,@\, t$
   (c) $\neg A \,@\, t, \neg B \,@\, t \vdash \neg(A \vee B) \,@\, t$
4. Ritardi:
   (a) $(\partial^k A) \,@\, t \vdash A \,@(t - k)$
   (b) $\neg A \,@\, t \vdash \neg(\partial^k A) \,@(t + k)$

Se gli unici fatti noti a priori appartengono agli insiemi $\mathcal{H}_G$ ed $\mathcal{H}_{in}$, queste dieci regole possono sostituire le venti iniziali senza modificare i risultati che si ottengono operando l'intersezione della chiusura rispetto ad esse con gli insiemi $\mathcal{K}_{out}$. Tra le regole rimanenti, il valore logico istantaneo prodotto durante l'inferenza è funzione della direzione di propagazione e si parlerà quindi solo di regole ascendenti ($\uparrow$), discendenti ($\downarrow$) o trasversali ($\hookrightarrow$), sottintendendo gli opportuni valori $\top$ o $\bot$. Questa proprietà, più che la riduzione del numero delle regole, consente la scrittura di una funzione inferenza locale molto compatta.

3.3 Reti di inferenza temporale

Considerando l'albero sintattico di $\bowtie G \in \dot{\mathcal{F}}$ e l'insieme dei suoi nodi V, si osserva che ad ogni sottoespressione C di G corrisponde un unico sottoalbero, dato che tutte le sottoespressioni di G sono distinte per definizione di $\dot{\mathcal{F}}$, e quindi un unico nodo $v_C \in V$ che ne costituisce la radice.

$\ell^i \,@\, t$	$(\neg \ell^j) \,@\, t$		Tipo
$\top$	$\cdots \leftarrow \bot$		$\hookrightarrow \bot$
$\cdots \leftarrow \bot$	$\top$		$\hookrightarrow \bot$
$A \,@\, t$	$B \,@\, t$	$(A \wedge B) \,@\, t$	Tipo
$\cdots \leftarrow \top$	$\cdots \leftarrow \top$	$\top$	$\downarrow \top$
$\bot$	$\cdots$	$\cdots \leftarrow \bot$	$\uparrow \bot$
$\cdots$	$\bot$	$\cdots \leftarrow \bot$	$\uparrow \bot$
$A \,@\, t$	$B \,@\, t$	$(A \vee B) \,@\, t$	Tipo
$\bot$	$\cdots \leftarrow \top$	$\top$	$\downarrow \top$
$\cdots \leftarrow \top$	$\bot$	$\top$	$\downarrow \top$
$\bot$	$\bot$	$\cdots \leftarrow \bot$	$\uparrow \bot$
$A \,@\, (t-k)$		$(\partial^k A) \,@\, t$	Tipo
$\cdots \leftarrow \top$		$\top$	$\downarrow \top$
$\bot$		$\cdots \leftarrow \bot$	$\uparrow \bot$

Tabella 3.3. Regole utili alla determinazione delle storie delle uscite

All'albero sintattico è possibile associare un grafo orientato (V, E), con $E \subseteq V \times V \mid \forall (v_i, v_j) \in E \;\; (v_j, v_i) \in E$, costruito in modo da far corrispondere ad ogni ramo dell'albero una coppia di archi opposti attraverso cui rappresentare il valore logico istantaneo delle sottoespressioni di G.

Infatti, chiamando $v_{\bowtie}$ la radice dell'albero sintattico e $\beta : V \setminus \{v_{\bowtie}\} \to V$ la funzione che associa ad ogni nodo dell'albero il suo genitore, si può definire la funzione di rappresentazione $\rho : D \to Z_D = E \times T$ come:

$$\forall\, C \,@\, t \in D \;\; \rho(C \,@\, t) = (\beta(v_C), v_C) \,@\, t$$

$$\forall\, \neg C \,@\, t \in D \;\; \rho(\neg C \,@\, t) = (v_C, \beta(v_C)) \,@\, t$$

con $v_C \neq v_{\bowtie}$ radice del sottoalbero sintattico a cui corrisponde la sottoespressione C di G. Conformemente a questa rappresentazione, si definisce la negazione su Z_D come:

$$\forall (v_i, v_j) \,@\, t \in Z_D \;\; \neg (v_i, v_j) \,@\, t = (v_j, v_i) \,@\, t \in Z_D$$

Si introduce infine la funzione $\alpha : V \to \{\bowtie, \wedge, \vee\} \cup \{\partial^k \mid k \in \mathbf{Z}\} \cup \tilde{\mathcal{L}}$ che associa ad ogni nodo dell'albero la sua classificazione sintattica come operatore o letterale.

Una definizione più formale di queste strutture di rappresentazione può essere data in forma ricorsiva. Immaginando di disporre di un insieme di nodi W sufficientemente ampio e di una qualunque funzione $v : \dot{\mathcal{G}} \cup \{\bowtie\} \to W$ soddisfacente $\forall\, A, B \in \dot{\mathcal{G}} \cup \{\bowtie\} \;\; A = B \iff v(A) = v(B)$, si definisce la radice del sottoalbero associato a C come $v_C = v(C)$. Ricordando che le funzioni monodimensionali α e β possono essere pensate come insiemi di coppie, si può allora scrivere:

1. $\forall\, \tilde{\ell} \in \tilde{\mathcal{L}}$:
   (a) $V_{\tilde{\ell}} = \{v_{\tilde{\ell}}\}$
   (b) $E_{\tilde{\ell}} = \emptyset$
   (c) $\alpha_{\tilde{\ell}} = \{(v_{\tilde{\ell}}, \tilde{\ell})\}$

   (d) $\beta_{\tilde{\ell}} = \emptyset$

2. $\forall (A \wedge B) \in \dot{\mathcal{G}}, \pi_A = (v_A, v_{A\wedge B}), \pi_B = (v_B, v_{A\wedge B})$:
   (a) $V_{A\wedge B} = V_A \cup V_B \cup \{v_{A\wedge B}\}$
   (b) $E_{A\wedge B} = E_A \cup E_B \cup \{\pi_A, \neg\pi_A, \pi_B, \neg\pi_B\}$
   (c) $\alpha_{A\wedge B} = \alpha_A \cup \alpha_B \cup \{(v_{A\wedge B}, \wedge)\}$
   (d) $\beta_{A\wedge B} = \beta_A \cup \beta_B \cup \{\pi_A, \pi_B\}$
3. $\forall (A \vee B) \in \dot{\mathcal{G}}, \pi_A = (v_A, v_{A\vee B}), \pi_B = (v_B, v_{A\vee B})$:
   (a) $V_{A\vee B} = V_A \cup V_B \cup \{v_{A\vee B}\}$
   (b) $E_{A\vee B} = E_A \cup E_B \cup \{\pi_A, \neg\pi_A, \pi_B, \neg\pi_B\}$
   (c) $\alpha_{A\vee B} = \alpha_A \cup \alpha_B \cup \{(v_{A\vee B}, \vee)\}$
   (d) $\beta_{A\vee B} = \beta_A \cup \beta_B \cup \{\pi_A, \pi_B\}$
4. $\forall\, \partial^k A \in \dot{\mathcal{G}}, \pi_A = (v_A, v_{\partial^k A})$:
   (a) $V_{\partial^k A} = V_A \cup \{v_{\partial^k A}\}$
   (b) $E_{\partial^k A} = E_A \cup \{\pi_A, \neg\pi_A\}$
   (c) $\alpha_{\partial^k A} = \alpha_A \cup \{(v_{\partial^k A}, \partial^k)\}$
   (d) $\beta_{\partial^k A} = \beta_A \cup \{\pi_A\}$
5. $\forall \bowtie G \in \dot{\mathcal{F}}, \pi_G = (v_G, v_\bowtie)$:
   (a) $V = V_G \cup \{v_\bowtie\}$
   (b) $E = E_G \cup \{\pi_G, \neg\pi_G\}$
   (c) $\alpha = \alpha_G \cup \{(v_\bowtie, \bowtie)\}$
   (d) $\beta = \beta_G \cup \{\pi_G\}$

Questa definizione ha senso solo a partire da una G contenente solo sottoespressioni distinte. Infatti, se una sottoespressione C vi compare in più di una occorrenza l'arco π_C non è unico, potendo valere ad esempio $(v_C, v_{A\wedge C})$ o $(v_C, v_{B\vee C})$, e questo trasforma β in una funzione a più valori.

Si possono fornire due definizioni equivalenti dell'insieme L contenente le foglie dell'albero sintattico, $L = \{v \in V \mid \alpha(v) \in \tilde{\mathcal{L}}\} = \{v \in V \mid \forall\, w \in V \quad v \neq \beta(w)\}$. Per la struttura delle formule ben formate risulta inoltre $\alpha(v_\bowtie) = \bowtie$ e $\alpha(V \setminus (\{v_\bowtie\} \cup L)) \subseteq \{\wedge, \vee\} \cup \{\partial^k \mid k \in \mathbf{Z}\}$. Anche la funzione di rappresentazione può essere definita in maniera immediata a partire da questa definizione ricorsiva, valendo:

$$\forall\, C \,@\, t \in D \;\; \rho(C \,@\, t) = \neg\pi_C \,@\, t$$

$$\forall\, \neg C \,@\, t \in D \;\; \rho(\neg C \,@\, t) = \pi_C \,@\, t$$

Dato un qualsiasi $v_o \in V \setminus (\{v_\bowtie\} \cup L)$, chiamando $v_p = \beta(v_o)$ il nodo genitore e $v_l, v_r \mid v_o = \beta(v_l) = \beta(v_r)$ i nodi figli, eventualmente coincidenti per l'operatore unario ritardo, è possibile riscrivere le regole ascendenti e discendenti[1] che concorrono alla determinazione delle storie delle uscite come di seguito:

1. Se $\alpha(v_o) = \wedge$ allora...
   (a) $(v_p, v_o) \,@\, t \vdash (v_o, v_l) \,@\, t, (v_o, v_r) \,@\, t$

[1] Le regole trasversali non vengono qui riscritte poiché la rappresentazione di ogni loro prodotto verrà ottenuta applicando il teorema di sostituzione.

(b) $(v_l, v_o) @ t \vdash (v_o, v_p) @ t$

(c) $(v_r, v_o) @ t \vdash (v_o, v_p) @ t$

2. Se $\alpha(v_o) = \vee$ allora...

(a) $(v_p, v_o) @ t, (v_l, v_o) @ t \vdash (v_o, v_r) @ t$

(b) $(v_p, v_o) @ t, (v_r, v_o) @ t \vdash (v_o, v_l) @ t$

(c) $(v_l, v_o) @ t, (v_r, v_o) @ t \vdash (v_o, v_p) @ t$

3. Se $\alpha(v_o) = \partial^k$ allora...

(a) $(v_p, v_o) @ t \vdash (v_o, v_l) @ (t - k)$

(b) $(v_l, v_o) @ t \vdash (v_o, v_p) @ (t + k)$

La scrittura $r_1, \ldots, r_n \vdash s_1, \ldots, s_m$ è utilizzata come sinonimo di $\{s_1, \ldots, s_m\} \subseteq \phi(\{r_1, \ldots, r_n\})$ e nei casi appena esaminati può essere letta anche come $\{s_1, \ldots, s_m\} = \delta(\{r_1, \ldots, r_n\})$. Nel seguito si utilizzerà anche la scrittura $r_1, \ldots, r_n \vdash \cdots \vdash s_1, \ldots, s_m$ per indicare che $\{s_1, \ldots, s_m\} \subseteq \widehat{\phi}(\{r_1, \ldots, r_n\})$ oppure $\{s_1, \ldots, s_m\} \subseteq \widehat{\phi}(\Omega_0 \cup \{r_1, \ldots, r_n\})$, con Ω_0 insieme di conoscenza a priori.

Queste regole si prestano ad essere implementate attraverso una funzione inferenza binaria:

$$\forall\, S \in \mathcal{Z}_D \;\; \phi(S) = S \cup \bigcup_{r \in S} f(r) \cup \bigcup_{r,s \in S} g(r, s)$$

Le opportune funzioni f e g vengono definite nelle rispettive figure 3.3 e 3.4. La verifica della loro correttezza può essere fatta enumerando i casi possibili alla luce delle precedenti regole.

1. Se $\alpha(v_o) = \wedge$ allora...

(a) Se $v_i = \beta(v_o)$ allora...

i. $\exists (v_o, v_j), (v_o, v_k) \in E \mid v_i \notin \{v_j, v_k\}, v_j \neq v_k$

ii. $F \leftarrow \{(v_o, v_j) @ t, (v_o, v_k) @ t\}$

(b) ...altrimenti $F \leftarrow \{(v_o, \beta(v_o)) @ t\}$

2. ...altrimenti, se $\alpha(v_o) = \partial^k$ allora...

(a) Se $v_i = \beta(v_o)$ allora...

i. $\exists (v_o, v_j) \in E \mid v_i \neq v_j$

ii. Se $t - k \in T$ allora $F \leftarrow \{(v_o, v_j) @ (t - k)\}$, altrimenti $F \leftarrow \emptyset$

(b) ...altrimenti ...

i. Se $t + k \in T$ allora $F \leftarrow \{(v_o, \beta(v_o)) @ (t + k)\}$, altrimenti $F \leftarrow \emptyset$

3. ...altrimenti $F \leftarrow \emptyset$

Figura 3.3. Calcolo di $F = f((v_i, v_o) @ t)$

Come precedentemente dimostrato, per una funzione inferenza binaria valgono $f(r) = \delta(\{r\})$ e $g(r, s) = \xi(\{r\}, \{s\})$. Inoltre $\lambda_S(r) = f(r) \cup \bigcup_{s \in \mu^*_S(r)} g(r, s) \setminus \phi(\mu^*_S(r))$ con $\mu^*(r) = \{s \in Z_D \mid g(r, s) \neq \emptyset\}$. Per $\alpha(v_o) = \wedge$ o $\alpha(v_o) = \partial^k$ vale $\mu^*((v_i, v_o) @ t) = \emptyset$ mentre per $\alpha(v_o) = \vee$ si ottiene $\mu^*((v_i, v_o) @ t) = \{(v, v_o) @ t \in Z_D \mid v_i \neq v\}$. L'espressione esatta della frontiera di uscita è più complessa ma osservando che $\nu^*((v_i, v_o) @ t) \subseteq \{(v_o, w) @ \tau \in Z_D \mid v_i \neq w\}$ e $\forall\, v \in V \;\; (v, v) \notin E$ si può affermare che $\forall\, r \in Z_D \;\; \mu^*(r) \cap \nu^*(r) = \emptyset$.

Dato $v_o \in V \setminus (\{v_\bowtie\} \cup L)$, se $\alpha(v_o) = \wedge$ o $\alpha(v_o) = \partial^k$ risulta $\phi(\mu^*_S((v_i, v_o) @ t)) = \phi(\emptyset) = \emptyset$ e quindi $\lambda_S(r) = f(r) \cup \bigcup_{s \in \mu^*_S(r)} g(r, s)$ per $r = (v_i, v_o) @ t$. Quando invece $\alpha(v_o) = \vee$, chiamando

1. Se $v_i \neq v_j$, $v_o = v'_o$, $t = t'$ e $\alpha(v_o) = \vee$ allora...
   (a) $\exists (v_o, v_k) \in E \mid v_k \notin \{v_i, v_j\}$
   (b) $G \leftarrow \{(v_o, v_k) @ t\}$
2. ...altrimenti $G \leftarrow \emptyset$

Figura 3.4. Calcolo di $G = g((v_i, v_o) @ t, (v_j, v'_o) @ t')$

$e_i = (v_i, v_o) \in E$, $e_j = (v_j, v_o) \in E$ e $e_k = (v_k, v_o) \in E$ gli archi incidenti su v_o, si verifica che $\phi(\mu^*_S(e_i @ t)) \subseteq \phi(\mu^*(e_i @ t)) = \phi(\{e_j @ t, e_k @ t\}) = \{e_j @ t, e_k @ t\} \cup g(e_j @ t, e_k @ t) = \{e_j @ t, e_k @ t, \neg e_i @ t\}$ ovvero $\phi(\mu^*_S(r)) \subseteq \mu^*(r) \cup \{\neg r\}$ per $r = e_i @ t$. La correttezza di ϕ assicura che $\forall r, s \in Z_D \;\; g(r, s) \neq \neg r$ ed osservando che $g(r, s) \subseteq \nu^*(r) \Rightarrow g(r, s) \cap \mu^*(r) = \emptyset$ si può scrivere $\bigcup_{s \in \mu^*_S(r)} g(r, s) \setminus \phi(\mu^*_S(r)) \supseteq \bigcup_{s \in \mu^*_S(r)} g(r, s) \setminus (\mu^*(r) \cup \{\neg r\}) = \bigcup_{s \in \mu^*_S(r)} g(r, s)$. In generale vale $\bigcup_{s \in \mu^*_S(r)} g(r, s) \setminus \phi(\mu^*_S(r)) \subseteq \bigcup_{s \in \mu^*_S(r)} g(r, s)$ ed il confronto di questa relazione con la precedente consente di concludere che $\bigcup_{s \in \mu^*_S(r)} g(r, s) \setminus \phi(\mu^*_S(r)) = \bigcup_{s \in \mu^*_S(r)} g(r, s)$, per cui anche in questo caso $\lambda_S(r) = f(r) \cup \bigcup_{s \in \mu^*_S(r)} g(r, s)$.

Il secondo termine della funzione di inferenza locale può essere non vuoto solo se $\alpha(v_o) = \vee$, nel qual caso $f(e_i @ t) = \emptyset$ e $\lambda_S(e_i @ t) = \bigcup_{s \in \{e_j @ t, e_k @ t\} \cap S} g(e_i @ t, s)$. Supponendo $e_i @ t \in S$ ed osservando che $g(e_i @ t, e_j @ t) = \{\neg e_k @ t\}$ e $g(e_i @ t, e_k @ t) = \{\neg e_j @ t\}$, se si verificasse $e_j @ t, e_k @ t \in S$ risulterebbe $\lambda_S(e_i @ t) = g(e_i @ t, e_j @ t) \cup g(e_i @ t, e_k @ t) = \{\neg e_j @ t, \neg e_k @ t\} \subseteq \phi(S)$ e quindi $e_j @ t, \neg e_j @ t, e_k @ t, \neg e_k @ t \in \phi(S)$, in contraddizione con l'ipotesi di correttezza di ϕ e S. Dunque $\mu^*_S(r)$ contiene al più un elemento tra $e_j @ t$ e $e_k @ t$ e questo permette di semplificare ulteriormente il calcolo di λ come mostrato sinteticamente in tabella 3.4 e nella funzione riportata in figura 3.5.

$\alpha(v_o)$	$\beta(v_o)$	$\delta(\{(v_i, v_o) @ t\})$	$\mu^*_S((v_i, v_o) @ t)$	$\xi(\{(v_i, v_o) @ t\}, \mu^*_S((v_i, v_o) @ t))$
$\wedge$	v_i	$\{(v_o, v_j) @ t, (v_o, v_k) @ t\}$	$\emptyset$	$\emptyset$
	v_j, v_k	$\{(v_o, \beta(v_o)) @ t\}$		
∂^k	v_i	$\{(v_o, v_j) @ (t - k)\}$	$\emptyset$	$\emptyset$
	v_j	$\{(v_o, v_j) @ (t + k)\}$		
$\vee$	v_i, v_j, v_k	$\emptyset$	$\emptyset$	$\emptyset$
			$\{(v_j, v_o) @ t\}$	$\{(v_o, v_k) @ t\}$
			$\{(v_k, v_o) @ t\}$	$\{(v_o, v_j) @ t\}$

Tabella 3.4. Componenti di $\lambda_S((v_i, v_o) @ t)$

A differenza delle precedenti, la proprietà che consente l'ultima semplificazione effettuata richiede la correttezza dell'insieme di conoscenza disponibile ed è utile formalizzarla come $\forall S \subseteq S^*, r \in S \;\; \exists s \in S \mid \lambda_S(r) = \lambda_{\{s\}}(r)$, con $S^* \in \mathcal{Z}^*_D$ insieme obiettivo. Quando $\mu^*_S(r) = \emptyset$ questa relazione è soddisfatta dalla scelta $r = s$ mentre i casi rimanenti sono coperti da $\mu^*_S(r) = \{s\}$.

Dato un qualunque insieme di nodi V su cui siano definite in qualche modo le funzioni α e β, la funzione di inferenza locale può essere applicata ad ogni grafo orientato costruito con questi, purché chiuso rispetto alla negazione e con grado dei nodi adeguato alla loro classificazione secondo α. I risultati di questa applicazione dipenderanno dalla struttura del grafo espressa in termini di E, α e β.

Una rete di inferenza temporale è dunque ogni quadrupla (V, E, α, β) soddisfacente:

1. Se $\alpha(v_o) = \wedge$ allora...
   (a) Se $v_i = \beta(v_o)$ allora...
      i. $\exists (v_o, v_j), (v_o, v_k) \in E \mid v_i \notin \{v_j, v_k\}, v_j \neq v_k$
      ii. $\Lambda \leftarrow \{(v_o, v_j) \mathbin{@} t, (v_o, v_k) \mathbin{@} t\}$
   (b) ...altrimenti $\Lambda \leftarrow \{(v_o, \beta(v_o)) \mathbin{@} t\}$
2. ...altrimenti, se $\alpha(v_o) = \partial^k$ allora...
   (a) Se $v_i = \beta(v_o)$ allora...
      i. $\exists (v_o, v_j) \in E \mid v_i \neq v_j$
      ii. Se $t - k \in T$ allora $\Lambda \leftarrow \{(v_o, v_j) \mathbin{@} (t - k)\}$, altrimenti $\Lambda \leftarrow \emptyset$
   (b) ...altrimenti ...
      i. Se $t + k \in T$ allora $\Lambda \leftarrow \{(v_o, \beta(v_o)) \mathbin{@} (t + k)\}$, altrimenti $\Lambda \leftarrow \emptyset$
3. ...altrimenti, se $\alpha(v_o) = \vee$ allora...
   (a) Se $\exists (v, v_o) \mathbin{@} t \in S \mid v_i \neq v$ allora...
      i. $\exists (v_o, w) \in E \mid w \notin \{v_i, v\}$
      ii. $\Lambda \leftarrow \{(v_o, w) \mathbin{@} t\}$
   (b) ...altrimenti $\Lambda \leftarrow \emptyset$
4. ...altrimenti $\Lambda \leftarrow \emptyset$

Figura 3.5. Calcolo di $\Lambda = \lambda_S((v_i, v_o) \mathbin{@} t)$

1. $E \subseteq V \times V \mid \forall (v_i, v_j) \in E \;\; (v_j, v_i) \in E$
2. $\alpha \subseteq V \times (\{\bowtie, \wedge, \vee\} \cup \{\partial^k \mid k \in \mathbf{Z}\} \cup \tilde{\mathcal{L}}) \mid \forall\, v_o \in V \;\; \exists!(v_o, o) \in \alpha$
3. $\beta \subseteq E \mid \forall\, v_o \in V \setminus \{v_\bowtie \in V \mid \alpha(v_\bowtie) = \bowtie\} \;\; \exists!(v_o, v_p) \in \beta$
4. $\forall\, v_o \in V \mid \alpha(v_o) \in \{\wedge, \vee\} \;\; \mathrm{card}\{(v_i, v_o) \in E\} = 3$
5. $\forall\, v_o \in V \mid \alpha(v_o) \in \{\partial^k \mid k \in \mathbf{Z}\} \;\; \mathrm{card}\{(v_i, v_o) \in E\} = 2$
6. $\forall\, v_o \in V \mid \alpha(v_o) \in \{\bowtie\} \cup \tilde{\mathcal{L}} \;\; \mathrm{card}\{(v_i, v_o) \in E\} = 1$

La coppia di funzioni $K = (\alpha, \beta)$ codifica la conoscenza strutturale che occorre alla sestupla $(K, E, I, O, T, \Omega_0)$ per individuare univocamente la configurazione dell'esecutore di specifiche al momento del lancio.

Le reti di inferenza temporale ammettono una rappresentazione grafica assai comoda, su cui è possibile fare inferenza anche manualmente. Ad ogni nodo viene associato un componente il cui simbolo dipende da α e la cui orientazione da β, come mostrato in figura 3.6.

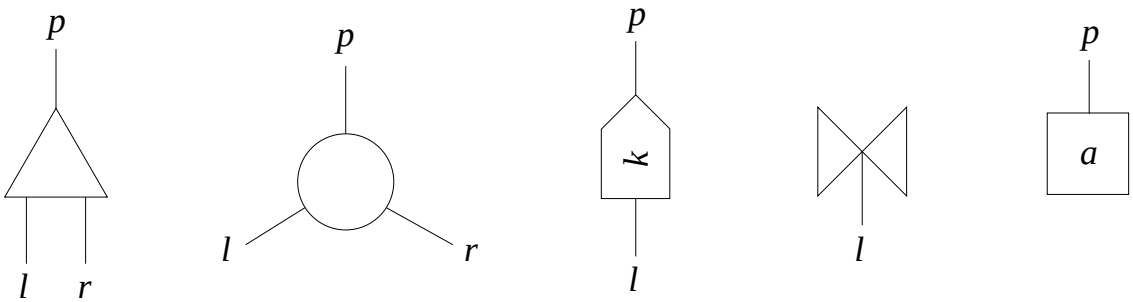

Figura 3.6. Da sinistra a destra, i simboli dei componenti *joint*, *gate*, *delay*, *root* e *leaf*

Ad ogni operatore $\wedge$ che compare nell'albero sintattico della specifica è associato un componente *joint*, ad ogni $\vee$ un *gate* e ad ogni ∂^k un *delay*, con k assunto uguale ad uno se non specificato diversamente all'interno del simbolo. Si noti che il simbolo del *gate* ha simmetria radiale, poiché la sua orientazione non influisce sul calcolo di λ, mentre quello del *joint* ha simmetria assiale, in quanto il suo comportamento è invariante rispetto allo scambio dei componenti figli l e r. In questo secondo caso, all'arco di collegamento con il componente genitore p si dà il nome di *terminale privilegiato* e lo si distingue spiccandolo da una punta del triangolo anzichè da un lato. Con un'analoga convenzione si distingue il terminale privilegiato del *delay*. Per completezza si associa un componente *root* all'operatore $\bowtie$, radice dell'albero sintattico, ed un *leaf* ad ogni letterale a che compare come sua foglia, anche se questi ultimi due componenti risulteranno successivamente superflui.

Tutti questi componenti sono connessi da archi semplici che potranno venire orientati qualora si rendesse disponibile conoscenza sulla formula loro associata in un particolare istante, ponendo una freccia incidente sul secondo nodo della coppia che ne rappresenta il valore logico nell'istante considerato. Gli archi a cui sono associate variabili di interesse sono etichettati con il nome della variabile ed una diversa freccia a metà lunghezza che ricorda il verso corrispondente al valore logico $\top$. Oltre al valore logico di una formula, ogni freccia indica la direzione in cui il processo d'inferenza propaga la conoscenza e per produrne di nuova basterà considerare il componente in cui entra la freccia ed aggiungervi le opportune freccie uscenti fino ad ottenere una tra le configurazioni invarianti elencate in figura 3.7. L'eventuale orientamento di un arco in entrambi i versi indica contraddizione e non può verificarsi nelle ipotesi finora assunte.

Il procedimento manuale non è direttamente applicabile ai ritardi ma richiede il disegno di una copia della rete per ciascun istante considerato. Per ognuna delle quattro configurazioni ammissibili

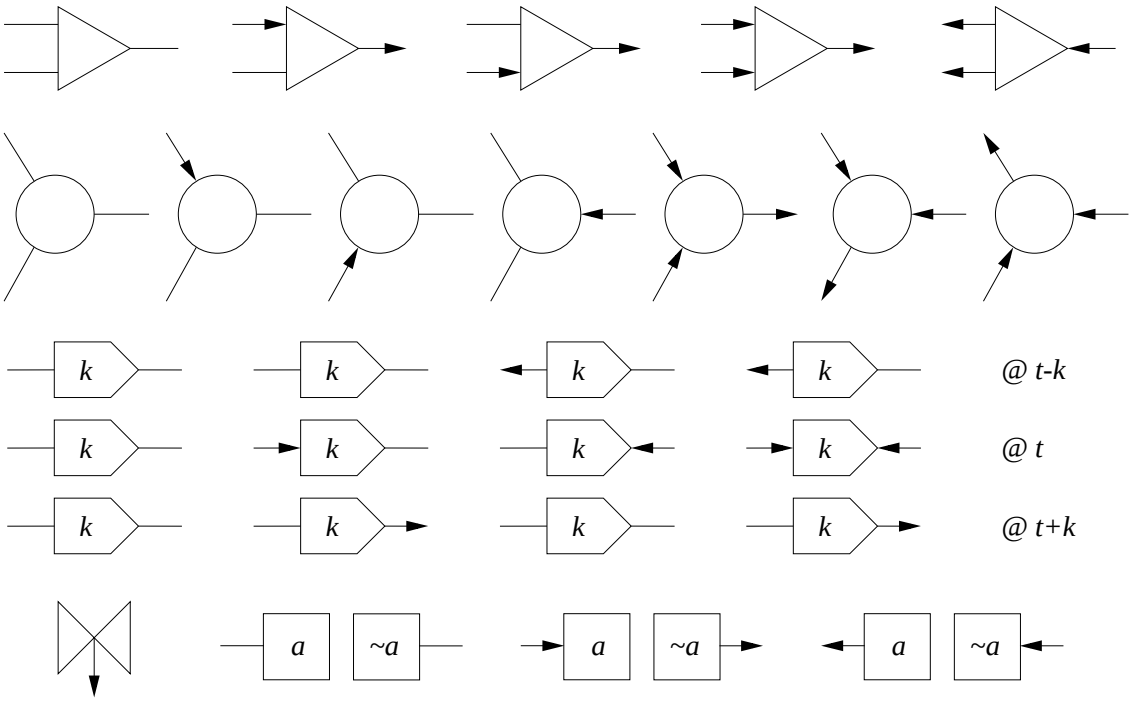

Figura 3.7. Configurazioni invarianti compatibili con i diversi componenti

come ingresso di un ritardo in un istante t assegnato, la figura riporta le configurazioni da esse implicate come uscita agli istanti $t-k$ e $t+k$. Dato che cambiare il segno di k equivale a scambiare i terminali del ritardo, quando il valore di k è negativo si preferisce procedere allo scambio per ottenere una maggiore leggibilità. Infine, le regole operanti sulle foglie valgono per ogni possibile coppia di letterali tra loro negati, e da sinistra a destra sono riportate le configurazioni relative ad una singola coppia nei casi di a ignoto, noto $\top$ e noto $\bot$.

Esempio In figura 3.8 sono presentate due possibili reti di definizione della funzione up_ℓ, tra loro equivalenti. Per ricavare la prima è sufficiente ridisegnare con diversa notazione l'albero sintattico riportato in figura 3.1, la seconda è invece frutto di un procedimento che verrà descritto nelle sezioni seguenti. Mentre nella prima rete la presenza dei letterali è sufficiente ad individuare le variabili in gioco, nella seconda sono state utilizzate freccie a metà lunghezza per etichettare gli archi di interesse. Inizializzando le due reti come mostrato in figura 3.9, cioè asserendo l'evento $up_\ell \,@\, t$, ed operando la chiusura si ottengono le configurazioni invarianti riportate più in basso nella stessa figura, da cui ad esempio si ricava la conoscenza dell'evento $\ell \,@\, t$. La figura 3.10 elenca i passi di applicazione della funzione ϕ che hanno portato al raggiungimento del risultato per la sola seconda rete, da non confondersi con le iterazioni dell'algoritmo di esecuzione che è invece basato sul calcolo di λ. La storia della rete è mostrata agli istanti $t-1$, t e $t+1$, sufficienti a verificare l'appartenenza alla chiusura di $\neg\ell \,@(t-1)$, $\neg up_\ell \,@(t-1)$ e $\neg up_\ell \,@(t+1)$. Alla stessa chiusura si giunge inizializzando la rete con la coppia di eventi $\neg\ell \,@(t-1)$, $\ell \,@\, t$ ed i passi necessari a verificarlo sono elencati in figura 3.11. In sintesi, tra le numerose produzioni che la rete implementa compaiono sia $up_\ell \,@\, t \vdash \cdots \vdash \neg\ell \,@(t-1), \ell \,@\, t, \neg up_\ell \,@(t-1), \neg up_\ell \,@(t+1)$ che $\neg\ell \,@(t-1), \ell \,@\, t \vdash \cdots \vdash \neg up_\ell \,@(t-1), up_\ell \,@\, t, \neg up_\ell \,@(t+1)$.

Così rappresentata, una rete di inferenza temporale ricorda una rete logica tradizionale, ma rispetto ad essa presenta alcuni vantaggi notevoli:

- Ogni arco ha tre possibili stati e può quindi rappresentare la non disponibilità di una particolare informazione

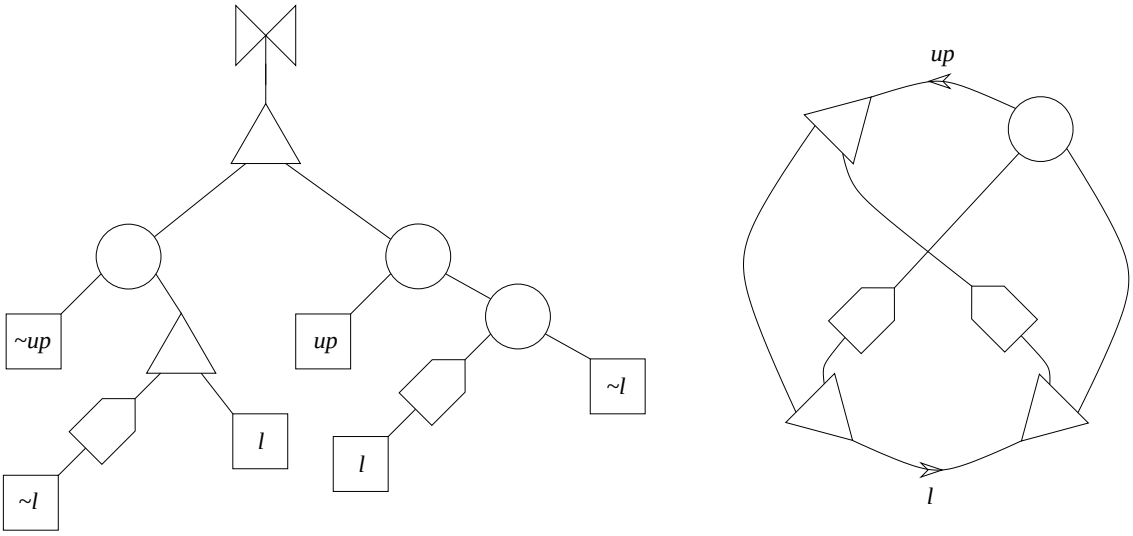

Figura 3.8. Due possibili reti di definizione della funzione up_ℓ

- Ingressi ed uscite non sono distinti in alcun modo e possono essere scambiati in qualsiasi momento, consentendo il calcolo di più funzioni logiche da parte della stessa rete
- Ogni componente riassume in sè la funzionalità di più componenti tradizionali e questo tende a produrre reti più compatte
- La conoscenza può propagarsi attraverso di esse anche lungo percorsi non causali

La reversibilità tra ingressi ed uscite e la compattezza ricordano il rapporto esistente tra una relazione, ad esempio $x + y - z = 0$, ed un insieme di funzioni equivalente, in questo caso la terna $z(x, y) = x + y$, $x(z, y) = z - y$ e $y(z, x) = z - x$. In questo senso, si può dire che i componenti di una rete di inferenza temporale esprimono *relazioni* tra gli archi circostanti e si comprende come questo permetta di operare inferenza causale o anticausale a seconda della diversa disponibilità delle variabili.

Una proprietà probabilmente unica di queste reti è la possibilità di considerare una variabile come ingresso su un certo intervallo temporale ed uscita su un secondo intervallo disgiunto dal primo. Questo può avere interesse se si dispone della storia parziale di un segnale e si desidera interpolarne le parti mancanti, supposto ovviamente che le parti presenti siano sufficienti e la specifica del segnale sufficientemente cogente.

L'esistenza di un algoritmo per costruire queste reti a partire dall'albero sintattico di una specifica non esclude la possibilità di disegnarle direttamente, aggirando ogni problema di completezza grazie ad un punto di vista esclusivamente operazionale. Più realisticamente, i due approcci possono essere combinati per costruire un primo prototipo per via sintattica, da rendere successivamente completo attraverso sostituzioni che garantiscano un aumento della conoscenza inferibile. Fortunatamente, queste sostituzioni hanno di solito l'aspetto di semplificazioni in quanto semplificare la rete significa avvicinare fatti topologicamente lontani e la funzione inferenza scelta, per quanto incompleta, risulta almeno localmente completa.

Si fa infine notare che i componenti *joint* e *delay* sono lineari, nel senso che per essi vale il principio di sovrapposizione degli effetti che in questo caso si scrive:

$$\forall\, R, S \in \mathcal{Z}_D \;\; \phi(R \cup S) = \phi(R) \cup \phi(S)$$

mentre i componenti *gate* e *joint* sono senza memoria, cioè soddisfacenti:

$$\forall\, S \in \mathcal{Z}_D, t \in T \;\; \phi(S \cap (E \times \{t\})) = \phi(S) \cap (E \times \{t\})$$

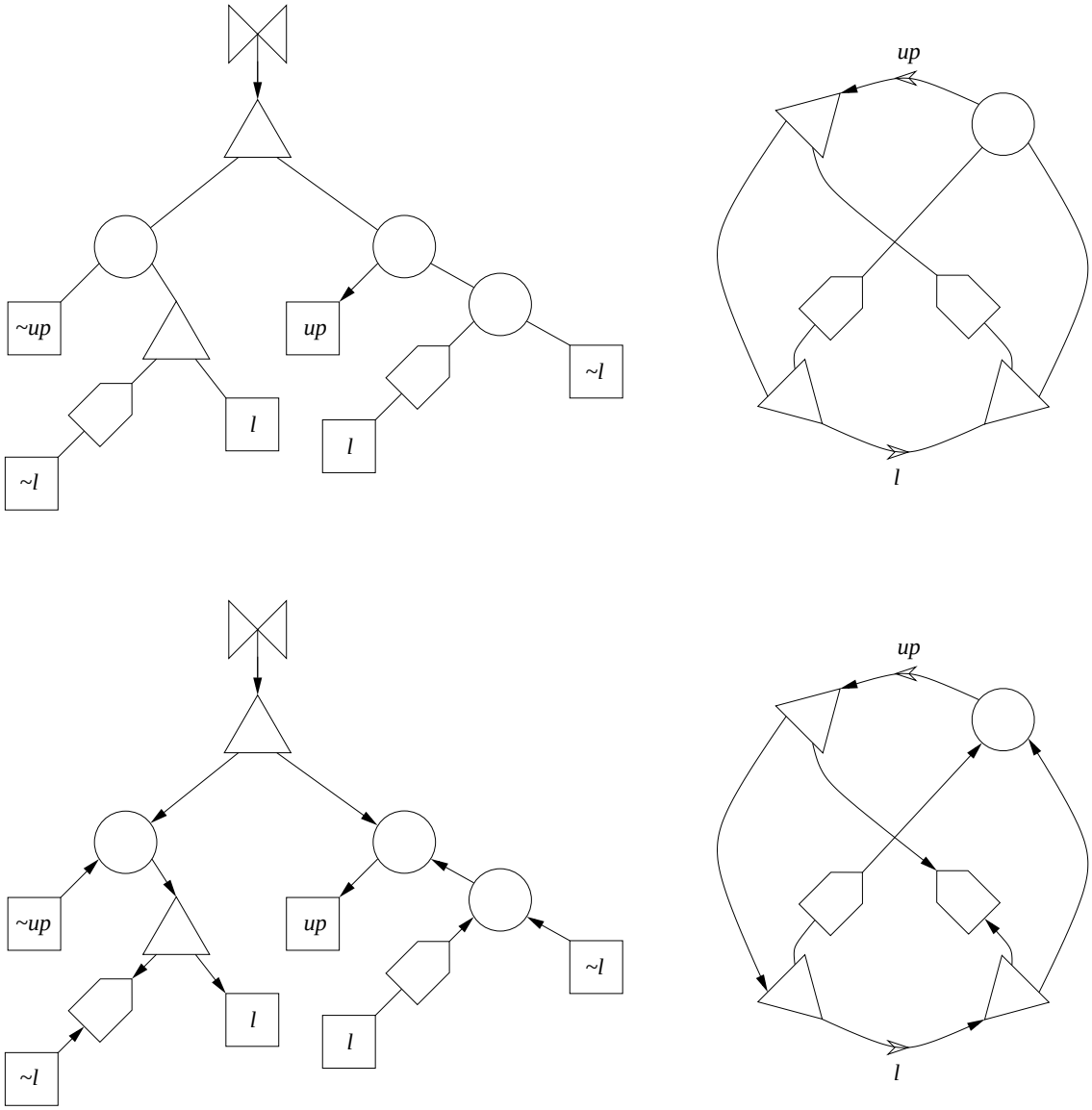

Figura 3.9. Due inizializzazioni equivalenti con le rispettive chiusure

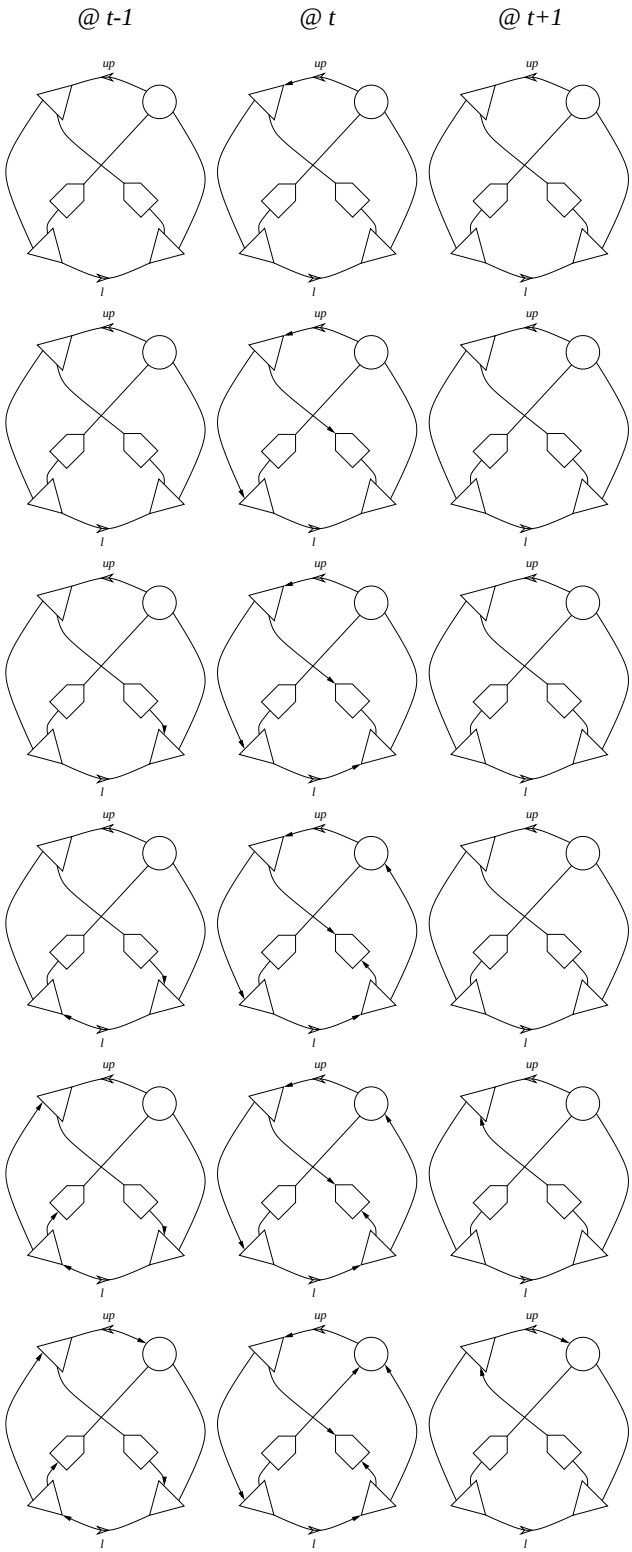

Figura 3.10. Passi di applicazione di ϕ alla precedente inizializzazione

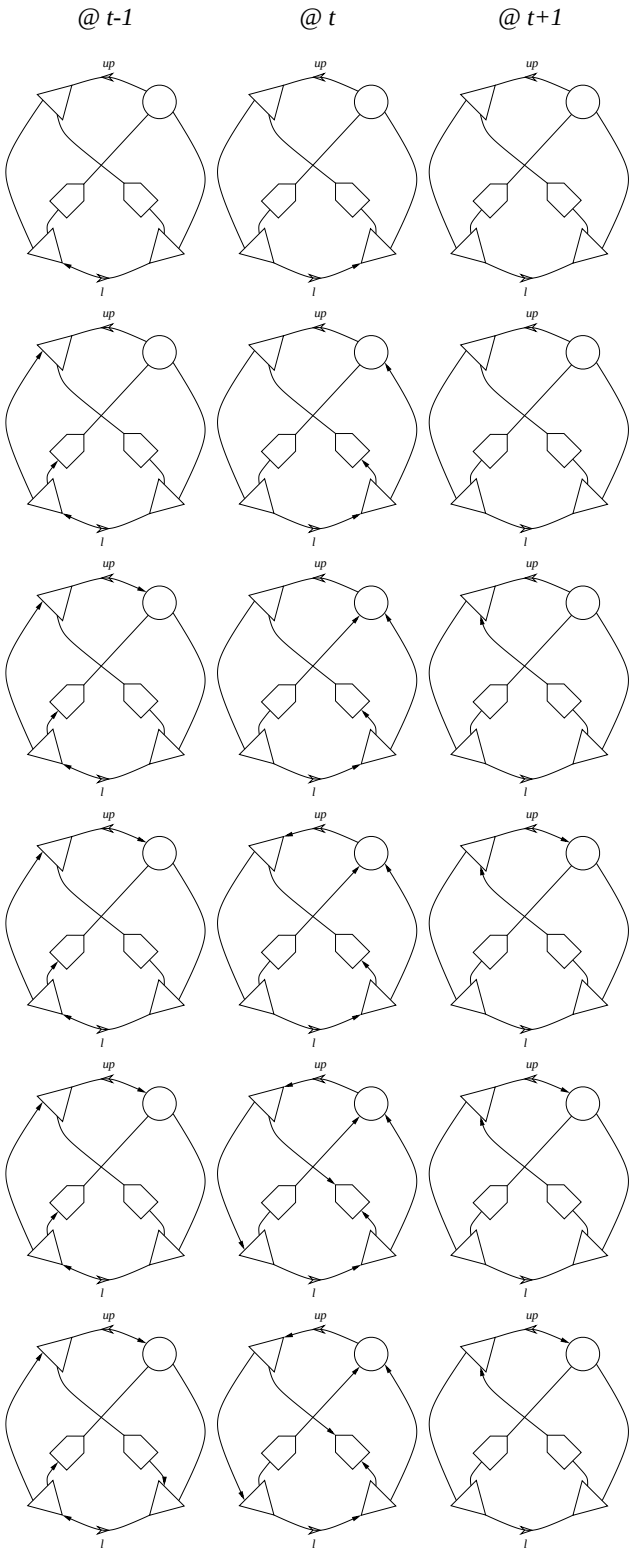

Figura 3.11. Passi di applicazione di ϕ ad una diversa inizializzazione

Le reti costruite con gli uni o con gli altri conservano queste proprietà e ciò si rivela utile durante la ricerca di sottoreti equivalenti tramite analisi visiva.

3.4 Teorema di sostituzione per reti di inferenza temporale

Operando un partizionamento dell'insieme dei nodi $V = V_X \cup V_Y \mid V_X \cap V_Y = \emptyset$, ad esempio tramite un taglio, e definendo:

$$E_X = \{(v, w) \in E \mid v \in V_X \vee w \in V_X\}$$

$$E_Y = \{(v, w) \in E \mid v \in V_Y \vee w \in V_Y\}$$

si può individuare l'insieme degli archi tagliati:

$$E_\cap = E_X \cap E_Y = \{(v, w) \in E \mid v \in V_X \iff w \in V_Y\}$$

e costruire gli insiemi di rappresentazione associati $Z_X = E_X \times T$, $Z_Y = E_Y \times T$ e $Z_\cap = E_\cap \times T$.

Ricordando poi la definizione di intorno $\upsilon(r) = \{r\} \cup \mu^*(r) \cup \nu^*(r)$, si può certamente affermare che $\upsilon((v_i, v_o) @ t) \subseteq \{(v_i, v_o) @ t\} \cup \{(v, v_o) @ t \in Z_D \mid v_i \neq v\} \cup \{(v_o, w) @ \tau \in Z_D \mid v_i \neq w\} \subseteq \{(v, w) @ \tau \in Z_D \mid v_o \in \{v, w\}\}$, quindi se $v_o \in V_X$ si verifica $\upsilon((v_i, v_o) @ t) \subseteq Z_X$, mentre $v_o \in V_Y$ implica $\upsilon((v_i, v_o) @ t) \subseteq Z_Y$, cioè $Z_\cap$ costituisce una interfaccia tra Z_X e Z_Y. Se il taglio non racchiude la rappresentazione di fatti di interesse, il teorema di sostituzione consente di rimpiazzare la parte del grafo interna ad esso, ad esempio E_X, con un nuovo sottografo $E_{X'}$ purché questo, sconnesso dal contesto ed inizializzato con un opportuno $\Omega_{X'} \subseteq Z_{X'} \setminus Z_Y$ e con tutte le possibili storie degli archi tagliati $S_\cap \subseteq Z_\cap$, continui a presentare alla chiusura uguali storie su $Z_\cap$.

Formalmente, la condizione che dovrà essere verificata per poter applicare una qualunque sostituzione è $\forall\, S_\cap \in \mathcal{Z}_\cap \;\; \widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_\cap \subseteq \widehat{\phi}_{X'}(\Omega_{X'} \cup S_\cap) \cap Z_\cap$ oppure $\widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_\cap = \widehat{\phi}_{X'}(\Omega_{X'} \cup S_\cap) \cap Z_\cap$ se si considera il teorema nella sua forma più stringente. Applicando il teorema ad una coppia di sottografi disgiunti si ha $V_X \cap V_Y = E_X \cap E_Y = \emptyset$ ed uno qualunque dei due sottografi può essere rimpiazzato con un qualunque altro sottografo disgiunto, ad esempio il grafo vuoto, senza modificare l'intersezione della chiusura con l'altro. Questo consente di unire due qualunque reti soddisfacenti $V_X \cap V_Y = E_X \cap E_Y = \emptyset$ secondo $(V, E, \alpha, \beta) = (V_X \cup V_Y, E_X \cup E_Y, \alpha_X \cup \alpha_Y, \beta_X \cup \beta_Y)$.

Definendo $E_{XY} = \{\underline{(v_x, v_y)} \in E \mid \underline{v_x} \in V_X, v_y \in V_Y\} \subseteq E_\cap$ ed il relativo $Z_{XY} = E_{XY} \times T \subseteq Z_\cap$ si osserva che $E_{YX} = \overline{E_{XY}}$, $Z_{YX} = \overline{Z_{XY}}$ e $Z_\cap = Z_X \cap Z_Y = Z_{XY} \cup Z_{YX}$. Inoltre $\forall (v_i, v_o) @ t \in Z_X \setminus Z_{XY} \;\; v_o \notin V_Y$ e questo comporta $\mu^*((v_i, v_o) @ t) \cap Z_{XY} \subseteq \{(v, v_o) @ t \in Z_D\} \cap \{(v, v_y) @ t \in Z_D \mid v_y \in V_Y\} = \emptyset$, mentre $\nu^*((v_i, v_o) @ t) \cap Z_{YX} \subseteq \{(v_o, w) @ \tau \in Z_D\} \cap \{(v_y, w) @ \tau \in Z_D \mid v_y \in V_Y\} = \emptyset$, cioè nel complesso $\forall\, r \in Z_X \setminus Z_{XY} \;\; \mu^*(r) \cap Z_{XY} = \nu^*(r) \cap Z_{YX} = \emptyset$.

Se vale la relazione:

$$\forall\, v_y \in V_Y \;\; (v_y, v_x), (v_y, v'_x) \in E_{YX} \Rightarrow v_x = v'_x$$

allora l'insieme Z_{YX} si dice una *porta di accesso* a Z_X. In questo caso $\forall (v_x, v_y) @ t \in Z_{XY}$ si verifica $\mu^*((v_x, v_y) @ t) \cap Z_X \subseteq \{(v, v_y) @ t \in Z_D \mid v_x \neq v\} \cap Z_{XY} = \{(v, v_y) @ t \in Z_D \mid v_x \neq v\} \cap \{(v'_x, v_y) @ t \in Z_{XY}\} = \{(v, v_y) @ t \in Z_D \mid v_x \neq v\} \cap \{(v_x, v_y) @ t\} = \emptyset$ e $\nu^*((v_x, v_y) @ t) \cap Z_X \subseteq \{(v_y, w) @ \tau \in Z_D \mid v_x \neq w\} \cap Z_{YX} = \{(v_y, w) @ \tau \in Z_D \mid v_x \neq w\} \cap \{(v_y, v'_x) @ \tau \in Z_{YX}\} = \{(v_y, w) @ \tau \in Z_D \mid v_x \neq w\} \cap \{(v_y, v_x) @ \tau \in Z_{YX}\} = \emptyset$, ovvero $\forall\, r \in Z_{XY} \;\; \mu^*(r) \cap Z_X = \nu^*(r) \cap Z_X = \emptyset$. Inoltre, ricordando che $Z_{XY}, Z_{YX} \subseteq Z_X$ il risultato $\forall\, r \in Z_X \setminus Z_{XY} \;\; \mu^*(r) \cap Z_{XY} = \nu^*(r) \cap Z_{YX} = \emptyset$ può essere riscritto come $\forall\, r \in Z_X \;\; \mu^*(r) \cap Z_{XY} = \nu^*(r) \cap Z_{YX} = \emptyset$.

Se Z_{YX} costituisce una porta di accesso a Z_X, le relazioni $\forall\, r \in Z_X \;\; \mu^*(r) \cap Z_{XY} = \emptyset$ e $\forall\, r \in Z_{XY} \;\; \nu^*(r) \cap Z_X = \emptyset$ soddisfano le ipotesi necessarie per applicare a $\widehat{\phi}_X$ un teorema dimostrato nella sezione sulle funzioni locali, la cui tesi garantisce che $\forall\, S_X \subseteq Z_X \;\; \widehat{\phi}_X(S_X) = S_X \cap Z_{XY} \cup \widehat{\phi}_X(S_X \setminus Z_{XY})$. Analogamente, da $\forall\, r \in Z_X \;\; \nu^*(r) \cap Z_{YX} = \emptyset$ un secondo teorema consente di concludere che $\forall\, S_X \subseteq Z_X \;\; \widehat{\phi}_X(S_X) = S_X \cap Z_{YX} \cup \widehat{\phi}_X(S_X) \setminus Z_{YX}$.

Si può allora dimostrare il seguente:

Teorema 3.4.1 (di sostituzione su reti) *Siano* $Z_X = E_X \times T \subseteq E_D \times T = Z_D, Z_{X'} = E_{X'} \times T \subseteq E_{D'} \times T = Z_{D'}, Z_Y = E_Y \times T \subseteq Z_D \cap Z_{D'} \mid Z_X = \overline{Z_X}, Z_{X'} = \overline{Z_{X'}}, Z_Y = \overline{Z_Y}, Z_X \cup Z_Y = Z_D, Z_{X'} \cup Z_Y = Z_{D'}$, $Z_{XY} = \{(v_x, v_y) @ t \in Z_D \mid v_x \in V_X, v_y \in V_Y\} = \{(v_{x'}, v_y) @ t \in Z_{D'} \mid v_{x'} \in V_{X'}, v_y \in V_Y\}$ *e* $Z_{YX} = \overline{Z_{XY}}$ *una porta di accesso comune a* Z_X *e* $Z_{X'}$*. Se esistono* $\Omega_X \subseteq Z_X \setminus Z_Y, \Omega_{X'} \subseteq Z_{X'} \setminus Z_Y$ *tali che* $\forall\, S_{YX} \subseteq Z_{YX}\ \ \widehat{\phi}_X(\Omega_X \cup S_{YX}) \cap Z_{XY} \subseteq \widehat{\phi}_{X'}(\Omega_{X'} \cup S_{YX}) \cap Z_{XY}$ *allora* $\forall\, S_Y \subseteq Z_Y\ \ \widehat{\phi}_D(\Omega_X \cup S_Y) \cap Z_Y \subseteq \widehat{\phi}_{D'}(\Omega_{X'} \cup S_Y) \cap Z_Y$*.*

Dimostrazione L'ipotesi di porta di accesso assicura che $\forall\, S_X \subseteq Z_X\ \ \widehat{\phi}_X(S_X) = S_X \cap Z_{XY} \cup \widehat{\phi}_X(S_X \setminus Z_{XY})$ e chiamando $S_\cap$ un qualunque sottoinsieme di $Z_\cap = Z_X \cap Z_Y$, $S_{XY} = S_\cap \cap Z_{XY}$ e $S_{YX} = S_\cap \cap Z_{YX}$ si verifica che $\widehat{\phi}_X(\Omega_X \cup S_\cap) = (\Omega_X \cup S_\cap) \cap Z_{XY} \cup \widehat{\phi}_X((\Omega_X \cup S_\cap) \setminus Z_{XY}) = S_{XY} \cup \widehat{\phi}_X(\Omega_X \cup (S_\cap \cap Z_{YX})) = S_{XY} \cup \widehat{\phi}_X(\Omega_X \cup S_{YX})$. Inoltre $\forall\, S_X \subseteq Z_X\ \ \widehat{\phi}_X(S_X) = S_X \cap Z_{YX} \cup \widehat{\phi}_X(S_X) \setminus Z_{YX}$ e nel caso esaminato $\widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_\cap = (\Omega_X \cup S_\cap) \cap Z_{YX} \cup (\widehat{\phi}_X(\Omega_X \cup S_\cap) \setminus Z_{YX}) \cap Z_\cap = S_{YX} \cup \widehat{\phi}_X(\Omega_X \cup S_\cap) \cap (Z_\cap \setminus Z_{YX}) = S_{YX} \cup \widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_{XY}$, per cui nel complesso $\widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_\cap = S_{YX} \cup \widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_{XY} = S_{YX} \cup S_{XY} \cup \widehat{\phi}_X(\Omega_X \cup S_{YX}) \cap Z_{XY} = S_\cap \cup \widehat{\phi}_X(\Omega_X \cup S_{YX}) \cap Z_{XY}$. Ragionando analogamente si ricava $\widehat{\phi}_{X'}(\Omega_{X'} \cup S_\cap) \cap Z_\cap = S_\cap \cup \widehat{\phi}_{X'}(\Omega_{X'} \cup S_{YX}) \cap Z_{XY}$ e se $\forall\, S_{YX} \subseteq Z_{YX}\ \ \widehat{\phi}_X(\Omega_X \cup S_{YX}) \cap Z_{XY} \subseteq \widehat{\phi}_{X'}(\Omega_{X'} \cup S_{YX}) \cap Z_{XY}$, il confronto di questi due risultati porta a concludere che $\forall\, S_\cap \in \mathcal{Z}_\cap\ \ \widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_\cap \subseteq \widehat{\phi}_{X'}(\Omega_{X'} \cup S_\cap) \cap Z_\cap$. Ciò soddisfa le ipotesi del teorema di sostituzione, da cui segue la tesi.

Gli archi appartenenti a E_{XY} saranno qualificati come *uscenti* dal taglio, mentre quelli di E_{YX} come *entranti* nel taglio. Individuato un taglio che racchiude il sottografo che si intende sostituire, se ogni arco uscente dal taglio incide su un diverso nodo del grafo circostante è sufficiente applicare ogni possibile combinazione di storie sugli archi entranti nel taglio, operare la restrizione della chiusura al sottografo e confrontare solo le storie degli archi uscenti dal taglio per poter applicare la sostituzione. Se invece il sottografo non ammette una porta di accesso e presenta coppie di archi uscenti incidenti sullo stesso nodo, si può aggirare l'ostacolo attraverso uno strumento alternativo che consente di aggiungere nuovi nodi al grafo o rimuoverli senza compromettere i risultati dell'inferenza. Inserendo questi nuovi nodi lungo gli archi candidati al taglio si può garantire l'esistenza di una porta di accesso al sottografo. Dato che questi nodi fittizi hanno anche altri usi di interesse, la restante parte di questa sezione ne introduce natura ed impiego.

La regola di riscrittura $[G]_{C \to \partial^0 C}$ non modifica la semantica di G, in quanto $C @ t \iff (\partial^0 C) @ t$, e le regole di inferenza implementate garantiscono che $(\partial^0 C) @ t \vdash C @ t$ e $\neg C @ t \vdash \neg(\partial^0 C) @ t$. Date le clausole $P, B, L, R \in \dot{\mathcal{G}}$ soddisfacenti $[P]_{C \to \star} \neq P$, $[P]_{B \to \star} \neq P$, $[C]_{B \to \star} = C$, $[B]_{C \to \star} = B$, $[C]_{L \to \star} \neq C$, $[L]_{C \to \star} = L$, $[C]_{R \to \star} \neq C$ e $[R]_{C \to \star} = R$, se è possibile applicare all'albero sintattico di $\bowtie G$ una tra le produzioni $a_1, \ldots, a_{n-1} \vdash a_n$:

1. $P @ \tau \vdash C @ t$
2. $P @ t, \neg B @ t \vdash C @ t$
3. $C @ t \vdash L @ \tau$
4. $C @ t, \neg R @ t \vdash L @ t$
5. $P @ t, \neg C @ t \vdash B @ t$
6. $\neg C @ t \vdash \neg P @ \tau$
7. $\neg C @ t, \neg B @ t \vdash \neg P @ t$
8. $\neg L @ \tau \vdash \neg C @ t$
9. $\neg L @ t, \neg R @ t \vdash \neg C @ t$

allora è possibile applicare all'albero sintattico di $\bowtie [G]_{C \to \partial^0 C}$ la produzione o coppia di produzioni corrispondente $b_1, \ldots, b_{n-1} \vdash \cdots \vdash b_n$:

1. $[P]_{C\to\partial^0 C} @ \tau \vdash (\partial^0 C) @ t$
2. $[P]_{C\to\partial^0 C} @ t, \neg B @ t \vdash (\partial^0 C) @ t$
3. $(\partial^0 C) @ t \vdash C @ t \vdash L @ \tau$
4. $(\partial^0 C) @ t, \neg R @ t \vdash C @ t, \neg R @ t \vdash L @ t$
5. $[P]_{C\to\partial^0 C} @ t, \neg(\partial^0 C) @ t \vdash B @ t$
6. $\neg(\partial^0 C) @ t \vdash \neg[P]_{C\to\partial^0 C} @ \tau$
7. $\neg(\partial^0 C) @ t, \neg B @ t \vdash \neg[P]_{C\to\partial^0 C} @ t$
8. $\neg L @ \tau \vdash \neg C @ t \vdash \neg(\partial^0 C) @ t$
9. $\neg L @ t, \neg R @ t \vdash \neg C @ t \vdash \neg(\partial^0 C) @ t$

Si osserva che ogni termine $b_k, k \in [1, n]$ può essere ottenuto riscrivendo il relativo a_k secondo $A_k @ t_k \to [A_k]_{C\to\partial^0 C} @ t_k$ o $\neg A_k @ t_k \to \neg[A_k]_{C\to\partial^0 C} @ t_k$, visto che $[B]_{C\to\star} = B$, $[L]_{C\to\star} = L$ e $[R]_{C\to\star} = R$ per ipotesi. Per quanto riguarda tutte le altre produzioni possibili, C non vi compare o vi compare solo come sottoespressione di una opportuna P e si verifica che se è possibile applicare a $\bowtie G$ una produzione $a_1, \ldots, a_{n-1} \vdash a_n$, con $a_k \notin \{C @ t, \neg C @ t\}, k \in [1, n]$, allora è possibile applicare a $\bowtie[G]_{C\to\partial^0 C}$ la produzione $b_1, \ldots, b_{n-1} \vdash b_n$ ottenuta attraverso la regola di riscrittura $a_k \to b_k$.

Si può dunque concludere che la riscrittura delle conclusioni di una generica produzione su $\bowtie G$ sono ottenibili anche operando su $\bowtie[G]_{C\to\partial^0 C}$ a partire dalla riscrittura delle sue premesse, direttamente come nell'ultimo caso discusso o per via transitiva come in alcuni dei casi precedenti. Operando la chiusura della riscrittura della conoscenza disponibile in base alla formula riscritta si produrrà allora la riscrittura di tutti gli eventi che sarebbero stati prodotti operando la chiusura della conoscenza disponibile in base alla formula di partenza, più alcuni della forma $C @ t$ o $\neg C @ t$.

La riscrittura ha un effetto topologico che può essere formalizzato. Dato W codominio di v sufficientemente ampio, si può individuare il nodo $w_{\partial^0 C} \in W \setminus V$ da inserire tra v_C e $\beta(v_C)$, cancellando la coppia di archi che li unisce e sostituendola con due nuove coppie di archi passanti per $w_{\partial^0 C}$. Chiamando $\pi_C = (v_C, \beta(v_C))$, $\pi'_C = (v_C, w_{\partial^0 C})$ e $\pi'_{\partial^0 C} = (w_{\partial^0 C}, \beta(v_C))$ si definisce la trasformazione operata sul grafo come:

1. $V' = V \cup \{w_{\partial^0 C}\}$
2. $E' = E \setminus \{\pi_C, \neg\pi_C\} \cup \{\pi'_C, \neg\pi'_C, \pi'_{\partial^0 C}, \neg\pi'_{\partial^0 C}\}$
3. $\alpha' = \alpha \cup \{(w_{\partial^0 C}, \partial^0)\}$
4. $\beta' = \beta \setminus \{\pi_C\} \cup \{\pi'_C, \pi'_{\partial^0 C}\}$

Come al solito, è possibile interpretare $\neg\pi'_C @ t$ e $\neg\pi'_{\partial^0 C} @ t$ come le nuove rappresentazioni ρ' di $C @ t$ e $(\partial^0 C) @ t$, ma da questo punto in poi si ignorerà questo aspetto a meno che gli elementi coinvolti non rappresentino fatti di interesse come variabili di ingresso o uscita. Se nessun evento $\neg\pi'_C @ t$ o $\pi'_{\partial^0 C} @ t$ è fornito come conoscenza iniziale o ingresso esterno, si può dimostrare che $\pi'_C @ t \in \widehat{\Phi} \iff \pi'_{\partial^0 C} @ t \in \widehat{\Phi}$ e $\neg\pi'_C @ t \in \widehat{\Phi} \iff \neg\pi'_{\partial^0 C} @ t \in \widehat{\Phi}$. Da un punto di vista operativo, risulta quindi ininfluente scegliere l'uno o l'altro come nuove rappresentazioni dell'evento sostituito. Individuato in $Z = W \times W \times T$ l'insieme di supporto di cui Z_D è sottoinsieme, si definirà direttamente $Z'_D = E' \times T \subseteq Z$, le relative funzioni inferenza saranno indicate con $\phi' : \mathcal{Z}'_D \to \mathcal{Z}'_D$, $\lambda'_S : Z'_D \to \mathcal{Z}'_D$ e così via.

L'inserimento di $w_{\partial^0 C}$ è una trasformazione invertibile, cioè $[[G]_{C\to\partial^0 C}]_{\partial^0 C\to C} = G$. Assegnato $w_{\partial^0 C}$, si può individuare v_C come il nodo soddisfacente $(v_C, w_{\partial^0 C}) \in E, v_C \neq \beta'(w_{\partial^0 C})$ ed assumendo le definizioni equivalenti $\pi_C = (v_C, \beta'(w_{\partial^0 C}))$, $\pi'_C = (v_C, w_{\partial^0 C})$ e $\pi'_{\partial^0 C} = (w_{\partial^0 C}, \beta'(w_{\partial^0 C}))$ si può formalizzare la trasformazione inversa come di seguito:

1. $V = V' \setminus \{w_{\partial^0 C}\}$
2. $E = E' \setminus \{\pi'_C, \neg\pi'_C, \pi'_{\partial^0 C}, \neg\pi'_{\partial^0 C}\} \cup \{\pi_C, \neg\pi_C\}$
3. $\alpha = \alpha' \setminus \{(w_{\partial^0 C}, \partial^0)\}$
4. $\beta = \beta' \setminus \{\pi'_C, \pi'_{\partial^0 C}, \neg\pi'_{\partial^0 C}\} \cup F_1 \cup F_2$
   (a) Se $\pi'_C \in \beta'$ allora $F_1 = \{\pi_C\}$, altrimenti $F_1 = \emptyset$
   (b) Se $\neg\pi'_{\partial^0 C} \in \beta'$ allora $F_2 = \{\neg\pi_C\}$, altrimenti $F_2 = \emptyset$

La determinazione di β merita un chiarimento. L'inserimento di un ritardo nullo tramite la regola di riscrittura garantisce $F_1 = \{\pi_C\}$, $F_2 = \emptyset$ e $\beta = \beta' \setminus \{\pi'_C, \pi'_{\partial^0 C}\} \cup \{\pi_C\}$. Tuttavia, la sua rimozione può avvenire anche dopo avere operato sostituzioni sul grafo che non hanno equivalenti sul piano sintattico e ne alterano la struttura consentendo l'esistenza di archi $\pi, \neg\pi \in \beta'$ o $e \in E, e \notin \beta', \neg e \notin \beta'$. Non si è invece ritenuto necessario generalizzare la trasformazione di inserimento a questi casi perché si può pensare di inserire tutti i ritardi che occorreranno prima di operare una qualunque sequenza di sostituzioni prive di un equivalente sintattico.

Per illustrare l'utilizzo di questi nodi fittizi è necessario considerare un un partizionamento di V in V_X e V_Y in base al quale individuare gli insiemi $E_X = \{(v_i, z), (z, v_i) \in E \mid v_i \in V_X\}$, $E_Y = \{(v_j, z), (z, v_j) \in E \mid v_j \in V_Y\}$, $E_{XY} = \{(v_i, v_j) \in E \mid v_i \in V_X, v_j \in V_Y\}$, $E_{YX} = \overline{E_{XY}}$ ed $E_\cap = E_X \cap E_Y = E_{XY} \cup E_{YX}$, che come dimostrato contiene gli archi le cui storie $Z_\cap = E_\cap \times T$ costituiscono una interfaccia tra $Z_X = E_X \times T$ e $Z_Y = E_Y \times T$. L'insieme E_{YX} può però non soddisfare la condizione che qualifica $Z_{YX} = E_{YX} \times T$ come porta di accesso a Z_X, cioè $\forall\, v_y \in V_Y \;\; (v_y, v_x), (v_y, v'_x) \in E_{YX} \Rightarrow v_x = v'_x$. Inoltre, se per maggior comodità di applicazione si desidera sostituire l'intero insieme dei nodi V_X con un nuovo insieme $U \subseteq W \setminus V$ su cui è costruita una rete equivalente, l'interfaccia $Z_\cap$ risulta inadeguata allo scopo in quanto contenente archi costruiti su alcuni elementi di V_X. Si può allora individuare un insieme di nodi $W_{\partial^0} \subseteq W \setminus (U \cup V)$ con $\operatorname{card} W_{\partial^0} = \operatorname{card} E_\cap / 2 = n_{\partial^0}$, i cui elementi $w_k \in W_{\partial^0}$ soddisfino $\forall\, k \in [1, n_{\partial^0}] \;\; \alpha(w_k) = \partial^0$ e $\forall\, h, k \in [1, n_{\partial^0}] \;\; h \neq k \Rightarrow w_h \neq w_k$ e possano essere inseriti su ognuno degli archi di $E_\cap$ secondo il procedimento appena visto.

Quando si desidera garantire l'esistenza di una porta di accesso per poter utilizzare il teorema di sostituzione nella sua ultima formulazione[2], l'inserimento dovrà avvenire a monte del taglio. Indicando con $(V', E', \alpha', \beta')$ la rete frutto dell'inserimento di tutti i nodi di W_{∂^0}, si può allora scegliere come nuovo partizionamento $V' = V_X \cup V'_Y = V_X \cup W_{\partial^0} \cup V_Y$ e, chiamando $E'_X = \{(v_i, z), (z, v_i) \in E' \mid v_i \in V_X\}$ ed $E'_Y = \{(v'_l, z), (z, v'_l) \in E' \mid v'_l \in W_{\partial^0} \cup V_Y\}$, si verifica che $E'_\cap = E'_X \cap E'_Y = \{(v_i, v'_l), (v'_l, v_i) \in E' \mid v_i \in V_X, v'_l \in W_{\partial^0} \cup V_Y\} = \{(v_i, w_k), (w_k, v_i) \in E' \mid v_i \in V_X, w_k \in W_{\partial^0}\}$ ovvero $E'_{YX} = \{(w_k, v_i) \in E' \mid v_i \in V_X, w_k \in W_{\partial^0}\}$, che soddisfa la condizione desiderata perché su ogni arco di E_{YX} è stato inserito un diverso w_k ed ad ogni $w_k \in W_{\partial^0}$ corrisponde dunque un unico $v_i \in V_X$. Una volta effettuata la sostituzione, i nodi fittizi potranno poi venire rimossi ricostituendo l'interfaccia di partenza $E_\cap$.

Quando invece si vuole sostituire l'intero insieme dei nodi racchiusi V_X, i ritardi nulli dovranno essere inseriti a valle del taglio in modo da costituire uno strato di interfaccia artificiale che resterà comune durante l'applicazione del teorema[3]. Indicando con $(V', E', \alpha', \beta')$ la rete frutto dell'inserimento di tutti i nodi di W_{∂^0}, si può stavolta scegliere come nuovo partizionamento $V' = V'_X \cup V_Y = V_X \cup W_{\partial^0} \cup V_Y$ e, chiamando $E'_X = \{(v'_l, z), (z, v'_l) \in E' \mid v'_l \in V_X \cup W_{\partial^0}\}$ ed $E'_Y = \{(v_j, z), (z, v_j) \in E' \mid v_j \in V_Y\}$, si verifica che $E'_\cap = E'_X \cap E'_Y = \{(v'_l, v_j), (v_j, v'_l) \in E' \mid v'_l \in V_X \cup W_{\partial^0}, v_j \in V_Y\} = \{(w_k, v_j), (v_j, w_k) \in E' \mid w_k \in W_{\partial^0}, v_j \in V_Y\}$, ottenendo $Z'_\cap = E'_\cap \times T$ interfaccia tra $Z'_X = E'_X \times T$ e $Z'_Y = E'_Y \times T$ priva di riferimenti ai nodi di V_X. A questo punto è possibile sostituire il sottografo costruito su $V'_X = V_X \cup W_{\partial^0}$ con un sottografo equivalente costruito su $V''_X = U \cup W_{\partial^0}$ e rimuovere infine i nodi di W_{∂^0} dal grafo risultante $(V'', E'', \alpha'', \beta'')$, invertendo il procedimento e rimpiazzando ogni quadrupla di archi

[2] Il teorema può comunque essere applicato direttamente ricorrendo ad una delle precedenti formulazioni.

[3] Identici risultati possono essere ottenuti semplicemente applicando il teorema ad un diverso taglio abbastanza ampio.

$(u_h, w_k), (w_k, u_h), (v_j, w_k), (w_k, v_j) \in E'' \mid u_h \in U, w_k \in W_{\partial^0}, v_j \in V_Y$ con una coppia di archi $(u_h, v_j), (v_j, u_h)$ costituente un collegamento diretto tra U e V_Y.

In realtà, nessuna di queste due applicazioni verrà mai effettuata esplicitamente. Sono state presentate in quanto costituenti una giustificazione rigorosa, sebbene non completamente formale, del modo intuitivo di operare sostituzioni sul grafo. Infatti, una volta disegnato un taglio attorno al sottografo da sostituire, tutto ciò che sta dentro potrà venir rimpiazzato a prescindere da tutto ciò che sta fuori, nonostante gli eventi di interfaccia siano definiti a partire da coppie di nodi di cui l'uno è interno e l'altro esterno al taglio. Analogamente, l'operazione di chiusura che garantisce l'equivalenza di due sottografi potrà essere effettuata ignorando gli eventuali prodotti dell'applicazione della funzione inferenza locale ad eventi incidenti su nodi esterni al taglio, anche quando questi prodotti appartenessero all'interfaccia. Questo permette di considerare gli archi tagliati alla stregua di terminali di un circuito e di interpretare la loro orientazione istantanea come qualifica del loro ruolo di ingresso o uscita per quello stesso istante.

3.5 Applicazione del teorema di sostituzione

Durante l'analisi fin qui condotta è stata sottintesa la necessità di implementare in qualche modo le regole di inferenza trasversali, ma la funzione λ presentata non risponde a questa esigenza in quanto $\forall\, v_o \in L, r = (v_i, v_o) \,@\, t \in Z_D, S \in \mathcal{Z}_D \;\; \lambda_S(r) = \emptyset$, con L insieme delle foglie dell'albero sintattico. Anziché aggiungere ulteriori casi che appesantiscano la funzione di inferenza locale, si farà uso del teorema di sostituzione. Usando la procedura appena descritta è possibile sostituire tutte le foglie di L ed ottenere attraverso una rete opportuna gli stessi risultati che si otterrebbero implementando le regole di inferenza trasversali.

Assegnata la specifica $\bowtie G$ ed una funzione $\ell \in \mathcal{L}$ che vi compare, questa avrà la forma di uno dei letterali $\ell^i, \neg\ell^j \in \alpha(L), i \in [1, m], j \in [1, n]$ e per semplicità si supporranno $m, n \geq 1$, condizione peraltro verificata da qualsiasi specifica aspiri alla completezza.

Si definisce allora la *specifica di collegamento* associata a ℓ come:

$$\bowtie J_\ell = \bowtie((\bigwedge_{i=1}^{m} \neg\ell^i) \vee (\bigwedge_{j=1}^{n} \ell^j))$$

Tutte le possibili applicazioni delle regole di inferenza selezionate su $\bowtie J_\ell \in \dot{\mathcal{F}}$ sono riassunte dalle seguenti produzioni:

1. $\forall\, h \in [2, m], t \in T \;\; (\bigwedge_{i=1}^{h} \neg\ell^i) \,@\, t \vdash (\bigwedge_{i=1}^{h-1} \neg\ell^i) \,@\, t, (\neg\ell^h) \,@\, t$
2. $\forall\, h \in [2, m], t \in T \;\; \neg(\bigwedge_{i=1}^{h-1} \neg\ell^i) \,@\, t \vdash \neg(\bigwedge_{i=1}^{h} \neg\ell^i) \,@\, t$
3. $\forall\, h \in [2, m], t \in T \;\; \neg(\neg\ell^h) \,@\, t \vdash \neg(\bigwedge_{i=1}^{h} \neg\ell^i) \,@\, t$
4. $\forall\, k \in [2, n], t \in T \;\; (\bigwedge_{j=1}^{k} \ell^j) \,@\, t \vdash (\bigwedge_{j=1}^{k-1} \ell^j) \,@\, t, \ell^k \,@\, t$
5. $\forall\, k \in [2, n], t \in T \;\; \neg(\bigwedge_{j=1}^{k-1} \ell^j) \,@\, t \vdash \neg(\bigwedge_{j=1}^{k} \ell^j) \,@\, t$
6. $\forall\, k \in [2, n], t \in T \;\; \neg(\ell^k) \,@\, t \vdash \neg(\bigwedge_{j=1}^{k} \ell^j) \,@\, t$
7. $\forall\, t \in T \;\; J_\ell \,@\, t, \neg(\bigwedge_{i=1}^{m} \neg\ell^i) \,@\, t \vdash (\bigwedge_{j=1}^{n} \ell^j) \,@\, t$
8. $\forall\, t \in T \;\; J_\ell \,@\, t, \neg(\bigwedge_{j=1}^{n} \ell^j) \,@\, t \vdash (\bigwedge_{i=1}^{m} \neg\ell^i) \,@\, t$

Osservando che $\bigwedge_{i=1}^{1} \neg\ell^i = \neg\ell^1$, $\bigwedge_{j=1}^{1} \ell^j = \ell^1$ e procedendo per induzione su m ed n si possono inoltre dimostrare le seguenti:

1. $\forall\, h \in [1, m], t \in T \;\; (\bigwedge_{i=1}^{m} \neg\ell^i) \,@\, t \vdash \cdots \vdash (\neg\ell^h) \,@\, t$
2. $\forall\, h \in [1, m], t \in T \;\; \neg(\neg\ell^h) \,@\, t \vdash \cdots \vdash \neg(\bigwedge_{i=1}^{m} \neg\ell^i) \,@\, t$

3. $\forall\, k \in [1,n], t \in T \;\; (\bigwedge_{j=1}^{n} \ell^j) \mathbin{@} t \vdash \cdots \vdash \ell^k \mathbin{@} t$

4. $\forall\, k \in [1,n], t \in T \;\; \neg(\ell^k) \mathbin{@} t \vdash \cdots \vdash \neg(\bigwedge_{j=1}^{n} \ell^j) \mathbin{@} t$

e ricordando infine che l'insieme di conoscenza a priori $\mathcal{H}_{J_\ell}$ contiene tutti i $J_\ell \mathbin{@} t \in D$ si può concludere che:

1. $\forall\, h \in [1,m], k \in [1,n], t \in T \;\; \neg(\neg\ell^h) \mathbin{@} t \vdash \cdots \vdash \ell^k \mathbin{@} t$

2. $\forall\, h \in [1,m], k \in [1,n], t \in T \;\; \neg(\ell^k) \mathbin{@} t \vdash \cdots \vdash (\neg\ell^h) \mathbin{@} t$

La specifica di collegamento implementa quindi per via transitiva regole di inferenza trasversali che sono duali rispetto a quelle che si desidera implementare su $\bowtie G$. Chiamando $(V_\triangle, E_\triangle, \alpha_\triangle, \beta_\triangle)$ la rete costruita a partire da $\bowtie G$ attraverso la funzione $v_\triangle$, si suppone di mappare le sottoespressioni di $\bowtie J_\ell$ attraverso una diversa funzione $v_\triangledown : \dot{\mathcal{G}} \cup \{\bowtie\} \to W \setminus V_\triangle$ soddisfacente $\forall\, A, B \in \dot{\mathcal{G}} \cup \{\bowtie\} \quad A = B \iff v_\triangledown(A) = v_\triangledown(B)$ e costruire così la rete di collegamento $(V_\triangledown, E_\triangledown, \alpha_\triangledown, \beta_\triangledown)$ associata alla funzione ℓ. Dato che $E_\triangle \cap E_\triangledown = \emptyset$ è una interfaccia tra $E_\triangle$ ed $E_\triangledown$, la chiusura che si ottiene operando su $(V, E, \alpha, \beta) = (V_\triangle \cup V_\triangledown, E_\triangle \cup E_\triangledown, \alpha_\triangle \cup \alpha_\triangledown, \beta_\triangle \cup \beta_\triangledown)$ coincide con l'unione delle chiusure che si sarebbero ottenute operando separatamente sulle due reti.

La rete (V, E, α, β) può essere modificata applicando il teorema di sostituzione nella sua forma più generale, cioè quella che garantisce la non diminuzione di conoscenza inferibile a seguito della sostituzione, e dimostrando separatamente la correttezza dei nuovi eventi che la sostituzione rende inferibili. Per far questo occorre definire $v^+_{\triangle,i} = v_\triangle(\ell^i)$, $v^-_{\triangle,j} = v_\triangle(\neg\ell^j)$, $\pi^+_{\triangle,i} = (v^+_{\triangle,i},\ \beta(v^+_{\triangle,i}))$, $\pi^-_{\triangle,j} = (v^-_{\triangle,j}, \beta(v^-_{\triangle,j}))$ ed i loro duali $v^+_{\triangledown,i} = v_\triangledown(\neg\ell^i)$, $v^-_{\triangledown,j} = v_\triangledown(\ell^j)$, $\pi^+_{\triangledown,i} = (v^+_{\triangledown,i},\ \beta(v^+_{\triangledown,i}))$, $\pi^-_{\triangledown,j} = (v^-_{\triangledown,j},\ \beta(v^-_{\triangledown,j}))$. Supponendo di aver già inserito un nodo $w^\pm_{\triangle,h}, w^\pm_{\triangledown,h} \in W_{\partial^0}$ su ogni arco che insiste su una foglia di uno dei due alberi sintattici, varrà $w^\pm_{\triangle,h} = \beta(v^\pm_{\triangle,h})$, $w^\pm_{\triangledown,h} = \beta(v^\pm_{\triangledown,h})$ con $\alpha(w^\pm_{\triangle,h}) = \alpha(w^\pm_{\triangledown,h}) = \partial^0$ e chiamando $e^\pm_{\triangle,h} = (w^\pm_{\triangle,h}, \beta(w^\pm_{\triangle,h}))$ e $e^\pm_{\triangledown,h} = (w^\pm_{\triangledown,h}, \beta(w^\pm_{\triangledown,h}))$, quanto fin qui dimostrato su $\bowtie J_\ell$ consente di scrivere:

1. $\forall\, i \in [1,m], j \in [1,n], t \in T \;\; \pi^+_{\triangledown,i} \mathbin{@} t \vdash e^+_{\triangledown,i} \mathbin{@} t \vdash \cdots \vdash \neg e^-_{\triangledown,j} \mathbin{@} t \vdash \neg\pi^-_{\triangledown,j} \mathbin{@} t$

2. $\forall\, i \in [1,m], j \in [1,n], t \in T \;\; \pi^-_{\triangledown,j} \mathbin{@} t \vdash e^-_{\triangledown,j} \mathbin{@} t \vdash \cdots \vdash \neg e^+_{\triangledown,i} \mathbin{@} t \vdash \neg\pi^+_{\triangledown,i} \mathbin{@} t$

ovvero $\pi^\pm_{\triangledown,h} \mathbin{@} t \vdash \cdots \vdash \neg\pi^\mp_{\triangledown,k} \mathbin{@} t$. Considerando adesso il taglio che racchiude i quattro nodi $v^\pm_{\triangle,h}, v^\pm_{\triangledown,h}, w^\pm_{\triangle,h}, w^\pm_{\triangledown,h}$, si identifica in $F^\pm_h = \{\pi^\pm_{\triangle,h}, \neg\pi^\pm_{\triangle,h}, \pi^\pm_{\triangledown,h}, \neg\pi^\pm_{\triangledown,h}\}$ l'insieme degli archi inclusi dal taglio, mentre l'insieme degli archi tagliati risulta $E^\pm_{\cap,h} = \{e^\pm_{\triangle,h}, \neg e^\pm_{\triangle,h}, e^\pm_{\triangledown,h}, \neg e^\pm_{\triangledown,h}\}$ ed individua l'interfaccia $Z^\pm_{\cap,h} = E^\pm_{\cap,h} \times T$ tra $Z^\pm_{X,h} = (E^\pm_{\cap,h} \cup F^\pm_h) \times T$ e $Z^\pm_{Y,h} = (E \setminus F^\pm_h) \times T$. Inoltre, partizionando l'interfaccia secondo $Z^\pm_{\cap,h} = Z^\pm_{XY,h} \cup Z^\pm_{YX,h} = (E^\pm_{XY,h} \cup E^\pm_{YX,h}) \times T$, con $E^\pm_{XY,h} = \{e^\pm_{\triangle,h}, e^\pm_{\triangledown,h}\}$ ed $E^\pm_{YX,h} = \{\neg e^\pm_{\triangle,h}, \neg e^\pm_{\triangledown,h}\}$, si osserva che $Z^\pm_{YX,h}$ costituisce una porta di accesso a $Z^\pm_{X,h}$ poiché $\beta(w^\pm_{\triangle,h}) \neq \beta(w^\pm_{\triangledown,h})$.

Quando $h = 1$, l'eventuale insieme di conoscenza a priori $\Omega_\ell = \rho_\triangle(\mathcal{H}_\ell) \subseteq I \times T$ che contiene la storia dell'ingresso $\ell \in \mathcal{L}$ sarà definito come $\Omega_\ell = \Omega^+_\ell \cup \Omega^-_\ell$, con $\Omega^+_\ell = \{\neg\pi^+_{\triangle,1} \mathbin{@} t^+ \in Z_D \mid \ell \mathbin{@} t^+ \in \mathcal{H}_\ell\}$ e $\Omega^-_\ell = \{\neg\pi^-_{\triangle,1} \mathbin{@} t^- \in Z_D \mid (\neg\ell) \mathbin{@} t^- \in \mathcal{H}_\ell\}$ perché $\rho_\triangle(\ell \mathbin{@} t) = \neg\pi^+_{\triangle,1} \mathbin{@} t$ e $\rho_\triangle((\neg\ell) \mathbin{@} t) = \neg\pi^-_{\triangle,1} \mathbin{@} t$, tale quindi da soddisfare $\Omega^\pm_\ell \subseteq F^\pm_1 \times T = Z^\pm_{X,1} \setminus Z^\pm_{Y,1}$. Dato che $E_\triangle$ ed $E_\triangledown$ sono disgiunti, l'appartenenza alla chiusura di $e^\pm_{\triangle,h} \mathbin{@} t$ o $\neg e^\pm_{\triangle,h} \mathbin{@} t$ è indipendente da quella di $e^\pm_{\triangledown,h} \mathbin{@} t$ o $\neg e^\pm_{\triangledown,h} \mathbin{@} t$ e la rete costruita su $E^\pm_{\cap,h} \cup F^\pm_h$ non compie alcuna inferenza visibile attraverso l'interfaccia. In altre parole, qualunque inizializzazione di $Z^\pm_{\cap,h}$ è invariante rispetto alla restrizione della chiusura a $Z^\pm_{X,h}$. Questo si verifica anche quando $\Omega_\ell \neq \emptyset$, valendo $\alpha(v^\pm_{\triangle,1}) \in \tilde{\mathcal{L}}$ e quindi $\forall\, r \in \Omega_\ell, S \in \mathcal{Z}_D \quad \lambda_S(r) = \lambda_S((w^\pm_{\triangle,1}, v^\pm_{\triangle,1}) \mathbin{@} t) = \emptyset$. Nel complesso, il comportamento del sottografo candidato alla sostituzione può dunque essere riassunto nella scrittura $\forall\, T_\triangle, T_\triangledown \subseteq T \;\; \widehat{\phi}_X(\Omega_X \cup \{\neg e^\pm_{\triangle,h}\} \times T_\triangle \cup \{\neg e^\pm_{\triangledown,h}\} \times T_\triangledown) \cap Z^\pm_{XY,h} = \emptyset$, con $\Omega_X = \Omega_\ell \cap Z^\pm_{X,h}$.

Definendo $e^\pm_{\sim,h} = (w^\pm_{\triangledown,h}, w^\pm_{\triangle,h})$ si può allora applicare la sostituzione:

1. $V' = V \setminus \{v^{\pm}_{\triangle,h}, v^{\pm}_{\triangledown,h}\}$
2. $E' = E \setminus \{\pi^{\pm}_{\triangle,h}, \neg\pi^{\pm}_{\triangle,h}, \pi^{\pm}_{\triangledown,h}, \neg\pi^{\pm}_{\triangledown,h}\} \cup \{e^{\pm}_{\sim,h}, \neg e^{\pm}_{\sim,h}\}$
3. $\alpha' = \alpha \setminus \{(v^{\pm}_{\triangle,h}, \alpha(v^{\pm}_{\triangle,h})), (v^{\pm}_{\triangledown,h}, \alpha(v^{\pm}_{\triangledown,h}))\}$
4. $\beta' = \beta \setminus \{\pi^{\pm}_{\triangle,h}, \pi^{\pm}_{\triangledown,h}\}$

che permette di operare le produzioni:

1. $\forall\, i \in [1,m], t \in T \;\; \neg e^{+}_{\triangle,i} \mathbin{@} t \vdash \neg e^{+}_{\sim,i} \mathbin{@} t \vdash e^{+}_{\triangledown,i} \mathbin{@} t$
2. $\forall\, i \in [1,m], t \in T \;\; \neg e^{+}_{\triangledown,i} \mathbin{@} t \vdash e^{+}_{\sim,i} \mathbin{@} t \vdash e^{+}_{\triangle,i} \mathbin{@} t$
3. $\forall\, j \in [1,n], t \in T \;\; \neg e^{-}_{\triangle,j} \mathbin{@} t \vdash \neg e^{-}_{\sim,j} \mathbin{@} t \vdash e^{-}_{\triangledown,j} \mathbin{@} t$
4. $\forall\, j \in [1,n], t \in T \;\; \neg e^{-}_{\triangledown,j} \mathbin{@} t \vdash e^{-}_{\sim,j} \mathbin{@} t \vdash e^{-}_{\triangle,j} \mathbin{@} t$

La correttezza di queste produzioni può essere verificata interpretando questi archi come $\neg e^{+}_{\triangle,i} \mathbin{@} t = \rho_{\triangle}(\partial^0(\ell^i) \mathbin{@} t)$, $\neg e^{-}_{\triangle,j} \mathbin{@} t = \rho_{\triangle}(\partial^0(\neg\ell^j) \mathbin{@} t)$, $\neg e^{+}_{\triangledown,i} \mathbin{@} t = \rho_{\triangledown}(\partial^0(\neg\ell^i) \mathbin{@} t)$ e $\neg e^{-}_{\triangledown,j} \mathbin{@} t = \rho_{\triangledown}(\partial^0(\ell^j) \mathbin{@} t)$. Ricordando che $Z^{\pm}_{YX,h}$ costituisce una porta di accesso a $Z^{\pm}_{X,h}$, per caratterizzare completamente il comportamento mostrato dal sottografo attraverso l'interfaccia basterà poi scrivere $\forall\, T_{\triangle}, T_{\triangledown} \subseteq T \;\; \widehat{\phi}_{X'}(\{\neg e^{\pm}_{\triangle,h}\} \times T_{\triangle} \cup \{\neg e^{\pm}_{\triangledown,h}\} \times T_{\triangledown}) \cap Z^{\pm}_{XY,h} = \{e^{\pm}_{\triangledown,h}\} \times T_{\triangle} \cup \{e^{\pm}_{\triangle,h}\} \times T_{\triangledown}$.

Il confronto delle precedenti produzioni con $e^{\pm}_{\triangledown,h} \mathbin{@} t \vdash \cdots \vdash \neg e^{\mp}_{\triangledown,k} \mathbin{@} t$ consente infine di concludere:

1. $\forall\, i \in [1,m], j \in [1,n], t \in T \;\; \neg e^{+}_{\triangle,i} \mathbin{@} t \vdash \neg e^{+}_{\sim,i} \mathbin{@} t \vdash \cdots \vdash e^{-}_{\sim,j} \mathbin{@} t \vdash e^{-}_{\triangle,j} \mathbin{@} t$
2. $\forall\, i \in [1,m], j \in [1,n], t \in T \;\; \neg e^{-}_{\triangle,j} \mathbin{@} t \vdash \neg e^{-}_{\sim,j} \mathbin{@} t \vdash \cdots \vdash e^{+}_{\sim,i} \mathbin{@} t \vdash e^{+}_{\triangle,i} \mathbin{@} t$

Questo risultato implementa per via transitiva le regole di inferenza trasversali. Per eliminare il ritardo nullo basterà infatti rimuovere i nodi $w^{\pm}_{\triangle,h}, w^{\pm}_{\triangledown,h}$, sostituendo agli archi $e^{\pm}_{\triangle,h}, \neg e^{\pm}_{\triangle,h}$, $e^{\pm}_{\sim,h}, \neg e^{\pm}_{\sim,h}$ ed $e^{\pm}_{\triangledown,h}, \neg e^{\pm}_{\triangledown,h}$ la coppia $e^{\pm}_{\approx,h}, \neg e^{\pm}_{\approx,h}$ con $e^{\pm}_{\approx,h} = (\beta(w^{\pm}_{\triangledown,h}), \beta(w^{\pm}_{\triangle,h}))$. Da un punto di vista operazionale, le storie negate degli archi $e^{\pm}_{\sim,h}$ o $e^{\pm}_{\approx,h}$ possono a tutti gli effetti essere considerate le nuove rappresentazioni delle storie dei letterali sostituiti ed anche per le seconde vale $\neg e^{\pm}_{\approx,h} \mathbin{@} t \vdash \cdots \vdash e^{\mp}_{\approx,k} \mathbin{@} t$.

Alla luce di questa osservazione, non è difficile determinare come possibili scelte per il nuovo insieme di conoscenza a priori $\Omega'_\ell = \{\neg e^{+}_{\sim,1} \mathbin{@} t^+, \neg e^{-}_{\sim,1} \mathbin{@} t^- \in Z_D \mid \ell \mathbin{@} t^+ \in \mathcal{H}_\ell, (\neg\ell) \mathbin{@} t^- \in \mathcal{H}_\ell\}$ oppure $\Omega''_\ell = \{\neg e^{+}_{\approx,1} \mathbin{@} t^+, \neg e^{-}_{\approx,1} \mathbin{@} t^- \in Z_D \mid \ell \mathbin{@} t^+ \in \mathcal{H}_\ell, (\neg\ell) \mathbin{@} t^- \in \mathcal{H}_\ell\}$. Dato che l'eventuale presenza di un insieme $\Omega_X = \Omega_\ell \cap Z^{\pm}_{X,h} \neq \emptyset$ non è visibile attraverso l'interfaccia prima della sostituzione, la scelta $\Omega'_\ell = \emptyset$ sarebbe stata sufficiente a soddisfare le ipotesi del teorema ma avrebbe impedito l'applicazione delle regole di inferenza trasversali agli ingressi.

Ragionando analogamente a quanto fatto per gli insiemi di ingresso $\Omega_\ell = \rho_{\triangle}(\mathcal{H}_\ell) \subseteq I \times T$, si possono definire gli insiemi di uscita $\Upsilon_\ell = \rho_{\triangle}(\mathcal{K}_\ell) \subseteq O \times T$ come $\Upsilon_\ell = \{\pi^{+}_{\triangle,1} \mathbin{@} t^+, \pi^{-}_{\triangle,1} \mathbin{@} t^- \in Z_D \mid \neg(\ell) \mathbin{@} t^+ \in \mathcal{K}_\ell, \neg(\neg\ell) \mathbin{@} t^- \in \mathcal{K}_\ell\}$ ed applicando le sostituzioni appena viste si ottengono i relativi $\Upsilon'_\ell = \{e^{+}_{\sim,1} \mathbin{@} t^+, e^{-}_{\sim,1} \mathbin{@} t^- \in Z_D \mid \neg(\ell) \mathbin{@} t^+ \in \mathcal{K}_\ell, \neg(\neg\ell) \mathbin{@} t^- \in \mathcal{K}_\ell\}$ e $\Upsilon''_\ell = \{e^{+}_{\approx,1} \mathbin{@} t^+, e^{-}_{\approx,1} \mathbin{@} t^- \in Z_D \mid \neg(\ell) \mathbin{@} t^+ \in \mathcal{K}_\ell, \neg(\neg\ell) \mathbin{@} t^- \in \mathcal{K}_\ell\}$.

Per applicare questo procedimento a tutti i letterali che compaiono nella specifica conviene introdurre l'insieme delle formule coinvolte:

$$\mathcal{A}_G = \{\bowtie G\} \cup \bigcup_{\ell \in \mathcal{L}_G} \{\bowtie J_\ell\}$$

dove

$$\mathcal{L}_G = \{\ell \in \mathcal{L} \mid [G]_{\ell^i \to \star} \neq G, i \in \mathbf{N}\} = \{\ell \in \mathcal{L} \mid [G]_{\neg\ell^i \to \star} \neq G, i \in \mathbf{N}\}$$

per l'ipotesi $m, n \geq 1$. Si può allora immaginare di mappare ogni elemento di questo insieme attraverso una diversa funzione v e di costruire la rete complessiva applicando il teorema di sostituzione all'unione delle reti rappresentanti le singole specifiche. Questo procedimento non viene illustrato perché troppo ricco di indici e banale estensione di quanto appena visto per un singolo $\ell \in \mathcal{L}_G$.

Prima di rappresentare le formule di $\mathcal{A}_G$ è però opportuno operarne una riscrittura secondo:

1. $\forall\, \tilde{\ell} \in \tilde{\mathcal{L}} \quad \bowtie \tilde{\ell} \rightarrow \bowtie \tilde{\ell}$
2. $\forall\, A, B \in \mathcal{G} \quad \bowtie (A \vee B) \rightarrow \bowtie (A \vee B)$
3. $\forall\, A, B \in \mathcal{G} \quad \bowtie (A \wedge B) \rightarrow \bowtie A, \bowtie B$
4. $\forall\, A \in \mathcal{G}, k \in \mathbf{Z} \quad \bowtie (\partial^k A) \rightarrow \bowtie A$

fino ad individuare un insieme invariante $\mathcal{A}_G^{\vee}$ che conserva la semantica di $\mathcal{A}_G$. Per semplicità si suppone inoltre che $\forall\, \tilde{\ell} \in \tilde{\mathcal{L}} \quad \bowtie \tilde{\ell} \notin \mathcal{A}_G^{\vee}$, dato che se così non fosse si potrebbe semplificare la specifica $\bowtie G$ basandosi sulla conoscenza a priori della verità di $\tilde{\ell}$ su tutto l'orizzonte. Assumendo questa ipotesi si può affermare che ogni elemento di $\mathcal{A}_G^{\vee}$ ha la forma $\bowtie (A \vee B)$, cioè quella di un'implicazione.

Risulta allora agevolata un'ulteriore applicazione del teorema di sostituzione per eliminare tutti i nodi $v_{\bowtie}$ soddisfacenti $\alpha(v_{\bowtie}) = \bowtie$ ed il relativo insieme di conoscenza iniziale. Chiamando v_o il nodo soddisfacente $\beta(v_o) = v_{\bowtie}$ e supponendo di aver già inserito gli opportuni ritardi nulli $w_l, w_r \in W_{\partial^0}$ con $\beta(w_l) = \beta(w_r) = v_o$ a monte dei rispettivi figli v_l, v_r con $\beta(v_l) = w_l$ e $\beta(v_r) = w_r$, si definiscono gli archi $e_{\bowtie} = (v_{\bowtie}, v_o), e_l = (w_l, v_o), e_r = (w_r, v_o), \pi_l = (v_l, w_l), \pi_r = (v_r, w_r) \in E$.

Considerando allora il taglio che racchiude i quattro nodi $v_{\bowtie}, v_o, w_l, w_r$, si identifica in $F = \{e_{\bowtie}, \neg e_{\bowtie}, e_l, \neg e_l, e_r, \neg e_r\}$ l'insieme degli archi inclusi dal taglio, mentre l'insieme degli archi tagliati risulta $E_{\cap} = \{\pi_l, \neg\pi_l, \pi_r, \neg\pi_r\}$ ed individua l'interfaccia $Z_{\cap} = E_{\cap} \times T$ tra $Z_X = (E_{\cap} \cup F) \times T$ e $Z_Y = (E \backslash F) \times T$. Anche in questo caso, il partizionamento $Z_{\cap} = Z_{XY} \cup Z_{YX} = (E_{XY} \cup E_{YX}) \times T$, con $E_{XY} = \{\neg\pi_l, \neg\pi_r\}$ ed $E_{YX} = \{\pi_l, \pi_r\}$, conduce a riconoscere in Z_{YX} una porta di accesso a Z_X poiché $v_l \neq v_r$.

Dato che $\alpha(v_o) = \vee$ per ipotesi, tutta l'inferenza potenzialmente compiuta dal sottografo candidato alla sostituzione si riassume nelle produzioni:

1. $\forall\, t \in T \quad e_{\bowtie} \,@\, t, \pi_l \,@\, t \vdash e_{\bowtie} \,@\, t, e_l \,@\, t \vdash \neg e_r \,@\, t \vdash \neg\pi_r \,@\, t$
2. $\forall\, t \in T \quad e_{\bowtie} \,@\, t, \pi_r \,@\, t \vdash e_{\bowtie} \,@\, t, e_r \,@\, t \vdash \neg e_l \,@\, t \vdash \neg\pi_l \,@\, t$

Chiamando $\Omega_X = \rho(\mathcal{H}_{A \vee B}) = \{e_{\bowtie}\} \times T$, se $\Omega_X \subseteq \Omega_0$ si può concludere che $\pi_l \,@\, t \vdash \cdots \vdash \neg\pi_r \,@\, t$ e $\pi_r \,@\, t \vdash \cdots \vdash \neg\pi_l \,@\, t$ su tutto l'orizzonte T. In breve, il comportamento mostrato dal sottografo attraverso l'interfaccia è completamente determinato dall'espressione $\forall\, T_l, T_r \subseteq T \;\; \widehat{\phi}_X(\Omega_X \cup \{\pi_l\} \times T_l \cup \{\pi_r\} \times T_r) \cap Z_{XY} = \{\neg\pi_r\} \times T_l \cup \{\neg\pi_l\} \times T_r$.

Chiamando $\pi_{\sim} = (w_l, w_r)$, si può allora operare la sostituzione:

1. $V' = V \setminus \{v_{\bowtie}, v_o\}$
2. $E' = E \setminus \{e_{\bowtie}, \neg e_{\bowtie}, e_l, \neg e_l, e_r, \neg e_r\} \cup \{\pi_{\sim}, \neg\pi_{\sim}\}$
3. $\alpha' = \alpha \setminus \{(v_{\bowtie}, \bowtie), (v_o, \vee)\}$
4. $\beta' = \beta \setminus \{\neg e_{\bowtie}, e_l, e_r\} \cup \{\pi_{\sim}, \neg\pi_{\sim}\}$

che implementa le produzioni equivalenti:

1. $\forall\, t \in T \quad \pi_l \,@\, t \vdash \pi_{\sim} \,@\, t \vdash \neg\pi_r \,@\, t$
2. $\forall\, t \in T \quad \pi_r \,@\, t \vdash \neg\pi_{\sim} \,@\, t \vdash \neg\pi_l \,@\, t$

e verifica $\forall T_l, T_r \subseteq T \;\; \widehat{\phi}_{X'}(\{\pi_l\} \times T_l \cup \{\pi_r\} \times T_r) \cap Z_{XY} = \{\neg\pi_r\} \times T_l \cup \{\neg\pi_l\} \times T_r$, ovvero $\forall T_l, T_r \subseteq T \;\; \widehat{\phi}_X(\Omega_X \cup \{\pi_l\} \times T_l \cup \{\pi_r\} \times T_r) \cap Z_{XY} = \widehat{\phi}_{X'}(\{\pi_l\} \times T_l \cup \{\pi_r\} \times T_r) \cap Z_{XY}$. L'insieme di conoscenza iniziale che soddisfa le ipotesi del teorema di sostituzione sarà dunque individuato da $\Omega_{X'} = \emptyset$, da cui $\Omega'_0 = \Omega_0 \setminus \Omega_X$. Al solito, la successiva rimozione dei nodi w_l e w_r porterà alla cancellazione degli archi $e_l, \neg e_l$, $e_r, \neg e_r$ e $\pi_\sim, \neg\pi_\sim$ ed alla creazione della nuova coppia di archi $\pi_\approx, \neg\pi_\approx$, con $\pi_\approx = (v_l, v_r)$.

Se non fosse stata operata la riscrittura che garantisce $\alpha(v_o) = \vee$ sarebbe stato comunque possibile applicare ricorsivamente il teorema di sostituzione anche ai casi $\alpha(v_o) \in \{\wedge, \partial^k\}$, ottenendo risultati analoghi. La figura 3.12 illustra le tre coppie di circuiti equivalenti che possono essere utilizzate allo scopo, con i tagli che ne individuano l'interfaccia evidenziati in tratteggio. La coppia centrale corrisponde al caso appena formalizzato, dove ai due componenti di partenza viene sostituito l'arco $\pi_\approx$.

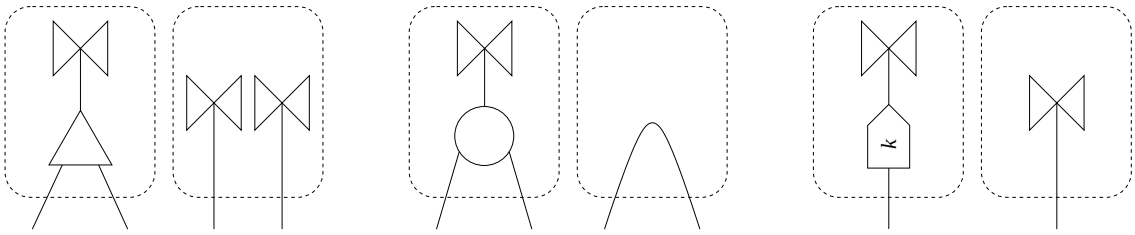

Figura 3.12. Le tre coppie di circuiti equivalenti utili nella sostituzione dei componenti *root*

Applicando questa sostituzione alla radice del sottografo associato ad una formula $\bowtie J_\ell \in \mathcal{A}^\vee_G$ che sia già stata collegata al resto della rete tramite gli opportuni archi $e^\pm_{\sim,h}$ si ottiene un risultato di notevole interesse. In questo caso si identificano in $\pi_l \,@\, t = \pi^+_{\triangledown,\wedge} \,@\, t$ e $\pi_r \,@\, t = \pi^-_{\triangledown,\wedge} \,@\, t$ le rappresentazioni delle formule $\neg(\bigwedge_{i=1}^m \neg\ell^i) \,@\, t$ e $\neg(\bigwedge_{j=1}^n \ell^j) \,@\, t$ e per quanto già dimostrato vale:

1. $\forall\, i \in [1,m], t \in T \;\; \neg\pi^+_{\triangledown,\wedge} \,@\, t \vdash \cdots \vdash \neg e^+_{\triangledown,i} \,@\, t \vdash e^+_{\sim,i} \,@\, t$
2. $\forall\, i \in [1,m], t \in T \;\; \neg e^+_{\sim,i} \,@\, t \vdash e^+_{\triangledown,i} \,@\, t \vdash \cdots \vdash \pi^+_{\triangledown,\wedge} \,@\, t$
3. $\forall\, j \in [1,n], t \in T \;\; \neg\pi^-_{\triangledown,\wedge} \,@\, t \vdash \cdots \vdash \neg e^-_{\triangledown,j} \,@\, t \vdash e^-_{\sim,j} \,@\, t$
4. $\forall\, j \in [1,n], t \in T \;\; \neg e^-_{\sim,j} \,@\, t \vdash e^-_{\triangledown,j} \,@\, t \vdash \cdots \vdash \pi^-_{\triangledown,\wedge} \,@\, t$

quindi nel complesso si verifica:

1. $\forall\, i \in [1,m], j \in [1,n], t \in T \;\; \neg e^+_{\sim,i} \,@\, t \vdash \cdots \vdash \pi^+_{\triangledown,\wedge} \,@\, t \vdash \pi_\sim \,@\, t \vdash \neg\pi^-_{\triangledown,\wedge} \,@\, t \vdash \cdots \vdash e^-_{\sim,j} \,@\, t$
2. $\forall\, i \in [1,m], j \in [1,n], t \in T \;\; \neg e^-_{\sim,j} \,@\, t \vdash \cdots \vdash \neg\pi^-_{\triangledown,\wedge} \,@\, t \vdash \neg\pi_\sim \,@\, t \vdash \pi^+_{\triangledown,\wedge} \,@\, t \vdash \cdots \vdash e^+_{\sim,i} \,@\, t$

Rimuovendo tutti i ritardi nulli si ottiene infine:

1. $\forall\, i \in [1,m], j \in [1,n], t \in T \;\; \neg e^+_{\approx,i} \,@\, t \vdash \cdots \vdash \pi_\approx \,@\, t \vdash \cdots \vdash e^-_{\approx,j} \,@\, t$
2. $\forall\, i \in [1,m], j \in [1,n], t \in T \;\; \neg e^-_{\approx,j} \,@\, t \vdash \cdots \vdash \neg\pi_\approx \,@\, t \vdash \cdots \vdash e^+_{\approx,i} \,@\, t$

Il legame tra premesse e conclusioni di tutte queste ultime produzioni era già noto, ma questa scrittura mette in evidenza il particolare ruolo giocato dagli archi $\pi_\sim$ o $\pi_\approx$, illustrato di seguito per il primo. Considerando il partizionamento $V_X = V_\triangledown \cup \bigcup_{i=1}^m \{w^+_{\triangle,i}\} \cup \bigcup_{j=1}^n \{w^-_{\triangle,j}\}$, $V_Y = V \setminus V_X$ ed relativi $E_X = E_\cap \cup F$, $E_Y = E_\triangle$ con $E_\cap = \bigcup_{i=1}^m \{e^+_{\triangle,i}, \neg e^+_{\triangle,i}\} \cup \bigcup_{j=1}^n \{e^-_{\triangle,j}, \neg e^-_{\triangle,j}\}$ e $F = E_\triangledown \cup \bigcup_{i=1}^m \{e^+_{\sim,i}, \neg e^+_{\sim,i}\} \cup \bigcup_{j=1}^n \{e^-_{\sim,j}, \neg e^-_{\sim,j}\}$, si applica il teorema di sostituzione in una forma degenere a $Z_X = E_X \times T$, senza cioè modificare l'insieme di rappresentazione Z_X ma

solo l'insieme di conoscenza a priori $\Omega_X \subseteq F \times T$ che garantisce l'uguaglianza delle chiusure all'interfaccia $Z_\cap = E_\cap \times T$ con $Z_Y = E_Y \times T$.

Tutti i componenti della rete generata dalla specifica $\bowtie J_\ell$ che non sono stati rimossi durante la precedente sostituzione soddisfano $\forall\, r, s \in Z_X \;\; g(r,s) = \xi(\{r\},\{s\}) = \emptyset$ ed il loro comportamento è dunque lineare, ovvero soddisfacente $\forall\, R, S \in Z_X \;\; \widehat{\phi}_X(R \cup S) = \widehat{\phi}_X(R) \cup \widehat{\phi}_X(S)$. In particolare, se $\widehat{\phi}_X(\Omega_X) \cap Z_\cap = \widehat{\phi}_X(\Omega'_X) \cap Z_\cap$ si verifica $\forall\, S_\cap \subseteq Z_\cap \;\; \widehat{\phi}_X(\Omega_X \cup S_\cap) \cap Z_\cap = \widehat{\phi}_X(\Omega_X) \cap Z_\cap \cup \widehat{\phi}_X(S_\cap) \cap Z_\cap = \widehat{\phi}_X(\Omega'_X) \cap Z_\cap \cup \widehat{\phi}_X(S_\cap) \cap Z_\cap = \widehat{\phi}_X(\Omega'_X \cup S_\cap) \cap Z_\cap$ e le ipotesi del teorema di sostituzione sono soddisfatte. In particolare, ogni insieme $\Omega_\ell = \{\neg e^+_{\sim,1} @ t^+, \neg e^-_{\sim,1} @ t^- \in Z_D \mid \ell @ t^+ \in \mathcal{H}_\ell, (\neg\ell) @ t^- \in \mathcal{H}_\ell\}$ che contiene la conoscenza a priori della storia di un ingresso $\ell \in \mathcal{L}$ verrà sostituito da un insieme $\Omega'_\ell = \{\pi_\sim @ t^+, \neg\pi_\sim @ t^- \in Z_D \mid \ell @ t^+ \in \mathcal{H}_\ell, (\neg\ell) @ t^- \in \mathcal{H}_\ell\}$, così da verificare $\widehat{\phi}_X(\Omega_\ell) \cap Z_\cap = \{e^-_{\triangle,j} @ t^+, e^+_{\triangle,i} @ t^- \in Z_D \mid \ell @ t^+ \in \mathcal{H}_\ell, (\neg\ell) @ t^- \in \mathcal{H}_\ell\} = \widehat{\phi}_X(\Omega'_\ell) \cap Z_\cap$ con $\Omega_\ell, \Omega'_\ell \subseteq Z_X \setminus Z_Y$.

Questo restituisce unicità alla rappresentazione della funzione ℓ, proprietà che era stata distrutta dal processo di ridenominazione dei letterali, e conduce ad una più elegante scelta degli insiemi di uscita, precedentemente definiti come $\Upsilon_\ell = \{e^+_{\sim,1} @ t^+, e^-_{\sim,1} @ t^- \in Z_D \mid \neg(\ell) @ t^+ \in \mathcal{K}_\ell, \neg(\neg\ell) @ t^- \in \mathcal{K}_\ell\}$, secondo $\Upsilon'_\ell = \{\neg\pi_\sim @ t^+, \pi_\sim @ t^- \in Z_D \mid \neg(\ell) @ t^+ \in \mathcal{K}_\ell, \neg(\neg\ell) @ t^- \in \mathcal{K}_\ell\}$. Si può infatti dimostrare che $\forall\, S_\cap \subseteq Z_\cap, \forall\, t^+, t^- \in T \;\; e^+_{\sim,1} @ t^+, e^-_{\sim,1} @ t^- \in \widehat{\phi}_X(\Omega'_X \cup S_\cap) \iff \neg\pi_\sim @ t^+, \pi_\sim @ t^- \in \widehat{\phi}_X(\Omega'_X \cup S_\cap)$, a garanzia di equivalenza tra le due definizioni. Naturalmente, dopo la rimozione dei ritardi nulli si individueranno in $\Omega''_\ell = \{\pi_\approx @ t^+, \neg\pi_\approx @ t^- \in Z_D \mid \ell @ t^+ \in \mathcal{H}_\ell, (\neg\ell) @ t^- \in \mathcal{H}_\ell\}$ e $\Upsilon''_\ell = \{\neg\pi_\approx @ t^+, \pi_\approx @ t^- \in Z_D \mid \neg(\ell) @ t^+ \in \mathcal{K}_\ell, \neg(\neg\ell) @ t^- \in \mathcal{K}_\ell\}$ i nuovi insiemi di ingresso ed uscita soddisfacenti le medesime proprietà.

Si giunge così ad identificare negli archi $\pi_\sim$ o $\pi_\approx$ una più conveniente rappresentazione delle funzioni $\ell \in \mathcal{L}$, siano esse ingressi o uscite di interesse, ed omettendo gli apici che distinguono i passi della sostituzione si adotta come nuova definizione di Ω''_ℓ e Υ''_ℓ la seguente:

$$\Omega_\ell = \Upsilon_\ell = \{\pi_\approx @ t, \neg\pi_\approx @ \bar{t} \in Z_D \mid \ell|_t = \top, \ell|_{\bar{t}} = \bot\}$$

In pratica, ad ogni coppia di archi $\pi_\approx, \neg\pi_\approx \in I$ verrà applicato un ingresso mentre l'esecutore produrrà in uscita la storia di ogni coppia $\pi_\approx, \neg\pi_\approx \in O$. Da un punto di vista grafico, entrambe verranno etichettate con il nome della variabile associata ed una freccia a metà lunghezza orientata nel verso corrispondente al valore logico $\top$. Infine, è interessante notare che se una funzione ℓ compare in una singola coppia di letterali $\ell^1, \neg\ell^1$, la composizione di queste sostituzioni degenera nel semplice collegamento dei terminali che si ottengono eliminando i nodi corrispondenti ai due letterali. In questo caso, l'arco di collegamento è il $\pi_\approx$ ($\neg\pi_\approx$) di $\bowtie J_\ell$.

Esempio Considerando la ormai familiare funzione up_ℓ, si identificano in $\bowtie J_{up_\ell} = \bowtie(\neg up^1_\ell \vee up^1_\ell)$ e $\bowtie J_\ell = \bowtie((\neg\ell^1 \wedge \neg\ell^2) \vee (\ell^1 \wedge \ell^2))$ le due specifiche di collegamento associate alle funzioni che compaiono nell'albero sintattico di $\bowtie G = \bowtie((\neg up^1_\ell \vee (\partial(\neg\ell^1) \wedge \ell^1)) \wedge (up^1_\ell \vee (\partial\,\ell^2 \vee \neg\ell^2)))$. L'insieme $\mathcal{A}_G = \{\bowtie G, \bowtie J_{up_\ell}, \bowtie J_\ell\}$ è riportato in figura 3.13 sotto forma di rete equivalente, con i tagli prescelti per la sostituzione evidenziati in tratteggio. Per semplicità grafica si suppone di procedere in ordine inverso rispetto a quanto teorizzato, operando cioè prima la rimozione delle radici e solo successivamente delle foglie, e di omettere le riscritture che generano $\mathcal{A}^\vee_G = \{\bowtie(\neg up^1_\ell \vee (\partial(\neg\ell^1) \wedge \ell^1)), \bowtie(up^1_\ell \vee (\partial\,\ell^2 \vee \neg\ell^2)), \bowtie J_{up_\ell}, \bowtie J_\ell\}$ in favore di una più flessibile applicazione del teorema di sostituzione alla sottorete generata da $\bowtie G$. Rimpiazzando le sottoreti incluse dai tagli con gli opportuni archi di collegamento $\pi_\approx$ si ottiene la rete di figura 3.14, dove le freccie a metà lunghezza evidenziano gli archi associati alle funzioni up_ℓ e ℓ. Accoppiando poi le foglie come mostrato in figura 3.15 e scegliendo come nuovi tagli quelli in tratteggio si può sostituire ad ogni coppia di foglie un arco di collegamento $e^\pm_\approx$ e giungere al risultato riportato nella parte destra della stessa figura. La rete così ottenuta è proprio quella già proposta in figura 3.8 come definizione alternativa di up_ℓ.

Esempio Applicando le precedenti trasformazioni all'albero sintattico di definizione della funzione $until_{a,b}$, riportato in figura 3.2, si ottiene la rete mostrata nell'angolo in alto a sinistra

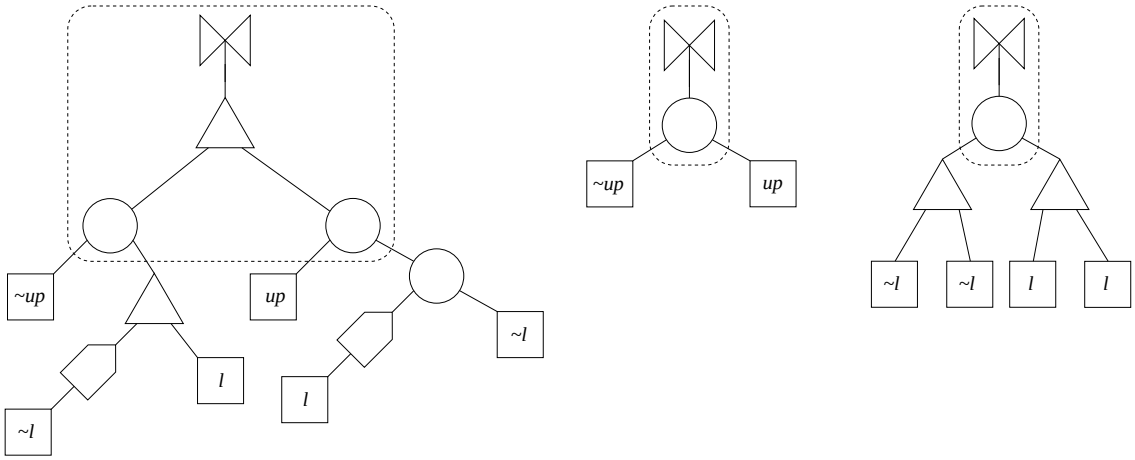

Figura 3.13. Rete di definizione della funzione up_ℓ con relative reti di collegamento

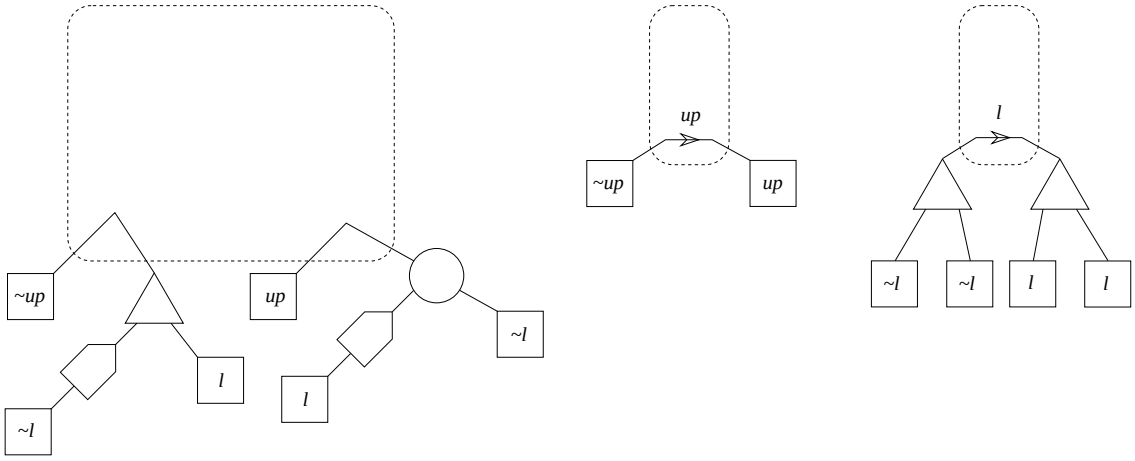

Figura 3.14. Esempio di sostituzione dei componenti *root*

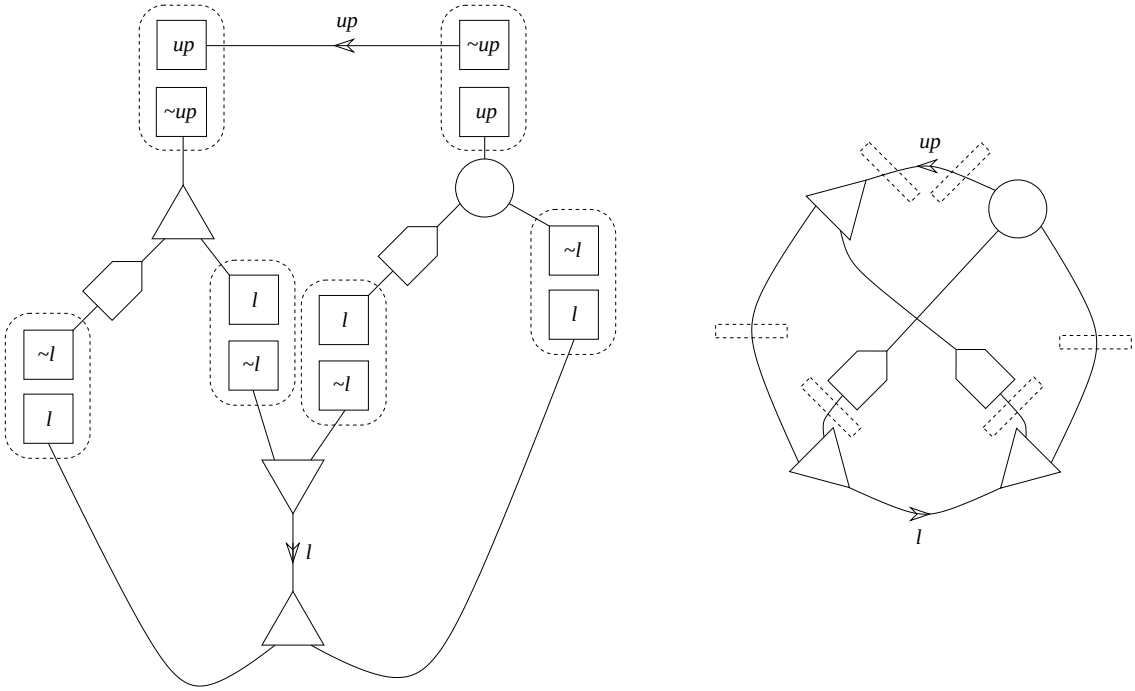

Figura 3.15. Esempio di sostituzione dei componenti *leaf*

di figura 3.16. Alla sua destra compare una possibile inizializzazione all'istante t e, in basso, la corrispondente chiusura agli istanti t e $t+1$. Interpretando l'orientazione degli archi in termini di formule logiche si verifica la capacità della rete di implementare la produzione $until_{a,b} @ t, \neg a @ t \vdash \cdots \vdash b @ t, until_{a,b} @ (t+1)$. Per ottenere la rete corrispondente alla funzione $since_{a,b}$ da quella appena ricavata è sufficiente scambiare i terminali dei due ritardi che vi compaiono e modificare l'etichetta con il nome dell'operatore.

Una volta rimosse radici e foglie, i requisiti caratteristici di una rete di inferenza temporale (V, E, α, β) si semplificano nel seguente modo:

1. $E \subseteq V \times V \mid \forall (v_i, v_j) \in E \;\; (v_j, v_i) \in E$
2. $\alpha \subseteq V \times (\{\wedge, \vee\} \cup \{\partial^k \mid k \in \mathbf{Z}\}) \mid \forall\, v_o \in V \;\; \exists!(v_o, o) \in \alpha$
3. $\beta \subseteq E \mid \forall\, v_o \in V \;\; \exists!(v_o, v_p) \in \beta$
4. $\forall\, v_o \in V \mid \alpha(v_o) \in \{\wedge, \vee\} \;\; \mathrm{card}\{(v_i, v_o) \in E\} = 3$
5. $\forall\, v_o \in V \mid \alpha(v_o) \in \{\partial^k \mid k \in \mathbf{Z}\} \;\; \mathrm{card}\{(v_i, v_o) \in E\} = 2$

Questi vincoli possono quindi essere assunti come definizione alternativa di rete senza perdere in generalità, dato che ogni rete conforme alla precedente definizione può essere manipolata attraverso le sostituzioni mostrate fino a soddisfarli.

3.6 Semplificazioni ed estensioni

Quelle presentate nella sezione precedente sono solo alcune delle possibili semplificazioni permesse dal teorema di sostituzione. La loro applicazione risulta particolarmente proficua se la specifica di partenza ha la forma di un prodotto di somme:

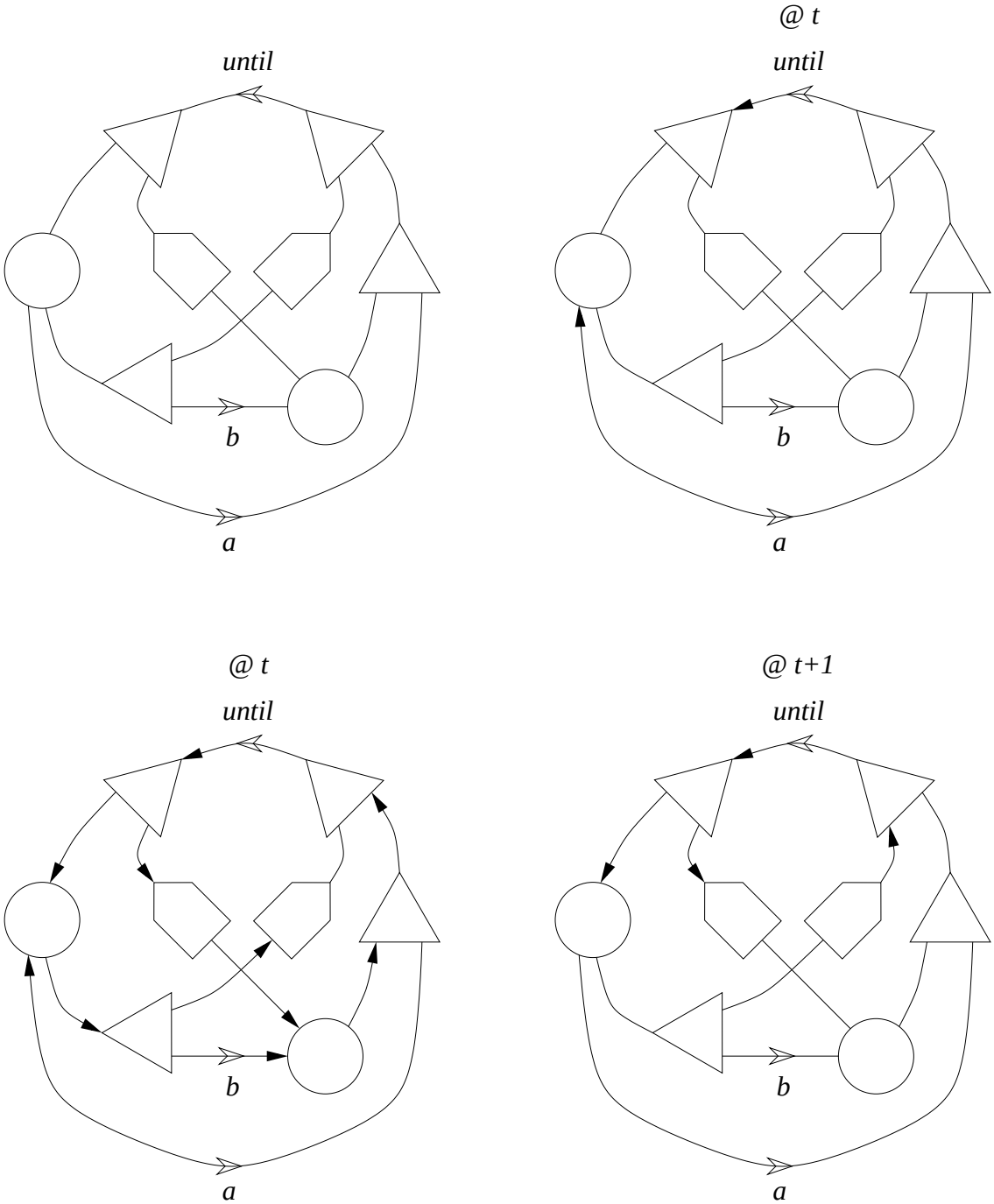

Figura 3.16. Rete di definizione della funzione $until_{a,b}$ ed esempi del suo comportamento

$$\bowtie G = \bowtie(\bigwedge_{i=1}^{m}(\bigvee_{j=1}^{n} \partial^{k_{i,j}}\, \tilde{\ell}_{i,j}))$$

Esempio Un meccanismo di mutua esclusione tra i tre stati $a, b, c \in \mathcal{L}$ può essere specificato in forma di prodotto di somme da $\bowtie((a \vee b \vee c) \wedge (\neg a \vee \neg b) \wedge (\neg b \vee \neg c) \wedge (\neg c \vee \neg a))$. Applicando il teorema di sostituzione alla rete generata dalla specifica si ottiene il risultato che compare in figura 3.17, composto da soli quattro componenti. Se invece fosse stata preferita la somma di prodotti semanticamente equivalente $\bowtie((a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge b \wedge \neg c) \vee (\neg a \wedge \neg b \wedge c))$, la rete risultante sarebbe stata composta da ben dieci componenti. Entrambe queste reti implementano un ugual repertorio di produzioni, elencato di seguito:

1. $\neg b \,@\, t, \neg c \,@\, t \vdash a \,@\, t$
2. $\neg c \,@\, t, \neg a \,@\, t \vdash b \,@\, t$
3. $\neg a \,@\, t, \neg b \,@\, t \vdash c \,@\, t$
4. $a \,@\, t \vdash \cdots \vdash \neg b \,@\, t, \neg c \,@\, t$
5. $b \,@\, t \vdash \cdots \vdash \neg c \,@\, t, \neg a \,@\, t$
6. $c \,@\, t \vdash \cdots \vdash \neg a \,@\, t, \neg b \,@\, t$

Dovendo gestire anche il caso di valore non disponibile, una rete logica convenzionale capace di compiere le stesse inferenze sugli ingressi a_{in}, b_{in}, c_{in} per produrre le uscite $a_{out}, b_{out}, c_{out}$ sarebbe composta da almeno dodici porte di tipo AND o OR a due valori, che si riducono a sei se si immagina di disporre di porte in grado di operare in logica a tre valori. Inoltre dal conteggio delle porte sono esclusi i nodi di diramazione di un segnale, che contribuiscono a rendere più complessa la topologia della rete logica, mentre una rete di inferenza temporale usa a tale scopo i componenti *joint*. Si osserva infine che i componenti delle reti di inferenza temporale hanno un comportamento più semplice ed intuitivo di quello delle porte logiche a tre valori.

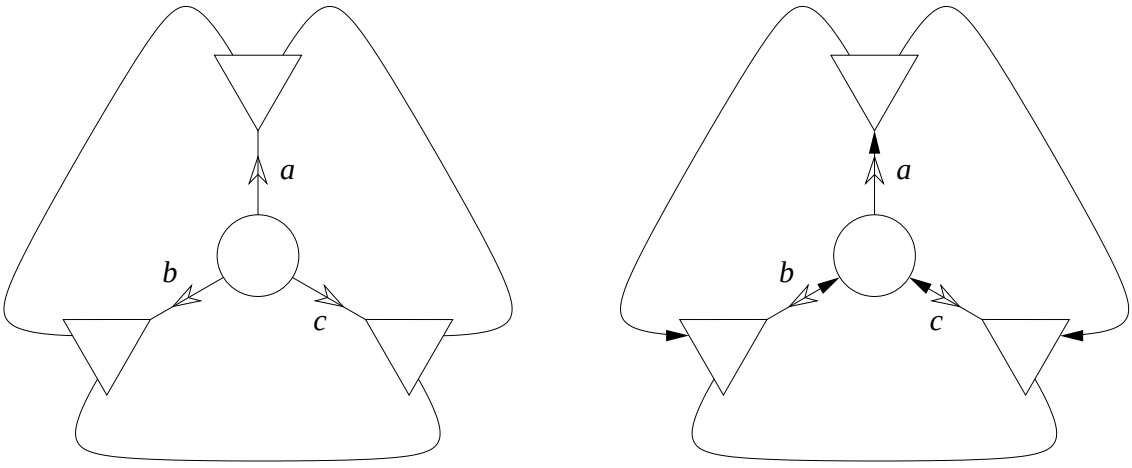

Figura 3.17. Rete di mutua esclusione tra tre stati ed una sua possibile chiusura

Non pare tuttavia opportuno limitarsi ad operare con specifiche in forma di prodotto di somme. Pur essendo sempre possibile ricondursi ad essa, infatti, può non essere conveniente farlo se si desidera minimizzare il numero di componenti della rete associata, proprio come avviene per le reti logiche tradizionali. Inoltre, il metodo proposto risulta vantaggioso anche in quanto non

costringe l'autore della specifica a conformarsi ad una particolare struttura ma lascia libertà di scegliere la forma che di volta in volta appare più espressiva. Sembra dunque maggiormente promettente la ricerca di riscritture o sostituzioni capaci di semplificare la rete con efficacia pari a quelle già presentate, anche quando la specifica di partenza non ha la forma di un prodotto di somme. Come verrà illustrato più avanti, il numero di componenti di una rete determina il minimo passo di discretizzazione capace di garantire il tempo reale e si comprende perché sia così importante mantenerlo limitato.

Durante la ricerca di una semplificazione si può procedere verificando l'equivalenza operazionale tra due circuiti, ma questo non è il solo criterio possibile. Il teorema è infatti applicabile anche quando la sostituzione con un circuito più semplice garantisce un incremento della conoscenza inferibile e si può dimostrare la correttezza della conoscenza in eccesso.

Esempio Dei quattro circuiti riportati in figura 3.18, dove alle funzioni a e b sono stati convenzionalmente associati archi entranti nel taglio, i primi due sono operazionalmente equivalenti in quanto entrambi implementano le produzioni $\forall\, t \in T \;\; a \,@\, t \vdash \cdots \vdash \neg b \,@\, t$ e $\forall\, t \in T \;\; b \,@\, t \vdash \cdots \vdash \neg a \,@\, t$ ed il secondo può quindi essere sostituito dal primo. Il terzo circuito implementa invece solo le produzioni $\forall\, t \in T \;\; b \,@\, t \vdash \cdots \vdash \neg a \,@\, t$, ma per la correttezza della funzione inferenza questo implica $b \,@\, t \Rightarrow \neg a \,@\, t$ e dunque $a \,@\, t \Rightarrow \neg b \,@\, t$. Si può allora sostituire anche questo circuito con il primo, certi della correttezza delle produzioni $\forall\, t \in T \;\; a \,@\, t \vdash \cdots \vdash \neg b \,@\, t$ che vengono così aggiunte a quelle già implementate. Le produzioni effettuate dal quarto circuito, $\forall\, t, t+k \in T \;\; a \,@\, t \vdash \cdots \vdash \neg b \,@\, t$ e $\forall\, t, t+k \in T \;\; b \,@\, t \vdash \cdots \vdash \neg a \,@\, t$, sono vincolate dall'appartenenza di $t+k$ all'orizzonte T ma ciò deriva solo dalla necessità pratica di limitare l'insieme di rappresentazione. Infatti la semantica della BTL è definita sull'intero insieme degli interi $\mathbf{Z}$ e ciò è sufficiente per poter procedere alla medesima sostituzione anche in quest'ultimo caso. I tre circuiti sostituiti possono essere generati da sottoespressioni come $\ell^i \wedge \ell^j$, $\ell^i \vee \ell^j$ o $\partial^{-k}(\partial^k A)$ e negli ultimi due casi la loro rimozione equivale all'implementazione delle produzioni $(\ell^i \vee \ell^j) \,@\, t \vdash \neg(\neg\ell^i \wedge \neg\ell^j) \,@\, t$ o $(\partial^{-k}(\partial^k A)) \,@\, t \vdash A \,@\, t$ per qualunque coppia $t, k \in \mathbf{Z}$, precedentemente non disponibili a causa dell'incompletezza della funzione inferenza.

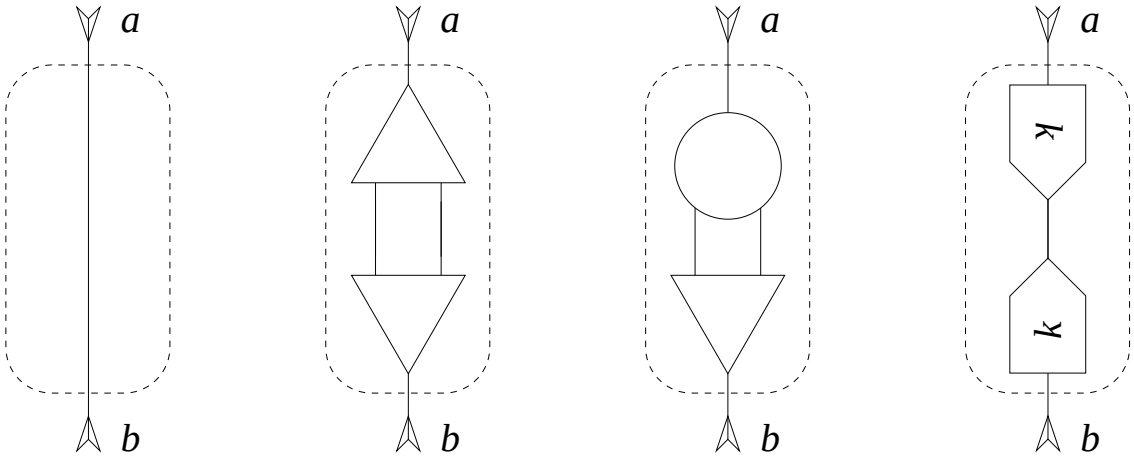

Figura 3.18. Quattro circuiti sottilmente diversi

Data $\bowtie G \in \mathcal{F}$ contenente la sottoespressione $C \in \mathcal{G} \setminus \tilde{\mathcal{L}}$ in più di una occorrenza e scelta una funzione $\ell \in \mathcal{L} \setminus \mathcal{L}_G$, una tecnica sintattica che contribuisce a semplificare la rete associata è la riscrittura della specifica secondo:

$$\bowtie G \rightarrow \bowtie((\neg\ell \vee C) \wedge [G]_{C \rightarrow \ell})$$

a cui seguirà una opportuna ridenominazione delle occorrenze del letterale ℓ, operazione che verrà sempre sottintesa nel seguito. Applicando le specifiche di collegamento a $\bowtie((\neg\ell \vee C) \wedge [G]_{C \rightarrow \ell})$

si implementano le regole di inferenza $\ell \,@\, t \vdash \neg(\neg\ell) \,@\, t \vdash C \,@\, t$ e $\neg C \,@\, t \vdash (\neg\ell) \,@\, t \vdash \neg(\ell) \,@\, t$, sufficienti a garantire la propagazione dei valori logici istantanei $\top$ e $\bot$ nelle direzioni appropriate. Date le clausole $P, B, L, R \in \mathcal{G}$ soddisfacenti $[P]_{C\to\star} \neq P$, $[C]_{P\to\star} = C$, $[P]_{B\to\star} \neq P$, $[B]_{P\to\star} = B$, $[C]_{L\to\star} \neq C$, $[L]_{C\to\star} = L$, $[C]_{R\to\star} \neq C$ e $[R]_{C\to\star} = R$, se è possibile applicare all'albero sintattico di $\bowtie G$ una tra le produzioni $a_1, \ldots, a_{n-1} \vdash a_n$:

1. $P \,@\, \tau \vdash C \,@\, t$
2. $P \,@\, t, \neg B \,@\, t \vdash C \,@\, t$
3. $C \,@\, t \vdash L \,@\, \tau$
4. $C \,@\, t, \neg R \,@\, t \vdash L \,@\, t$
5. $P \,@\, t, \neg C \,@\, t \vdash B \,@\, t$
6. $\neg C \,@\, t \vdash \neg P \,@\, \tau$
7. $\neg C \,@\, t, \neg B \,@\, t \vdash \neg P \,@\, t$
8. $\neg L \,@\, \tau \vdash \neg C \,@\, t$
9. $\neg L \,@\, t, \neg R \,@\, t \vdash \neg C \,@\, t$

allora è possibile applicare all'albero sintattico di $\bowtie((\neg\ell \vee C) \wedge [G]_{C\to\ell})$ la produzione o sequenza di produzioni corrispondente $b_1, \ldots, b_{n-1} \vdash \cdots \vdash b_n$:

1. $[P]_{C\to\ell} \,@\, \tau \vdash \ell \,@\, t$
2. $[P]_{C\to\ell} \,@\, t, \neg[B]_{C\to\ell} \,@\, t \vdash \ell \,@\, t$
3. $\ell \,@\, t \vdash \neg(\neg\ell) \,@\, t \vdash C \,@\, t \vdash L \,@\, \tau$
4. $\ell \,@\, t, \neg R \,@\, t \vdash \neg(\neg\ell) \,@\, t, \neg R \,@\, t \vdash C \,@\, t, \neg R \,@\, t \vdash L \,@\, t$
5. $[P]_{C\to\ell} \,@\, t, \neg(\ell) \,@\, t \vdash [B]_{C\to\ell} \,@\, t$
6. $\neg(\ell) \,@\, t \vdash \neg[P]_{C\to\ell} \,@\, \tau$
7. $\neg(\ell) \,@\, t, \neg[B]_{C\to\ell} \,@\, t \vdash \neg[P]_{C\to\ell} \,@\, t$
8. $\neg L \,@\, \tau \vdash \neg C \,@\, t \vdash (\neg\ell) \,@\, t \vdash \neg(\ell) \,@\, t$
9. $\neg L \,@\, t, \neg R \,@\, t \vdash \neg C \,@\, t \vdash (\neg\ell) \,@\, t \vdash \neg(\ell) \,@\, t$

Si osserva che ogni termine $b_k, k \in [1, n]$ può essere ottenuto riscrivendo il relativo a_k secondo $A_k \,@\, t_k \to [A_k]_{C\to\ell} \,@\, t_k$ o $\neg A_k \,@\, t_k \to \neg[A_k]_{C\to\ell} \,@\, t_k$, visto che $[L]_{C\to\star} = L$ e $[R]_{C\to\star} = R$ per ipotesi. Per quanto riguarda tutte le altre produzioni possibili, C non vi compare o vi compare solo come sottoespressione di una opportuna P e si verifica che se è possibile applicare a $\bowtie G$ una produzione $a_1, \ldots, a_{n-1} \vdash a_n$, con $a_k \notin \{C \,@\, t, \neg C \,@\, t\}, k \in [1, n]$, allora è possibile applicare a $\bowtie((\neg\ell \vee C) \wedge [G]_{C\to\ell})$ la produzione $b_1, \ldots, b_{n-1} \vdash b_n$ ottenuta attraverso la regola di riscrittura $a_k \to b_k$.

Si può dunque concludere che la riscrittura delle conclusioni di una generica produzione su $\bowtie G$ sono ottenibili anche operando su $\bowtie((\neg\ell \vee C) \wedge [G]_{C\to\ell})$ a partire dalla riscrittura delle sue premesse, direttamente come nell'ultimo caso discusso o per via transitiva come in alcuni dei casi precedenti. Operando la chiusura della riscrittura della conoscenza disponibile in base alla formula riscritta si produrrà allora la riscrittura di tutti gli eventi che sarebbero stati prodotti operando la chiusura della conoscenza disponibile in base alla formula di partenza, più alcuni della forma $(\neg\ell) \,@\, t$, $\neg(\neg\ell) \,@\, t$, $C \,@\, t$ o $\neg C \,@\, t$.

Una tecnica simile può essere impiegata anche quando la sottoespressione C è unica, ma è retta da un operatore la cui riscrittura in BTL è possibile solo in forma ricorsiva. Per tradurre

l'operatore mancante si sostituisce ad ogni occorrenza di C un letterale ℓ estraneo alla specifica e si congiunge il risultato con $\neg\ell \vee C'$, con C' soddisfacente $\forall\, t \in \mathbf{Z}\ \ C \,@\, t \iff C' \,@\, t$, secondo $\bowtie G \rightarrow \bowtie((\neg\ell \vee C') \wedge [G]_{C \rightarrow \ell})$. Oltre alle sottoespressioni di C che costituiscono gli argomenti dell'operatore tradotto, la clausola C' conterrà tutti i riferimenti al letterale ℓ richiesti dalla definizione ricorsiva.

Per illustrare l'uso di queste due tecniche ed estendere la BTL con costrutti più potenti si introducono gli insiemi:

$$\mathcal{G}^{\diamond} = \{C ::= \tilde{\ell} \parallel A \wedge B \parallel A \vee B \parallel \partial^k A \parallel A \,@[x,y] \parallel A \,?[x,y] \parallel \mathrm{until}(A,B) \parallel \mathrm{since}(A,B) \parallel \neg\,\mathrm{until}(\neg A, \neg B) \parallel \neg\,\mathrm{since}(\neg A, \neg B) \parallel (A) \mid \tilde{\ell} \in \tilde{\mathcal{L}}, A, B \in \mathcal{G}^{\diamond}, k, x, y \in \mathbf{Z}, x < y\}$$

$$\mathcal{F}^{\diamond} = \{F ::= C \,@\, t \parallel \neg C \,@\, t \parallel \bowtie C \mid C \in \mathcal{G}^{\diamond}, t \in \mathbf{Z}\}$$

La semantica dei nuovi operatori, già anticipata negli esempi della sezione 3.1, può essere così riassunta:

1. $\forall\, A \in \mathcal{G}^{\diamond}, x, y, t \in \mathbf{Z}, x < y\ \ (A \,@[x,y]) \,@\, t \iff \bigwedge_{\tau=x}^{y} A \,@(t+\tau)$
2. $\forall\, A \in \mathcal{G}^{\diamond}, x, y, t \in \mathbf{Z}, x < y\ \ (A \,?[x,y]) \,@\, t \iff \bigvee_{\tau=x}^{y} A \,@(t+\tau)$
3. $\forall\, A, B \in \mathcal{G}^{\diamond}, t \in \mathbf{Z}\ \ \mathrm{until}(A,B) \,@\, t \iff A \,@\, t \vee (B \,@\, t \wedge \mathrm{until}(A,B) \,@(t+1))$
4. $\forall\, A, B \in \mathcal{G}^{\diamond}, t \in \mathbf{Z}\ \ \mathrm{since}(A,B) \,@\, t \iff A \,@\, t \vee (B \,@\, t \wedge \mathrm{since}(A,B) \,@(t-1))$
5. $\forall\, A, B \in \mathcal{G}^{\diamond}, t \in \mathbf{Z}\ \ (\neg\,\mathrm{until}(\neg A, \neg B)) \,@\, t \iff A \,@\, t \wedge (B \,@\, t \vee (\neg\,\mathrm{until}(\neg A, \neg B)) \,@(t+1))$
6. $\forall\, A, B \in \mathcal{G}^{\diamond}, t \in \mathbf{Z}\ \ (\neg\,\mathrm{since}(\neg A, \neg B)) \,@\, t \iff A \,@\, t \wedge (B \,@\, t \vee (\neg\,\mathrm{since}(\neg A, \neg B)) \,@(t-1))$

Si definiscono inoltre alcune regole di riscrittura che vanno ad aggiungersi alle precedenti, consentendo la traduzione di intervalli della forma $[x,x]$ e la distribuzione della negazione sugli argomenti dei nuovi operatori:

1. $\forall\, A \in \mathcal{G}^{\diamond}, x \in \mathbf{Z}\ \ A \,@[x,x] \rightarrow \partial^{-x} A$
2. $\forall\, A \in \mathcal{G}^{\diamond}, x \in \mathbf{Z}\ \ A \,?[x,x] \rightarrow \partial^{-x} A$
3. $\forall\, A \in \mathcal{G}^{\diamond}, x, y \in \mathbf{Z}, x < y\ \ \neg(A \,@[x,y]) \rightarrow (\neg A) \,?[x,y]$
4. $\forall\, A \in \mathcal{G}^{\diamond}, x, y \in \mathbf{Z}, x < y\ \ \neg(A \,?[x,y]) \rightarrow (\neg A) \,@[x,y]$
5. $\forall\, A, B \in \mathcal{G}^{\diamond}\ \ \neg(\mathrm{until}(A,B)) \rightarrow \neg\,\mathrm{until}(\neg(\neg A), \neg(\neg B))$
6. $\forall\, A, B \in \mathcal{G}^{\diamond}\ \ \neg(\mathrm{since}(A,B)) \rightarrow \neg\,\mathrm{since}(\neg(\neg A), \neg(\neg B))$
7. $\forall\, A, B \in \mathcal{G}^{\diamond}\ \ \neg(\neg\,\mathrm{until}(\neg A, \neg B)) \rightarrow \mathrm{until}((\neg A), (\neg B))$
8. $\forall\, A, B \in \mathcal{G}^{\diamond}\ \ \neg(\neg\,\mathrm{since}(\neg A, \neg B)) \rightarrow \mathrm{since}((\neg A), (\neg B))$

Gli operatori $@[x,y]$ e $?[x,y]$ ammettono riscrittura in BTL secondo $A \,@[x,y] \rightarrow \bigwedge_{\tau=x}^{y} \partial^{-\tau} A$ e $A \,?[x,y] \rightarrow \bigvee_{\tau=x}^{y} \partial^{-\tau} A$, ma una semplice applicazione di queste regole non è conveniente poiché comporta la costruzione di $y - x$ copie della sottoespressione A. Per ovviare a questo problema si può però utilizzare la regola $\bowtie G \rightarrow \bowtie((\neg\ell \vee A) \wedge [G]_{A \rightarrow \ell})$ e rendere A di nuovo unica. Per quanto riguarda gli operatori di tipo until e since, esprimibili in BTL solo in forma ricorsiva, si adotta invece una regola di tipo $\bowtie G \rightarrow \bowtie((\neg\ell \vee C') \wedge [G]_{C \rightarrow \ell})$, dove C' è ricavata direttamente dalla semantica del singolo operatore. Il seguente elenco copre tutti i casi che si possono verificare durante la riscrittura dei nuovi operatori e tiene già conto della successiva ridenominazione dei letterali:

1. $\forall\, A \in \mathcal{G}^{\diamond}, x, y \in \mathbf{Z}, x < y\ \ \bowtie G \rightarrow \bowtie((\neg\ell^1 \vee A) \wedge [G]_{A \,@[x,y] \rightarrow \bigwedge_{\tau=x}^{y} \partial^{-\tau}\, \ell^{\tau-x+1}})$
2. $\forall\, A \in \mathcal{G}^{\diamond}, x, y \in \mathbf{Z}, x < y\ \ \bowtie G \rightarrow \bowtie((\neg\ell^1 \vee A) \wedge [G]_{A \,?[x,y] \rightarrow \bigvee_{\tau=x}^{y} \partial^{-\tau}\, \ell^{\tau-x+1}})$

3. $\forall\, A, B \in \mathcal{G}^{\diamond} \quad \bowtie G \rightarrow \bowtie((\neg \ell^1 \vee (A \vee (B \wedge \partial^{-1}\, \ell^2))) \wedge [G]_{\mathrm{until}(A,B) \rightarrow \ell^1})$

4. $\forall\, A, B \in \mathcal{G}^{\diamond} \quad \bowtie G \rightarrow \bowtie((\neg \ell^1 \vee (A \vee (B \wedge \partial\, \ell^2))) \wedge [G]_{\mathrm{since}(A,B) \rightarrow \ell^1})$

5. $\forall\, A, B \in \mathcal{G}^{\diamond} \quad \bowtie G \rightarrow \bowtie((\neg \ell^1 \vee (A \wedge (B \vee \partial^{-1}\, \ell^2))) \wedge [G]_{\neg\, \mathrm{until}(\neg A, \neg B) \rightarrow \ell^1})$

6. $\forall\, A, B \in \mathcal{G}^{\diamond} \quad \bowtie G \rightarrow \bowtie((\neg \ell^1 \vee (A \wedge (B \vee \partial\, \ell^2))) \wedge [G]_{\neg\, \mathrm{since}(\neg A, \neg B) \rightarrow \ell^1})$

La funzione ℓ che compare nelle prime due regole coincide con la a dell'esempio relativo alle funzioni $at_{a,[x,y]}$ e $qm_{a,[x,y]}$ esposto nella sezione 3.1, mentre la ℓ delle rimanenti quattro regole ha un ruolo analogo a quello della funzione $until_{a,b}$ presentata nella stessa sezione. Effettuando il collegamento di ℓ, ad ogni nuovo operatore viene a corrispondere un circuito caratteristico a cui è comodo associare un componente equivalente sul piano operazionale. Durante la stesura della specifica questi nuovi componenti possono essere utilizzati al posto degli operatori corrispondenti, proprio come avviene con i componenti elementari. La rete così prodotta risulta più compatta ed espressiva, ma prima di procedere alla sua esecuzione è necessario sostituire ogni nuovo componente con il rispettivo circuito equivalente.

Per maggior praticità si definisce innanzitutto un componente che implementa la congiunzione estesa ad un qualunque numero di termini, operazionalmente equivalente ad un albero di componenti *joint* soddisfacente $\forall (v_i, v_j) \in E, v_i, v_j \in V_{\wedge} \;\; (v_i, v_j) \in \beta \iff (v_j, v_i) \notin \beta$, con $V_{\wedge}$ insieme dei nodi interni al componente. Il componente duale, cioè la disgiunzione estesa, può invece essere pensato come un generico albero di componenti *gate* comunque orientati. Nella parte alta di figura 3.19 ne sono riportati simboli e circuiti equivalenti, nel caso di cinque termini.

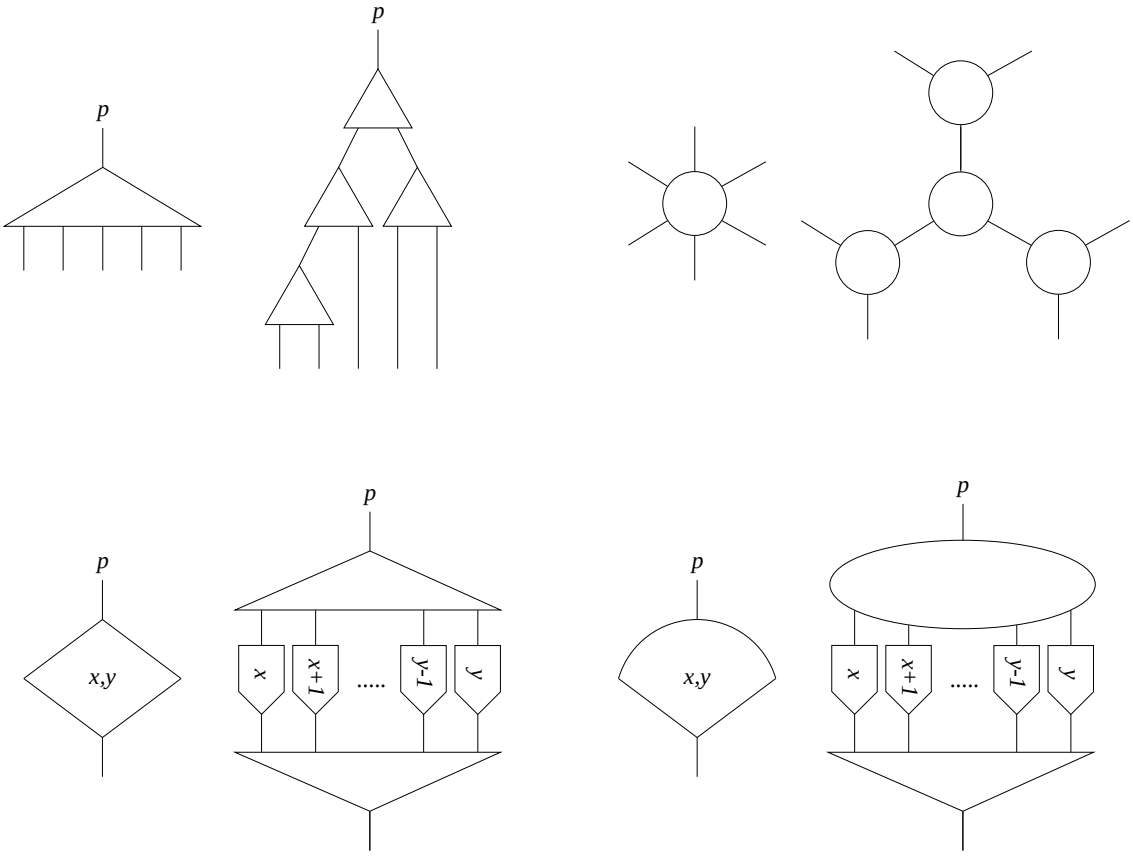

Figura 3.19. Circuiti generati da $\bigwedge_{i=1}^{5} A_i$, $\bigvee_{i=1}^{5} A_i$, $A\,@[x,y]$ e $A\,?[x,y]$

Utilizzando questi due componenti non è difficile disegnare i circuiti corrispondenti agli operatori $@[x, y]$ e $?[x, y]$, riportati nella parte bassa della stessa figura assieme ai simboli che li rappresentano sinteticamente. Si osserva come il collegamento dei letterali abbia prodotto un albero di *joint* rovesciato che riduce a due il numero di terminali di ogni circuito, garantendo complessità spaziale lineare anche in presenza di operatori nidificati. Più precisamente, ogni operatore definito sull'intervallo $[x, y]$ genera un numero di componenti elementari pari a $3 \cdot (y - x) + 1$, indipendentemente dalla sua collocazione all'interno dell'albero sintattico.

Infine, la figura 3.20 mostra da sinistra a destra e dall'alto in basso simboli e circuiti equivalenti per gli operatori until$(\ldots, \ldots)$, $\neg$ until$(\neg \ldots, \neg \ldots)$, since$(\ldots, \ldots)$ e $\neg$ since$(\neg \ldots, \neg \ldots)$. Ogni circuito è costituito da quattro componenti elementari e presenta tre terminali non intercambiabili, dove ai componenti figli l e r sono associati rispettivamente il primo ed il secondo argomento di ogni operatore. Si nota che i circuiti relativi a until e since possono essere trasformati l'uno nell'altro semplicemente scambiando i terminali contraddistinti dai segni $+$ e $-$. Questi circuiti costituiscono una rappresentazione in forma chiusa di operatori la cui definizione sintattica è possibile solo ricorsivamente, risultato ottenuto grazie all'utilizzo di un componente *joint* privo di corrispondente sintattico.

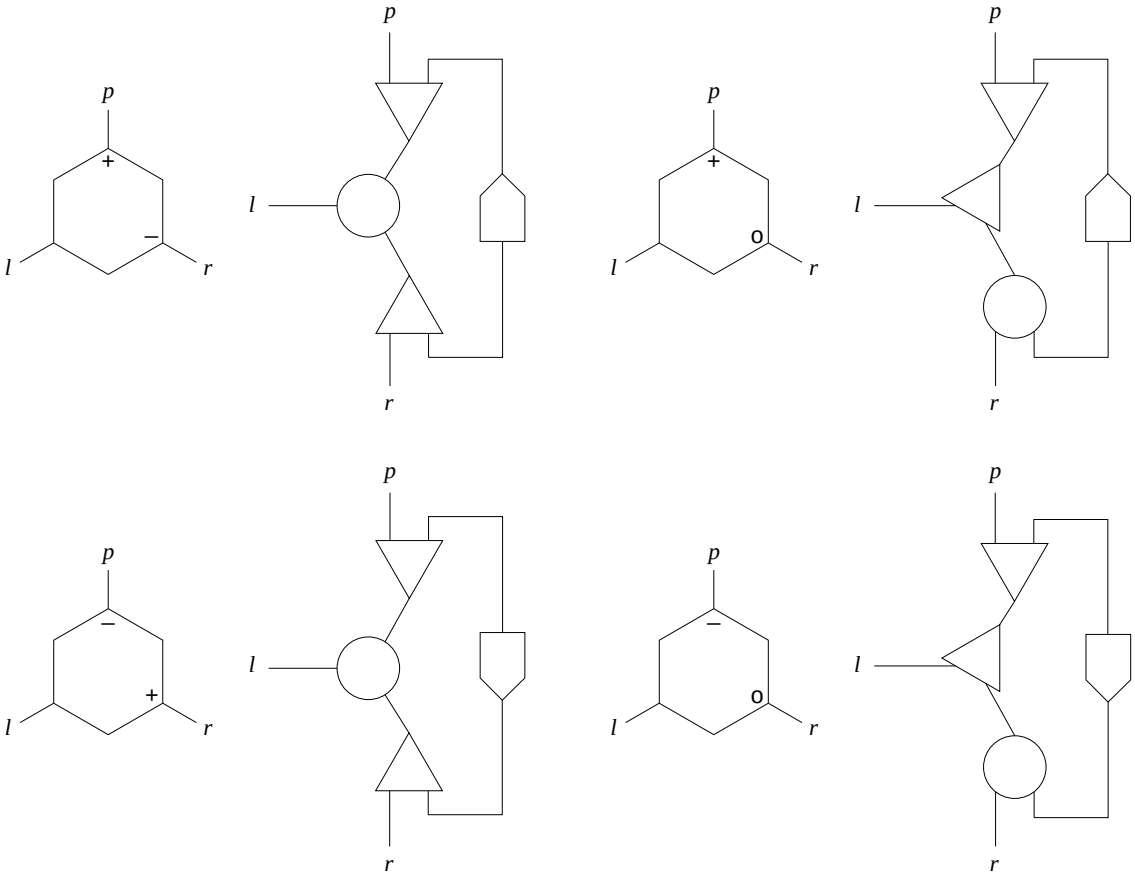

Figura 3.20. Circuiti generati da until(A, B), $\neg$ until$(\neg A, \neg B)$, since(A, B) e $\neg$ since$(\neg A, \neg B)$

3.7 Applicazione del teorema di tempo reale

Chiamando $V_\triangle \subseteq W$ l'insieme di nodi generato dalla specifica $\bowtie G \in \dot{\mathcal{F}}$ ed $U_\triangledown \subseteq W \setminus V_\triangle$ l'unione degli insiemi di nodi $V_\triangledown$ generati da tutte le specifiche di collegamento $\bowtie J_\ell \in \dot{\mathcal{F}} \mid \ell \in \mathcal{L}_G$, le regole di riscrittura che definiscono $\mathcal{A}_G^\vee$ producono un insieme di reti il cui insieme complessivo di nodi V non supera in cardinalità $\operatorname{card} V_\triangle + \operatorname{card} U_\triangledown$. Dato che le specifiche di collegamento sono rappresentate da alberi binari a parte l'operatore $\bowtie$, il numero di foglie di ognuno di essi è $\operatorname{card} L_\triangledown = 2^h$ ed il numero di nodi $\operatorname{card} V_\triangledown = 2^{h+1} - 1$, con h altezza dell'albero, e quindi nel complesso $\operatorname{card} V_\triangledown = 2^{\log_2 \operatorname{card} L_\triangledown + 1} - 1 = 2 \cdot \operatorname{card} L_\triangledown - 1$. Per quanto riguarda l'albero di $\bowtie G$, questo è al più binario e quindi soddisfacente $\operatorname{card} V_\triangle \geq 2 \cdot \operatorname{card} L_\triangle - 1$. Considerando allora l'unione $N_\triangledown$ di tutti gli insiemi $L_\triangledown$, varrà $\operatorname{card} U_\triangledown \leq 2 \cdot \operatorname{card} N_\triangledown - 1$ e dato che ad ogni elemento di $N_\triangledown$ corrisponde una foglia di $L_\triangle$ questo equivale a scrivere che $\operatorname{card} U_\triangledown \leq 2 \cdot \operatorname{card} L_\triangle - 1 \leq \operatorname{card} V_\triangle$, da cui in definitiva $\operatorname{card} V \leq \operatorname{card} V_\triangle + \operatorname{card} U_\triangledown \leq 2 \cdot \operatorname{card} V_\triangle$.

Dato che per un qualunque albero il numero dei nodi supera di uno quello dei rami e che le reti sono costituite da coppie di archi ed archi negati, si avrà $\operatorname{card} E_\triangle = 2 \cdot (\operatorname{card} V_\triangle - 1) \leq 2 \cdot \operatorname{card} V_\triangle$, $\operatorname{card} E_\triangledown = 2 \cdot (\operatorname{card} V_\triangledown - 1) \leq 2 \cdot \operatorname{card} V_\triangledown$ e chiamando $F_\triangledown$ l'unione di tutti gli $E_\triangledown$, varrà certamente $\operatorname{card} F_\triangledown \leq 2 \cdot \operatorname{card} U_\triangledown$. L'insieme di archi E generato da $\mathcal{A}_G^\vee$ non supera in cardinalità $\operatorname{card} E_\triangle + \operatorname{card} F_\triangledown$, e nel complesso si ha $\operatorname{card} E \leq \operatorname{card} E_\triangle + \operatorname{card} F_\triangledown \leq 2 \cdot (\operatorname{card} V_\triangle + \operatorname{card} U_\triangledown) \leq 4 \cdot \operatorname{card} V_\triangle$. Questa proprietà si conserva anche applicando il teorema di sostituzione come descritto, in quanto ad ogni inserimento di un ritardo nullo corrisponde la sua successiva rimozione e tutte le sostituzioni proposte riducono la cardinalità dell'insieme degli archi E.

La funzione di inferenza locale precedentemente proposta non è causale, ma è facile modificarla in modo da renderla tale. La definizione della funzione causale associata $\lambda_S^\circ : Z_D \to \mathcal{Z}_D$ è infatti:

$$\lambda_S^\circ(e \mathbin{@} t) = \lambda_S(e \mathbin{@} t) \cap (E \times [t, +\infty))$$

In figura 3.21 ne viene riportato l'algoritmo di calcolo, soddisfacente la condizione richiesta grazie ad un controllo esplicito $e \mathbin{@} t \preceq s$ sugli elementi s da restituire. Si sottolinea come quest'ultima modifica renda ineseguibile un'ampia gamma di reti. Alcune di esse, in cui l'ineseguibilità è frutto dell'incompletezza della funzione inferenza, possono essere trasformate in modo da divenire eseguibili. Questo viene fatto solitamente applicando il teorema di sostituzione in modo da spostare o cancellare i ritardi, con regole identiche a quelle adottate nei diagrammi a blocchi. Unica avvertenza in questo caso sarà quella di modificare le etichette degli archi corrispondenti a variabili d'interesse in modo da rispecchiare la nuova semantica.

In altri casi l'ineseguibilità è strutturale, in quanto frutto della non causalità della specifica. Disponendo ad esempio della storia delle funzioni a e b fino ad un certo istante, in generale non si potrà inferire la storia della funzione $\mathrm{until}(a, b)$ fino a quello stesso istante ma sarà necessario attendere la disponibilità di valori relativi ad istanti successivi. In questo caso, nessuna funzione inferenza di tipo causale può garantire la determinazione dell'intera storia di $\mathrm{until}(a, b)$ per ogni possibile storia di a e b. La storia della funzione gemella $\mathrm{since}(a, b)$ risulta invece determinabile anche solo attraverso inferenza causale.

Supponendo che le funzioni α e β siano valutabili in tempo costante e che la determinazione degli archi $(v, v_o), (v_o, w) \in E$ possa avvenire con complessità $O(M_\mu + M_\nu)$, ad esempio sostituendo ad ogni nodo un riferimento ad una struttura che ne descrive tipo e topologia circostante, si ha $M_\mu = 2$, $M_\nu = 2$ e nel complesso si può dire che l'intera funzione λ° è valutabile in tempo costante. Per quanto dimostrato precedentemente, la soglia minima del passo di discretizzazione capace di garantire il tempo reale è allora $\epsilon_m = O(B_\nu + \operatorname{card} I) \cdot \operatorname{card} E$ sotto decongestione iniziale o una delle altre ipotesi equivalenti esposte nella sezione sulla complessità computazionale. Dato che $\operatorname{card} E \leq 4 \cdot \operatorname{card} V_\triangle$, si può concludere:

$$\epsilon_m = O(\operatorname{card} V_\triangle \cdot (B_\nu + \operatorname{card} I))$$

Il termine $\operatorname{card} V_\triangle$ è il numero di operatori e di letterali che compaiono in $\bowtie G$ e $\operatorname{card} I$ è il numero degli ingressi che la procedura deve acquisire durante l'esecuzione. Per valutare il termine $B_\nu = \max_{e \mathbin{@} t \in Z_D, e' \mathbin{@} t' \in \nu^*(e \mathbin{@} t)} |t' - t|$ occorre osservare che $\prod_T(\nu^*((v_i, v_o) \mathbin{@} t)) = \{t\}$

1. Se $\alpha(v_o) = \wedge$ allora...
   (a) Se $v_i = \beta(v_o)$ allora...
      i. $\exists (v_o, v_j), (v_o, v_k) \in E \mid v_i \notin \{v_j, v_k\}, v_j \neq v_k$
      ii. $\Lambda \leftarrow \{(v_o, v_j) \mathbin{@} t, (v_o, v_k) \mathbin{@} t\}$
   (b) ...altrimenti $\Lambda \leftarrow \{(v_o, \beta(v_o)) \mathbin{@} t\}$
2. ...altrimenti, se $\alpha(v_o) = \partial^k$ allora...
   (a) Se $v_i = \beta(v_o)$ allora...
      i. $\exists (v_o, v_j) \in E \mid v_i \neq v_j$
      ii. Se $t - k \in [t, t_M]$ allora $\Lambda \leftarrow \{(v_o, v_j) \mathbin{@} (t - k)\}$, altrimenti $\Lambda \leftarrow \emptyset$
   (b) ...altrimenti ...
      i. Se $t + k \in [t, t_M]$ allora $\Lambda \leftarrow \{(v_o, \beta(v_o)) \mathbin{@} (t + k)\}$, altrimenti $\Lambda \leftarrow \emptyset$
3. ...altrimenti, se $\alpha(v_o) = \vee$ allora...
   (a) Se $\exists (v, v_o) \mathbin{@} t \in S \mid v_i \neq v$ allora...
      i. $\exists (v_o, w) \in E \mid w \notin \{v_i, v\}$
      ii. $\Lambda \leftarrow \{(v_o, w) \mathbin{@} t\}$
   (b) ...altrimenti $\Lambda \leftarrow \emptyset$
4. ...altrimenti $\Lambda \leftarrow \emptyset$

Figura 3.21. Calcolo di $\Lambda = \lambda_S^\circ((v_i, v_o) \mathbin{@} t)$

per $\alpha(v_o) \in \{\wedge, \vee\}$ mentre $\prod_T(\nu^*((v_i, v_o) @ t)) = \{t \pm k\}$ per $\alpha(v_o) = \partial^k$. Si può allora interpretare B_ν come il massimo ritardo che compare in $\bowtie G$, cioè:

$$B_\nu = \max_{(v,\partial^k)\in\alpha} |k|$$

Capitolo 4

Un esecutore di specifiche logiche

4.1 Implementazione

Nei capitoli precedenti è stato mostrato come si possa applicare l'algoritmo di chiusura progressiva alle reti di inferenza temporale ottenendo, nel caso di inferenza causale, garanzia di tempo reale per una scelta opportuna del passo di discretizzazione. Punto debole di questo approccio risulta certamente la scelta di una funzione inferenza incompleta, scelta peraltro resa necessaria dalla complessità esponenziale di un qualunque algoritmo che aspiri alla completezza. Per superare questo problema si può tentare di caratterizzare l'insieme delle specifiche eseguibili, per cui cioè è garantita la produzione dell'intera storia di tutte le uscite per qualunque combinazione di storie di ingressi e condizioni iniziali, ma questa strada è ricca di difficoltà teoriche. Si possono individuare condizioni sufficienti per l'eseguibilità, ma quelle fino ad oggi trovate risultano molto restrittive e molte specifiche in effetti eseguibili non le soddisfano. In mancanza di strumenti teorici adeguati, la sperimentazione può comunque fornire l'evidenza di un insieme di specifiche eseguibili sufficientemente ampio. Una volta completata la specifica del sistema desiderato, tecniche di dimostrazione automatica possono essere impiegate per verificare l'appartenenza delle uscite alla chiusura delle possibili combinazioni di ingressi e condizioni iniziali rispetto alle regole di inferenza implementate. Questo secondo problema è evidentemente più semplice del precedente, in quanto non prescinde dalla struttura della particolare specifica.

La strada che è stata in effetti percorsa ha richiesto l'implementazione di un esecutore di specifiche logiche conforme all'algoritmo teorico e la creazione di alcune reti di prova che si sono mostrate sempre capaci di produrre le uscite richieste a partire da segnali di ingresso casuali. Per la codifica è stato scelto il linguaggio C ANSI e tutte le funzioni esterne al programma appartengono alla libreria standard. Questo, assieme ad opportuni accorgimenti progettuali come la scelta del compilatore GNU, garantisce un'ottima portabilità sulle piattaforme più diverse attraverso la semplice ricompilazione. La piattaforma preferita per lo sviluppo è stato un computer compatibile Intel con sistema operativo Linux, quest'ultimo scelto per le sue caratteristiche di robustezza e velocità che risultano fattori critici in molte applicazioni tempo reale. Per compiere i test necessari sono stati realizzati altri programmi di corredo, tra cui un generatore/visualizzatore di segnali casuali ed un compilatore[1] da BTL al formato di descrizione della rete accettato dall'esecutore.

L'esecutore effettivamente implementato ha corrisposto alle attese. Può acquisire ingressi da file predeterminati al momento del lancio o progressivamente costruiti da altre applicazioni[2], mantenendosi in attesa di nuova informazione sulla fine dei file medesimi e registrando in altri file le uscite inferibili a mano a mano che queste vengono determinate dall'algoritmo di chiusura progressiva. In particolare, i file di ingresso ed uscita hanno lo stesso formato e questo permette il lancio di più esecutori in parallelo che si scambiano conoscenza in tempo reale. Dato che non è

[1] Autore Pierfrancesco Bellini.

[2] Ad esempio un programma di acquisizione dati in tempo reale.

evidente in quale misura l'ipotesi di inferenza causale sia restrittiva ai fini dell'esecuzione di una specifica generica, il programma consente di scegliere se operare o meno sotto questa ipotesi.

4.2 TINX

Il funzionamento dell'esecutore, `tinx` (Temporal Inference Network eXecutor), può essere così riassunto:

- Al lancio:
  - Legge gli argomenti forniti sulla linea di comando e configura i relativi parametri di esecuzione
  - Carica un file con estensione `.tin` contenente una descrizione della topologia della rete ed un elenco di ingressi ed uscite, ed alloca le opportune strutture di rappresentazione
  - Se richiesto, carica un file con estensione `.evl` contenente l'insieme di conoscenza iniziale Ω_0
- Ad ogni passo del ciclo di esecuzione:
  - Sceglie un evento $e \,@\, \sigma(\Delta)$ su cui fare inferenza e, se richiesto, lo visualizza e/o lo registra in un file di log con estensione `.evl`
  - Calcola le conseguenze ad un passo dell'elemento scelto e, se ad esse sono associate uscite, registra il risultato in opportuni file con estensione `.io`
  - Tenta di leggere nuovi dati d'ingresso dai file `.io` specificati e li gestisce come se fossero risultato d'inferenza
  - Se richiesto, controlla che la nuova conoscenza prodotta sia compatibile con quella precedentemente acquisita e termina con condizione di errore in caso contrario
- Termina con successo nei seguenti casi:
  - Alla chiusura della conoscenza disponibile, se da ogni file di ingresso è stato letto un carattere di fine flusso (`.`)
  - Immediatamente, se da almeno un file di ingresso è stato letto un carattere di Escape (codice esadecimale `1b`)
  - Immediatamente, se il programma ha ricevuto un segnale `SIGINT`, che di solito è un `Ctrl-C`
- Alla terminazione (con successo):
  - Aggiunge in coda ai file di uscita un carattere di fine flusso (`.`)
  - Disalloca tutte le risorse allocate

La sintassi della linea di comando è:

`tinx` [`-cdilv`] [`-I` *state*] [`-L` *log*] [*base*]
oppure
`tinx -h`

dove *base* è il nome senza estensione del file `.tin` contenente la descrizione della rete e dei file di I/O, che se non specificato viene assunto uguale a `default`, mentre gli switch opzionali hanno i seguenti significati:

- `-c`: Usa solo regole d'inferenza causali
- `-d`: Visualizza su `stdout` l'ultimo evento selezionato $e \,@\, \sigma(\Delta)$ ed un simbolo che indica se la conoscenza acquisita fino a quell'istante abbia raggiunto la chiusura (`*`) o meno (`>`)

- `-h`: Stampa la sintassi della linea di comando
- `-i`: Carica l'insieme di conoscenza a priori Ω_0 che contiene le condizioni iniziali per l'esecuzione. Se nessun nome di file viene specificato attraverso l'apposito switch `-I`, questo viene assunto uguale alla concatenazione di *base* con il suffisso `_ic.evl`
- `-I` *state*: Attiva lo switch `-i` e specifica come nome del file da caricare la concatenazione di *state* con l'estensione `.evl`
- `-l`: Registra gli eventi selezionati $e \,@\, \sigma(\Delta)$ in un file di log. Se nessun nome di file viene specificato attraverso l'apposito switch `-L`, questo viene assunto uguale alla concatenazione di *base* con il suffisso `_log.evl`
- `-L` *log*: Attiva lo switch `-l` e specifica come nome del file di log la concatenazione di *log* con l'estensione `.evl`
- `-v`: Verifica la coerenza della conoscenza inferita durante l'esecuzione

Il formato del file contenente la descrizione della rete e dei file di I/O è descritto in figura 4.1, mentre le figure 4.4 e 4.5 ne riportano due semplici esempi relativi alle specifiche di up_ℓ ed $until_{a,b}$.

node, *parent*, *son*, *from*, *to* ::= Stringa alfanumerica
arc ::= `(` *from* `,` *to* `)`
delay, *offset* ::= Intero con segno
class ::= `G` ‖ `J` ‖ `D` [*delay*]
component ::= *node* `:` *class* `;` *parent* [`,` *son* [`,` *son*]]
components ::= *component* [*components*]
io_mode ::= `!` ‖ `?`
filename ::= Nome di file valido privo di estensione
stream ::= *io_mode filename arc* [`@` *offset*]
streams ::= *stream* [*streams*]
filename`.tin` ::= [*components*] [*streams*]

Figura 4.1. Grammatica di un file `.tin`

La prima parte del file `.tin` è costituito da una lista di componenti. Il nome di ogni nodo è seguito da una tra le costanti `G` (*gate*), `J` (*joint*) e `D` (*delay*) che ne codificano il tipo. Il parametro *delay* è assunto uguale ad 1 se non specificato diversamente, mente il nodo *parent* è obbligatorio in quanto determina l'orientazione del componente. Se uno od entrambi i nodi *son* non sono specificati, il programma tenta di completare la topologia della rete utilizzando le informazioni associate agli altri componenti e termina con condizione di errore in caso di insuccesso.

Nella seconda parte del file `.tin` compaiono i riferimenti ad ingressi ed uscite d'interesse. Le dichiarazioni `!` (input) e `?` (output) qualificano il file `.io` seguente come di ingresso o uscita e vi associano un arco *arc* e la sua negazione. Ogni record del file `.io` è messo in corrispondenza con un evento $arc\,@(offset + pos)$ o con il suo negato, dove *offset* è assunto uguale a 0 se non specificato diversamente e *pos* è la posizione del record espressa in numero di record a partire dall'inizio del file. Ciò permette di gestire segnali di ingresso disponibili a partire da istanti arbitrari.

Il formato dei file contenenti la conoscenza a priori ed il log degli eventi selezionati è riassunto in figura 4.2. Come esempio, in figura 4.6 sono riportate le prime venti linee del log di una esecuzione di `up.tin`.

Dato che ogni evento acquisito viene prima o poi selezionato, il file di log contiene la chiusura dell'insieme di conoscenza disponibile in un formato ricaricabile al lancio. Questo consente di operare la chiusura dell'insieme Ω_0 in assenza di ingressi per sostituirlo con un nuovo insieme $\widehat{\Omega}_0$, in modo da ridurre la produzione di nuova conoscenza durante le prime iterazioni della successiva esecuzione e rendere più rapido il raggiungimento della decongestione.

Infine, i file di I/O hanno il formato riportato in figura 4.3. Il valore logico `1` corrisponde all'affermazione dell'arco associato al file in un istante dipendente dalla sua posizione, mentre lo `0`

from, *to* ::= Stringa alfanumerica
arc ::= `(` *from* `,` *to* `)`
time ::= Intero senza segno
event ::= *arc* `@` *time*
events ::= *event* [*events*]
filename ::= Nome di file valido privo di estensione
filename`.evl` ::= [*events*]

Figura 4.2. Grammatica di un file `.evl`

alla sua negazione. In figura 4.7 sono riportati a titolo di esempio i files coinvolti in una esecuzione di `until.tin`.

bit ::= `0` ‖ `1`
bits ::= *bit* [*bits*]
filename ::= Nome di file valido privo di estensione
filename`.io` ::= [*bits*] `.`

Figura 4.3. Grammatica di un file `.io`

Il punto che termina ogni file di I/O è necessario poiché il carattere `EOF` viene interpretato dall'esecutore come indice del fatto che tutto ciò che poteva essere letto dal file durante le precedenti scansioni degli ingressi è già stato letto, ma nuovi dati possono ancora venire aggiunti in coda al file in un qualunque istante futuro ed essere acquisiti dall'esecutore durante una successiva scansione degli ingressi. Il punto finale esclude proprio questa possibilità, consentendo un più comodo uso del programma in modalità batch. A differenza dell'aggiunta di un nuovo dato in coda, eventuali modifiche alla sezione del file già scandita non vengono rilevate per evidenti ragioni di efficienza.

```
g0:     G ; j2
j0:     J ; g0, j1
j1:     J ; j2
j2:     J ; j1
d0:     D ; g0, j1
d1:     D ; j0, j2

! l      (j1, j2)
? up_l   (g0, j0)
```

Figura 4.4. `up.tin`, specifica di up_ℓ

Per rappresentare la rete di inferenza temporale è stata utilizzata una tecnica di allocazione sparsa dei nodi. All'interno di ogni nodo v è memorizzato un vettore di dimensione predefinita pari a $\operatorname{card} T$ per ognuno degli archi e_i entranti nel nodo, indicizzato nel tempo t e contenente la storia dell'arco associato, espressa in termini di appartenenza dell'evento $e_i \,@\, t$ all'insieme di quelli già selezionati, S, o semplicemente noti, Φ. Ogni nodo contiene anche la sua classificazione $\alpha(v)$ ed un elenco degli archi $\neg e_i$ uscenti dallo stesso, che sono così subito disponibili.

Ogni arco e_i è rappresentato tramite una coppia costituita da un puntatore al nodo v in cui entra e da un indice i che individua all'interno del nodo il vettore di storia appropriato o l'arco uscente corrispondente $\neg e_i$, indice che vale zero se l'arco è di collegamento con il genitore $\beta(v)$.

Oltre all'arco su cui è definito, ogni evento $e_i \,@\, t$ contiene un indice temporale t che localizza le relative informazioni nel vettore di storia individuato dall'arco e_i. Come preannunciato nella sezione sulla complessità computazionale, l'insieme Δ degli eventi noti ma non ancora selezionati è rappresentato da una lista concatenata ed ordinata nel tempo di liste concatenate di archi, le

```
ju:     J ; jv
jv:     J ; ju
ga:     G ; ju, ja
ja:     J ; jv
gb:     G ; ja, jb
jb:     J ; ga
du:     D ; ju, gb
dv:     D ; jv, jb

! a             (ga, ja)
! b             (jb, gb)
? until_ab      (jv, ju)
```

Figura 4.5. `until.tin`, specifica di $until_{a,b}$

```
(j1, j2)@0
(j2, g0)@0
(j2, d1)@0
(d1, j0)@1
(j0, g0)@1
(j2, j1)@1
(j1, j0)@1
(j1, d0)@1
(d0, g0)@2
(j1, j2)@2
(j2, g0)@2
(g0, j0)@2
(j0, d1)@2
(d1, j2)@1
(j0, j1)@2
(j2, d1)@2
(d1, j0)@3
(j0, g0)@3
(j1, j2)@3
(j2, g0)@3
```

Figura 4.6. Le prime venti linee del file `up_log.evl`

```
11100001100000111100100101010001001101011010011001010111001000011.

01011111110100010111110110100011111011100101111111011000000010110.

11111111100000111111100111110011111111111111111111101111100100111.
```

Figura 4.7. Contenuto dei files `a.io`, `b.io` ed `until_ab.io`

cui strutture di supporto sono anch'esse interne ai nodi e più precisamente ai vettori di storia. Tutta la memoria necessaria viene così allocata prima che abbia inizio il processo di inferenza e non viene mai disallocata durante l'esecuzione, evitando chiamate a funzioni del sistema operativo che introdurrebbero ritardi variabili e potenziale violazione del tempo reale.

Qualora lo si ritenesse opportuno per ragioni di prestazioni, non presenta particolare difficoltà la riscrittura delle routine di I/O in modo da sfruttare meccanismi di comunicazione interprocesso. Si tenga tuttavia presente che la libreria standard ANSI non prevede primitive di comunicazione interprocesso, che sono specifiche del sistema operativo, mentre la soluzione adottata risulta portabile. Se l'esecutore dovesse invece girare come unico task, ad esempio all'interno di un microcontrollore, sarebbe necessario riscrivere le routine di I/O per gestire l'interfaccia con l'hardware controllato. In entrambi i casi, è bene ricordare che il programma emette ogni risultato non appena riesce a determinarlo e si rende quindi indispensabile un buffer in uscita che possa essere svuotato alla velocità richiesta dal passo di discretizzazione.

4.3 Programmi di corredo

4.3.1 BTL2TIN

Il compilatore `btl2tin` è stato realizzato da Pierfrancesco Bellini facendo uso di `lex` e `yacc` per tradurre una specifica BTL nella rete di inferenza temporale corrispondente. Il programma esegue anche la distribuzione della negazione sugli argomenti degli operatori e la rimozione dei componenti *root* e *leaf*, applicando le sostituzioni presentate nel capitolo precedente.

La sintassi della linea di comando è:

`btl2tin` *name*

dove *name* è il nome senza estensione del file `.btl` contenente il sorgente della specifica, mentre al file contenente la rete generata sarà assegnato ugual nome ed estensione `.tin`. Dato che i caratteri usati per rappresentare gli operatori della BTL non sono presenti sulle comuni tastiere, si è resa necessaria una loro riscrittura. La figura 4.8 riassume la grammatica riconosciuta dal compilatore.

function ::= Stringa alfanumerica
filename ::= Nome di file valido privo di estensione
var ::= *function* ‖ *filename*
delay, *min*, *max*, *offset* ::= Intero con segno
interval ::= `[` *min* `,` *max* `]`
clause ::= *var* ‖ `~` *clause* ‖ *clause* `&` *clause* ‖ *clause* `|` *clause* ‖ `#` [*delay*] *clause* ‖ `(` *clause* `)` ‖
‖ *clause* `-->` *clause* ‖ *clause* `<-->` *clause* ‖ *clause* `==` *clause* ‖
‖ *clause* `@` *interval* ‖ *clause* `?` *interval* ‖ `until(` *clause* `,` *clause* `)` ‖ `since(` *clause* `,` *clause* `)`
clauses ::= *clause* `;` [*clauses*]
signal ::= *filename* [`@` *offset*]
signals ::= *signal* [`,` *signals*]
time ::= Intero senza segno
sample ::= [`~`] *var* `@` *time* ‖ [`~`] *var* `@ [` *time* `,` *time* `]`
samples ::= *sample* [`,` *samples*]
filename`.btl` ::= [`inputs:` *signals*] [`outputs:` *signals*] [`init:` *samples*] *clauses*

Figura 4.8. Grammatica di un file `.btl`

Il sorgente può inoltre contenere linee di commento, precedute dal prefisso `//`. In testa si trovano le dichiarazioni dei file di ingresso (`inputs:`) e di uscita (`outputs:`), che finiscono in coda al file *name*`.tin`, e dello stato iniziale (`init:`) che viene registrato nell'apposito file *name*`_ic.evl`. La corrispondenza tra gli operatori BTL e quelli accettati da `btl2tin` è sintetizzata in tabella 4.1.

Come esempio di sorgente, la figura 4.9 mostra la specifica di un meccanismo di arbitraggio tra due competitori per una risorsa esclusiva. Gli ingressi `rq1` o `rq2` comunicano le richieste di

BTL	`btl2tin`
$\neg A$	`~`A
$A \wedge B$	A `&` B
$A \vee B$	A `|` B
$\partial^k A$	`#`k A
(A)	`(`A`)`
$\neg A \vee B$	A `-->` B
$(\neg A \vee B) \wedge (\neg B \vee A)$	A `<-->` B
$(\neg A \vee B) \wedge (\neg B \vee A)$	A `==` B
$A \,@[x, y]$	A `@ [`x`,` y`]`
$A \,?[x, y]$	A `? [`x`,` y`]`
$\text{until}(A, B)$	`until(`A`,` B`)`
$\text{since}(A, B)$	`since(`A`,` B`)`
$C \,@\, t$	C `@` t
$\neg C \,@\, t$	`~`C `@` t
$C \,@\, t_1, \ldots, C \,@\, t_2$	C `@ [`t_1`,` t_2`]`
$\neg C \,@\, t_1, \ldots, \neg C \,@\, t_2$	`~`C `@ [`t_1`,` t_2`]`
$\bowtie(\bigwedge_{k=1}^{n} C_k)$	C_1 `;` $\ldots$ `;` C_k `;` $\ldots$ `;` C_n `;`

Tabella 4.1. Corrispondenza tra gli operatori BTL e quelli accettati da `btl2tin`

appropriazione della risorsa all'esecutore, che ne garantisce la disponibilità attraverso le uscite `gnt1` o `gnt2`, favorendo il primo competitore in caso di richiesta contemporanea. Se disponibile, la risorsa viene allocata in modo esclusivo dall'istante successivo a quello della richiesta fino all'istante successivo a quello del suo rilascio, comunicato dal proprietario attraverso gli ingressi `ex1` o `ex2`. Se la risorsa è già stata allocata dall'altro competitore, il richiedente viene favorito nella scelta successiva in modo da garantire l'alternanza degli accessi. La rete generata da questa specifica conta ben 134 componenti, ma è ragionevole supporre che semplificazioni appropriate ne possano ridurre significativamente il numero, e risulta eseguibile anche solo attraverso inferenza causale. Altri esempi di sorgenti, tra cui quello di un meccanismo produttore-consumatore, sono riportati in appendice B.

```
inputs: rq1, ex1, rq2, ex2
outputs: gnt1, gnt2
init: wait_req1 @0, wait_req2 @0, ~sosp2 @0

up_ex1 == (ex1 & # ~ex1);
wait_req1 --> ( req1 & (~gnt1 & ~gnt2) & #-1 (gnt1 & ~gnt2)) |
              ( req1 & (~gnt1 &  gnt2) & #-1 wait_free1) |
              (~req1 &  ~gnt1 & #-1 wait_req1);
gnt1 --> ( up_ex1 & #-1 (wait_req1 & ~gnt1)) |
         (~up_ex1 & #-1 (gnt1 & ~gnt2));
wait_free1 --> (~gnt2 & gnt1) |
               ( gnt2 & #-1 wait_free1);

up_ex2 == (ex2 & # ~ex2);
wait_req2 --> ( req2 & (~gnt1 & ~gnt2) & #-1 (gnt2 & ~gnt1)) |
              ( req2 & ( gnt1 & ~gnt2) & #-1 wait_free2) |
              (~req2 & ~gnt2 & #-1 wait_req2);
gnt2 --> ( up_ex2 & #-1 (wait_req2 & ~gnt2)) |
         (~up_ex2 & #-1 (gnt2 & ~gnt1));
wait_free2 --> (~gnt1 & gnt2) |
               ( gnt1 & #-1 wait_free2);

 rq1 & ~(rq2 | sosp2) -->  req1 & ~req2 & #-1 ~sosp2;
~rq1 &  (rq2 | sosp2) --> ~req1 &  req2 & #-1 ~sosp2;
~rq1 & ~(rq2 | sosp2) --> ~req1 & ~req2 & #-1 ~sosp2;
 rq1 &  (rq2 | sosp2) -->  req1 & ~req2 & #-1  sosp2;
```

Figura 4.9. Specifica di un meccanismo di arbitraggio

4.3.2 TINT

Il programma `tint` (Temporal Interface Network Tester) è un dimostrativo senza particolari pretese che mostra come interfacciarsi all'esecutore. Genera segnali booleani casuali nel formato di uno o più file `.io` in ingresso all'esecutore e legge le uscite da esso prodotte a mano a mano che queste si rendono disponibili, visualizzando una finestra scorrevole sulle loro storie. Gli ingressi sono colorati in rosso e le uscite in verde attraverso sequenze ANSI per il controllo del terminale.

Le informazioni sui file da generare o acquisire possono venir lette dallo stesso file `.tin` che contiene la configurazione dell'esecutore. Al lancio, `tint` crea nuovi file di ingresso vuoti, eventualmente sovrascrivendo i precedenti, e cancella quelli di uscita per mettersi in attesa della loro creazione da parte dell'esecutore, che deve essere quindi lanciato successivamente.

La sintassi della linea di comando è:

`tint` [`-q`] [`-t` *step*] [*base*]
oppure
`tint -h`

dove *base* è il nome senza estensione del file `.tin` contenente la descrizione dei file di I/O, che se non specificato viene assunto uguale a `default`, mentre gli switch opzionali hanno i seguenti significati:

- `-h`: Stampa la sintassi della linea di comando
- `-q`: Genera i segnali ma non li visualizza
- `-t` *step*: Attende almeno un numero *step* di secondi tra la generazione di un campione dei segnali ed il successivo, 0.05 se non diversamente specificato. Questo valore è puramente indicativo, dato che la struttura del programma non ne garantisce il funzionamento in tempo reale

La figura 4.10 riporta un esempio dell'output di `tint`, dove i simboli `-` e `#` rappresentano rispettivamente i valori logici `0` e `1`. I segnali mostrati sono legati dalla specifica già esposta in figura 4.9.

```
-#-####---##-#--###---#-----####-#-####-##-##-#--#----#--##-#-- [I] rq1.io
##-####-#-#-###-#-#-----#-####----#-#--##-#####--#--####---#--# [I] ex1.io
#-#-#-####--#-##-#-##-#-##-#-###-#-##-#------##---#######-#-#-# [I] rq2.io
##-----#-##-#-#--#---#---##--#---#--##---###-##-#--#--#--##--#- [I] ex2.io
-###----#--##--##-#----##-----#####--###--#-######-----#####--# [O] gnt1.io
#---####-#---##--#--##---#--##-----##---##--------##-##-----##- [O] gnt2.io
```

Figura 4.10. Esempio di output del programma `tint`

4.4 Un esempio di applicazione

Il nocciolo del motore a curvatura dell'astronave *Enterprise* è circondato da un campo di contenimento che impedisce all'antimateria di venire a contatto con la superficie interna del reattore. Quando il valore del campo scende sotto una soglia critica per almeno 3 secondi si rischia di perdere il controllo della reazione, situazione di pericolo evidenziata dalla spia *danger* sulla console dell'ingegnere di bordo, mentre alla condizione di sotto-soglia è associato il segnale *critical*. Anche dopo il ripristino dei normali valori di campo, il rischio persiste per altri 5 secondi. L'ingresso in una situazione di pericolo viene accompagnato da un caratteristico allarme acustico, indicato con *alert*, che continua a suonare finché il computer non riceve un segnale *acknowledge* da parte dell'ingegnere. Se una situazione di pericolo dura almeno 60 secondi consecutivi senza che l'allarme sia stato nel frattempo spento, il computer provvede a farlo ed assumendo che l'equipaggio non sia in grado di intervenire genera il segnale *shutdown* per spengere la reazione ed uscire dalla curvatura. In questo caso, per poter riavviare il motore è richiesta la pressione dell'apposito tasto *reset*.

Assumendo un passo di discretizzazione ϵ pari ad un secondo, il comportamento del sistema di controllo dell'astronave può essere schematizzato con l'insieme di macchine a stati comunicanti di figura 4.11, estesa con i contatori L, M e N che scandiscono con il loro incremento il passare del tempo e sono stati introdotti per non dover rappresentare esplicitamente un numero eccessivo di stati. I nomi dei segnali sono stati abbreviati con la loro lettera iniziale, con l'eccezione del segnale *acknowledge* a cui è associata la lettera k per distinguerlo da *alert*, e sono stati introdotti i segnali intermedi u, v e w. Un insieme consistente di stati iniziali è stato poi contrassegnato attraverso un token interno ad ogni stato.

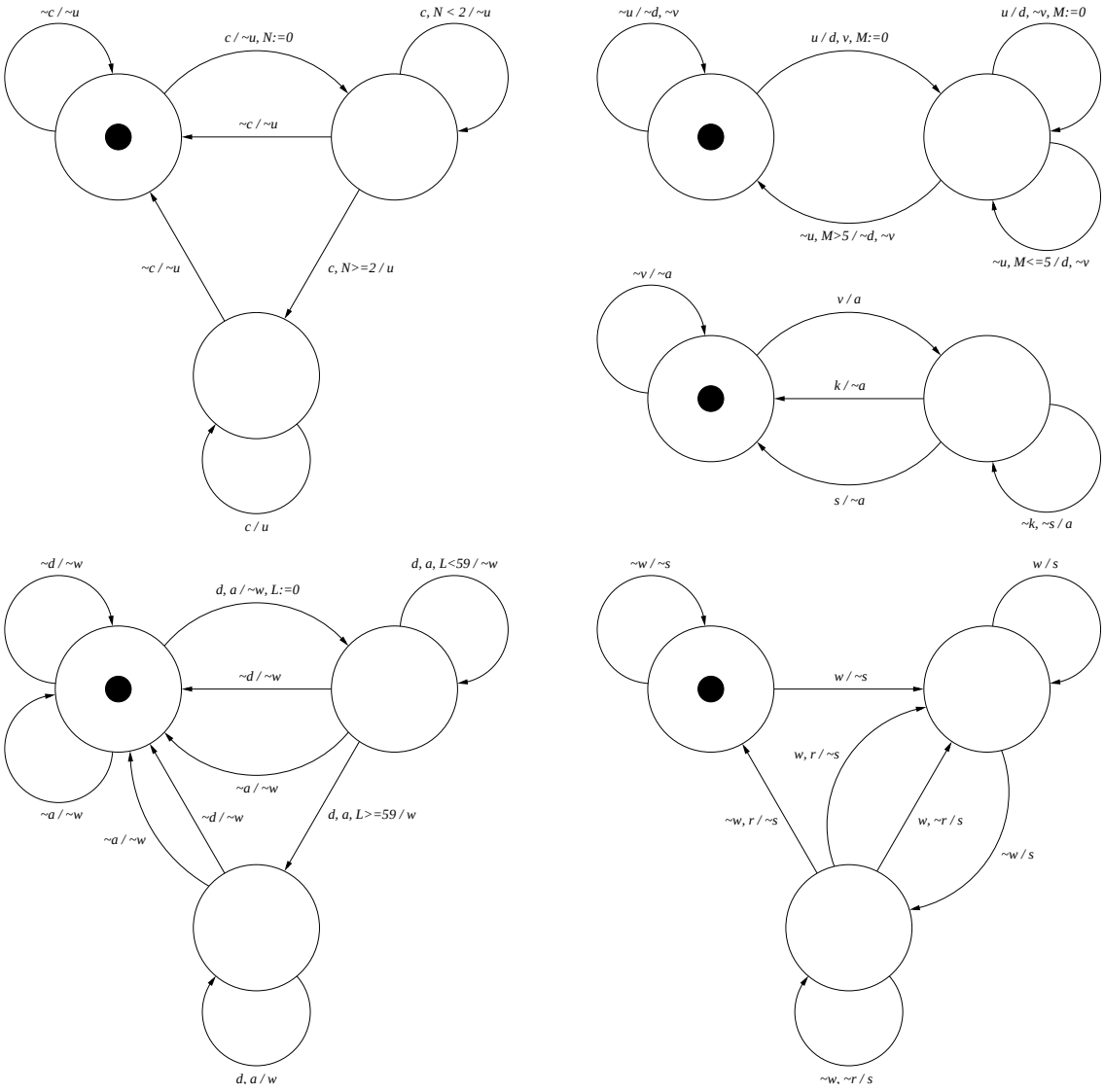

Figura 4.11. Macchine a stati che riproducono il comportamento del sistema di controllo

Nel progettare un simile automa è facile commettere errori. Un approccio descrittivo consente maggiore astrazione dal dettaglio e richiede la specifica del comportamento voluto attraverso una congiunzione di doppie implicazioni:

1. $danger \iff (critical\,@[-2,0])\,?[-5,0]$
2. $alert \iff \text{since}(\partial(\neg danger) \wedge danger, \neg acknowledge \wedge \neg shutdown)$
3. $shutdown \iff \text{since}((danger \wedge alert)\,@[-60,-1], \neg reset)$

che possono essere tradotte in BTL come:

$$
\begin{aligned}
&\bowtie((\neg danger \vee (critical\,@[-2,0])\,?[-5,0]) \wedge \\
&\quad (danger \vee (\neg critical\,?[-2,0])\,@[-5,0]) \wedge \\
&\quad (\neg alert \vee \text{since}(\partial(\neg danger) \wedge danger, \neg acknowledge \wedge \neg shutdown)) \wedge \\
&\quad (alert \vee \neg\,\text{since}(\neg(\partial\, danger \vee \neg danger), \neg(acknowledge \vee shutdown))) \wedge \\
&\quad (\neg shutdown \vee \text{since}((danger \wedge alert)\,@[-60,-1], \neg reset)) \wedge \\
&\quad (shutdown \vee \neg\,\text{since}(\neg((\neg danger \vee \neg alert)\,?[-60,-1]), \neg(reset))))
\end{aligned}
$$

oppure trovare una più immediata espressione nel sorgente accettato dal compilatore `btl2tin` e riportato in figura 4.12.

```
inputs: critical @2, ack @2, reset @2
outputs: danger @2, alert @2, shutdown @2
init:
      ack @1,
      reset @1,
      ~danger @[0,6]

danger   == (critical @[-2,0]) ?[-5,0];
alert    == since(# ~danger & danger, ~ack & ~shutdown);
shutdown == since((danger & alert) @[-60,-1], ~reset);
```

Figura 4.12. Specifica del sistema di controllo

Per migliorare la leggibilità della rete di inferenza temporale generata da questa specifica si è provveduto al collegamento dei soli letterali che non compaiono in più di una implicazione, in questo caso gli ingressi *critical*, *acknowledge* e *reset*. Il risultato è costituito dalle tre sottoreti di figura 4.13, ognuna delle quali corrisponde ad una diversa implicazione. La simmetria che si può osservare dipende dal fatto che ogni implicazione è doppia, mentre è accidentale il fatto che siano stati collegati tutti e soli gli ingressi. Scelti questi ultimi ed assegnate le opportune condizioni iniziali, questa rete può essere eseguita e si mostra in grado di svolgere il compito assegnato.

Le condizioni iniziali sono state determinate attraverso simulazione manuale. Altre condizioni scelte sulla base di considerazioni astratte non hanno dato il risultato sperato a causa dell'incompletezza della funzione inferenza, ma dall'esame della rete è emerso un diverso modo di scrivere la specifica che non risente dell'inconveniente. Questo evidenzia il ruolo che il linguaggio grafico adottato può svolgere durante la messa a punto di una specifica eseguibile.

TINX si è rivelato capace di determinare completamente la storia delle uscite a partire dall'istante di disponibilità degli ingressi, specificato nel sorgente attraverso l'apposito offset. I risultati di due esecuzioni sono visualizzati in figura 4.14 tramite un programma di manipolazione dei segnali, con l'origine posta in corrispondenza dell'istante di lancio dell'esecutore. I tracciati in alto corrispondono ad una situazione in cui l'equipaggio riesce per due volte a riprendere il controllo della reazione, mentre quelli in basso ne descrivono una peggiore che costringe il computer ad ordinare lo spengimento di emergenza.

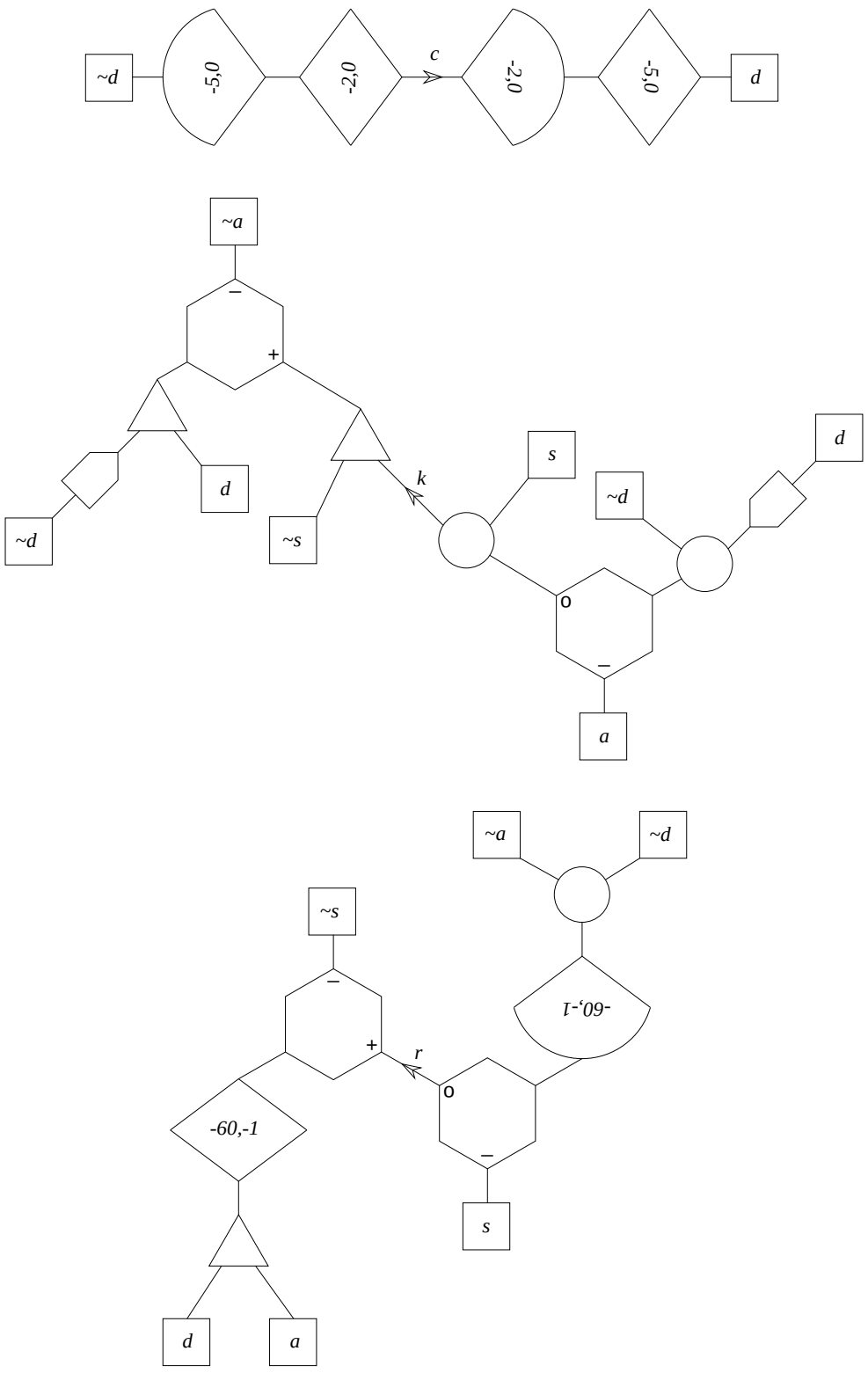

Figura 4.13. Rete generata dalla specifica del sistema di controllo

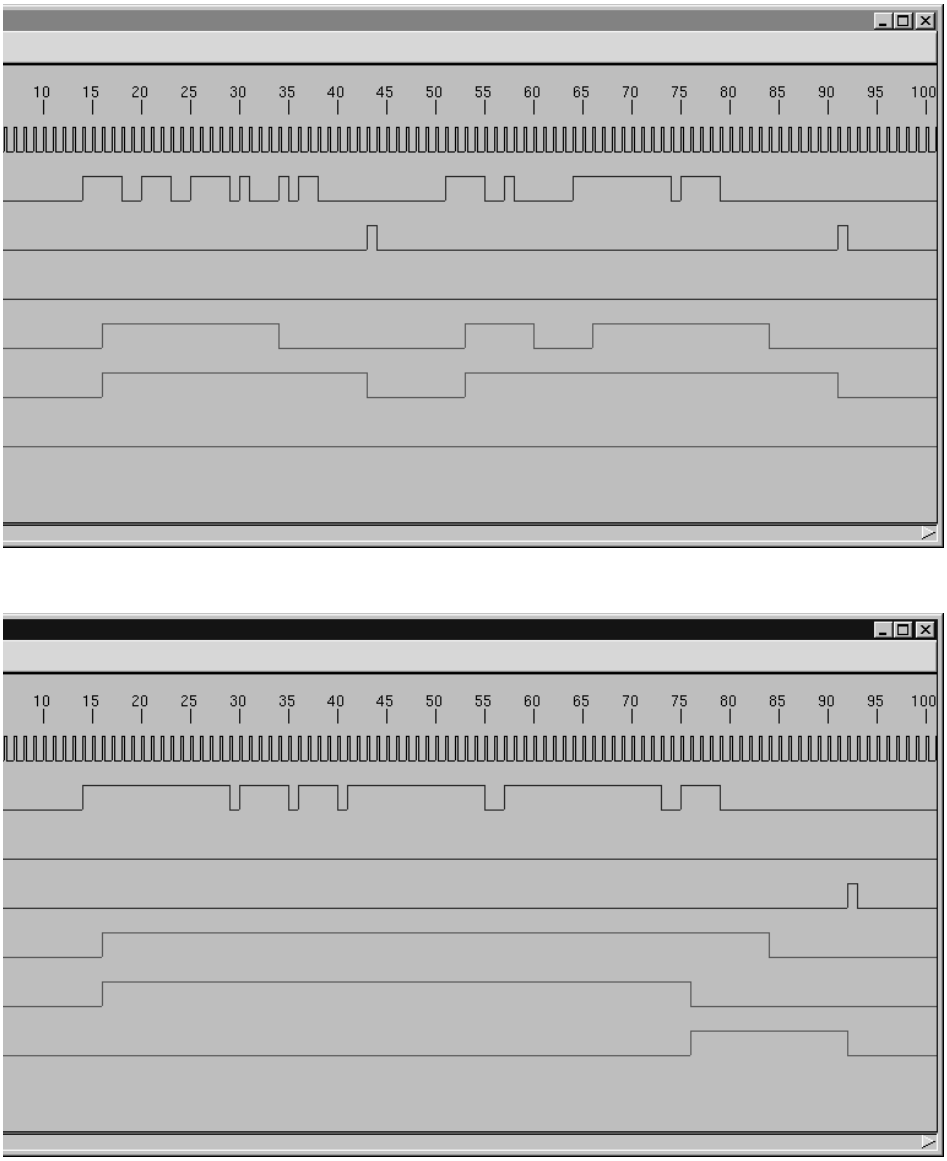

Figura 4.14. Due esempi di esecuzioni

Numerosi altri esperimenti sono stati compiuti abilitando l'opzione di inferenza causale, con risultati analoghi. Il sistema specificato verrà quindi eseguito in tempo reale, pur di disporre di un hardware sottostante adeguato al passo di discretizzazione.

Capitolo 5

Conclusioni

Il risultato teorico più rilevante dimostrato in questa tesi è probabilmente il teorema di tempo reale. Per questa ragione viene riformulato di seguito, in una versione molto compatta e leggermente più generale in modo da consentirne una più ampia applicazione. Tra le varianti introdotte, la definizione di funzione di inferenza locale è stata resa meno stringente per permettere la scelta della forma di più comodo calcolo nei diversi casi, ma continua a garantire alla procedura di chiusura progressiva le medesime proprietà. Anche i vincoli sulla struttura dei campioni in ingresso sono stati parimenti rilassati. Sebbene non sia fornita una nuova dimostrazione del teorema, per convincersi della sua validità è sufficiente integrare la precedente dimostrazione con alcuni risultati minori esposti nello stesso capitolo.

Chiamando:

- E un insieme finito di asserzioni logiche istantanee
- T un intervallo temporale discreto
- Ω^* l'insieme obiettivo
- Ω_0 l'insieme di conoscenza iniziale
- Φ l'insieme di conoscenza disponibile
- ω la funzione di ingresso
- ϕ la funzione di inferenza
- λ una funzione di inferenza locale associata a ϕ

ed indicando con $x^{(k)}$ il valore della variabile x dopo k iterazioni e con $\prod_T$ l'operatore di proiezione su T, se valgono le ipotesi:

1. $\Omega_0 \subseteq \Omega^* \subseteq E \times T$
2. $\forall\, k \in [1, \infty)\ \ \Omega_k = \omega^{(k)} \subseteq \Omega^*$
3. $\phi(\emptyset) = \emptyset$
4. $\forall\, S \subseteq \Omega^*\ \ S \subseteq \phi(S) \subseteq \Omega^*$
5. $\forall\, R, S \subseteq \Omega^*\ \ R \subseteq S \Rightarrow \phi(R) \subseteq \phi(S)$
6. $\forall\, r \in \Omega^*, S \subseteq \Omega^*\ \ \phi(\{r\} \cup S) \setminus (\{r\} \cup \phi(S)) \subseteq \lambda(r \mid S) \subseteq \Omega^*$
7. $\forall\, e \mathbin{@} t \in \Omega^*, S \subseteq \Omega^*\ \ e' \mathbin{@} t' \in \lambda(e \mathbin{@} t \mid S) \Rightarrow t \leq t'$
8. $\forall\, k \in [0, \infty)\ \ \tau_k = \min(\prod_T(\Omega_k) \cup \{+\infty\})$

9. $\forall\, i, j \in [1, \infty)\ \ i < j \Rightarrow 0 \leq (\tau_j - \tau_i) \cdot \operatorname{card} E < j - i$

allora la procedura di chiusura progressiva:

1. $\Delta \leftarrow \Omega_0$
2. $\Phi \leftarrow \Omega_0$
3. Finché $\operatorname{card} \Phi < \operatorname{card} \Omega^*$ ripeti...
   (a) $\Psi \leftarrow \omega(\bullet) \setminus \Phi$
   (b) Se $\Delta \neq \emptyset$ allora...
      i. $\exists\, e \,@\, t \in \Delta \mid \forall\, e' \,@\, t' \in \Delta\ \ t \leq t'$
      ii. $\Psi \leftarrow \Psi \cup \lambda(e \,@\, t \mid \Phi \setminus \Delta) \setminus \Phi$
      iii. $\Delta \leftarrow \Delta \setminus \{e \,@\, t\}$
   (c) $\Delta \leftarrow \Delta \cup \Psi$
   (d) $\Phi \leftarrow \Phi \cup \Psi$

produce la conoscenza inferibile rispettando i vincoli di ordinamento definiti dalla relazione:

$$\forall\, k \geq \frac{d \cdot (d+1)}{2} + 1,\ d = \max\{0, \tau_1 - \tau_0\} \cdot \operatorname{card} E,\ \forall\, t \leq \tau_k,\ \forall\, n \geq k + \operatorname{card} E - 1$$
$$e \,@\, t \in \lim_{i \to \infty} \phi^i\Big(\bigcup_{h=0}^{\infty} \Omega_h\Big) \Rightarrow e \,@\, t \in \Phi^{(n)} \subseteq \Omega^*$$

Esprimendo il teorema attraverso termini più intuitivi, se:

- La funzione inferenza determina conclusioni relative ad un presente o futuro ignoto a partire da premesse relative ad un presente o passato noto
- La procedura di chiusura progressiva esegue entro un tempo minore del passo di discretizzazione un numero di iterazioni pari al numero di asserzioni logiche istantanee scelto per caratterizzare la configurazione del sistema specificato

allora da un certa iterazione in poi il ritardo ingresso-uscita introdotto dal processo d'inferenza è minore del passo di discretizzazione.

La generalità delle ipotesi e la rilevanza pratica della tesi suggerisce un vasto ambito di applicazione. Non esistendo in letteratura alcun risultato analogo, questo teorema costituisce un punto di riferimento con cui dovrà confrontarsi qualunque motore di inferenza soggetto a vincoli di tempo reale.

Per quanto riguarda le reti di inferenza temporale, è assai più difficile fare una sintesi. La teoria in cui si inquadrano è probabilmente suscettibile di miglioramento, ma già sufficiente ad evidenziare alcune loro caratteristiche notevoli:

- Presentano analogie con le reti logiche, ma descrivono relazioni anziché funzioni e prescindono quindi dai concetti di ingresso o uscita
- Possono essere accoppiate strettamente alla sintassi di una specifica e rappresentarne così la semantica, ma un loro uso non sintattico consente maggiore libertà espressiva
- Ammettono una immediata interpretazione operazionale che le rende adatte ad implementare macchine sequenziali, anche in presenza di parziale disponibilità delle storie degli ingressi
- Possono essere usate per specificare sistemi non causali
- La loro dimensione è indipendente dall'estensione dell'orizzonte temporale e dai ritardi

Appare inoltre rassicurante il fatto che alcune delle reti presentate siano state realizzate intuitivamente e solo in un secondo momento ne sia stata giustificata la struttura a partire dalla sintassi della loro definizione. Se poi questo linguaggio grafico costituisca una reale alternativa ai formalismi in uso, solo la pratica potrà stabilirlo.

Per concludere, una considerazione di carattere generale. Con l'evoluzione della tecnologia è ragionevole attendersi una sempre maggiore diffusione di linguaggi "non convenzionali", capaci di semplificare l'interazione tra uomo e macchina fino a lasciare al primo solo il compito di formulare gli obiettivi desiderati. Ancor prima di servire per comunicarli, la traduzione di questi desideri in un simile linguaggio dovrebbe aiutare a comprenderli con maggiore chiarezza ed in questo senso i linguaggi logici forniscono un utile contributo. Il loro studio mostra come spesso un "fine" determini il "mezzo" e viceversa, fino a rendere assai labile la distinzione tra questi due concetti. Trascurando il secondo, difficilmente si otterrà ciò che davvero si desidera, ma al più solo ciò che si è chiesto.

Appendice A

Sorgenti C

A.1 tinx.h

```
/*
  TINX - Temporal Inference Network eXecutor
  Design & coding by Andrea Giotti, 1998-1999
*/

/* #define NDEBUG */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <signal.h>
#include <assert.h>

typedef char bool;

#define FALSE 0
#define TRUE (!FALSE)

typedef int d_time;

#define TIME_FMT "%d"
#define TIME_LIMIT 1024

typedef enum link_code
{
  no_link = -1,
  parent,
  left_son,
  right_son,
  LINK_CODES_NUMBER
} link_code;

typedef struct node node;

typedef struct arc
{
  node *vp;
  link_code lc;
} arc;

typedef struct event
{
  arc e;
  d_time t;
} event;
```

```
#define ev_eq(R, S) ((R).e.vp == (S).e.vp && (R).e.lc == (S).e.lc && \
                     (R).t == (S).t)

#if defined NDEBUG
#define valid(S) ((S).e.vp)
#else
#define valid(S) ((S).e.vp && (S).e.lc >= 0 && (S).e.lc < LINK_CODES_NUMBER && \
                  (S).t >= 0 && (S).t < TIME_LIMIT)
#endif

typedef struct record
{
  bool stated;
  bool chosen;
  event other;
  event next;
} record;

typedef enum stream_class
{
  input_stream,
  output_stream,
  STREAM_CLASSES_NUMBER
} stream_class;

typedef struct stream stream;

typedef struct link
{
  arc e;
  /* Attributes of arc_neg(e) */
  record history[TIME_LIMIT];
  stream *io_stream[STREAM_CLASSES_NUMBER];
} link;

#define MAX_STRLEN 32
#define MAX_NAMELEN 24
#define MAX_NAMELEN_C "23"

struct stream
{
  char name[MAX_NAMELEN];
  stream_class class;
  link *pin;
  d_time offset;
  char file_name[MAX_STRLEN];
  FILE *fp;
  stream *next_ios;
  stream *prev_ios;
  bool open;
};

typedef enum node_class
{
  null_op = -1,
  gate,
  joint,
  delay,
  NODE_CLASSES_NUMBER
} node_class;

#define CLASS_SYMBOLS "GJD"

struct node
{
  char name[MAX_NAMELEN];
```

```
  node_class class;
  d_time k;
  link pin[LINK_CODES_NUMBER];
  node *vp;
};

#define link_of(E) ((E).vp->pin[(E).lc])
#define arc_neg(E) (link_of(E).e)
#define record_of(S) (link_of((S).e).history[(S).t])
#define stated(S) (record_of(S).stated)
#define chosen(S) (record_of(S).chosen)
#define other(S) (record_of(S).other)
#define next(S) (record_of(S).next)

#define HASH_SIZE 8191 /* Prime */
#define HASH_FMT "%X"
#define hash(X) (strtoul((X), NULL, 36) % HASH_SIZE)

typedef struct k_base
{
  event focus;
  node *network;
  node *table[HASH_SIZE];
  stream *io_stream[STREAM_CLASSES_NUMBER];
  bool strictly_causal;
  bool soundness_check;
  bool trace_focus;
  bool echo_stdout;
  FILE *logfp;
  bool idle;
} k_base;

#define BLANKS " \t\n\r"
#define SEPARATORS BLANKS"(,;:?!)@"
#define NAME_FMT "%"MAX_NAMELEN_C"[^"SEPARATORS"]"
#define OP_FMT "%c"
#define ARG_FMT TIME_FMT

#define DEFAULT_NAME "default"
#define STATE_SUFFIX "_ic"
#define LOG_SUFFIX "_log"
#define NETWORK_EXT ".tin"
#define EVENT_LIST_EXT ".evl"
#define STREAM_EXT ".io"

#define LO_CHAR '0'
#define HI_CHAR '1'
#define END_CHAR '.'
#define TERM_CHAR '\x1b' /* Escape */

/* Protos */

event ev_neg(event s);
arc arc_between(node *vp, node *wp);

void state(k_base *kb, event s);
event choose(k_base *kb);
void process(k_base *kb, event s);
bool loop(k_base *kb);

event input(stream *ios);
void output(event s);
bool scan_inputs(k_base *kb);
void trace(k_base *kb);
stream *open_stream(char *name, stream_class class, arc e, d_time offset);
void close_stream(stream *ios);
```

```
node *alloc_node(char *name);
node *name2node(k_base *kb, char *name);
void thread_network(node *network);
k_base *open_base(char *base_name, char *logfile_name,
                  bool strictly_causal, bool soundness_check, bool echo_stdout);
void close_base(k_base *kb);
void init_state(k_base *kb, char *state_name);

void trap(void);
void run(char *base_name, char *state_name, char *logfile_name,
         bool strictly_causal, bool soundness_check, bool echo_stdout);

/* End of protos */
```

A.2 tinx.c

```
/*
  TINX - Temporal Inference Network eXecutor
  Design & coding by Andrea Giotti, 1998-1999
*/

#include "tinx.h"

#define VER "1.00"

const event null_event = {{NULL, no_link}, -1};

const char class_symbol[NODE_CLASSES_NUMBER] = CLASS_SYMBOLS;

volatile sig_atomic_t irq = FALSE;

/******** Tools ********/

event ev_neg(event s)
{
  event r;

  assert(valid(s));

  r.e = arc_neg(s.e);
  r.t = s.t;

  return r;
}

arc arc_between(node *vp, node *wp)
{
  arc e;
  link_code lc;

  assert(vp && wp);

  e.vp = wp;
  e.lc = no_link;

  for(lc = parent; lc < LINK_CODES_NUMBER; lc++)
    if(vp == wp->pin[lc].e.vp)
      {
        e.lc = lc;
        break;
      }

  return e;
}

/******** Inference engine ********/

void state(k_base *kb, event s)
{
  event q, r;

  assert(kb);
  assert(valid(s));
  assert(!stated(s));

  if(kb->soundness_check && stated(ev_neg(s)))
    {
      fprintf(stderr, "(%s, %s)@"TIME_FMT": Soundness violation\n",
              arc_neg(s.e).vp->name, s.e.vp->name, s.t);
      close_base(kb);
      exit(EXIT_FAILURE);
```

```
    }

  stated(s) = TRUE;

  q = null_event;
  r = kb->focus;
  while(valid(r) && r.t < s.t)
    {
      q = r;
      r = next(r);
    }

  if(valid(q))
    next(q) = s;
  else
    kb->focus = s;

  if(r.t == s.t)
    {
      assert(valid(r));

      other(s) = r;
      next(s) = next(r);
    }
  else
    {
      other(s) = null_event;
      next(s) = r;
    }

#if !defined NDEBUG
  printf("\t(%s, %s)@"TIME_FMT"\n", arc_neg(s.e).vp->name, s.e.vp->name, s.t);
#endif

  if(link_of(s.e).io_stream[output_stream])
    output(s);
}

event choose(k_base *kb)
{
  event r, s;

  assert(kb);

  s = kb->focus;

  if(valid(s))
    {
      assert(!chosen(s));

      chosen(s) = TRUE;
      r = other(s);

      if(valid(r))
        {
          next(r) = next(s);
          kb->focus = r;
        }
      else
        kb->focus = next(s);

#if !defined NDEBUG
      printf("(%s, %s)@"TIME_FMT"\n", arc_neg(s.e).vp->name, s.e.vp->name, s.t);
#endif
    }

  return s;
```

```
}

void process(k_base *kb, event s)
{
  event r;
  link_code lc1, lc2;

  assert(kb);
  assert(valid(s));

  switch(s.e.vp->class)
    {
    case gate:

      lc1 = (s.e.lc + 1) % LINK_CODES_NUMBER;
      lc2 = (s.e.lc + 2) % LINK_CODES_NUMBER;

      if(s.e.vp->pin[lc1].history[s.t].chosen)
        {
          r.e = s.e.vp->pin[lc2].e;
          r.t = s.t;

          if(!stated(r))
            state(kb, r);
        }
      else
        if(s.e.vp->pin[lc2].history[s.t].chosen)
          {
            r.e = s.e.vp->pin[lc1].e;
            r.t = s.t;

            if(!stated(r))
              state(kb, r);
          }

      break;

    case joint:

      r.t = s.t;

      if(s.e.lc == parent)
        {
          r.e = s.e.vp->pin[left_son].e;

          if(!stated(r))
            state(kb, r);

          r.e = s.e.vp->pin[right_son].e;

          if(!stated(r))
            state(kb, r);
        }
      else
        {
          r.e = s.e.vp->pin[parent].e;

          if(!stated(r))
            state(kb, r);
        }

      break;

    case delay:

      if(s.e.lc == parent)
        {
```

```
          r.e = s.e.vp->pin[left_son].e;
          r.t = s.t - s.e.vp->k;
        }
      else
        {
          r.e = s.e.vp->pin[parent].e;
          r.t = s.t + s.e.vp->k;
        }

      if(kb->strictly_causal)
        {
          if(r.t >= s.t && r.t < TIME_LIMIT && !stated(r))
            state(kb, r);
        }
      else
        if(r.t >= 0 && r.t < TIME_LIMIT && !stated(r))
          state(kb, r);

      break;

    default:
      assert(FALSE);
    }
}

bool loop(k_base *kb)
{
  event s;

  if(kb->trace_focus)
    trace(kb);

  s = choose(kb);

  if(valid(s))
    {
      process(kb, s);
      scan_inputs(kb);
      return TRUE;
    }
  else
    return scan_inputs(kb);
}

/******** Input/Output ********/

event input(stream *ios)
{
  event s;
  char c;

  s.t = ftell(ios->fp);
  if(s.t == -1)
    {
      perror(ios->file_name);
      exit(EXIT_FAILURE);
    }
  else
    s.t += ios->offset;

  c = fgetc(ios->fp);

  switch(c)
    {
    case EOF:
      if(ferror(ios->fp))
        {
```

```
          perror(ios->file_name);
          exit(EXIT_FAILURE);
        }
      else
        {
          clearerr(ios->fp);
          s = null_event;
        }
      break;

    case '\0':
      s = null_event;
      break;

    case LO_CHAR:
      if(s.t < 0 || s.t >= TIME_LIMIT)
        s = null_event;
      else
        s.e = ios->pin->e;
      break;

    case HI_CHAR:
      if(s.t < 0 || s.t >= TIME_LIMIT)
        s = null_event;
      else
        s.e = arc_neg(ios->pin->e);
      break;

    case END_CHAR:
      ios->open = FALSE;
      s = null_event;
      break;

    case TERM_CHAR:
      irq = TRUE;
      s = null_event;
      break;

    default:
      fprintf(stderr, "%s, %c (dec %d): Invalid character in stream\n",
              ios->file_name, c, c);
      exit(EXIT_FAILURE);
    }

  return s;
}

void output(event s)
{
  stream *ios;

  ios = link_of(s.e).io_stream[output_stream];

  if(s.t - ios->offset >= 0)
    {
      if(fseek(ios->fp, s.t - ios->offset, SEEK_SET))
        {
          perror(ios->file_name);
          exit(EXIT_FAILURE);
        }

      if(fputc((ios->pin->e.vp == s.e.vp)? LO_CHAR : HI_CHAR, ios->fp) == EOF)
        {
          perror(ios->file_name);
          exit(EXIT_FAILURE);
        }
```

```
      if(fflush(ios->fp))
        {
          perror(ios->file_name);
          exit(EXIT_FAILURE);
        }
    }
}

bool scan_inputs(k_base *kb)
{
  stream *ios, *next_ios;
  event s;

  ios = kb->io_stream[input_stream];
  while(ios)
    {
      if(ios->open)
        {
          s = input(ios);

          if(valid(s) && !stated(s))
            state(kb, s);

          ios = ios->next_ios;
        }
      else
        {
          next_ios = ios->next_ios;

          if(next_ios)
            next_ios->prev_ios = ios->prev_ios;

          if(ios->prev_ios)
            ios->prev_ios->next_ios = next_ios;
          else
            kb->io_stream[input_stream] = next_ios;

          close_stream(ios);
          ios = next_ios;
        }
    }

  return (kb->io_stream[input_stream] != NULL);
}

void trace(k_base *kb)
{
  event s;

  if(valid(kb->focus))
    {
      s = kb->focus;
      kb->idle = FALSE;

      if(kb->echo_stdout)
        {
          printf(" > (%s, %s)@"TIME_FMT"        \r",
                 arc_neg(s.e).vp->name, s.e.vp->name, s.t);
          fflush(stdout);
        }

      if(kb->logfp)
        {
          if(fprintf(kb->logfp, "(%s, %s)@"TIME_FMT"\n",
                     arc_neg(s.e).vp->name, s.e.vp->name, s.t) < 0)
            {
              perror(NULL);
```

```
            exit(EXIT_FAILURE);
          }

        if(fflush(kb->logfp))
          {
            perror(NULL);
            exit(EXIT_FAILURE);
          }
      }
  }
else
  if(!kb->idle)
    {
      kb->idle = TRUE;

      if(kb->echo_stdout)
        {
          printf(" *\r");
          fflush(stdout);
        }
    }
}

stream *open_stream(char *name, stream_class class, arc e, d_time offset)
{
  stream *ios;

  ios = malloc(sizeof(stream));
  if(!ios)
    {
      perror(NULL);
      exit(EXIT_FAILURE);
    }

  strcpy(ios->name, name);
  ios->class = class;

  ios->pin = &link_of(e);
  ios->pin->io_stream[class] = ios;
  link_of(ios->pin->e).io_stream[class] = ios;

  ios->offset = offset;

  strcpy(ios->file_name, name);
  strcat(ios->file_name, STREAM_EXT);

  ios->fp = fopen(ios->file_name, (class == input_stream)? "r" : "w");

  if(!ios->fp)
    {
      perror(ios->file_name);
      exit(EXIT_FAILURE);
    }

  ios->open = TRUE;

  return ios;
}

void close_stream(stream *ios)
{
  ios->pin->io_stream[ios->class] = NULL;
  link_of(ios->pin->e).io_stream[ios->class] = NULL;

  if(ios->class == output_stream &&
     (fseek(ios->fp, 0, SEEK_END) || fputc(END_CHAR, ios->fp) == EOF))
    {
```

```
      perror(ios->file_name);
      exit(EXIT_FAILURE);
    }

  if(fclose(ios->fp))
    {
      perror(ios->file_name);
      exit(EXIT_FAILURE);
    }

  free(ios);
}

/******** Network setup ********/

node *alloc_node(char *name)
{
  node *vp;
  link_code lc;
  d_time t;

  vp = malloc(sizeof(node));
  if(!vp)
    {
      perror(NULL);
      exit(EXIT_FAILURE);
    }

  strcpy(vp->name, name);
  vp->class = null_op;

  for(lc = 0; lc < LINK_CODES_NUMBER; lc++)
    {
      vp->pin[lc].e.vp = NULL;
      vp->pin[lc].e.lc = no_link;

      for(t = 0; t < TIME_LIMIT; t++)
        {
          vp->pin[lc].history[t].stated = FALSE;
          vp->pin[lc].history[t].chosen = FALSE;
          vp->pin[lc].history[t].other = null_event;
          vp->pin[lc].history[t].next = null_event;
        }

      vp->pin[lc].io_stream[input_stream] = NULL;
      vp->pin[lc].io_stream[output_stream] = NULL;
    }

  return vp;
}

node *name2node(k_base *kb, char *name)
{
  node *vp;
  int h;

  if(*name)
    {
      h = hash(name);

      if(kb->table[h])
        {
          vp = kb->table[h];
          if(strcmp(vp->name, name))
            {
              fprintf(stderr, "%s, %s: Node names generate duplicate hashes\n",
                      vp->name, name);
```

```
              exit(EXIT_FAILURE);
            }
        }
      else
        {
          vp = alloc_node(name);
          assert(vp);

          vp->vp = kb->network;
          kb->network = vp;

          kb->table[h] = vp;

#if !defined NDEBUG
          printf("%s -> "HASH_FMT"\n", name, h);
#endif
        }
      return vp;
    }
  else
    return NULL;
}

void thread_network(node *network)
{
  node *vp;
  link_code lc, lc1;

  for(vp = network; vp; vp = vp->vp)
    for(lc = 0; lc < LINK_CODES_NUMBER && vp->pin[lc].e.vp; lc++)
      if(vp->pin[lc].e.lc < 0)
        {
          for(lc1 = 0; lc1 < LINK_CODES_NUMBER; lc1++)
            {
              if(!vp->pin[lc].e.vp->pin[lc1].e.vp)
                vp->pin[lc].e.vp->pin[lc1].e.vp = vp;

              if(vp->pin[lc].e.vp->pin[lc1].e.vp == vp &&
                 vp->pin[lc].e.vp->pin[lc1].e.lc < 0)
                {
                  vp->pin[lc].e.vp->pin[lc1].e.lc = lc;
                  vp->pin[lc].e.lc = lc1;

                  break;
                }
            }
          if(lc1 == LINK_CODES_NUMBER)
            {
              fprintf(stderr, "%s: Arc mismatch in node declarations\n",
                      vp->name);
              exit(EXIT_FAILURE);
            }
        }

  for(vp = network; vp; vp = vp->vp)
    {
      for(lc = 0; lc < LINK_CODES_NUMBER && vp->pin[lc].e.lc >= 0; lc++);

      switch(vp->class)
        {
        case gate:
        case joint:
          assert(lc <= 3);
          if(lc < 3)
            {
              fprintf(stderr, "%s: Undefined arc in node declarations\n",
                      vp->name);
```

```
              exit(EXIT_FAILURE);
            }
          break;

        case delay:
          if(lc < 2)
            {
              fprintf(stderr, "%s: Undefined arc in node declarations\n",
                      vp->name);
              exit(EXIT_FAILURE);
            }
          else
            if(lc > 2)
              {
                fprintf(stderr, "%s: Arc mismatch in node declarations\n",
                        vp->name);
                exit(EXIT_FAILURE);
              }
          break;

        default:
          fprintf(stderr, "%s: Reference to undefined node\n", vp->name);
          exit(EXIT_FAILURE);
        }

#if !defined NDEBUG
      printf("%s: "OP_FMT""TIME_FMT" ; %s", vp->name,
             class_symbol[vp->class], vp->k, vp->pin[parent].e.vp->name);

      if(vp->pin[left_son].e.vp)
        printf(", %s", vp->pin[left_son].e.vp->name);

      if(vp->pin[right_son].e.vp)
        printf(", %s", vp->pin[right_son].e.vp->name);

      printf("\n");
#endif
    }
}

k_base *open_base(char *base_name, char *logfile_name,
                  bool strictly_causal, bool soundness_check, bool echo_stdout)
{
  k_base *kb;
  FILE *fp;
  char file_name[MAX_STRLEN], name[MAX_NAMELEN], type[MAX_NAMELEN],
       up[MAX_NAMELEN], left[MAX_NAMELEN], right[MAX_NAMELEN],
       name_v[MAX_NAMELEN], name_w[MAX_NAMELEN];
  char c;
  d_time k;
  node *vp, *wp;
  node_class nclass;
  stream *ios;
  stream_class sclass;
  arc e;

  kb = malloc(sizeof(k_base));
  if(!kb)
    {
      perror(NULL);
      exit(EXIT_FAILURE);
    }

  kb->focus = null_event;
  kb->network = NULL;

  memset(kb->table, 0, sizeof(node *) * HASH_SIZE);
```

```
memset(kb->io_stream, 0, sizeof(stream *) * STREAM_CLASSES_NUMBER);

strcpy(file_name, base_name);
strcat(file_name, NETWORK_EXT);
fp = fopen(file_name, "r");
if(!fp)
  {
    perror(file_name);
    exit(EXIT_FAILURE);
  }

*left = *right = '\0';
while(fscanf(fp, " "NAME_FMT" : "NAME_FMT" ; "NAME_FMT" , "
                 NAME_FMT" , "NAME_FMT" ",
             name, type, up, left, right) >= 3)
  {
    k = 1;
    sscanf(type, OP_FMT""ARG_FMT, &c, &k);

    nclass = strchr(class_symbol, toupper(c)) - class_symbol;
    if(nclass < 0)
      {
        fprintf(stderr, "%s, %s, "OP_FMT": Invalid node class\n",
                file_name, name, c);
        exit(EXIT_FAILURE);
      }

    if(k <= -TIME_LIMIT || k >= TIME_LIMIT)
      {
        fprintf(stderr, "%s, %s, "TIME_FMT": Delay out of range\n",
                file_name, name, k);
        exit(EXIT_FAILURE);
      }

    vp = name2node(kb, name);
    if(vp->class >= 0)
      {
        fprintf(stderr, "%s, %s: Duplicate node\n", file_name, name);
        exit(EXIT_FAILURE);
      }

    vp->class = nclass;
    vp->k = k;

    vp->pin[parent].e.vp = name2node(kb, up);
    vp->pin[left_son].e.vp = name2node(kb, left);
    vp->pin[right_son].e.vp = name2node(kb, right);

    *left = *right = '\0';
  }

if(ferror(fp))
  {
    perror(file_name);
    exit(EXIT_FAILURE);
  }

thread_network(kb->network);

k = 0;
while(fscanf(fp, " "OP_FMT" "NAME_FMT" ( "NAME_FMT" , "NAME_FMT" ) @ "
                 TIME_FMT" ",
             &c, name, name_v, name_w, &k) >= 4)
  {
    vp = kb->table[hash(name_v)];
    wp = kb->table[hash(name_w)];
```

```
    if(!vp || !wp)
      {
        fprintf(stderr, "%s, %s: Undefined node\n",
                file_name, !vp? name_v : name_w);
        exit(EXIT_FAILURE);
      }

    e = arc_between(vp, wp);
    if(e.lc < 0)
      {
        fprintf(stderr, "%s, (%s, %s): Undefined arc\n",
                file_name, name_v, name_w);
        exit(EXIT_FAILURE);
      }

    if(k <= -TIME_LIMIT || k >= TIME_LIMIT)
      {
        fprintf(stderr, "%s, "TIME_FMT": Offset out of range\n",
                file_name, k);
        exit(EXIT_FAILURE);
      }

    switch(c)
      {
      case '!':
        sclass = input_stream;
        break;

      case '?':
        sclass = output_stream;
        break;

      default:
        fprintf(stderr, "%s, "OP_FMT": Invalid stream class\n", file_name, c);
        exit(EXIT_FAILURE);
      }

    if(link_of(e).io_stream[sclass])
      {
        fprintf(stderr, "%s, "OP_FMT"%s (%s, %s): Duplicate stream\n",
                file_name, c, name, name_v, name_w);
        exit(EXIT_FAILURE);
      }

    ios = open_stream(name, sclass, e, k);

    ios->next_ios = kb->io_stream[sclass];
    ios->prev_ios = NULL;

    if(ios->next_ios)
      ios->next_ios->prev_ios = ios;

    kb->io_stream[sclass] = ios;

    k = 0;
  }

if(ferror(fp))
  {
    perror(file_name);
    exit(EXIT_FAILURE);
  }

if(!feof(fp))
  {
    fprintf(stderr, "%s: Parser error\n", file_name);
    exit(EXIT_FAILURE);
```

```
    }

  if(fclose(fp))
    {
      perror(file_name);
      exit(EXIT_FAILURE);
    }

  kb->strictly_causal = strictly_causal;
  kb->soundness_check = soundness_check;
  kb->trace_focus = echo_stdout || logfile_name;
  kb->echo_stdout = echo_stdout;

  if(logfile_name)
    {
      strcpy(file_name, logfile_name);
      strcat(file_name, EVENT_LIST_EXT);
      kb->logfp = fopen(file_name, "w");
      if(!kb->logfp)
        {
          perror(file_name);
          exit(EXIT_FAILURE);
        }
    }

  kb->idle = TRUE;

  return kb;
}

void close_base(k_base *kb)
{
  node *vp, *wp;
  stream *ios, *next_ios;
  stream_class class;

  if(kb->logfp && fclose(kb->logfp))
    {
      perror(NULL);
      exit(EXIT_FAILURE);
    }

  for(class = 0; class < STREAM_CLASSES_NUMBER; class++)
    {
      ios = kb->io_stream[class];
      while(ios)
        {
          next_ios = ios->next_ios;
          close_stream(ios);
          ios = next_ios;
        }
    }

  vp = kb->network;
  while(vp)
    {
      wp = vp->vp;
      free(vp);
      vp = wp;
    }

  free(kb);
}

void init_state(k_base *kb, char *state_name)
{
  FILE *fp;
```

```
  char file_name[MAX_STRLEN], name_v[MAX_NAMELEN], name_w[MAX_NAMELEN];
  node *vp, *wp;
  event s;

  strcpy(file_name, state_name);
  strcat(file_name, EVENT_LIST_EXT);
  fp = fopen(file_name, "r");
  if(!fp)
    {
      perror(file_name);
      exit(EXIT_FAILURE);
    }

  while(fscanf(fp, " ( "NAME_FMT" , "NAME_FMT" ) @ "TIME_FMT" ",
               name_v, name_w, &s.t) == 3)
    {
      vp = kb->table[hash(name_v)];
      wp = kb->table[hash(name_w)];

      if(!vp || !wp)
        {
          fprintf(stderr, "%s, %s: Undefined node\n",
                  file_name, !vp? name_v : name_w);
          exit(EXIT_FAILURE);
        }

      s.e = arc_between(vp, wp);
      if(s.e.lc < 0)
        {
          fprintf(stderr, "%s, (%s, %s): Undefined arc\n",
                  file_name, name_v, name_w);
          exit(EXIT_FAILURE);
        }

      if(s.t < 0 || s.t >= TIME_LIMIT)
        {
          fprintf(stderr, "%s, "TIME_FMT": Time out of range\n",
                  file_name, s.t);
          exit(EXIT_FAILURE);
        }

      if(!stated(s))
        state(kb, s);
    }

  if(ferror(fp))
    {
      perror(file_name);
      exit(EXIT_FAILURE);
    }

  if(!feof(fp))
    {
      fprintf(stderr, "%s: Parser error\n", file_name);
      exit(EXIT_FAILURE);
    }

  if(fclose(fp))
    {
      perror(file_name);
      exit(EXIT_FAILURE);
    }
}

/******** Toplevel ********/

void trap()
```

```
{
  irq = TRUE;
}

void run(char *base_name, char *state_name, char *logfile_name,
         bool strictly_causal, bool soundness_check, bool echo_stdout)
{
  k_base *kb;

  kb = open_base(base_name, logfile_name,
                 strictly_causal, soundness_check, echo_stdout);
  if(state_name)
    init_state(kb, state_name);

  signal(SIGINT, (void (*)())&trap);
  while(!irq && loop(kb));
  signal(SIGINT, SIG_DFL);

  close_base(kb);
}

int main(int argc, char **argv)
{
  char *base_name, *state_name, *logfile_name, *option;
  char default_state_name[MAX_STRLEN], default_logfile_name[MAX_STRLEN];
  bool strictly_causal, soundness_check, echo_stdout;
  int i;

  base_name = state_name = logfile_name = NULL;
  strictly_causal = soundness_check = echo_stdout = FALSE;

  for(i = 1; i < argc; i++)
    {
      if(*argv[i] == '-')
        {
          option = argv[i] + 1;
          switch(*option)
            {
            case 'h':
              fprintf(stderr, "Usage: %s [-cdilv] [-I state] [-L log] [base]\n",
                      argv[0]);
              exit(EXIT_SUCCESS);
              break;

            case 'I':
              if(*(++option))
                {
                  fprintf(stderr, "%s, %c: Invalid command line option"
                                  " (%s -h for help)\n",
                          argv[i], *option, argv[0]);
                  exit(EXIT_FAILURE);
                }

              if(state_name && state_name != default_state_name)
                {
                  fprintf(stderr, "%s: Duplicate option\n", argv[i]);
                  exit(EXIT_FAILURE);
                }

              if(++i < argc)
                state_name = argv[i];
              else
                {
                  fprintf(stderr, "%s: Missing argument\n", argv[--i]);
                  exit(EXIT_FAILURE);
                }
              break;
```

```
      case 'L':
        if(*(++option))
          {
            fprintf(stderr, "%s, %c: Invalid command line option"
                            " (%s -h for help)\n",
                    argv[i], *option, argv[0]);
            exit(EXIT_FAILURE);
          }

        if(logfile_name && logfile_name != default_logfile_name)
          {
            fprintf(stderr, "%s: Duplicate option\n", argv[i]);
            exit(EXIT_FAILURE);
          }

        if(++i < argc)
          logfile_name = argv[i];
        else
          {
            fprintf(stderr, "%s: Missing argument\n", argv[--i]);
            exit(EXIT_FAILURE);
          }
        break;

      default:
        do
          {
            switch(*option)
              {
              case 'c':
                strictly_causal = TRUE;
                break;

              case 'd':
                echo_stdout = TRUE;
                break;

              case 'i':
                if(!state_name)
                  state_name = default_state_name;
                break;

              case 'l':
                if(!logfile_name)
                  logfile_name = default_logfile_name;
                break;

              case 'v':
                soundness_check = TRUE;
                break;

              default:
                fprintf(stderr, "%s, %c: Invalid command line option"
                                " (%s -h for help)\n",
                        argv[i], *option, argv[0]);
                exit(EXIT_FAILURE);
              }
          }
        while(*(++option));
      }
  }
else
  {
    if(!base_name)
      base_name = argv[i];
    else
```

```
            fprintf(stderr, "%s: Extra argument ignored\n", argv[i]);
        }
    }

  if(!base_name)
    base_name = DEFAULT_NAME;

  if(state_name == default_state_name)
    {
      strcpy(state_name, base_name);
      strcat(state_name, STATE_SUFFIX);
    }

  if(logfile_name == default_logfile_name)
    {
      strcpy(logfile_name, base_name);
      strcat(logfile_name, LOG_SUFFIX);
    }

  printf("\nTINX "VER" - Temporal Inference Network eXecutor\n"
         "Design & coding by Andrea Giotti, 1998-1999\n\n");

  run(base_name, state_name, logfile_name,
      strictly_causal, soundness_check, echo_stdout);

  if(irq)
    printf("\nExecution interrupted\n");
  else
    printf("\nEnd of execution\n");

  return EXIT_SUCCESS;
}
```

A.3 tint.c

```
/*
  TINT - Temporal Inference Network Tester
  Design & coding by Andrea Giotti, 1998-1999
*/

#include <time.h>

#if !defined CLOCKS_PER_SEC
#define CLOCKS_PER_SEC CLK_TCK
#endif

#include "tinx.h"

#define VER "0.50"

#define MAX_FILES 32
#define HORIZON_SIZE 64

#define SKIP_FMT "%*[^?!]"

#define DISPLAY_LO_CHAR '-'
#define DISPLAY_HI_CHAR '#'

/* ANSI control sequences */
#define ESC                     "\x1b"
#define CSI                     ESC "["
#define HOME                    CSI "H"
#define ERASE2EOS               CSI "J"
#define ERASE2EOL               CSI "K"
#define CLS                     HOME ERASE2EOS
#define LOCATE(i, j)            CSI i ";" j "H"
#define SET_COLORS(fg, bg)      CSI "1;3" fg ";4" bg "m"
#define RESET_COLORS            CSI "m"

#define DEFAULT_STEP .05

int main(int argc, char *argv[])
{
  FILE *bp, *fp[MAX_FILES], *gp[MAX_FILES];
  int fn, gn, i;
  float step;
  char name[MAX_STRLEN],
       fnames[MAX_FILES][MAX_STRLEN], gnames[MAX_FILES][MAX_STRLEN];
  d_time t, k, tau[MAX_FILES];
  char memory_f[MAX_FILES][HORIZON_SIZE], memory_g[MAX_FILES][HORIZON_SIZE];
  char c, ic, oc;
  clock_t deadline;
  bool quiet;

  *name = '\0';
  quiet = FALSE;
  step = DEFAULT_STEP;

  for(i = 1; i < argc; i++)
    {
      if(*argv[i] == '-')
        {
          switch(*(argv[i] + 1))
            {
            case 'h':
              fprintf(stderr, "Usage: %s [-q] [-t step] [base]\n", argv[0]);
              exit(EXIT_SUCCESS);
              break;

            case 'q':
```

```
            quiet = TRUE;
            break;

          case 't':
            if(++i < argc)
              {
                step = atof(argv[i]);
                if(step < 0)
                  {
                    fprintf(stderr, "%f: Argument out of range\n", step);
                    exit(EXIT_FAILURE);
                  }
              }
            else
              {
                fprintf(stderr, "%s: Missing argument\n", argv[--i]);
                exit(EXIT_FAILURE);
              }
            break;

          default:
            fprintf(stderr,
                    "%s: Invalid command line option (%s -h for help)\n",
                    argv[i], argv[0]);
            exit(EXIT_FAILURE);
          }
      }
    else
      {
        if(!*name)
          strcpy(name, argv[i]);
        else
          fprintf(stderr, "%s: Extra argument ignored\n", argv[i]);
      }
  }

if(!*name)
  strcpy(name, DEFAULT_NAME);

printf("\nTINT "VER" - Temporal Inference Network Tester\n"
       "Design & coding by Andrea Giotti, 1998-1999\n\n");

strcat(name, NETWORK_EXT);
bp = fopen(name, "r");

if(!bp)
  {
    perror(name);
    exit(EXIT_FAILURE);
  }

fn = gn = 0;
while(fscanf(bp, SKIP_FMT" "OP_FMT" "NAME_FMT, &c, name) == 2)
  {
    switch(c)
      {
      case '!':
        if(fn >= MAX_FILES)
          {
            fprintf(stderr, "Too many input files\n");
            exit(EXIT_FAILURE);
          }
        strcpy(fnames[fn], name);
        strcat(fnames[fn], STREAM_EXT);

        fp[fn] = fopen(fnames[fn], "w");
        if(!fp[fn])
```

```
          {
            perror(fnames[fn]);
            exit(EXIT_FAILURE);
          }
        fn++;
        break;

      case '?':
        if(!quiet)
          {
            if(gn >= MAX_FILES)
              {
                fprintf(stderr, "Too many output files\n");
                exit(EXIT_FAILURE);
              }
            strcpy(gnames[gn], name);
            strcat(gnames[gn], STREAM_EXT);

            remove(gnames[gn]);
            gn++;
          }
        break;

      default:
        assert(FALSE);
      }
  }

if(ferror(bp))
  {
    perror(name);
    exit(EXIT_FAILURE);
  }

if(fclose(bp))
  {
    perror(name);
    exit(EXIT_FAILURE);
  }

for(i = 0; i < fn; i++)
  for(k = 0; k < HORIZON_SIZE; k++)
    memory_f[i][k] = ' ';

if(!quiet)
  {
    printf("Waiting for TINX host...\n");

    for(i = 0; i < gn; i++)
      {
        do
          gp[i] = fopen(gnames[i], "r");
        while(!gp[i]);

        for(k = 0; k < HORIZON_SIZE; k++)
          memory_g[i][k] = ' ';

        tau[i] = 0;
      }

    printf(CLS);
  }

t = 0;
deadline = 0;
for(;;)
  {
```

```
if(clock() >= deadline)
  {
    deadline = clock() + CLOCKS_PER_SEC * step;

    if(!quiet)
      {
        printf(HOME""SET_COLORS("1","0"));

        for(i = 0; i < fn; i++)
          {
            memory_f[i][t % HORIZON_SIZE] = ' ';

            for(k = t; k < t + HORIZON_SIZE; k++)
              putchar(memory_f[i][k % HORIZON_SIZE]);

            printf(" [I] %s\n", fnames[i]);
          }

        printf(SET_COLORS("2","0"));

        for(i = 0; i < gn; i++)
          {
            while(tau[i] < t)
              {
                ic = fgetc(gp[i]);

                if(ic == EOF)
                  {
                    if(ferror(gp[i]))
                      {
                        perror(gnames[i]);
                        exit(EXIT_FAILURE);
                      }
                    else
                      {
                        clearerr(gp[i]);
                        break;
                      }
                  }

                switch(ic)
                  {
                  case LO_CHAR:
                    memory_g[i][tau[i] % HORIZON_SIZE] = DISPLAY_LO_CHAR;
                    break;

                  case HI_CHAR:
                    memory_g[i][tau[i] % HORIZON_SIZE] = DISPLAY_HI_CHAR;
                    break;

                  default:
                    memory_g[i][tau[i] % HORIZON_SIZE] = ' ';
                    break;
                  }

                tau[i]++;
              }

            memory_g[i][t % HORIZON_SIZE] = ' ';

            for(k = t; k < t + HORIZON_SIZE; k++)
              putchar(memory_g[i][k % HORIZON_SIZE]);

            printf(" [O] %s\n", gnames[i]);
          }

        printf(RESET_COLORS);
```

```
            fflush(stdout);
          }

        if(t >= TIME_LIMIT)
          break;

        for(i = 0; i < fn; i++)
          {
            oc = (rand() <= RAND_MAX / 2)? LO_CHAR : HI_CHAR;

            if(fputc(oc, fp[i]) == EOF)
              {
                perror(fnames[i]);
                exit(EXIT_FAILURE);
              }

            if(fflush(fp[i]))
              {
                perror(fnames[i]);
                exit(EXIT_FAILURE);
              }

            memory_f[i][t % HORIZON_SIZE] = (oc == LO_CHAR)?
                                            DISPLAY_LO_CHAR : DISPLAY_HI_CHAR;
          }

        t++;
      }
  }

  for(i = 0; i < fn; i++)
    {
      if(fputc(END_CHAR, fp[i]) == EOF)
        {
          perror(fnames[i]);
          exit(EXIT_FAILURE);
        }

      if(fclose(fp[i]))
        {
          perror(fnames[i]);
          exit(EXIT_FAILURE);
        }
    }

  if(!quiet)
    for(i = 0; i < gn; i++)
      if(fclose(gp[i]))
        {
          perror(gnames[i]);
          exit(EXIT_FAILURE);
        }

  printf("\nEnd of generation\n");

  return EXIT_SUCCESS;
}
```

A.4 graph.h

```
#ifndef GRAPH_H
#define GRAPH_H

#define TRUE 1
#define FALSE 0

typedef enum {n_and, n_or, n_not, n_delay, n_letteral,
              n_at, n_happen, n_until, n_since, n_notuntil, n_notsince,
              n_deleted} nodeType;

typedef struct {
  int min,max;
} Interval;

typedef struct _node {
  nodeType type;
  union {
    int delay_value;
    char *letteral_name;
    Interval interval;
  } data;
  struct _node *n1,*n2,*up;
  int neg;
  int mark;
  char *class;
} Node;

Node* newNode(void);
void deleteNode(Node* n);

void newArc(Node*,Node*);
void newArc1(Node*,Node*);

Node* newAnd(Node*,Node*);
Node* newAnd1(Node*,Node*,char*);
Node* newOr(Node*,Node*);
Node* newDelay(Node*,int);
Node* newNot(Node*);
Node* newLetteral(char*);
Node* newAt(Node*,Interval);
Node* newHappen(Node*,Interval);
Node* newUntil(Node*, Node*);
Node* newSince(Node*, Node*);

Node* newDeepCopy(Node*,Node*);

void printTree(Node*);

Node* Subst(Node*);

Node* notElim(Node*,int);
void upClose(Node*);
void downClose(void);

Node *firstLetteral(void);
Node *findCmpLetteral(void);
Node *nextLetteral(void);
Node *nextCmpLetteral(void);

void outputGraph(FILE*);

typedef enum {io_input,io_output} ioType;

typedef struct _inout {
  ioType type;
```

```
  char *name;
  Node *n1,*n2;
  int at;
  int mark;
} inout;

inout* addInOut(ioType,char* name,int at);
void setupInOut(char *name,Node*,Node*);

void outputInOut(FILE*);

typedef struct _initc {
  int tf;
  char *name;
  int atMin,atMax;
  Node *n1,*n2;
  int mark;
} initc;

initc* addIC(int, char* name, int atMin, int atMax);
void setupIC(char *name,Node*,Node*);
void outputIC(FILE*);

#endif
```

A.5 btl2tin.c

```
#include <stdio.h>
#include <string.h>
#include <assert.h>

#include "graph.h"

#define MAX_NODES 1000
#define MAX_INOUTS 100
#define MAX_INITS 100

extern FILE *yyin;
int yyparse(void);
int lineno=1;

inout inouts[MAX_INOUTS];
initc inits[MAX_INITS];
Node nodes[MAX_NODES];
int nnodes=0,ninouts=0,ninits=0;
int minIC=0;

FILE *ftin=NULL;
FILE *fic=NULL;

void main(int ac, char** av)
{
  char btl_name[100],
       tin_name[100],
       ic_name[100];

  if(ac<2)
    return;

  strcpy(btl_name,av[1]);
  strcat(btl_name,".btl");
  yyin=fopen(btl_name,"r");
  if(yyin==NULL)
  {
    perror(btl_name);
    return;
  }
  strcpy(tin_name,av[1]);
  strcat(tin_name,".tin");
  ftin=fopen(tin_name,"w");
  if(ftin==NULL)
  {
    perror(tin_name);
    return;
  }

  strcpy(ic_name,av[1]);
  strcat(ic_name,"_ic.evl");
  fic=fopen(ic_name,"w");
  if(fic==NULL)
  {
    perror(ic_name);
    return;
  }

  yyparse();

  fclose(yyin);

}

Node* newNode()
```

```
{
  if(nnodes<MAX_NODES)
  {
    Node *n=&nodes[nnodes++];
    n->n1=NULL;
    n->n2=NULL;
    n->up=NULL;
    n->mark=0;
    n->class=NULL;
    return n;
  }
  return NULL;
}

Node* newCopy(Node* n1)
{
  Node* n=newNode();
  if(n)
  {
    *n=*n1;
    n->n1=NULL;
    n->n2=NULL;
    n->up=NULL;
    n->class=strdup(n->class);
    n->mark=0;
  }
  return n;
}

Node* newDeepCopy(Node* n1,Node* up)
{
  Node* n=newNode();
  if(n)
  {
    *n=*n1;
    n->n1=NULL;
    n->n2=NULL;
    n->up=NULL;
    n->class=strdup(n->class);
    n->mark=0;
    if(n1->n1)
      n->n1=newDeepCopy(n1->n1,n);
    if(n1->n2)
      n->n2=newDeepCopy(n1->n2,n);
    n->up=up;
  }
  return n;
}

void newArc(Node* from, Node* to)
{
  assert(from);
  assert(to);

  if(from->n1==NULL)
    from->n1=to;
  else
    from->n2=to;
  to->up=from;
}

void newArc1(Node* from, Node* to)
{
  assert(from);
  assert(to);

  if(from->n1==NULL)
```

```
    from->n1=to;
  else
    from->n2=to;
  //to->up=from;
}

Node* newAnd(Node* n1,Node* n2)
{
  Node* nodo=newNode();
  if(nodo!=NULL)
  {
    nodo->type=n_and;
    nodo->class="A";
    newArc(nodo,n1);
    newArc(nodo,n2);
  }
  return nodo;
}

Node* newAnd1(Node* n1,Node* n2,char *class)
{
  Node* nodo=newNode();
  if(nodo!=NULL)
  {
    nodo->type=n_and;
    nodo->class=strdup(class);
    newArc1(nodo,n1);
    newArc1(nodo,n2);
  }
  return nodo;
}

Node* newOr(Node* n1,Node* n2)
{
  Node* nodo=newNode();
  if(nodo!=NULL)
  {
    nodo->type=n_or;
    nodo->class="O";
    newArc(nodo,n1);
    newArc(nodo,n2);
  }
  return nodo;
}

Node* newDelay(Node* n,int v)
{
  Node* nodo=newNode();
  if(nodo!=NULL)
  {
    nodo->type=n_delay;
    nodo->class="D";
    nodo->data.delay_value=v;
    newArc(nodo,n);
  }
  return nodo;
}

Node* newNot(Node* n)
{
  Node* nodo=newNode();
  if(nodo!=NULL)
  {
    nodo->type=n_not;
    nodo->class="~";
    newArc(nodo,n);
  }
```

```
  return nodo;
}

Node* newLetteral(char* l)
{
  Node* nodo=newNode();
  if(nodo!=NULL)
  {
    nodo->type=n_letteral;
    nodo->class="L";
    nodo->data.letteral_name=l;
    nodo->neg=0;
  }
  return nodo;
}

Node* newAt(Node* n,Interval i)
{
  Node* nodo=newNode();
  if(nodo!=NULL)
  {
    nodo->type=n_at;
    nodo->class="@";
    nodo->data.interval=i;
    newArc(nodo,n);
  }
  return nodo;
}

Node* newHappen(Node* n,Interval i)
{
  Node* nodo=newNode();
  if(nodo!=NULL)
  {
    nodo->type=n_happen;
    nodo->class="?";
    nodo->data.interval=i;
    newArc(nodo,n);
  }
  return nodo;
}

Node* newUntil(Node* n1,Node* n2)
{
  Node* nodo=newNode();
  if(nodo!=NULL)
  {
    nodo->type=n_until;
    nodo->class="U";
    newArc(nodo,n1);
    newArc(nodo,n2);
  }
  return nodo;
}

Node* newSince(Node* n1,Node* n2)
{
  Node* nodo=newNode();
  if(nodo!=NULL)
  {
    nodo->type=n_since;
    nodo->class="S";
    newArc(nodo,n1);
    newArc(nodo,n2);
  }
  return nodo;
}
```

```
void deleteNode(Node* n)
{
  n->type=n_deleted;
}

void printTree(Node *root)
{
  assert(root);

  switch(root->type)
  {
    case n_and:
      fprintf(stderr,"(");
      printTree(root->n1);
      fprintf(stderr," & ");
      printTree(root->n2);
      fprintf(stderr,")");
      break;
    case n_or:
      fprintf(stderr,"(");
      printTree(root->n1);
      fprintf(stderr," | ");
      printTree(root->n2);
      fprintf(stderr,")");
      break;
    case n_not:
      fprintf(stderr,"~ ");
      printTree(root->n1);
      break;
    case n_delay:
      fprintf(stderr,"#%d ",root->data.delay_value);
      printTree(root->n1);
      break;
    case n_at:
      fprintf(stderr,"(");
      printTree(root->n1);
      fprintf(stderr," @ [%d,%d]",root->data.interval.min,
                                  root->data.interval.max);
      fprintf(stderr,")");
      break;
    case n_happen:
      fprintf(stderr,"(");
      printTree(root->n1);
      fprintf(stderr," ? [%d,%d]",root->data.interval.min,
                                  root->data.interval.max);
      fprintf(stderr,")");
      break;
    case n_until:
      fprintf(stderr,"until(");
      printTree(root->n1);
      fprintf(stderr," , ");
      printTree(root->n2);
      fprintf(stderr,")");
      break;
    case n_since:
      fprintf(stderr,"since(");
      printTree(root->n1);
      fprintf(stderr," , ");
      printTree(root->n2);
      fprintf(stderr,")");
      break;
    case n_notuntil:
      fprintf(stderr,"~until(~");
      printTree(root->n1);
      fprintf(stderr," , ~");
      printTree(root->n2);
```

```
      fprintf(stderr,")");
      break;
    case n_notsince:
      fprintf(stderr,"~since(~");
      printTree(root->n1);
      fprintf(stderr," , ~");
      printTree(root->n2);
      fprintf(stderr,")");
      break;
    case n_letteral:
      fprintf(stderr,"%s%s",root->neg?"!":"",root->data.letteral_name);
      break;
  }
}

Node* notElim(Node *root,int swap)
{
  Node* newroot=root;
  assert(root);

  switch(root->type)
  {
    case n_and:
      if(swap)
      {
        root->type=n_or;
      }
      notElim(root->n1,swap);
      notElim(root->n2,swap);
      break;
    case n_or:
      if(swap)
      {
        root->type=n_and;
      }
      notElim(root->n1,swap);
      notElim(root->n2,swap);
      break;
    case n_not:
      if(root->up==NULL)
      {
        newroot=root->n1;
      }
      else
      {
        if(root->up->n1==root)
          root->up->n1=root->n1;
        else
          root->up->n2=root->n1;
      }
      root->n1->up=root->up;

      notElim(root->n1,!swap);

      deleteNode(root);
      break;
    case n_delay:
      notElim(root->n1,swap);
      break;
    case n_letteral:
      root->neg=swap;
      break;
    case n_at:
      if(swap)
      {
        root->type=n_happen;
      }
```

```
      notElim(root->n1,swap);
      break;
    case n_happen:
      if(swap)
      {
        root->type=n_at;
      }
      notElim(root->n1,swap);
      break;
    case n_until:
      if(swap)
      {
        root->type=n_notuntil;
      }
      notElim(root->n1,swap);
      notElim(root->n2,swap);
      break;
    case n_notuntil:
      if(swap)
      {
        root->type=n_until;
      }
      notElim(root->n1,swap);
      notElim(root->n2,swap);
      break;
    case n_since:
      if(swap)
      {
        root->type=n_notsince;
      }
      notElim(root->n1,swap);
      notElim(root->n2,swap);
      break;
    case n_notsince:
      if(swap)
      {
        root->type=n_since;
      }
      notElim(root->n1,swap);
      notElim(root->n2,swap);
      break;
  }
  return newroot;
}

void upClose(Node* root)
{
  assert(root);

  switch(root->type)
  {
    case n_and:
      upClose(root->n1);
      upClose(root->n2);
      deleteNode(root);
      break;
    case n_or:
      root->n1->up=root->n2;
      root->n2->up=root->n1;
      deleteNode(root);
      break;
    case n_delay:
      upClose(root->n1);
      deleteNode(root);
      break;
    case n_letteral:
      fprintf(stderr,"warning: letteral %s is always %s\n",
```

```
              root->data.letteral_name,root->neg?"false":"true");
      deleteNode(root);
      break;
    default:
      printf("warning: up closing node type %d\n",root->type);
      break;
  }
}

int pos=0,next=0,posCmp,nextCmp;

Node *firstLetteral()
{

  for(;pos<nnodes; pos++)
    if(nodes[pos].type==n_letteral && !nodes[pos].mark)
    {
      next=pos;
      //printf("firstLetteral n%d\n",pos);
      return &nodes[pos];
    }
  return NULL;
}

Node *findCmpLetteral()
{
  char *l;
  int neg;

  l=nodes[pos].data.letteral_name;
  neg=nodes[pos].neg;

  for(posCmp=pos+1;posCmp<nnodes; posCmp++)
    if(nodes[posCmp].type==n_letteral &&
       !nodes[posCmp].mark &&
       nodes[posCmp].neg!=neg &&
       strcmp(nodes[posCmp].data.letteral_name,l)==0)
    {
      nextCmp=posCmp;
      //printf("findCmpLetteral n%d\n",posCmp);
      return &nodes[posCmp];
    }
  return NULL;
}

Node *nextLetteral()
{
  char *l;
  int neg;

  l=nodes[pos].data.letteral_name;
  neg=nodes[pos].neg;

  next++;

  for(;next<nnodes;next++)
    if(nodes[next].type==n_letteral &&
       !nodes[next].mark &&
       nodes[next].neg==neg &&
       strcmp(nodes[next].data.letteral_name,l)==0)
    {
      //printf("nextLetteral n%d\n",next);
      return &nodes[next];
    }
  return NULL;
}
```

```
Node *nextCmpLetteral()
{
  char *l;
  int neg;

  l=nodes[posCmp].data.letteral_name;
  neg=nodes[posCmp].neg;

  nextCmp++;

  for(;nextCmp<nnodes;nextCmp++)
    if(nodes[nextCmp].type==n_letteral &&
       !nodes[nextCmp].mark &&
       nodes[nextCmp].neg==neg &&
       strcmp(nodes[nextCmp].data.letteral_name,l)==0)
    {
      //printf("nextCmpLetteral n%d\n",nextCmp);
      return &nodes[nextCmp];
    }
  return NULL;
}

void downClose()
{
  Node *l1,*l2,*lc1,*lc2;
  inout *io;
  char class[100];
  pos=0;

  while(l1=firstLetteral())
  {
    Node *and1=l1->up,*and2=NULL,*newand;
    int neg=l1->neg;
    char *lname=nodes[pos].data.letteral_name;

    class[0]='A';
    class[1]=neg?'0':'1';
    class[2]='\0';

    assert(l1->type==n_letteral && !l1->mark);

    l1->mark=1;
    while(l2=nextLetteral())
    {
      newand=newAnd1(and1,l2->up,class);
      if(and1!=l1->up)
        and1->up=newand;

      if(and1->n1==l1)
        and1->n1=newand;
      else if(and1->n2==l1)
        and1->n2=newand;
      else if(and1->up==l1)
        and1->up=newand;

      if(l2->up->n1==l2)
        l2->up->n1=newand;
      else if(l2->up->n2==l2)
        l2->up->n2=newand;
      else if(l2->up->up==l2)
        l2->up->up=newand;

      l2->mark=1;
      and1=newand;
    }

    lc1=findCmpLetteral();
```

```
if(lc1==NULL)
{
  fprintf(stderr,"ERROR: letteral %s without complement\n",l1->data.letteral_name);
  exit(1);
}

class[0]='A';
class[1]=!neg?'0':'1';
class[2]='\0';

and2=lc1->up;
lc1->mark=1;
while(lc2=nextCmpLetteral())
{
  newand=newAnd1(and2,lc2->up,class);
  if(and2!=lc1->up)
    and2->up=newand;

  if(and2->n1==lc1)
    and2->n1=newand;
  else if(and2->n2==lc1)
    and2->n2=newand;
  else if(and2->up==lc1)
    and2->up=newand;

  if(lc2->up->n1==lc2)
    lc2->up->n1=newand;
  else if(lc2->up->n2==lc2)
    lc2->up->n2=newand;
  else if(lc2->up->up==lc2)
    lc2->up->up=newand;

  lc2->mark=1;
  and2=newand;
}

if(and2!=lc1->up)
  and2->up=and1;
else
{
  if(lc1->up->n1==lc1)
    lc1->up->n1=and1;
  else if(lc1->up->n2==lc1)
    lc1->up->n2=and1;
  else if(lc1->up->up==lc1)
    lc1->up->up=and1;
}
if(and1!=l1->up)
  and1->up=and2;
else
{
  if(l1->up->n1==l1)
    l1->up->n1=and2;
  else if(l1->up->n2==l1)
    l1->up->n2=and2;
  else if(l1->up->up==l1)
    l1->up->up=and2;
}

if(!neg)
{
  setupInOut(lname,and1,and2);
  setupIC(lname,and1,and2);
}
else
{
  setupInOut(lname,and2,and1);
```

```
      setupIC(lname,and2,and1);
    }
  }
  pos=0;
  assert(firstLetteral()==NULL);
}

void outputGraph(FILE* fp)
{
  int i;

  for(i=0; i<nnodes; i++)
  {
    switch(nodes[i].type)
    {
      case n_and:
        fprintf(fp,"%s%d: J ; %s%d, %s%d, %s%d\n",nodes[i].class,i,
               nodes[i].up->class,nodes[i].up-nodes,
               nodes[i].n1->class,nodes[i].n1-nodes,
               nodes[i].n2->class,nodes[i].n2-nodes);
        assert(nodes[i].n1->type!=n_letteral);
        assert(nodes[i].n2->type!=n_letteral);
        assert(nodes[i].n2!=nodes[i].n1);
        break;
      case n_or:
        fprintf(fp,"%s%d: G ; %s%d, %s%d, %s%d\n",nodes[i].class,i,
               nodes[i].up->class,nodes[i].up-nodes,
               nodes[i].n1->class,nodes[i].n1-nodes,
               nodes[i].n2->class,nodes[i].n2-nodes);
        assert(nodes[i].n1->type!=n_letteral);
        assert(nodes[i].n2->type!=n_letteral);
        assert(nodes[i].n2!=nodes[i].n1);
        break;
      case n_delay:
        fprintf(fp,"%s%d: D%d ; %s%d, %s%d\n",nodes[i].class,i,nodes[i].data.delay_value,
               nodes[i].up->class,nodes[i].up-nodes,
               nodes[i].n1->class,nodes[i].n1-nodes);
        assert(nodes[i].n1->type!=n_letteral);
        break;
/*      case n_letteral:
        printf("n%d: L %s%s; n%d\n",i,nodes[i].neg?"!":"",
               nodes[i].data.letteral_name,nodes[i].up-nodes);*/
    }
  }
  outputInOut(fp);
}

inout* addInOut(ioType type,char* name,int at)
{
  if(ninouts<MAX_INOUTS)
  {
    inout *n=&inouts[ninouts++];
    n->n1=NULL;
    n->n2=NULL;
    n->name=name;
    n->type=type;
    n->at=at;
    return n;
  }
  return NULL;
}

void setupInOut(char *name,Node* n1,Node* n2)
{
  int i;
  for(i=0; i<ninouts; i++)
  {
```

```
    if(strcmp(inouts[i].name,name)==0)
    {
      inout* n=&inouts[i];

      n->n1=n1;
      n->n2=n2;
      return;
    }
  }
  return;
}

void outputInOut(FILE* fp)
{
  int i;
  fprintf(fp,"\n\n");
  for(i=0; i<ninouts; i++)
  {
    inout* io=&inouts[i];

    if(io->n1 && io->n2)
    {
       fprintf(fp,"%c %s (%s%d,%s%d) @ %d\n",
              io->type==io_input ? '!':'?',io->name,
              io->n1->class,io->n1-nodes,
              io->n2->class,io->n2-nodes,
              io->at-minIC);
       assert(io->n1->type!=n_letteral);
       assert(io->n2->type!=n_letteral);
    }
    else
      fprintf(stderr,"warning: %s %s not defined\n",
              io->type==io_input ? "input":"output",io->name);
  }
}

initc* addIC(int tf,char* name,int atMin,int atMax)
{
  if(ninits<MAX_INITS)
  {
    initc *n=&inits[ninits++];
    n->n1=NULL;
    n->n2=NULL;
    n->name=name;
    n->tf=tf;
    n->atMin=atMin;
    n->atMax=atMax;
    if(atMin<minIC)
      minIC=atMin;
    //printf("IC: %s%s@%d\n",n->tf ? "":"~",n->name,n->at);
    return n;
  }
  return NULL;
}

void setupIC(char* name, Node* n1, Node* n2)
{
  int i;

  for(i=0;i<ninits;i++)
  {
    initc* n=&inits[i];

    if(strcmp(name,n->name)==0)
    {
      if(n->tf)
      {
```

```
        n->n1=n1;
        n->n2=n2;
      }
      else
      {
        n->n1=n2;
        n->n2=n1;
      }
    }
  }
}

void outputIC(FILE* fp)
{
  int i,at;

  for(i=0;i<ninits;i++)
  {
    initc* n=&inits[i];

    if(n->n1 && n->n2)
    {
      for(at=n->atMin; at<=n->atMax; at++)
        fprintf(fp,"(%s%d,%s%d)@%d\n",
                n->n1->class,n->n1-nodes,
                n->n2->class,n->n2-nodes,
                at-minIC);
      assert(n->n1->type!=n_letteral);
      assert(n->n2->type!=n_letteral);
    }
    else
      fprintf(stderr,"warning: letteral %s not valid in initial conditions\n",n->name);
  }
}

yyerror(char* s)
{
  fprintf(stderr,"ERROR line %d: %s\n",lineno,s);
  return 0;
}

int yywrap()
{
  return 1;
}


Node* AtSubst(Node* atNode, Node* root)
{
  int min,max,i;
  Node *n,*newroot,*nletteral;
  char atID[20];
  static atNum=1;

  assert(atNode->type==n_at);

  sprintf(atID,"at_%d",atNum++);

  min=atNode->data.interval.min;
  max=atNode->data.interval.max;

  n=newDelay(newLetteral(strdup(atID)),-min);
  for(i=min+1;i<=max; i++)
  {
    n=newAnd(n,newDelay(newLetteral(strdup(atID)),-i));
  }
```

```
  n->up=atNode->up;
  if(atNode->up!=NULL)
    if(n->up->n1==atNode)
      n->up->n1=n;
    else
      n->up->n2=n;


  nletteral=newLetteral(strdup(atID));
  nletteral->neg=1;

  newroot=newAnd(root,newOr(nletteral,atNode->n1));

  deleteNode(atNode);
  return newroot;
}

Node* HappenSubst(Node* happenNode, Node* root)
{
  int min,max,i;
  Node *n,*newroot,*nletteral;
  char happenID[20];
  static happenNum=1;

  assert(happenNode->type==n_happen);

  sprintf(happenID,"happen_%d",happenNum++);

  min=happenNode->data.interval.min;
  max=happenNode->data.interval.max;

  n=newDelay(newLetteral(strdup(happenID)),-min);
  for(i=min+1;i<=max; i++)
  {
    n=newOr(n,newDelay(newLetteral(strdup(happenID)),-i));
  }

  n->up=happenNode->up;
  if(happenNode->up!=NULL)
    if(n->up->n1==happenNode)
      n->up->n1=n;
    else
      n->up->n2=n;


  nletteral=newLetteral(strdup(happenID));
  nletteral->neg=1;

  newroot=newAnd(root,newOr(nletteral,happenNode->n1));
  deleteNode(happenNode);
  return newroot;
}

Node* UntilSubst(Node* untilNode, Node* root)
{
  int min,max,i;
  Node *n,*newroot,*nletteral;
  char untilID[20];
  static untilNum=1;

  assert(untilNode->type==n_until);

  sprintf(untilID,"until_%d",untilNum++);

  n=newLetteral(strdup(untilID));

  n->up=untilNode->up;
```

```
  if(untilNode->up!=NULL)
    if(n->up->n1==untilNode)
      n->up->n1=n;
    else
      n->up->n2=n;


  nletteral=newLetteral(strdup(untilID));
  nletteral->neg=1;

  newroot=newAnd(root,
                 newOr(nletteral,
                       newOr(untilNode->n1,
                             newAnd(untilNode->n2,
                                    newDelay(newLetteral(strdup(untilID)),
                                             -1)))));
  deleteNode(untilNode);
  return newroot;
}

Node* NotUntilSubst(Node* nuntilNode, Node* root)
{
  int min,max,i;
  Node *n,*newroot,*nletteral;
  char nuntilID[20];
  static nuntilNum=1;

  assert(nuntilNode->type==n_notuntil);

  sprintf(nuntilID,"nuntil_%d",nuntilNum++);

  n=newLetteral(strdup(nuntilID));

  n->up=nuntilNode->up;
  if(nuntilNode->up!=NULL)
    if(n->up->n1==nuntilNode)
      n->up->n1=n;
    else
      n->up->n2=n;


  nletteral=newLetteral(strdup(nuntilID));
  nletteral->neg=1;

  newroot=newAnd(root,
                 newOr(nletteral,
                       newAnd(nuntilNode->n1,
                             newOr(nuntilNode->n2,
                                   newDelay(newLetteral(strdup(nuntilID)),
                                             -1)))));
  deleteNode(nuntilNode);
  return newroot;
}

Node* SinceSubst(Node* sinceNode, Node* root)
{
  int min,max,i;
  Node *n,*newroot,*nletteral;
  char sinceID[20];
  static sinceNum=1;

  assert(sinceNode->type==n_since);

  sprintf(sinceID,"since_%d",sinceNum++);

  n=newLetteral(strdup(sinceID));
```

```
  n->up=sinceNode->up;
  if(sinceNode->up!=NULL)
    if(n->up->n1==sinceNode)
      n->up->n1=n;
    else
      n->up->n2=n;


  nletteral=newLetteral(strdup(sinceID));
  nletteral->neg=1;

  newroot=newAnd(root,
                 newOr(nletteral,
                       newOr(sinceNode->n1,
                             newAnd(sinceNode->n2,
                                    newDelay(newLetteral(strdup(sinceID)),
                                             1)))));
  deleteNode(sinceNode);
  return newroot;
}

Node* NotSinceSubst(Node* nsinceNode, Node* root)
{
  int min,max,i;
  Node *n,*newroot,*nletteral;
  char nsinceID[20];
  static nsinceNum=1;

  assert(nsinceNode->type==n_notsince);

  sprintf(nsinceID,"nsince_%d",nsinceNum++);

  n=newLetteral(strdup(nsinceID));

  n->up=nsinceNode->up;
  if(nsinceNode->up!=NULL)
    if(n->up->n1==nsinceNode)
      n->up->n1=n;
    else
      n->up->n2=n;


  nletteral=newLetteral(strdup(nsinceID));
  nletteral->neg=1;

  newroot=newAnd(root,
                 newOr(nletteral,
                       newAnd(nsinceNode->n1,
                             newOr(nsinceNode->n2,
                                   newDelay(newLetteral(strdup(nsinceID)),
                                            1)))));
  deleteNode(nsinceNode);
  return newroot;
}

Node* Subst(Node *root)
{
  int i;

  for(i=0;i<nnodes; i++)
    switch(nodes[i].type)
    {
      case n_at:
        root=AtSubst(&nodes[i],root);
        break;
      case n_happen:
        root=HappenSubst(&nodes[i],root);
```

```
        break;
      case n_until:
        root=UntilSubst(&nodes[i],root);
        break;
      case n_notuntil:
        root=NotUntilSubst(&nodes[i],root);
        break;
      case n_since:
        root=SinceSubst(&nodes[i],root);
        break;
      case n_notsince:
        root=NotSinceSubst(&nodes[i],root);
        break;
    }
  return root;
}
```

A.6 btl.l

```
/*
 */

%{

#include <string.h>
#include "graph.h"
#include "y.tab.h"

//#define LDEBUG

/* Questo serve per il debugging */
#ifdef LDEBUG
        #define RETURNIT(a,b) {printf("%s :\"%s\"\n",(b), yytext); return((a));}
#else
        #define RETURNIT(a,b) {return((a));}
#endif

extern int lineno;

%}

DIGIT   [0-9]
ID      [a-zA-Z_][a-zA-Z0-9_\.]*
NL      \n

%%


"&"     RETURNIT(AND, "and");

"|"     RETURNIT(OR,"or");

"~"     RETURNIT(NOT,"not");

"-->"   RETURNIT(IMPLY,"imply");

"=="    RETURNIT(IFF,"iff");

"<-->"  RETURNIT(IFF,"iff");

"#"     RETURNIT(DELAY,"delay");

"("     RETURNIT(OPENB,"openb");

")"     RETURNIT(CLOSEB,"closeb");

"inputs:" RETURNIT(INPUTS,"inputs");

"outputs:" RETURNIT(OUTPUTS,"outputs");

"init:" RETURNIT(INIT,"init");

","     RETURNIT(COMMA,"comma");

";"     RETURNIT(SEMICOLON,"semicolon");

"@"     RETURNIT(AT,"at");

"?"     RETURNIT(HAPPEN,"happen");

"["     RETURNIT(OPENSQB,"[");

"]"     RETURNIT(CLOSESQB,"]");
```

```
"until" RETURNIT(UNTIL,"until");

"since" RETURNIT(SINCE,"since");

{ID}    {
          yylval.string=strdup(yytext);
          RETURNIT(LETTERAL, "letterale");
        }

{NL}    { lineno++; }

[-]?{DIGIT}+  {
          sscanf(yytext,"%d", &yylval.integer);
          RETURNIT(INTEGER, "integer");
        }

[ \t]+ ;

"//"[^\n]* { /* a Line comment */ }

.       { return yytext[0]; }

%%
```

A.7 btl.y

```
/* $Id$ */

%{
#include <stdio.h>
#include <string.h>

#include "graph.h"

extern int lineno;
extern FILE *yyin;
extern int yy_init;
extern yyerror(char *s); // aggiunta
extern int yywrap();

extern FILE* ftin;
extern FILE* fic;

// Prototipi delle funzioni
int yylex(void);
int yyparse(void);

ioType iotype;

%}

%union {
     int integer;
     Node* btl_node;
     char* string;
     Interval interval;
}

%token INPUTS
%token OUTPUTS
%token COMMA
%token INIT
%token AT

%token AND
%token OR
%token NOT
%token IMPLY
%token IFF
%token DELAY
%token OPENB
%token CLOSEB
%token <string>  LETTERAL
%token <integer> INTEGER
%token SEMICOLON
%token OPENSQB
%token CLOSESQB
%token HAPPEN
%token UNTIL
%token SINCE

%type <btl_node> spec,btl_expr,btl_iff,btl_imply,btl_and,btl_or,btl_athappen,btl_term
%type <interval> interval

%%

file:
    prologue spec
    {
      Node *n=$2;
```

```
      fprintf(stderr,"\nBTL_expr: ");
      printTree(n);
      fprintf(stderr,"\nBTL_notElim:");
      n=notElim(n,0);
      printTree(n);
      fprintf(stderr,"\n");

      fprintf(stderr,"\nBTL_Subst:");
      n=Subst(n);
      printTree(n);
      fprintf(stderr,"\n");

      upClose(n);
      downClose();
      outputGraph(ftin);
      outputIC(fic);
    }

prologue:
     inputs outputs init

inputs:
    /* empty */
  | INPUTS { iotype=io_input; } io_list

outputs:
    /* empty */
  | OUTPUTS { iotype=io_output; } io_list

io_el:
    LETTERAL
    {
      addInOut(iotype,$1,0);
    }
  | LETTERAL AT INTEGER
    {
      addInOut(iotype,$1,$3);
    }

io_list:
    io_el
  | io_el COMMA io_list

init:
    /* empty */
  | INIT init_list

init_list:
    init_el
  | init_el COMMA init_list

init_el:
    LETTERAL AT INTEGER
    {
      addIC(TRUE,$1,$3,$3);
    }
  | NOT LETTERAL AT INTEGER
    {
      addIC(FALSE,$2,$4,$4);
    }
  | LETTERAL AT interval
    {
      addIC(TRUE,$1,$3.min,$3.max);
    }
  | NOT LETTERAL AT interval
    {
      addIC(FALSE,$2,$4.min,$4.max);
```

```
    }

spec:
    btl_expr SEMICOLON
    {
      $$=$1;
    }
  | btl_expr SEMICOLON spec
    {
      //printf(" and");
      $$=newAnd($1,$3);
    }

btl_expr:
    btl_iff

btl_iff:
    btl_imply
  | btl_imply IFF btl_iff
    {
      //printf(" iff");
      $$=newAnd(newOr(newNot($1),$3),
                newOr(newNot(newDeepCopy($3,NULL)),
                      newDeepCopy($1,NULL)));
    }

btl_imply:
    btl_and
  | btl_and IMPLY btl_imply
    {
      //printf(" imply");
      $$=newOr(newNot($1),$3);
    }

btl_and:
    btl_or
  | btl_or AND btl_and
    {
      //printf(" and");
      $$=newAnd($1,$3);
    }

btl_or:
    btl_athappen
  | btl_athappen OR btl_or
    {
      //printf(" or");
      $$=newOr($1,$3);
    }

btl_athappen:
    btl_term
  | btl_term AT interval
    {
      //printf(" at");
      $$=newAt($1,$3);
    }
  | btl_term HAPPEN interval
    {
      //printf(" happen");
      $$=newHappen($1,$3);
    }

btl_term:
    LETTERAL
    {
      //printf("<%s>",yytext);
```

```
      $$=newLetteral($1);
    }
  | NOT btl_term
    {
      //printf(" not");
      $$=newNot($2);
    }
  | DELAY btl_term
    {
      //printf(" delay");
      $$=newDelay($2,1);
    }
  | DELAY INTEGER btl_term
    {
      //printf(" delay%d",$2);
      $$=newDelay($3,$2);
    }
  | OPENB btl_expr CLOSEB
    {
      $$=$2;
    }
  | UNTIL OPENB btl_expr COMMA btl_expr CLOSEB
    {
      //printf(" until");
      $$=newUntil($3,$5);
    }
  | SINCE OPENB btl_expr COMMA btl_expr CLOSEB
    {
      //printf(" since");
      $$=newSince($3,$5);
    }

interval:
  OPENSQB INTEGER COMMA INTEGER CLOSESQB
  {
    Interval i;

    i.min = $2;
    i.max = $4;

    $$=i;
  }
```

Appendice B

Sorgenti BTL

B.1 Mutua esclusione

Arbitro

```
inputs: rq1, ex1, rq2, ex2
outputs: gnt1, gnt2
init: wait_req1@0, wait_req2@0, ~sosp2@0

up_ex1 == (ex1 & # ~ex1);
wait_req1 --> ( req1 & (~gnt1 & ~gnt2) & #-1 (gnt1 & ~gnt2)) |
              ( req1 & (~gnt1 &  gnt2) & #-1 wait_free1) |
              (~req1 &  ~gnt1 & #-1 wait_req1);
gnt1 --> ( up_ex1 & #-1 (wait_req1 & ~gnt1)) |
         (~up_ex1 & #-1 (gnt1 & ~gnt2));
wait_free1 --> (~gnt2 & gnt1) |
               ( gnt2 & #-1 wait_free1);

up_ex2 == (ex2 & # ~ex2);
wait_req2 --> ( req2 & (~gnt1 & ~gnt2) & #-1 (gnt2 & ~gnt1)) |
              ( req2 & ( gnt1 & ~gnt2) & #-1 wait_free2) |
              (~req2 & ~gnt2 & #-1 wait_req2);
gnt2 --> ( up_ex2 & #-1 (wait_req2 & ~gnt2)) |
         (~up_ex2 & #-1 (gnt2 & ~gnt1));
wait_free2 --> (~gnt1 & gnt2) |
               ( gnt1 & #-1 wait_free2);

 rq1 & ~(rq2 | sosp2) -->  req1 & ~req2 & #-1 ~sosp2;
~rq1 &  (rq2 | sosp2) --> ~req1 &  req2 & #-1 ~sosp2;
~rq1 & ~(rq2 | sosp2) --> ~req1 & ~req2 & #-1 ~sosp2;
 rq1 &  (rq2 | sosp2) -->  req1 & ~req2 & #-1  sosp2;
```

Primo competitore

```
inputs: gnt1
outputs: rq1,ex1
init: loop1@0

loop1 --> ~rq1 & ~ex1 & #-1 (~rq1 & ~ex1 &#-1(rq1 & ~ex1 & #-1wait_gnt1));
wait_gnt1 --> (gnt1 & ~ex1 & ~rq1 & #-1(~ex1 &~rq1) & #-2(~ex1&~rq1) &
               #-3(ex1 & ~rq1 & #-1 loop1))|
              (~gnt1 & ~rq1 &~ex1 & #-1 wait_gnt1);
```

Secondo competitore

```
inputs: gnt2
outputs: rq2,ex2
init: loop2@0
```

```
loop2 --> ~rq2 & ~ex2 & #-1(rq2 & ~ex2 &#-1 wait_gnt2);
wait_gnt2 --> (gnt2 & ~ex2 & ~rq2 & #-1(~ex2&~rq2) & #-2(ex2 & ~rq2 &#-1 loop2))|
              (~gnt2 & ~rq2 &~ex2 & #-1 wait_gnt2);
```

B.2 Produttore-consumatore

Produttore

```
inputs: sended
outputs: send
init: loop@0

loop --> ~send & #-1 (~send &#-1(send & #-1wait));
wait --> (sended & loop)|
         (~sended & ~send & #-1 wait);
```

Consumatore

```
inputs: received
outputs: receive
init: loop@0

loop --> receive & #-1(wait);
wait --> (received & ~receive &
                    #-1(~receive) &
                    #-2(~receive) &
                    #-3(~receive & #-1 loop))|
         (~received & ~receive & #-1 wait);
```

Trasmettitore

```
inputs: ack,send
outputs: msg,sended
init: ~wait_ack@0

~send & ~wait_ack --> ~msg & ~sended & #-1 ~wait_ack;
 send & ~wait_ack -->  msg & ~sended & #-1 wait_ack;

wait_ack --> ( ack &  sended & ~msg & #-1 ~wait_ack) |
             (~ack & ~sended & ~msg & #-1  wait_ack);
```

Ricevitore

```
inputs: msg,receive
outputs: ack,received,wait_msg,pmsg
init: ~wait_msg@0,~pmsg@0,~ack@0,~received@0

~receive & ~wait_msg --> #-1(~ack & ~received & ~wait_msg);
 receive & ~wait_msg & ~(msg|pmsg) --> #-1(~ack & ~received & wait_msg);
 receive & ~wait_msg &  (msg|pmsg) --> #-1(ack & received) &
                                       #-1(~wait_msg & ~pmsg);
~receive & ~wait_msg &  msg --> #-1 pmsg & #-1 ~wait_msg;

 pmsg & ~receive --> #-1pmsg;
~pmsg & ~receive & ~msg--> #-1~pmsg;
 pmsg & receive --> #-1~pmsg;

wait_msg --> ( msg &  ~received &  ~ack &
               #-1(received & ack & ~wait_msg & ~pmsg)) |
             (~msg & ~pmsg & ~received & ~ack & #-1wait_msg);
```

B.3 Altri sorgenti

Controllo di tempo massimo

```
inputs: e
outputs: ok,timeout
init: wait@4

start-->wait;
wait & e --> ok & #-1 start;
wait & ~e & (#1e | #2e | #3e)  --> ~ok & ~timeout & #-1 wait;
wait & ~e & #1~e & #2~e & #3~e --> ~ok & timeout & #-1 wait;
```

Segnale periodico

```
inputs: start
outputs: p
init: ~started@0,~start@-1

begin <--> start & #~start;

(begin | started) --> #-1 started;
~begin & ~started --> #-1 (~started & ~p);

begin & ~started --> ~p & #-1~p & #-2~p & #-3 ~p & #-4 p;
p --> #-1~p & #-2~p & #-3~p & #-4 p;
```

Segnale periodico condizionato

```
inputs: start
outputs: p
init: free@0

begin <--> (start & #~start);
end   <--> (~start & #start);

free & ~start --> #-1 free;
~free & start --> #-1~free;

begin & free --> until_end & ~p & #-1~p & #-2~p & #-3 ~p & #-4 p;

until_end --> (end & until_p) |
              (~end & (p --> #-1~p & #-2~p & #-3~p & #-4 p)
                & ~until_p & #-1 until_end);
  // fino a che start diventa basso p e' periodico

until_p  -->  ((p & ~free
                & (~start --> #-1 free & until_beg)
                & (start --> #-1 recover)
                ) |
               ((~p & ~free) & #-1 until_p));
  // dopo il prossimo p vero, p e' falso fino al prossimo begin (until_beg)

recover --> ((end & #-1 free & #until_beg) | (~end & ~free & ~p & #-1 recover));
  // fino a end not free e not p e quando end cerca begin.

until_beg  --> #-1 (begin | ((~p) & until_beg));
  // fino a begin p e' falso
```

Bibliografia

[1] D. Gabbay, "The declarative past and imperative future," in *Proc. of the International Conference on Temporal Logic in Specification* (B. Banieqbal, H. Barringer, and P. Pnueli, eds.), (Altrincham, UK), pp. 409–448, Springer Verlag, LNCS n.398, 8-10 April 1987.

[2] M. Fisher and R. Owens, "From the past to the future: Executing temporal logic programs," in *Proc. of the Conference on Logic Programming and Automated Reasoning (LPAR)*, (St. Petersberg, Russia), Springer Verlarg, LNCS N.624, July 1992.

[3] S. Merz, "Efficiently executable temporal logic programs," in *Executable Modal and Temporal Logics: Proceedings of the IJCAI '93 Workshop, Chamberry, France, August 1993* (M. Fisher and R. Owens, eds.), pp. 1–20, Lecture Notes in Artificial Intelligence, Springer Verlag LNCS 897, 1995.

[4] B. C. Moszkowski, *Executing Temporal Logic Programs.* PhD thesis, Cambridge University, 1986.

[5] M. Fisher and R. Owens, "An introduction to executable modal and temporal logics," in *Executable Modal and Temporal Logics: Proceedings of the IJCAI '93 Workshop, Chamberry, France, August 1993* (M. Fisher and R. Owens, eds.), pp. 1–20, Lecture Notes in Artificial Intelligence, Springer Verlag LNCS 897, 1995.

[6] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens, "Metatem: A framework for programming in temporal logic," in *in Proc. of REX Workshop on Stepwise Refinement of Distributed Systems: Models Formalism, Correctness*, (Mook, Netherlands), Springer Verlag, LNCS n.430, June 1989.

[7] R. Mattolini, *TILCO: a Temporal Logic for the Specification of Real-Time Systems (TILCO: una Logica Temporale per la Specifica di Sistemi di Tempo Reale).* PhD thesis, 1996.

[8] R. Mattolini and P. Nesi, "An interval logic for real-time system specification," *IEEE Transactions on Software Engineering, in press*, 1999.

[9] P. Bellini, R. Mattolini and P. Nesi, "Temporal logics for real-time system specification," *ACM Computing Surveys, in press*, 1999.

[10] J. S. Ostroff, *Temporal Logic for Real-Time Systems.* Taunton, Somerset, England: Research Studies Press LTD., Advanced Software Development Series, 1, 1989. NO.

[11] R. Alur and T. A. Henzinger, "Real time logics: complexity and expressiveness," in *Proc. of 5th Annual IEEE Symposium on Logic in Computer Science, LICS 90*, (Philadelphia, USA), pp. 390–401, IEEE, June 1990. YES in Kavi92, TR STANCS901307, Dept. of Comp. Science and Med, Stanford.

[12] P. M. MelliarSmith, "Extending interval logic to real time systems," in *Proc. of Temporal Logic Specification United Kingdom, (B. Banieqbal, H. Barringer, A. Pnueli, eds)*, pp. 224–242, Springer Verlag, Lecture Notes in Computer Sciences, LNCS 398, April 1987.

[13] R. R. Razouk and M. M. Gorlick, "A real-time interval logic for reasoning about executions of real-time programs," in *Proc. of the ACM/SIGSOFT'89, Tav.3*, pp. 10–19, ACM Press, Dec. 1989.

[14] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna, "A graphical interval logic for specifying concurrent systems," *ACM Transactions on Software Engineering and Methodology*, vol. 3, pp. 131–165, April 1994.

[15] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Systems Journal*, vol. 2, pp. 255–299, 1990.

[16] J. F. Allen and G. Ferguson, "Actions and events in interval temporal logic," tech. rep., University of Rochester Computer Science Department, TR-URCSD 521, Rochester, New York 14627, July 1994.

[17] R. Gotzhein, "Temporal logic and applications – a tutorial," *Computer Networks and ISDN Systems, North-Holland*, vol. 24, pp. 203–218, 1992.

[18] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt, "A interval logic for higher-level temporal reasoning," in *Proc. of the 2nd Annual ACM Symp. Principles of Distributed Computing, Lecture Notes in Computer Science, LNCS N. 164*, (Montreal Canada), pp. 173–186, Springer Verlag, ACM NewYork, 17-19 Aug. 1983.

[19] R. Alur and T. A. Henzinger, "A really temporal logic," in *30th IEEE FOCS*, 1989.

[20] A. Pnueli, "The temporal logic of programs," in *18th IEEE FOCS*, 1977.

[21] J. Y. Halpern and Y. Shoham, "A propositional modal logic of time intervals," in *Proc. of the 1st IEEE Symp. on Logic in Computer Science*, pp. 274–292, IEEE Press, 1986.

[22] M. Ben-Ari, *Mathematical Logic for Computer Science*. New York: Prentice Hall, 1993.

[23] G. Bucci, M. Campanai, and P. Nesi, "Tools for specifying real-time systems," *Journal of Real-Time Systems*, vol. 8, pp. 117–172, March 1995.

[24] J. S. Ostroff, "Formal methods for the specification and design of real-time safety critical systems," *Journal of Systems and Software*, pp. 33–60, April 1992.

[25] A. D. Stoyenko, "The evolution and state-of-the-art of real-time languages," *Journal of Systems and Software*, pp. 61–84, April 1992.