Università degli Studi di Brescia

Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

Specifica e diagnosi di sistemi attivi complessi

Relatore:
Chiar.mo Prof. Gianfranco Lamperti

Laureando:
Giulio Quarenghi
Matricola 91667

Anno Accademico 2014-2015

Indice

Ringraziamenti

Vorrei ringraziare tutte le persone che, direttamente o indirettamente, hanno permesso il raggiungimento di questo traguardo, anche se costituisce per me solo un nuovo punto di partenza, una tappa lungo il mio viaggio.
Vorrei ringraziare il prof. Lamperti, per la chiarezza, la disponibilità, i preziosi consigli. A lui va la mia riconoscenza e la mia profonda stima.
I miei genitori, che mi hanno permesso di ottenere questo risultato, con grandi sacrifici e sostegno incondizionato.
I miei fratelli, per essere sempre stati per me un esempio da seguire.
Martina, per avermi supportato e sopportato in questi ultimi quattro anni e più, per avermi incoraggiato nell'ultimo periodo e per avermi fatto capire ciò che veramente è importante.
I miei amici, in particolare Andrea e Boris, e tutti gli altri, per essermi sempre stati vicino.
I miei amici e compagni di università, per aver fatto in modo che questi lunghi anni, che solo ora sembrano essere passati così in fretta, non saranno mai dimenticati.

Giulio

Capitolo 1

Introduzione

Il lavoro di questa tesi si focalizza sulla diagnosi, la cui esecuzione automatica costituisce un ramo di forte interesse nell'ambito dell'intelligenza artificiale. Data un'osservazione riguardante il comportamento corrente di un sistema fisico, il ragionamento diagnostico permette di trovare eventuali guasti in tale comportamento. Questo, tipicamente, avviene mettendo in evidenza le discrepanze tra il comportamento normale e quello osservato, isolando le cause di tali differenze, chiamati guasti. Nello specifico, il lavoro svolto ha inizio dalle conoscenze del settore riguardante la diagnosi di sistemi attivi. Questi ultimi sono un particolare tipo di sistemi a eventi discreti, nel quale l'evoluzione dello stato è asincrona. Un sistema attivo si suppone formato da componenti che comunicano tra loro attraverso la consumazione e la generazione di eventi. La tesi svolta estende la specifica di un sistema attivo, come suggerito dalla fiorente ricerca degli ultimi anni, raggruppando i componenti interconnessi in sottosistemi, i quali conducono una vita propria eccetto ciò che riguarda il verificarsi di eventi particolari, scaturiti da determinate dinamiche interne. Un sistema attivo complesso è una rete di sistemi attivi, connessi tra loro in maniera gerarchica, formando un albero. Questa topologia di modello è suggerita in modo naturale da molti sistemi reali fisici, biologici, economici e sociali, nei quali l'evoluzione del sistema può essere vista a diversi livelli di astrazione. Si suppone che un sistema attivo possa influenzare il comportamento di un sistema attivo appartenente al livello immediatamente superiore nell'albero. Sebbene l'applicazione originaria di sistemi siffatti nasca principalmente nell'ambito della diagnosi di reti elettriche, essi si prestano ad essere utilizzati in un qualsiasi contesto dal quale sia possibile estrapolare una stratificazione di comportamenti. Per citare un esempio, un sistema attivo complesso potrebbe in teoria costituire un modello per un sistema biologico: nel livello più basso della gerarchia vi sono le singole interazioni fra le cellule, ad un livello più alto i tessuti, aumentando il livello di astrazione vi è il funzionamento degli organi e degli apparati. A fronte della definizione di questa nuova classe di sistemi, il lavoro svolto include una implementazione software riguardante la dichiarazione di problemi concreti di questo tipo per mezzo di un particolare

linguaggio di specifica. Data una specifica, un compilatore, dopo aver effettuato i controlli necessari, genera le strutture dati atte a rappresentare l'istanza. L'elaborazione prosegue attraverso due metodi diagnostici risolutivi: un metodo greedy, che estende l'algoritmo noto nell'ambito dei sistemi attivi tradizionali a quello dei sistemi attivi complessi, calcolando l'evoluzione del sistema nella sua interezza, e un nuovo metodo lazy che, traendo vantaggio dalla topologia del sistema, calcola le diagnosi globali combinando le diagnosi di ogni singolo sistema attivo che compone la gerarchia con quelle delle interfacce dei nodi inferiori a esso collegati. Lo scopo di questa tesi è quello di mostrare, attraverso risultati sperimentali, la maggiore efficienza di questo nuovo metodo risolutivo. La tesi è articolata nei capitoli seguenti:

- nel *capitolo 2* è introdotto il concetto di automa a stati finiti, importante in questo lavoro sia nei modelli proposti che negli algoritmi risolutivi;
- nel *capitolo 3* è presentato lo stato dell'arte riguardante i sistemi attivi;
- nel *capitolo 4* sono fornite le definizioni e i metodi inerenti i sistemi attivi complessi;
- nel *capitolo 5* è descritta l'implementazione riferita alla specifica e al calcolo diagnostico, nelle modalità greedy e lazy;
- nel *capitolo 6* sono forniti i risultati sperimentali che mostrano un confronto tra i due metodi e approfondiscono l'approccio lazy;
- nel *capitolo 7* sono inferite le conclusioni di questo lavoro e alcune considerazioni riguardanti i possibili sviluppi futuri;
- l'*appendice A* contiene la grammatica, in notazione BNF, del linguaggio di specifica;
- l'*appendice B* presenta un esempio di specifica di un sistema;
- l'*appendice C* fornisce le istruzione di installazione del software sviluppato.

Capitolo 2

Automi a stati finiti

Nell'ambito della diagnosi di sistemi a eventi discreti in generale e dei sistemi attivi (complessi) in particolare, il comportamento del sistema è descritto da successioni di eventi. A questo livello di astrazione, le sequenze di transizioni che si verificano possono essere rappresentate da un automa a stati finiti. Anche il comportamento dei singoli componenti del sistema, come si vedrà in seguito, può essere visto attraverso un automa. La semplicità e l'intuitività degli automi rende agevole il loro utilizzo in problemi di questo tipo, nonché nella più ampia branca dell'intelligenza artificiale. Gli automi a stati finiti sono utili per una grande varietà di scopi, ad esempio per scansionare dei testi alla ricerca di parole o pattern, oppure nei compilatori per compiere la fase di analisi lessicale. Gli automi sono anche detti riconoscitori, poiché ricevendo una stringa in ingresso, possono confermare o meno l'appartenenza di tale stringa al linguaggio definito dall'automa. Esistono due principali tipi di automi a stati finiti:

- automi a stati finiti deterministici (DFA);
- automi a stati finiti non deterministici (NFA).

In questo capitolo vengono presentati definizioni ed esempi riguardanti entrambe le classi di automi. In particolare, è descritto come convertire espressioni regolari in NFA e come convertire NFA in DFA. Da ultimo viene fornito il concetto di automa minimo, cioè un automa caratterizzato dal minor numero di stati possibile, utile per una implementazione efficiente in termini di risorse computazionali.

2.1 DFA

Un automa a stati finiti deterministico (DFA: deterministic finite automaton) è una quintupla:

$$D = (\Sigma, S, t, s_0, F)$$

dove:

- Σ è un alfabeto, ovvero l'insieme dei simboli di input;
- S è un insieme finito di stati;
- t è la funzione di transizione (deterministica) $t : S \times \Sigma \to S$ che associa, ad uno stato e un simbolo di input, un nuovo stato;
- s_0 è lo stato iniziale;
- $F \subseteq S$ è un insieme di stati finali.

Un automa di questo tipo è detto deterministico poiché la sua funzione di transizione non permette l'esistenza, a partire da un medesimo stato, di due transizioni caratterizzate dallo stesso simbolo: l'automa, istantaneamente, si trova in un singolo stato.
Esistono due principali rappresentazioni per gli automi:

- Tabella delle transizioni, rappresentazione tabellare della funzione di transizione dove nelle righe si trovano gli stati di partenza, mentre nelle colonne i simboli (o viceversa). L'elemento in una cella della tabella rappresenta lo stato destinazione della transizione, a partire dallo stato della riga corrispondente e in base al simbolo di input nella relativa colonna; particolari notazioni grafiche possono essere utilizzate per contraddistinguere lo stato iniziale e gli stati finali.
- Diagramma delle transizioni, un grafo nel quale:
  - ogni stato è rappresentato da un vertice;
  - ogni transizione è un arco orientato che connette due vertici e possiede una label indicante il simbolo della transizione;
  - una freccia entrante in un vertice rappresenta lo stato iniziale;
  - ogni vertice associato ad uno stato finale è descritto da un doppio contorno, mentre gli stati non finali hanno come vertice un contorno singolo.

Esempio 2.1. Si consideri un DFA con $\Sigma = \{a, b\}$, $S = \{s_0, s_1, s_2\}$, stato iniziale s_0, insieme di stati finali $F = \{s_2\}$ e funzione di transizione t che si evince dalle rappresentazioni in tabella 2.1 e figura 2.1.

	a	b
$\rightarrow s_0$	s_1	s_0
s_1	s_1	s_2
$*s_2$	s_2	s_2

Tabella 2.1: Rappresentazione tramite tabella di un DFA

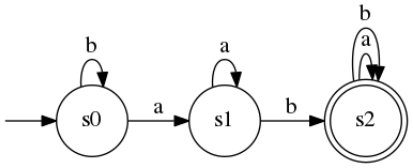


Figura 2.1: Rappresentazione grafica di un DFA

2.1.1 Linguaggio di un DFA

Il linguaggio di un DFA è l'insieme di tutte le stringhe che il DFA accetta. Sia $a_1, a_2, \ldots, a_n$ una sequenza di simboli. A partire dallo stato iniziale s_0, mediante la funzione di transizione t si ha $s_1 = t(s_0, a_1)$ come stato raggiunto a fronte dell'ingresso a_1. Procedendo allo stesso modo per altri simboli, $s_i = t(s_{i-1}, a_i)$, si viene a creare una sequenza di stati raggiunti $s_1, s_2, \ldots, s_n$. La sequenza è accettata se e solo se si raggiunge uno stato finale $s_n \in F$. É possibile introdurre il concetto di *funzione di transizione estesa* che restituisce lo stato raggiunto quando a partire da uno stato si ha una sequenza di simboli d'ingresso, invece che un simbolo soltanto. La funzione di transizione estesa $\hat{t} : S \times \Sigma^* \mapsto S$ è definita come:

$$\hat{t}(s, \underline{\omega}) = \begin{cases} s & \underline{\omega} = \epsilon, \\ t(\hat{t}(q, \underline{x}), a) & \underline{\omega} = \underline{x}\, a \end{cases}$$

Dunque una stringa $\underline{\omega} = a_1\, a_2\, \ldots\, a_n$ appartiene al linguaggio definito dall'automa se e solo se $\hat{t}(s_0, \underline{\omega}) \in F$. Di conseguenza si definisce linguaggio del DFA $D = (S, \Sigma, t, s_0, F)$:

$$L(D) = \{\, \underline{\omega} \mid \hat{t}(s_0, \underline{\omega}) \in F \,\}.$$

2.2 NFA

Un automa a stati finiti non deterministico (NFA: nondeterministic finite automaton) è una quintupla:

$$N = (\Sigma, S, t, s_0, F)$$

dove:

- Σ è un alfabeto, ovvero l'insieme dei simboli di input;
- S è un insieme finito di stati;
- t è la funzione di transizione (non deterministica) $t : S \times (\Sigma \cup \{\epsilon\}) \to 2^S$ che associa, ad uno stato e un simbolo di input (oppure al simbolo vuoto ϵ), un insieme di stati;
- s_0 è lo stato iniziale;
- $F \subseteq S$ è un insieme di stati finali.

La differenza tra gli automi deterministici e quelli non deterministici riguarda la funzione di transizione, che in quest'ultima classe di automi presenta due forme di non determinismo: da un lato, la presenza di ϵ-transizioni può causare il cambiamento di stato dell'automa senza che venga consumato alcun simbolo in input; d'altro canto, la possibile presenza di transizioni uscenti dal medesimo stato con la stessa label fornisce una scelta non deterministica nella simulazione dell'automa. Il teorema che è analizzato nella sezione successiva mostra che è sempre possibile, partendo da un NFA, ricostruire un corrispondente DFA equivalente. L'utilizzo di automi non deterministici è dettato dalla maggior facilità di modellazione nell'ambito di alcuni algoritmi specifici, nonché da procedure note che permettono di costruire un automa non deterministico equivalente ad una espressione regolare data [1].
Le possibili rappresentazioni di NFA sono simili a quelle viste per i DFA, con piccole variazioni: nella rappresentazione tabellare ogni cella, invece che contenere un singolo stato, racchiude un insieme di stati, e come simboli si aggiunge una colonna che esplicita le transizioni scatenate dal simbolo vuoto; nella rappresentazione grafica, si ricorre a frecce con la label ϵ per indicare le ϵ-transizioni.

Esempio 2.2. Un esempio di NFA è rappresentato in figura 2.2. Esso è caratterizzato da $\Sigma = \{a, b\}$, $S = \{s_0, s_1, s_2, s_3, s_4\}$, stato iniziale s_0, insieme di stati finali $F = \{s_4\}$. La relativa funzione di transizione è specificata in tabella 2.2.

[1] Si noti che, come illustrato in [1], è altresì possibile partendo da un'espressione regolare costruire direttamente un DFA.

	a	b	ϵ
$\rightarrow s_0$	$\emptyset$	$\{s_4\}$	$\{s_1, s_2, s_3\}$
s_1	$\{s_4\}$	$\emptyset$	$\{s_3\}$
s_2	$\emptyset$	$\{s_3\}$	$\emptyset$
$*s_3$	$\emptyset$	$\{s_4\}$	$\emptyset$
$*s_4$	$\{s_3\}$	$\emptyset$	$\emptyset$

Tabella 2.2: Rappresentazione tramite tabella di un NFA

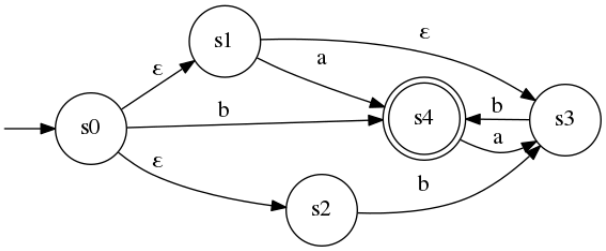


Figura 2.2: Rappresentazione grafica di un NFA

2.2.1 ϵ-closure

La ϵ-closure di uno stato s è l'insieme degli stati raggiunti a partire da s tramite transizioni di stato senza simboli di input, ovvero mediante ϵ-transizioni. Formalmente:

$$\epsilon\text{-}closure(s) = \begin{cases} \{s\} & t(s,\epsilon) = \emptyset \\ \{s\} \cup \bigcup_{s_i \in A} \epsilon\text{-}closure(s_i) & A = t(s,\epsilon) \neq \emptyset \end{cases}$$

É inoltre possibile introdurre la nozione di ϵ-closure per insiemi di stati, $\epsilon\text{-}closure^*$:

$$\epsilon\text{-}closure^*(\mathbb{T} \subseteq S) = \bigcup_{s_i \in \mathbb{T}} \epsilon\text{-}closure(s_i)$$

2.2.2 Linguaggio di un NFA

Attraverso il concetto di ϵ-closure è possibile definire in modo conciso la *funzione di transizione estesa* per stringhe non contenenti il simbolo ϵ. Si definisce $\hat{t} : S \times \Sigma^* \mapsto 2^S$:

$$\hat{t}(s, \underline{\omega}) = \begin{cases} \epsilon\text{-}closure(s) & \omega = \epsilon \\ \epsilon\text{-}closure^*(A) & \omega = \underline{x}\, a\,,\ A = \bigcup_{s_i \in B} t(s_i, a)\,,\ B = \hat{t}(s, \underline{x}) \end{cases}$$

Dunque una stringa $\underline{\omega} = a_1\, a_2\, \ldots\, a_n$ appartiene al linguaggio definito dall'automa se e solo se $\hat{t}(s_0, \underline{\omega}) \in F$.

Di conseguenza si definisce linguaggio del NFA $N = (S, \Sigma, t, s_0, F)$:

$$L(N_\epsilon) = \{\, \underline{\omega} \mid \hat{t}(s_0, \underline{\omega}) \cap F \neq \emptyset \,\}.$$

2.3 Subset construction

A causa del loro non determinismo, spesso i NFA necessitano di essere convertiti in DFA per essere utilizzati e simulati in modo più intuitivo. La tecnica nota in letteratura per ottenere questa conversione è chiamata *subset construction*. L'idea generale di questo algoritmo è che ad ogni stato del risultante DFA corrisponda un insieme di stati del NFA di partenza. Il primo problema che bisogna affrontare è quello di gestire le ϵ-transizioni. Queste particolari transizioni indicano in quali stati l'automa può trovarsi contemporaneamente. L'operazione di ϵ-closure permette di trovare, a partire da un particolare stato, tutti gli stati raggiungibili senza la consumazione di simboli in ingresso. Il primo passo della subset construction, quindi, consiste nell'assegnare allo stato iniziale del DFA risultante la ϵ-closure dello stato iniziale del NFA in esame. L'algoritmo prosegue calcolando, per ogni possibile input ricevuto a partire da ogni stato del NFA contenuto nello stato attuale del DFA, l'insieme degli stati raggiungibili, su cui viene effettuata la ϵ-closure. Il procedimento procede in questo modo sino a che tutti i nuovi stati generati nel DFA non sono stati processati. Lo pseudocodice è fornito nell'algoritmo 1; la funzione $move(T, a)$ calcola l'insieme di tutti gli stati del NFA per i quali vi è una transizione, mediante il simbolo a, per qualche stato $s \in T$. É possibile, teoricamente, che il numero di stati del DFA sia esponenziale nel numero di stati del NFA: tale numero è limitato superiormente dalla cardinalità dell'insieme potenza degli stati dell'automa non deterministico, cioè dal numero di tutti i possibili sottoinsiemi degli stati. Fortunatamente questo caso pessimo raramente si presenta e per linguaggi reali spesso il numero di stati dei due automi è simile e il comportamento esponenziale non sussiste.

Algorithm 1 Algoritmo subset construction

SubsetConstruction(N)

```
Sia s_0 lo stato iniziale del NFA, Dstates gli stati del DFA risultante
Sia Dtran la funzione di transizione del DFA
while ∃s ∈ Dstates non marcato do
  marca s
  for all simbolo di input a do
    U = ε − closure(move(T, a))
    if U ∉ Dstates then
      aggiungi U come stato non marcato in Dstates
    end if
    Dtran(T, a) = U
  end for
end while
```

2.4 Linguaggi

Un automa a stati finiti possiede delle transizioni i cui simboli costituiscono l'alfabeto di un linguaggio e una stringa ottenuta percorrendo un cammino dallo stato iniziale ad uno stato finale si dice che appartiene al linguaggio. Una stringa caratterizzata da nessun simbolo dell'alfabeto è detta stringa vuota e si denota con ϵ. Data una stringa s, la sua lunghezza $|s|$ è il numero di simboli contenuti in essa, tenendo conto di eventuali occorrenze multiple del medesimo simbolo. Per convenzione, la lunghezza della stringa vuota è zero.

Definizione 2.1. Un linguaggio definito su un insieme alfabeto Σ è un insieme di stringhe di lunghezza finita formate da simboli appartenenti a Σ.

L'operazione fondamentale per la costruzione di linguaggi è quella della concatenazione. Se x e y sono stringhe, la loro concatenazione è la stringa xy. La stringa vuota è l'elemento identità di tale operazione, poiché per ogni stringa s, $s\epsilon = \epsilon s = s$.
Un'altra operazione che caratterizza i linguaggi è la chiusura di Kleene, denotata L^*, che è l'insieme di tutte le stringhe finite che si ottengono concatenando elementi dell'alfabeto, compresa la stringa vuota.
Sui linguaggi è altresì possibile applicare tutte le operazioni insiemistiche quali l'unione, l'intersezione, la differenza e il complemento. Un linguaggio può essere visto come un modo formale di descrivere il comportamento di un sistema, specificando tutte le possibili sequenze di eventi che esso è in grado di generare. Un automa è un formalismo in grado di rappresentare un linguaggio. In particolare, nel seguito della trattazione, ci si focalizzerà sui cosiddetti linguaggi regolari, ovvero su quei linguaggi che possono essere rappresentati tramite automi con un numero finito di stati.

2.5 Espressioni regolari

Le espressioni regolari costituiscono quell'insieme di linguaggi che possono essere rappresentati da automi a stati finiti (grammatica di tipo 3 della gerarchia di Chomsky). Le espressioni regolari descrivono tutti i linguaggi costruiti applicando ai simboli appartenenti ad un alfabeto Σ gli operatori di unione, concatenazione e chiusura. Le espressioni regolari sono generate ricorsivamente dalle sottoespressioni che le formano, secondo le seguenti regole.

1. ϵ è una espressione regolare che denota il linguaggio $L(\epsilon) = \{\epsilon\}$, contenente la sola stringa vuota;

2. se a è un simbolo in Σ, allora a è un'espressione regolare che denota il linguaggio $L(a) = \{a\}$, contenente la sola stringa di lunghezza unitaria a;

3. se x e y sono espressioni regolari che denotano rispettivamente i linguaggi $L(x)$ e $L(y)$, allora:

- (x) è l'espressione regolare che denota il linguaggio $L(x)$;
- $(x)|(y)$ è l'espressione regolare che denota il linguaggio $L(x) \cup L(y)$ (unione);
- $(x)(y)$ è l'espressione regolare che denota il linguaggio $L(x)L(y) = \{xy|x \in L(x), y \in L(y)\}$ (concatenazione);
- $(x)^*$ è l'espressione regolare che denota il linguaggio $(L(x))^*$: ripetizione zero o più volte di stringhe in $L(x)$ (chiusura di Kleene).

Altri operatori possono essere definiti opportunamente in base al dominio applicativo, generando espressioni regolari estese.

2.5.1 Costruzione di Thompson

La costruzione di *McNaughton-Yamada-Thompson* (o semplicemente costruzione di Thompson) è in grado di convertire una espressione regolare in un NFA caratterizzato dal medesimo linguaggio. L'algoritmo opera scomponendo l'espressione regolare in un albero sintattico e costruendo un NFA per ogni sottoespressione, in maniera bottom-up, utilizzando le ϵ-transizioni come "collante" per unire i sottoautomi. L'automa risultante permette di riconoscere se una stringa appartiene o meno al linguaggio relativo all'espressione regolare di partenza.
La regola base del metodo, rappresentata in figura 2.3, genera un NFA da una sottoespressione atomica, composta da un solo simbolo a o dal simbolo nullo ϵ. L'automa generato è composto da due stati, uno iniziale e l'altro finale, e una transizione in corrispondenza del simbolo atomico dallo stato iniziale allo stato finale. Le regole induttive permettono di ricavare ricorsivamente l'intero automa. Si supponga che $N(s)$ e $N(t)$ siano gli NFA ricavati dalle espressioni regolari s e t rispettivamente.

1. Si supponga $r = s|t$. L'automa $N(r)$ equivalente all'espressione regolare r è costruito in figura 2.4. Sono creati due nuovi stati, uno iniziale e uno finale. Due ϵ-transizioni uniscono lo stato iniziale del nuovo automa agli stati iniziali di $N(r)$ e $N(s)$. Si noti che gli stati finali di $N(r)$ e $N(s)$ non sono finali nel nuovo automa, e una ϵ-transizione collega ciascuno al nuovo stato finale. Il linguaggio dell'automa generato è l'unione dei linguaggi dei due automi di partenza.

2. Si supponga $r = st$. L'automa $N(r)$ equivalente all'espressione regolare r è costruito in figura 2.5. Lo stato iniziale di $N(s)$ diviene in nuovo stato iniziale di $N(r)$, mentre lo stato finale di $N(t)$ costituisce il nuovo stato finale dell'automa risultante $N(r)$. Il linguaggio dell'automa generato è la concatenazione dei linguaggi dei due automi di partenza.

3. Si supponga $r = s^*$. L'automa $N(r)$ equivalente all'espressione regolare r è costruito in figura 2.6. Sono creati due nuovi stati, uno iniziale e uno finale. A partire dallo

stato iniziale, vengono poste due ϵ-transizioni: una verso lo stato iniziale di $N(s)$, l'altra verso il nuovo stato finale. Inoltre è aggiunta una ϵ-transizione dallo stato finale per $N(s)$ al suo stato iniziale per $N(s)$. Il linguaggio dell'automa generato dalla chiusura di Kleene dei linguaggi dei due automi di partenza.

4. Si supponga $r = (s)$. Allora il linguaggio di r coincide con quello di s, e come automa $N(r)$ può essere utilizzato l'automa di partenza $N(s)$.

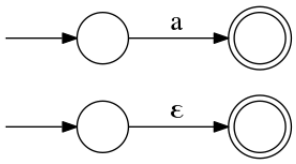


Figura 2.3: NFA per espressioni regolari atomiche

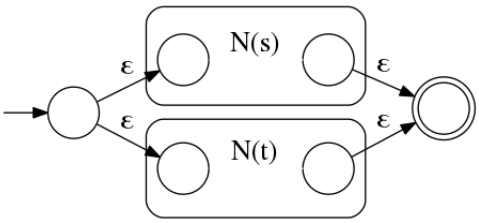


Figura 2.4: NFA per l'unione di due espressioni regolari

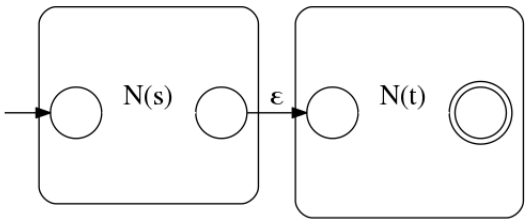


Figura 2.5: NFA per la concatenazione di due espressioni regolari

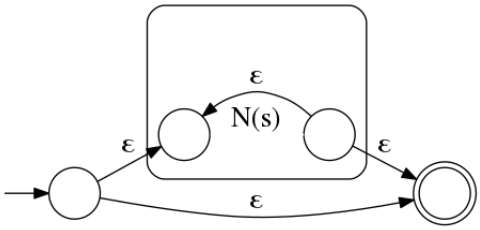


Figura 2.6: NFA per la chiusura di un'espressione regolare

2.6 Minimizzazione degli stati

Dato un linguaggio, possono esistere più automi deterministici che lo riconoscono. Si cerca sempre di scegliere l'automa che presenta il minor numero di stati, dato che l'implementazione di quest'ultimo consente di risparmiare memoria e tempo nella simulazione. Esiste sempre un unico DFA minimo per ogni linguaggio regolare, a meno di una ridenominazione del nome degli stati. Questo particolare automa può essere costruito raggruppando insieme quegli stati che sono tra loro equivalenti. Per capire come verificare tale equivalenza, si ricorre al concetto di distinguibilità.

Definizione 2.2. La stringa x distingue lo stato s dallo stato t se esattamente uno degli stati raggiunti partendo da s e da t seguendo il cammino con label x è uno stato finale. Quindi, lo stato s è distinguibile dallo stato t se esiste una stringa che li distingue. In particolare, la stringa vuota distingue ogni stato finale da qualsiasi stato non finale.

L'algoritmo di minimizzazione degli stati funziona partizionando gli stati dell'automa in gruppi di stati indistinguibili. In ogni istante, l'algoritmo mantiene partizioni di stati che non sono ancora stati distinti. Inizialmente gli stati sono divisi in due macro-partizioni, una contenente gli stati non finali, l'altra contente quelli finali. La divisione iniziale viene raffinata individuando un simbolo di input che distingue tra loro alcuni degli stati appartenenti ad un gruppo, generando in questo modo una ulteriore suddivisione interna. La procedura viene applicata fino a quando nessun gruppo, per ogni simbolo di input, può essere scisso nuovamente. Infine ogni gruppo di stati, scegliendo per ogni insieme uno stato come rappresentante, viene fuso in un singolo stato che appartiene all'automa minimo risultante.

Capitolo 3

Sistemi Attivi

In questo capitolo vengono presentate definizioni, modelli ed esempi riguardanti i sistemi attivi tradizionali[2]. I sistemi attivi rappresentano una classe specifica di sistemi a eventi discreti[3]. Ad un certo livello di astrazione, generalmente, qualsiasi sistema fisico può essere modellato attraverso un comportamento discreto. Un sistema, infatti, non è continuo o discreto di per sé, anche se si può prestare o meno ad una certa scelta nel modello da adottare. I sistemi attivi sono asincroni; questo significa che gli eventi generati dai componenti sono immagazzinati nei link prima di essere consumati (in maniera asincrona). Il modello dei sistemi attivi include due tipi fondamentali di elementi: i componenti e i link. Un sistema attivo è una rete di componenti connessi gli uni agli altri per mezzo di link uscenti da terminali di output di alcuni componenti ed entranti nei terminali di input di altri componenti. Il comportamento di ogni componente è descritto da un automa a stati finiti, le cui transizioni tra gli stati sono compiute in base alla consumazione di determinati eventi disponibili nei terminali di input. L'esecuzione di una transizione causa la generazione di eventi trasmessi ai terminali di output del componente. É altresì possibile che alcune transizioni non vengano innescate da alcun evento particolare presente nel modello: in questo caso si assume che l'evento scatenante provenga dal mondo esterno al sistema. Il modello comportamentale del componente è assunto essere completo, nel senso che racchiude sia le transizioni normali, sia quelle di guasto. Un sistema attivo è quindi caratterizzato da una topologia (collegamenti tra componenti) e dal comportamento dei singoli componenti e dei link. Questi ultimi, in un problema generale, potrebbero avere differenti capacità di immagazzinamento (in termini di numero di eventi) e comportamenti diversi nel caso siano colmi (la cosiddetta politica di saturazione). Il modello globale del sistema è quindi implicitamente dato dalla topologia dell'insieme e dai comportamenti dei singoli componenti e dei link.

3.1 Definizioni

3.1.1 Componenti

I componenti sono i costituenti base dei sistemi attivi. Ogni componente è caratterizzato da due modelli:

1. *modello topologico*, secondo il quale un componente è descritto da un insieme di terminali di input, da cui gli eventi in ingresso sono consumati, e da un insieme di terminali di output, dai quali gli eventi in uscita sono generati;

2. *modello comportamentale*, descritto da un automa a stati finiti che ingloba sia il comportamento normale sia le transizioni di guasto, dotato di una funzione di transizione che mappa uno stato e un evento in ingresso in un nuovo stato, generando un sottoinsieme (eventualmente vuoto) di eventi nei terminali di uscita.

Definizione 3.1. Un modello di un componente è un automa:

$$M_c = (S, E_{in}, I, E_{out}, O, \tau)$$

dove S è l'insieme degli stati, E_{in} è l'insieme degli eventi in ingresso, I è l'insieme dei terminali di input, E_{out} è l'insieme degli eventi in uscita, O è l'insieme dei terminali di output, e τ è la funzione di transizione (non deterministica):

$$\tau : S \times (E_{in} \times I) \times 2^{(E_{out} \times O)} \to 2^S.$$

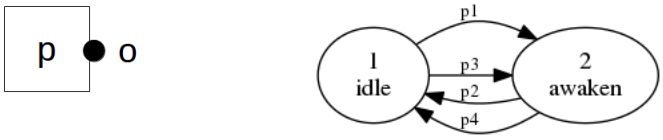
Un evento è una coppia (e, θ), dove e è un ingresso (uscita) e θ un terminale di ingresso (uscita). Un componente particolare può essere visto come una istanza di un modello di componente. Una transizione t da uno stato s ad uno stato s' è innescata da un evento in ingresso (e, x) disponibile al terminale di input x e genera l'insieme (possibilmente vuoto) di eventi $\{(e_1, y_1), \ldots, (e_n, y_n)\}$ in corrispondenza dei terminali di output $y_1, \ldots, y_n$, in simboli:

$$t = s \xrightarrow{(e,x) \Rightarrow (e_1,y_1),\ldots,(e_n,y_n)} s'.$$

É implicitamente definito un terminale di input virtuale, chiamato In, attraverso il quale giungono gli eventi esterni al sistema.

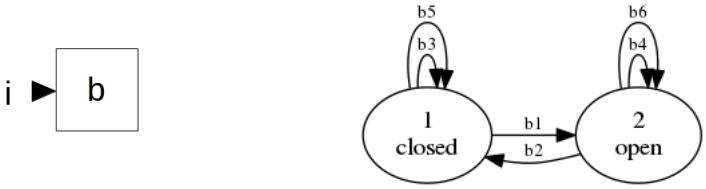
Esempio 3.1. Si consideri l'esempio di un sistema attivo costituito da un meccanismo di protezione per una linea elettrica. Il componente p rappresenta la protezione (*protection device*), ovvero un sensore che si attiva quando rileva bassa tensione ai capi della linea. Il componente b rappresenta il *breaker*, ovvero il dispositivo che, comandato dalla protezione, si apre isolando la linea elettrica e prevenendo il cortocircuito, oppure si chiude riconnettendo la rete. Nella figure 3.1 e 3.2 sono rappresentati, rispettivamente, i modelli relativi alla protezione e quelli riferiti al breaker. Il componente di protezione

non possiede terminali di input e ha un terminale di output o, mentre il breaker dispone solamente di un terminale di input i. La protezione può essere nello stato inattivo *idle* quando non viene rilevato alcun cortocircuito, oppure nello stato *awaken*, nel momento in cui la tensione si abbassa oltre una certa soglia. Quando viene rilevato un cortocircuito, il componente p può compiere le transizioni p_1, che genera un evento *op* per comandare l'apertura del breaker, o p_3, che a causa di un malfunzionamento genera invece un evento *cl* per comandare la chiusura del breaker. Quando il cortocircuito svanisce, il componente torna nello stato *idle* percorrendo la transizione p_2, che genera l'evento *cl*, oppure tramite p_4, che genera in maniera difettosa l'evento *op*. Il breaker può essere aperto (*open*) o chiuso (*closed*). Quando è chiuso e un evento *op* è disponibile in corrispondenza del terminale i, il breaker b può percorrere la transizione b_1 che lo apre o b_3 con la quale resta chiuso (guasto). Quando il componente è aperto, le transizioni possibili sono b_2 se il breaker si chiude, b_4 se a causa di un guasto rimane aperto. Le transizioni b_5 e b_6 non cambiano lo stato del componente. Un sunto delle transizioni relative ai due componenti è fornito nelle tabelle 3.1 e 3.2.



(a) topologia protezione (b) comportamento protezione

Figura 3.1: Modello del componente p



(a) topologia breaker (b) comportamento breaker

Figura 3.2: Modello del componente b

3.1.2 Link

Nell'ambito dei sistemi attivi, i componenti sono tra loro connessi per mezzo di link. Ogni link esce da un terminale di output y di un componente c e entra in un terminale di input x di un componente c'. Analogamente a quanto avviene per i componenti, anche i link sono caratterizzati da un modello, che costituisce un'astrazione del link specifico appartenente al sistema.

Transizione	**Descrizione**
$p_1 = idle \xrightarrow{(sh,In)\Rightarrow(op,o)} awaken$	Rileva un cortocircuito e genera l'evento *op*
$p_2 = awaken \xrightarrow{(ok,In)\Rightarrow(cl,o)} idle$	Rileva la fine del cortocircuito e genera l'evento *cl*
$p_3 = idle \xrightarrow{(sh,In)\Rightarrow(cl,o)} awaken$	Rileva un cortocircuito ma genera l'evento *cl*
$p_4 = awaken \xrightarrow{(ok,In)\Rightarrow(op,o)} idle$	Rileva la fine del cortocircuito ma genera l'evento *op*

Tabella 3.1: Transizioni relative alla protezione

Transizione	**Descrizione**
$b_1 = closed \xrightarrow{(op,i)} open$	Consuma l'evento *op* e apre
$b_2 = open \xrightarrow{(cl,i)} closed$	Consuma l'evento *cl* e chiude
$b_3 = closed \xrightarrow{(op,i)} closed$	Consuma l'evento *op* ma rimane chiuso
$b_4 = open \xrightarrow{(cl,i)} open$	Consuma l'evento *cl* ma rimane aperto
$b_5 = closed \xrightarrow{(cl,i)} closed$	Consuma l'evento *cl*
$b_6 = open \xrightarrow{(op,i)} open$	Consuma l'evento *op*

Tabella 3.2: Transizioni relative al breaker

Definizione 3.2. Un modello di un link è una quadrupla

$$M_l = (x, y, z, w)$$

dove x è il terminale di input, y il terminale di output, z la dimensione e w la politica di saturazione.

Un particolare link l è un'istanza di un modello siffatto, e consiste quindi in un canale di comunicazione unidirezionale fra due componenti distinti c e c', dove un terminale di output y di c e un terminale di input x di c' coincidono rispettivamente con l'input e l'output del link l. La dimensione z rappresenta il numero massimo di eventi che possono essere accodati nel link. Indichiamo con $|l|$ la configurazione corrente del link. Se il numero di eventi attualmente memorizzati coincide con la dimensione, il link si dice essere saturo. Quando il link è saturo, la semantica legata al compimento delle transizioni è dettata dalla politica di saturazione w, la quale può essere:

- *lose*: l'evento, non potendo essere memorizzato, viene perso;
- *override*: l'evento sovrascrive l'ultimo evento nel link;
- *wait*: la transizione non viene portata a termine fintanto che il link permane nello stato di saturazione, ovvero fino a quando almeno un evento nel link non viene consumato.

Nell'ambito di questa tesi, si considererà il caso particolare di link di dimensione unitaria e politica di saturazione *wait* , ovvero $z = 1$ e $w = wait$. Tale supposizione permette di visualizzare le configurazioni dei link dell'intero sistema come la tupla del contenuto dei terminali di input di ogni componente del sistema. Implicitamente questo significa che ogni evento generato è inserito istantaneamente, se libero, nel terminale (o nei terminali) di input liberi collegati al terminale di output. Se un terminale di destinazione è occupato, la transizione non viene compiuta.

Esempio 3.2. Con riferimento all'esempio 3.1, in figura 3.3 è presentato il link *pb* che collega il terminale di output o della protezione p al terminale di input i del breaker b.

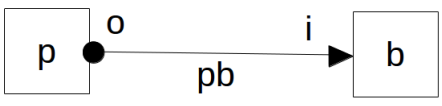


Figura 3.3: Modello del sistema attivo $\overline{A}$

3.1.3 Sistema attivo

Un sistema attivo è una rete di componenti interconnessi per mezzo di link. Ogni componente ed ogni link del sistema sono caratterizzati da un modello e quindi, in generale, più elementi potrebbero avere il medesimo modello. Si assume che più link possano uscire da un terminale di output di un componente, mentre al massimo un link possa entrare in un terminale di input. Un sistema potrebbe contenere dei terminali che non sono connessi con alcun link: questi vengono chiamati terminali scollegati.

Definizione 3.3. Un sistema attivo è una tripla

$$A = (C, L, D)$$

dove C è l'insieme dei componenti, L l'insieme dei link tra terminali di componenti in C, e D è l'insieme dei terminali scollegati. Quest'ultimo insieme può essere visto come l'unione di due insiemi disgiunti:

$$D = D_{on} \cup D_{off}$$

dove D_{on} è l'insieme dei terminali on scollegati, mentre D_{off} è l'insieme dei terminali off scollegati.

Se $D_{on} \neq \emptyset$, il sistema A è detto aperto, altrimenti è detto chiuso. Nel caso A sia chiuso ($D_{on} = \emptyset$), nessun evento è disponibile nei terminali di input di D_{off}, mentre gli eventi generati nei terminali di output di D_{off} sono persi. D'altro canto, se A è aperto, si assume che i terminali scollegati in D_{on} siano connessi attraverso link all'esterno del sistema A; quest'ultimo, in altre parole, è come se fosse incorporato in un altro sistema più grande (sconosciuto). Quindi gli eventi generati in terminali di output in D_{on} sono immagazzinati all'interno dei link (sconosciuti) esterni al sistema A. Analogamente, il componente in corrispondenza del quale si trova un terminale di input in D_{on} è sensibile a eventi disponibili in quel terminale.

Esempio 3.3. Con riferimento agli esempi 3.1 e 3.2, il sistema rappresentato in figura 3.3 è un sistema attivo $\overline{A} = (\{p, b\}, \{pb\}, \emptyset)$ nel quale non è presente alcun terminale scollegato.

3.1.4 Traiettoria

Un sistema attivo può essere pensato come una macchina che si trova in uno stato quiescente o in uno stato reattivo. Se si trova nello stato quiescente, i componenti non effettuano alcuna transizione, dal momento che nessun evento è disponibile nei terminali di ingresso. In corrispondenza del verificarsi di un determinato evento, sia esso proveniente dal mondo esterno, sia da un terminale di input scollegato appartenente a D_{on}, il sistema evolve nella sua fase reattiva. Dato che il comportamento del sistema è asincrono (non dipende dal tempo), la reazione, detta *traiettoria* (o *storia*), consiste in una sequenza di transizioni

compiute da componenti presenti nel sistema. Ogni transizione di un componente porta il sistema in un nuovo stato, dove uno stato è identificato dallo stato corrente di ogni componente e dallo stato attuale di ogni link. In altre parole, uno stato del sistema attivo è una coppia (S, Q), dove S è la n-pla degli stati dei componenti, mentre Q è la m-pla delle configurazioni dei link, ovvero la m-pla del contenuto dei terminali di input di tutti i componenti. In questo lavoro di tesi, il focus è sulla cosiddetta diagnosi a posteriori, che avviene quando il sistema ha percorso una traiettoria partendo da uno stato quiescente e tornando in uno stato di quiete, nel quale cioè tutti i terminali di input dei componenti sono vuoti. In questo contesto, una traiettoria completa può essere vista come una sequenza di transizioni che, partendo dallo stato iniziale (quiescente), termina in uno stato finale (anch'esso quiescente). Una transizione del sistema può essere scritta come:

$$T = (S, Q) \xrightarrow{t(c)} (S', Q'),$$

dove $t(c)$ identifica univocamente la transizione t appartenente al modello del componente c. Assumendo che $a_0 = (S_0, Q_0)$ sia lo stato iniziale del sistema, la traiettoria ottenuta partendo da a_0 è la sequenza di stati del sistema determinata dall'occorrenza di transizioni $t_1, \ldots, t_k$ attuabili da parte dei singoli componenti:

$$h = a_0 \xrightarrow{t_1(c_1)} a1 \xrightarrow{t_2(c_2)} a2 \ldots \xrightarrow{t_k(c_k)} a_k.$$

Ogni stato (non iniziale) dipende quindi dallo stato precedente e dalla particolare transizione che lo porta allo stato corrente, ovvero la coppia $(a_{i-1}, t_i(c_i))$. Assumendo che si parta dallo stato iniziale a_0 questo ci permette di identificare una traiettoria per mezzo delle sole transizioni dei componenti:

$$h = [t_1(c_1), t_2(c_2), \ldots, t_k(c_k)].$$

Esempio 3.4. Con riferimento al sistema attivo in figura 3.3, si assuma che lo stato iniziale sia la composto dalla tripla $(idle, closed, \epsilon)$ indicante rispettivamente gli stati dei componenti p e b (figure 3.1 e 3.2) e il contenuto inizialmente vuoto del link pb, ovvero del terminale di input i del breaker. Una possibile traiettoria è la seguente:

$$\overline{h} = (idle, closed, \epsilon) \xrightarrow{p_1(p)} (awaken, closed, op) \xrightarrow{b_1(b)} (awaken, open, \epsilon) \xrightarrow{p_2(p)} (idle, open, cl) \xrightarrow{b_2(b)} (idle, closed, \epsilon)$$

che corrisponde alla seguente dinamica:

1. la protezione rileva un cortocircuito e comanda l'apertura del breaker;
2. il breaker si apre;
3. la protezione rileva la fine del cortocircuito e comanda la chiusura del breaker;
4. il breaker di chiude.

Si noti come lo stato finale della traiettoria $\overline{h}$ coincida con lo stato iniziale della medesima traiettoria. Questa ciclicità permette di inferire la presenza di infinite traiettorie per il sistema attivo in esame, in quanto percorrere un ciclo un diverso numero di volte porta alla percorrenza di traiettorie diverse. Da osservare, inoltre, come una siffatta dinamica sia esente da guasti: nelle sezioni successive verranno analizzati casi più generali, contraddistinti da malfunzionamenti, e saranno fornite le informazioni necessarie al fine di discriminare le transizioni normali da quelle di guasto.

3.1.5 Behavior Space

Dato un sistema attivo ed il suo stato iniziale, possono esservi molte traiettorie percorribili, persino infinite. Analogamente a quanto avviene per il comportamento del singolo componente, che può essere descritto nel suo modello come un automa a stati finiti, anche per quanto riguarda le traiettorie dell'intero sistema si può seguire un procedimento analogo. Un automa a stati finiti rappresenta infatti, attraverso un numero di stati e di transizioni finiti, un numero di traiettorie che può essere infinito. Questo è dovuto alla possibile presenza di ciclicità all'interno dell'automa. Tale automa è chiamato *behavior space* e ha come alfabeto l'intero insieme delle transizioni dei singoli componenti del sistema. Il linguaggio del *behavior space* $Bsp(A)$ con stato iniziale a_0 coincide con l'insieme di tutte le possibili traiettorie del sistema A partendo dallo stato a_0.

Definizione 3.4. Sia $A = (C, L, D)$ un sistema attivo, dove C è l'insieme di n componenti, mentre L è l'insieme di m link. Il behavior space di A è il DFA

$$Bsp(A) = (\Sigma, \alpha, \tau, a_0, \alpha_f)$$

dove:

1. Σ è l'alfabeto, dato dall'unione delle transizioni dei componenti in C;
2. α è l'insieme degli stati (S, Q), con $S = (s_1, \ldots, s_n)$ una n-pla di stati dei componenti in C, e $Q = (q_1, \ldots, q_m)$ una m-pla del contenuto dei terminali di input di tutti i componenti;
3. $a_0 = (S_0, Q_0)$ è uno stato iniziale quiescente, nel quale tutti i terminali di input sono vuoti, cioè $Q_0 = (\epsilon, \ldots, \epsilon)$;
4. α_f è l'insieme degli stati finali, ovvero stati in cui $Q = (\epsilon \ldots \epsilon)$;
5. τ è la funzione di transizione deterministica, $\tau : \alpha \times \Sigma \to \alpha$, tale che $(S, Q) \xrightarrow{t(c)} (S', Q') \in \tau$, dove $S = (s_1, \ldots, s_n)$, $Q = (q_1, \ldots, q_m)$, $S' = (s'_1, \ldots, s'_n)$, $Q' = (q'_1, \ldots, q'_m)$ e

$$t(c) = s \xrightarrow{(e,x)|\{(e_1,y_1),\ldots,(e_p,y_p)\}} s'$$

se e solo se:

- $x = In$, cioè l'evento è disponibile sul terminale di input virtuale sensibile agli eventi esterni al sistema, oppure $x \in D_{on}$, o e è pronto al terminale x all'interno di un link del sistema;
- Per ogni $i \in [1 \dots n]$, abbiamo

$$s'_i = \begin{cases} s' & \text{se } c_i = c \\ s_i & \text{altrimenti} \end{cases}$$

  cioè per ogni transizione del *behavior space* cambia lo stato relativo al singolo componente coinvolto nella transizione;
- Q' differisce da Q per le seguenti condizioni:
  - $Q(x) = e$, $Q'(x) = \epsilon$, cioè l'evento in input è consumato;
  - $\forall (e_j, y_j), j \in [1 \dots p], Q(y_j) = \epsilon$, $Q'(y_j) = e_j$, cioè gli eventi di uscita sono inseriti nei terminali di input connessi ai terminali di output coinvolti nella transizione.

Il *behavior space*, contenendo tutte le possibili traiettorie del sistema, assume dimensioni considerevoli anche con la presenza di pochi componenti e relativi terminali di input. Questo perché la sua dimensione ha una complessità esponenziale nel numero di componenti e link, dato che uno stato del *behavior space* è caratterizzato, nel caso peggiore, da tutte le possibili combinazioni di stati dei componenti e contenuto dei link [1]. Per questo motivo, si assume che un automa siffatto non possa essere generato perché troppo costoso, in termini di tempo di esecuzione ma soprattutto di memoria. Nella risoluzione del problema di diagnosi, infatti, verrà generato un automa più contenuto, chiamato semplicemente *behavior*.

Esempio 3.5. Il *behavior space* del sistema attivo $\overline{A}$ dell'esempio 3.3 è riportato in figura 3.4. Si considera come stato iniziale a_0 lo stato nel quale la protezione è inattiva, il breaker è chiuso e il link *pb* è vuoto. Gli archi evidenziati rappresentano la particolare traiettoria $\overline{h}$ introdotta nell'esempio 3.4, mentre i nodi con un doppio contorno rappresentano gli stati finali α_f nei quali l'unico link *pb* del sistema è vuoto.

[1] Si tratta di un limite di complessità superiore. In base alle particolari transizioni coinvolte, infatti, non tutte le combinazione sono possibili stati del *behavior space*.

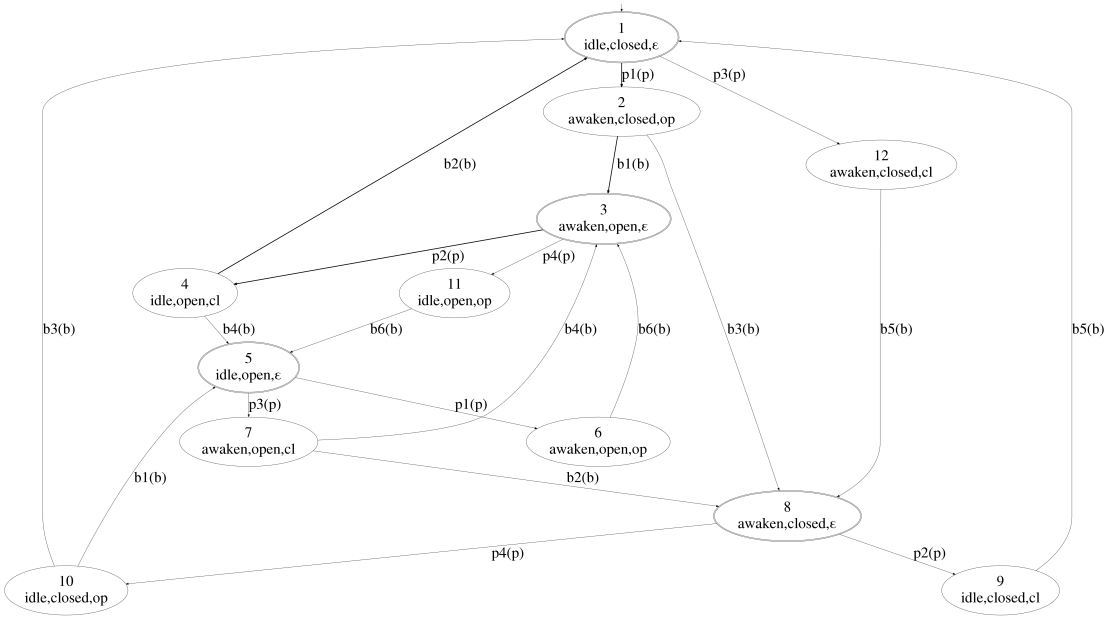


Figura 3.4: Behavior space del sistema attivo $\overline{A}$

3.2 Problema diagnostico

Un sistema attivo, inizialmente, si trova in uno stato quiescente, ovvero in uno stato in cui i link sono vuoti. Il sistema passa alla fase reattiva nel momento in cui riceve dei particolari eventi esterni. Una volta che l'occorrenza di un tale evento si verifica, la reazione del sistema è composta da una sequenza di transizioni che formano una traiettoria. Dato che ogni componente è modellato dal suo comportamento completo, ovvero sia dai suoi comportamenti normali sia quelli di guasto, sono necessarie delle informazioni aggiuntive, in grado di determinare se le transizioni facenti parte della traiettoria siano di guasto o normali. Un ulteriore problema consiste nella non completa osservabilità tipica dei sistemi reali: una traiettoria, in generale, non è percepita come effettivamente è, ma quello che è visibile consiste in una proiezione di un suo sottoinsieme osservabile. Una sequenza di label osservabili costituisce una traccia della reale traiettoria. Inoltre, una label associata ad una transizione di un componente potrebbe non identificare univocamente la transizione, poiché più transizioni possono condividere la medesima label di osservazione. Si noti come i modelli comportamentali dei singoli componenti non effettuino una distinzione tra ciò che osservabile e quello che non lo è: per questo motivo è necessaria una specifica esplicita di questa informazione. Una ulteriore componente significativa del problema è l'osservazione, solo a seguito della quale può essere formulata una diagnosi. Un'osservazione consiste in una sequenza di label osservabili.

3.2.1 Viewer

Una transizione è osservabile se, in corrispondenza della sua esecuzione, genera una label osservabile, altrimenti la transizione è non osservabile. La specifica dell'osservabilità di un sistema attivo è una corrispondenza tra transizioni (osservabili) e rispettive label.

Definizione 3.5. Sia A un sistema attivo e Ω un dominio di label osservabili. Un viewer V di A è una funzione suriettiva da un sottoinsieme di transizioni di componenti a Ω.

Si noti che, dal momento che la funzione di osservabilità è suriettiva, possono esservi più transizioni che vengono mappate nella stessa label. Questo aspetto è dovuto alla limitata osservabilità di un sistema così caratterizzato.

Definizione 3.6. Sia h una traiettoria di un sistema attivo A, e V un viewer per A. La traccia di h basata su V, scritta $h_{[V]}$, è la sequenza di etichette osservabili:

$$h_{[V]} = [l | t \in h, (t, l) \in V]$$

In altre parole, la traccia di una traiettoria h di un sistema attivo è la proiezione delle transizioni osservabili di h nelle corrispondenti label osservabili definite nel viewer V.

Esempio 3.6. In tabella 3.3 è rappresentato un possibile *viewer* per il sistema attivo $\overline{A}$ dell'esempio 3.3. Si noti che le uniche transizioni osservabili del breaker sono b_1 e b_2. Per quanto riguarda la protezione, nonostante tutte le transizioni siano osservabili, la stessa etichetta *awk* è associata alle transizioni p_1 e p_3, mentre la label *ide* è associata alle transizioni p_2 e p_4.

Transizione	p_1	p_2	p_3	p_4	b_1	b_2	b_3	b_4	b_5	b_6
Label osservabile	*awk*	*ide*	*awk*	*ide*	*opb*	*clb*				

Tabella 3.3: *Viewer* per il sistema attivo $\overline{A}$

3.2.2 Osservazione temporale

L'osservazione temporale di un sistema attivo è una sequenza di label osservabili, dunque una traccia. Si noti che ad una stessa traccia possono corrispondere più traiettorie, a causa della presenza di transizioni non osservabili o di transizioni indistinguibili poiché mappate nella medesima label.

Esempio 3.7. Con riferimento all'esempio 3.3 e al *viewer* definito nella tabella 3.3, una possibile osservazione del sistema attivo $\overline{A}$ è la seguente:

$$\overline{O} = [awk, opb, ide]$$

3.2.3 Ruler

La distinzione tra comportamento normale e difettoso viene fornita esplicitando quali tra le transizioni dei componenti sono corrette e quali di guasto. In questo modo una transizione $a \xrightarrow{t(c)} a'$ di A è di guasto se e solo se la transizione t del componente c è di guasto. L'informazione che permette di individuare i guasti è fornita dal ruler.

Definizione 3.7. Sia A un sistema attivo e Φ un dominio di etichette di guasto. Un ruler R per A è una funzione, in generale suriettiva, da un sottoinsieme di transizioni di componenti a Φ.

Si noti che, per convenienza, la funzione di mapping potrebbe essere biettiva; tuttavia alcune transizioni di guasto possono non essere distinte le une dalle altre, in base al livello di precisione che si vuole dare alla diagnosi. Per esempio, due o più transizioni di guasto relative allo stesso componente possono essere mappate nella medesima label di guasto: in questo modo la diagnosi rileverà un generico guasto al componente, che in alcuni casi reali può essere una informazione sufficiente. Si noti che la specifica del viewer, del ruler e dello stato iniziale viene data posteriormente al modello del sistema. Questo perché, in linea generale, differenti problemi di diagnosi, anche legati allo stesso sistema, possono essere caratterizzati da ontologie diverse, ognuna delle quali con un particolare modo di specificare le transizioni osservabili e di guasto. Un approccio ibrido, che è quello utilizzato in questo lavoro di tesi (esteso al caso di sistemi attivi complessi), consiste nel definire opzionalmente viewer, ruler e stato iniziale in fase di modellazione, fornendo la possibilità di ridefinirli nella successiva fase diagnostica.

Esempio 3.8. In tabella 3.4 è rappresentato un possibile *ruler* per il sistema attivo $\overline{A}$ dell'esempio 3.3. La protezione ha come etichette di guasto *fop*, che si verifica quando la transizione p_3 fallisce nel comandare l'apertura del breaker, e *fcp*, che si verifica quando la transizione p_4 fallisce nel comandare la chiusura del breaker. Il breaker è caratterizzato da due guasti: la label *nob* si riferisce alla mancata apertura del breaker nella transizione b_3, mentre *ncb* alla sua mancata chiusura nella transizione b_4.

Transizione	p_1	p_2	p_3	p_4	b_1	b_2	b_3	b_4	b_5	b_6
Label di guasto			*fop*	*fcp*			*nob*	*ncb*		

Tabella 3.4: *Ruler* per il sistema attivo $\overline{A}$

Definizione 3.8. Sia h una traiettoria di un sistema attivo A, e R un ruler per A. La diagnosi di h basata su R, scritta $h_{[R]}$, è l'insieme delle label di guasto:

$$h_{[R]} = \{f | t \in h, (t, f) \in R\}.$$

Esempio 3.9. Con riferimento all'esempio 3.3, si consideri la traiettoria:

$$h = [p_1, b_1, p_4, b_6]$$

In base al *ruler* $\overline{R}$ definito nella tabella 3.4, la diagnosi della traiettoria h è l'insieme $h_{[\overline{R}]} = \{fcp\}$, ottenuto dal guasto in corrispondenza della transizione p_4.

3.2.4 Problema diagnostico

Definizione 3.9. Sia A un sistema attivo, V un viewer di A, O una osservazione temporale per A relativa ad una traiettoria che parte dallo stato a_0, e R un ruler per A. La quadrupla

$$P(A) = (a_0, V, O, R)$$

è un problema di diagnosi per A.

Concettualmente, è possibile definire la soluzione di un problema diagnostico nel seguente modo.

Definizione 3.10. Sia $P(A) = (a_0, V, O, R)$ un problema di diagnosi per il sistema attivo A, e $Bsp(A)$ il behavior space con stato iniziale a_0. La soluzione del problema di diagnosi $\Delta(P(A))$, è l'insieme di diagnosi:

$$\Delta(P(A)) = \{\delta | \delta = h_{[R]}, h \in Bsp(A), h_{[V]} = O\}$$

In altre parole, la soluzione di un problema diagnostico è l'insieme costituito dagli insiemi di guasti rilevati lungo le traiettorie la cui traccia è consistente con l'osservazione temporale.

Esempio 3.10. Si consideri il *behavior space* del sistema attivo $\overline{A}$ in figura 3.4, l'osservazione dell'esempio 3.7 $\overline{O} = [awk, opb, ide]$, il viewer $\overline{V}$ e il ruler $\overline{R}$ riportati rispettivamente nelle tabelle 3.3 e 3.4. Le possibili traiettorie consistenti con l'osservazione sono:

- $h_1 = [p_1, b_1, p_2, b_4]$, a cui corrisponde la diagnosi $h_{1,[\overline{R}]} = \{ncb\}$;
- $h_2 = [p_1, b_1, p_4, b_6]$, a cui corrisponde la diagnosi $h_{2,[\overline{R}]} = \{fcp\}$.

La soluzione del problema diagnostico $P(\overline{A}) = (a_0, \overline{V}, \overline{O}, \overline{R})$ è data dalla composizione delle diagnosi di queste traiettorie:

$$\Delta(P(\overline{A}) = \{\{ncb\}, \{fcp\}\}.$$

Il significato delle due soluzioni trovate è il seguente: o il breaker b non riesce a chiudersi, oppure la protezione p invia il comando sbagliato al breaker.

3.3 Diagnosi greedy

La soluzione di un problema di diagnosi, presentata nella sezione precedente, è definita come l'insieme delle label di guasto che vengono generate dalle traiettorie consistenti con l'osservazione temporale. Questa definizione, tuttavia, non presenta una tecnica che permetta di trovare tale insieme nella pratica, anche perché si suppone che il *behavior space* non sia disponibile per la diagnosi. Scopo di questo paragrafo è quello di fornire, a fronte della definizione vista in precedenza, un metodo efficace che permetta di ottenere l'insieme delle diagnosi candidate in modo corretto e completo. Come prima osservazione, è essenziale notare che per generare le diagnosi candidate bisogna analizzare le traiettorie del sistema che sono consistenti con l'osservazione temporale. Per fare questo, bisogna poter generare un sotto-automa del behavior space contenente le sole traiettorie consistenti con l'osservazione temporale. La prima fase della diagnosi greedy consiste nella costruzione del behavior Bhv, il cui linguaggio coincide esattamente con il sottoinsieme del linguaggio del behavior space consistente con l'osservazione temporale. Successivamente è necessario associare ad ogni traiettoria del behavior la corrispondente diagnosi. Questa procedura potrebbe essere infinita, dato che un automa può avere ciclicità e quindi dare luogo ad un insieme infinito di traiettorie. Tuttavia, è bene notare come l'insieme delle diagnosi è sempre finito, poiché limitato superiormente dal numero di label di guasto specificate nel ruler: nello specifico, l'insieme delle diagnosi è un sottoinsieme di 2^{Φ}, cioè l'insieme potenza del dominio di label di guasto Φ. Nella pratica, l'insieme finito di traiettorie da considerare sono tutte quelle in cui i cicli sono attraversati al massimo una volta. Questo perché percorrere un ciclo più di una volta non cambia le diagnosi, dato che quelle ivi presenti sono già state collezionate durante l'iterazione precedente del ciclo. Il secondo punto della diagnosi greedy consiste nel generare l'insieme di diagnosi associate ad ogni nodo del behavior, ovvero praticare la cosiddetta decorazione del behavior. Una volta che il behavior è stato costruito e decorato, l'ultima fase della diagnosi greedy prevede la distillazione delle diagnosi, compiuta selezionando unicamente le diagnosi delle traiettorie consistenti con l'osservazione temporale. Per fare questo è importante distinguere, nella costruzione del behavior, tra stati finali e stati non finali. Uno stato del behavior è finale se e solo se tutte le traiettorie che giungono in quello stato hanno consumato interamente l'osservazione, cioè hanno percorso delle transizioni le cui label osservabili hanno prodotto una sequenza pari a quella dell'osservazione data.

3.3.1 Ricostruzione del behavior

Il primo passo della diagnosi greedy consiste nella ricostruzione del behavior.

Definizione 3.11. Sia $P(A) = (a_0, V, O, R)$ un problema diagnostico per un sistema attivo $A = (C, L, D)$, dove C è un insieme di n componenti, mentre L è un insieme di m link. Sia $I(O)$ la sequenza di indici $[0 \dots i_f]$ che rappresentano la consumazione dell'osservazione

di lunghezza i_f del sistema, con 0 che rappresenta l'indice corrispondente all'osservazione iniziale nulla. Il behavior spurio di $P(A)$ è l'automa deterministico

$$Bhv^s(P(A)) = (\Sigma, B^s, \tau^s, \beta_0, \beta_f)$$

dove:

1. Σ è l'alfabeto, costituito dall'unione delle transizioni dei componenti in C;
2. B^S è l'insieme di stati (S, Q, i), con $S = (s_1, \ldots, s_n)$ la n-pla di stati di tutti i componenti in C, $Q = (q_1, \ldots, q_m)$ la m-pla del contenuto di tutti i link in L, e i l'indice corrente della sequenza di osservazione;
3. $\beta_0 = (S_0, Q_0, 0)$ è lo stato iniziale, dove $a_0 = (S_0, Q_0)$ e 0 è l'indice dell'osservazione iniziale nulla;
4. β_f è l'insieme degli stati finali (S_f, Q_f, i_f), dove i_f è l'indice finale della sequenza di osservazione e $Q_f = \{\epsilon, \ldots, \epsilon\}$, ovvero tutti i link devono essere vuoti;
5. τ^s è la funzione di transizione, $\tau^s : B^s \times \Sigma \to B^s$, dove $(S, Q, i) \xrightarrow{t(c)} (S', Q', i') \in \tau^s$ e

   $$t(c) = s \xrightarrow{(e,x)|\{(e_1,y_1),\ldots,(e_p,y_p)\}} s'$$

   con e disponibile in corrispondenza del terminale di input x, oppure nullo se rappresenta un evento esterno o relativo ad un terminale scollegato in D_{on}. La transizione è effettuabile se e solo se:

   - Per ogni $j \in [1 \ldots n]$:

     $$s'_j = \begin{cases} s' & \text{se } c_j = c \\ s_j & \text{altrimenti} \end{cases}$$

     cioè per ogni transizione del behavior space cambia lo stato relativo al singolo componente coinvolto nella transizione;
   - Q' differisce da Q per le seguenti condizioni:
     - $Q(x) = e$, $Q'(x) = \epsilon$, cioè l'evento in input è consumato;
     - $\forall (e_j, y_j), j \in [1 \ldots p], Q(y_j) = \epsilon$, $Q'(y_j) = e_j$, cioè gli eventi di uscita sono inseriti nei terminali di input connessi ai terminali di output coinvolti nella transizione;
   - nel caso $t(c)$ sia osservabile, essa è presente nel viewer V associata alla label l e quest'ultima è una label contenuta nella sequenza di osservazione O in corrispondenza dell'indice $i + 1$, successivo a quello corrente. In questo caso l'indice dell'osservazione deve essere aggiornato ($i' = i + 1$). Nel caso invece $t(c)$ non sia osservabile, l'indice dell'osservazione rimane immutato ($i' = i$).

Il behavior $Bhv(P(A)) = (\Sigma, B, \tau, \beta_0, \beta_f)$ è ottenuto rimuovendo dal behavior spurio tutti gli stati e tutte le transizioni che non appartengono a nessun cammino tra lo stato iniziale e uno degli stati finali (operazione di trim). La definizione di behavior è ottenuta dalla definizione di behavior space, secondo le seguenti variazioni:

1. ogni stato del behavior include un campo aggiuntivo, l'indice i della sequenza di osservazione;
2. la funzione di transizione richiede un requisito aggiuntivo di consistenza con l'osservazione temporale, nel caso la transizione sia osservabile;
3. nel behavior gli stati finali, oltre ad avere tutti i link vuoti, indicano il raggiungimento della completa consumazione della sequenza osservata;
4. nel passaggio dal behavior spurio al behavior, sono mantenuti solo gli stati e le transizioni incluse in un cammino dallo stato iniziale ad uno stato finale.

Secondo le definizioni viste in precedenza, quindi, il linguaggio del behavior costituisce un sottoinsieme del linguaggio del behavior space, in quanto costituito unicamente da quelle traiettorie che sono consistenti con l'osservazione temporale. L'algoritmo 2 riporta lo pseudocodice della procedura di ricostruzione del behavior, in accordo con la definizione data.

Esempio 3.11. Con riferimento al problema diagnostico nell'esempio 3.10 in figura 3.5 è riportato il behavior del problema diagnostico $P(\overline{A})$. Le transizioni e gli stati tratteggiati costituiscono la parte spuria dell'automa, in quanto non appartengono a nessun cammino dallo stato iniziale a uno stato finale. Uno stato è finale se, oltre ad avere i link vuoti, ha un indice di osservazione pari a 3, che costituisce l'indice finale della sequenza di osservazione $\overline{O} = [awk, opb, ide]$ di lunghezza 3. Si noti la differenza di questo automa con il behavior space della figura 3.4. La diversa grandezza degli automi può essere evidenziata meglio in sistemi attivi con un numero maggiore di componenti e link, a causa della crescita esponenziale degli automi.

3.3.2 Decorazione del behavior

Il secondo passo della ricostruzione greedy consiste nella decorazione del behavior. Il linguaggio dell'automa ottenuto nel passo precedente è composto da stringhe, ognuna delle quali rappresenta una traiettoria consistente con l'osservazione temporale. Dato che il behavior racchiude tutte e sole le traiettorie la cui traccia rispetto al viewer è una traccia candidata dell'osservazione, la soluzione del problema diagnostico è l'insieme delle diagnosi relative a tali traiettorie. Dato che le traiettorie potrebbero avere dei cicli, dando luogo ad un numero infinito di percorrenze, l'idea è quella di marcare ogni stato del behavior

Algorithm 2 Algoritmo di ricostruzione del behavior

BuildBhv($P(A) = (a_0, V, O, R)$**)**

$\beta_0 \leftarrow (S_0, Q_0, 0)$, dove $a_0 = (S_0, Q_0)$
$B \leftarrow \{\beta_0\}$
$\tau \leftarrow \emptyset$
repeat
  Scegli uno stato non ancora marcato $\beta = (S, Q, i) \in B$
  for all $t(c)$ effettuabile in β **do**
    if $t(c)$ non è osservabile in V **then**
      $i' \leftarrow i$
    else if $t(c)$ è osservabile in V tramite la label l **and** $l \in O$ in corrispondenza dell'indice $i + 1$ **then**
      $i' \leftarrow i + 1$
    else
      continue
    end if
    $S' \leftarrow S$ e aggiorna lo stato del componente c di $t(c)$
    $Q' \leftarrow Q$ e aggiorna lo stato dei link coinvolti in $t(c)$
    $\beta' \leftarrow (S', Q', i')$
    if $\beta' \notin B$ **then**
      inserisci β' in B
    end if
    Inserisci la transizione $\beta \xrightarrow{t(c)} \beta'$ in τ
  end for
  Marca lo stato β
until tutti gli stati in B sono marcati
$\beta_f \leftarrow \{\beta_f | \beta_f \in B, \beta_f = (S, Q, i_f), i_f$ indice finale di $O\}$
Rimuovi da B e da τ, rispettivamente, tutti gli stati e le transizioni non inclusi in un cammino da β_0 ad uno stato finale

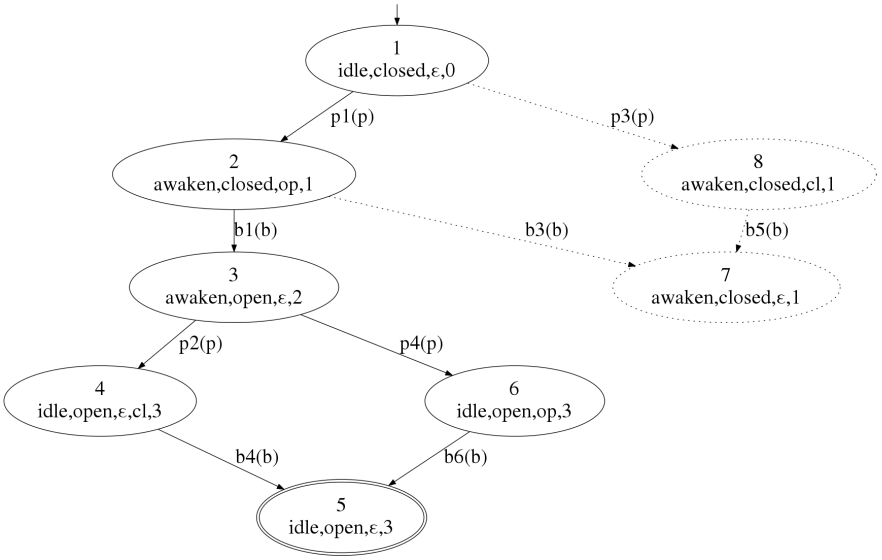


Figura 3.5: Behavior del problema diagnostico $P(\overline{A})$. Le transizioni e gli stati tratteggiati indicano la parte spuria dell'automa

con l'insieme di diagnosi relative a tutte le traiettorie che terminano in quello stato. Il fatto di avere ciclicità non è un problema, poiché la percorrenza di più cicli non accresce la diagnosi già collezionata in precedenza. Il processo di marcare ogni stato con le relative diagnosi è detto decorazione.

Definizione 3.12. Sia $Bhv(P(A))$ il behavior di un problema diagnostico $P(A)$. Il behavior decorato $Bhv^{\star}(P(A))$ è il DFA ottenuto dal behavior marcando ogni stato β con un insieme diagnostico $\Delta(\beta)$ basato sull'applicazione delle seguenti regole:

1. lo stato iniziale è decorato con l'insieme diagnostico vuoto $\delta(\beta_0) = \{\emptyset\}$;
2. per ogni transizione $\beta \xrightarrow{t(c)} \beta'$, per ogni $\delta \in \Delta(\beta)$, se $(t(c), f) \in R$ allora $\delta \cup \{f\} \in \Delta(\beta')$, altrimenti $\delta \in \Delta(\beta')$.

La prima regola è quella base, in cui si associa alla traiettoria nulla la diagnosi vuota. Se la decorazione di uno stato β include una diagnosi δ allora esiste almeno una traiettoria h, che termina in β, la cui diagnosi è δ. Conseguentemente, esiste una traiettoria $h \cup [t(c)]$ terminante in β' la cui diagnosi è δ se $(t(c), f) \notin R$, oppure la diagnosi è l'estensione di δ tramite la label di guasto f, associata alla transizione $t(c)$ nel ruler R. Quest'ultima regola è induttiva e deve essere applicata continuamente allo stato successivo della transizione fino a che la decorazione non aggiunge nuove diagnosi, raggiungendo una condizione di stabilità. L'algoritmo di decorazione, in forma ricorsiva, è fornito nell'algoritmo 3.

Esempio 3.12. Con riferimento al problema diagnostico dell'esempio 3.10, in figura 3.6 è rappresentato il relativo behavior decorato. Nella tabella 3.5 sono riportate le chiamate

Algorithm 3 Algoritmo di decorazione del behavior

Decorate($Bhv(P(A))$**)**
  Marca lo stato iniziale β_0 con $\{\emptyset\}$
  Marca tutti gli altri stati non iniziali con l'insieme vuoto $\emptyset$
  $Dec(\beta_0, \{\emptyset\})$

  Function $Dec(\beta, D)$
  for all $t(c), \beta \xrightarrow{t(c)} \beta'$ **do**
    $D^+ \leftarrow \emptyset$
    for all $\delta \in D$ **do**
      if $(t(c), f) \in R$ **then**
        $\delta' \leftarrow \delta \cup \{f\}$
      else
        $\delta' \leftarrow \delta$
      end if
      if $\delta' \notin \Delta(\beta')$ **then**
        inserire δ' in $\Delta(\beta')$ e in D^+
      end if
    end for
    if $D^+ \neq \emptyset$ **then**
      $Dec(\beta', D^+)$
    end if
  end for

effettuate dall'algoritmo alla procedura ricorsiva *Dec*. L'ordine di considerazione delle transizioni è, rispetto all'automa del behavior decorato, da sinistra a destra.

N	Chiamata	$\Delta(1)$	$\Delta(2)$	$\Delta(3)$	$\Delta(4)$	$\Delta(5)$	$\Delta(6)$
1	$(1, \{\emptyset\})$	$\{\emptyset\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$(2, \{\emptyset\})$	$\{\emptyset\}$	$\{\emptyset\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
3	$(3, \{\emptyset\})$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\emptyset$	$\emptyset$	$\emptyset$
4	$(4, \{\emptyset\})$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\emptyset$	$\emptyset$
5	$(5, \{\{ncb\}\})$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\{ncb\}\}$	$\emptyset$
6	$(6, \{\{fcp\}\})$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\{ncb\}\}$	$\{\{fcp\}\}$
7	$(5, \{\{fcp\}\})$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\emptyset\}$	$\{\{ncb\}, \{fcp\}\}$	$\{\{fcp\}\}$

Tabella 3.5: Applicazione dell'algoritmo di decorazione per ricavare $Bhv^\star(P(\overline{A}))$

3.3.3 Distillazione delle diagnosi

Una volta che il behavior è stato costruito e decorato, la soluzione del problema diagnostico $P(A)$ può essere determinata calcolando l'unione delle diagnosi associate a stati finali del behavior decorato:

$$\Delta(P(A))) = \{\delta | \delta \in \Delta(\beta_f), \beta_f \text{ finale in } Bhv^\star(P(A))\}.$$

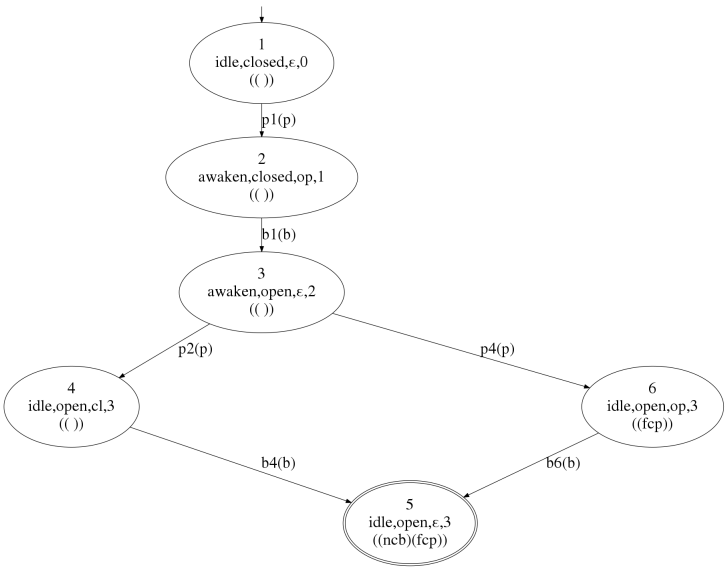


Figura 3.6: Behavior decorato del problema diagnostico $P(\overline{A})$

Esempio 3.13. Considerando il behavior decorato in figura 3.6, la distillazione delle diagnosi avviene considerando l'insieme delle diagnosi relative all'unico stato finale 5, cioè

$$\Delta(P(\overline{A})) = \{\{ncb\}, \{fcp\}\}$$

Si noti come la soluzione trovata è la stessa intuitivamente ricavata dalle definizioni nell'esempio 3.10.

Capitolo 4

Sistemi Attivi Complessi

In questo capitolo vengono presentate definizioni e modelli che descrivono i sistemi attivi complessi, visti come estensione dei sistemi attivi tradizionali descritti precedentemente. Un sistema attivo complesso è una gerarchia di sistemi attivi tra loro comunicanti. Questo modello si ispira a molti sistemi naturali, fisici, economici e sociali, dei quali è possibile fornire un modello stratificato, in base a diversi livelli di astrazione. Nei sistemi biologici, ad esempio, il comportamento può essere descritto a basso livello dalle interazioni fra le cellule, oppure ad un livello più alto tra i tessuti, o ancora in maniera più astratta studiando il funzionamento degli organi. Nell'ambito dei linguaggi di programmazione a oggetti, ad esempio, a basso livello vi sono le singole istruzioni, a livello più alto i metodi, ad un più alto livello ancora le classi e le interfacce. L'attributo "complesso" si riferisce, più che alla quantità di componenti presenti nel sistema (che comunque possono essere molti), all'organizzazione gerarchica e alla conseguente stratificazione del comportamento globale del sistema. Ogni livello della gerarchia di sistemi siffatti ha una evoluzione indipendente dagli altri, eccetto alcune informazioni che giungono dai livelli sottostanti. Ogni livello, in altre parole, ha un comportamento emergente che è impredicibile dalla semplice conoscenza del comportamento dei singoli componenti. Queste informazioni che giungono ai livelli superiori sono rese nel modello dei sistemi attivi complessi attraverso pattern event, che si verificano quando determinate configurazioni delle transizioni dei componenti sottostanti si verificano, soddisfacendo particolari espressioni regolari.

4.1 Definizioni

Un sistema attivo complesso (SAC) è una gerarchia di sistemi attivi. In un SAC, vi sono due tipi di comunicazione: tra componenti dello stesso sistema attivo e tra differenti sistemi attivi. I componenti di un sistema attivo interagiscono tra loro in base a determinati eventi che giungono o da componenti adiacenti (eventi interni), o da altri sistemi attivi collegati (pattern event) o dal mondo esterno al sistema (eventi esterni). Questi eventi innescano delle transizioni nei componenti, le quali a loro volta generano nuovi eventi, formando una sequenza di transizioni detta traiettoria del SAC. Analogamente ai sistemi attivi tradizionali, un sistema attivo complesso può essere in uno stato quiescente, nel quale nessun evento si verifica e conseguentemente non vengono innescate transizioni dei componenti. Il SAC diviene reattivo nel momento in cui si verifica un evento esterno che è consumato da un componente. In tale circostanza la transizione di un componente modifica lo stato del sistema, dove lo stato è caratterizzato dalla composizione di tutti gli stati dei componenti, dagli eventi non ancora consumati presenti nei terminali di input, e dagli stati relativi agli automi (pattern space) responsabili del matching delle espressioni regolari generanti i pattern event.

4.1.1 Nodi

Un sistema attivo complesso è composto da un insieme di sistemi attivi che interagiscono tra loro. Ogni singolo sistema attivo è detto nodo, in quanto costituisce un nodo nella gerarchia che descrive la topologia del sistema. Il modello di un nodo, a cui più istanze di nodi possono fare riferimento, possiede tutti gli attributi di un sistema attivo, con l'aggiunta di alcuni elementi:

- un insieme I di terminali di input, nei quali giungono pattern event generati da nodi del livello inferiore;
- un insieme O di terminali di output, nei quali vengono generati i pattern event del nodo corrente, che vengono inviati a uno o più nodi del livello superiore;
- un insieme P di dichiarazioni di pattern;
- un insieme L_{patt} di link uscenti da terminali di input del nodo ed entranti in terminali di input dei componenti interni al nodo stesso.

Quest'ultima caratteristica aggiuntiva consente l'invio dei pattern event in ingresso al nodo ai componenti particolari predisposti a gestire tali eventi.

4.1.2 Pattern

Una dichiarazione di pattern è una quadrupla (p, l, r, o) dove:

- p è il nome del pattern event;
- l è il linguaggio relativo al pattern;
- r è una espressione regolare;
- $o \in O$ è un terminale di output in corrispondenza del quale il pattern event viene generato.

Il linguaggio l del pattern può variare dal linguaggio minimo, cioè costituito dalle sole transizioni coinvolte nell'espressione regolare r, al linguaggio massimo corrispondente all'insieme di tutte le transizioni dei componenti appartenenti al nodo. L'appartenenza a diversi linguaggi conferisce una semantica differente a quei pattern le cui espressioni regolari utilizzano operatori insiemistici, come ad esempio il *not*. Si nota facilmente che il complemento di un elemento rispetto a linguaggi differenti ha come risultato un insieme di transizioni che varia in base all'insieme a cui appartiene.

4.1.3 Link tra nodi

Un sistema attivo complesso, oltre che da un insieme di nodi, è caratterizzato dalla topologia con cui i nodi sono collegati tra loro. Si assume che tale topologia generi un albero. I nodi sono tra loro connessi per mezzo di link. I link tra nodi sono definiti analogamente ai link tra componenti del singolo sistema attivo, con la differenza che i primi connettono tra loro sistemi attivi. In un SAC ogni link esce da un terminale di output o di un nodo n e entra in un terminale di input i di un nodo n'. I link sono caratterizzati da un modello, che costituisce un'astrazione del link specifico in esame.

Definizione 4.1. Un modello di un link è una quadrupla

$$M_{l_c} = (i, o, z, w)$$

dove i è il terminale di input, o il terminale di output, z la dimensione e w la politica di saturazione.

Un particolare link l_c è un'istanza di un modello siffatto, e consiste quindi in un canale di comunicazione unidirezionale fra due nodi del sistema n e n', dove un terminale di output o di n e un terminale di input i di n' coincidono rispettivamente con l'input e l'output del link l_c. La dimensione z rappresenta il numero massimo di eventi che possono essere accodati nel link. Indichiamo con $|l_c|$ la configurazione corrente del link. Se il numero di eventi attualmente memorizzati coincide con la dimensione, il link si dice essere saturo. Quando il link è saturo, la semantica legata al compimento delle transizioni è dettata dalla politica di saturazione w (si veda il paragrafo 3.1.2). Come per i link tra componenti, anche per quanto riguarda i link tra nodi, nel seguito della trattazione, si farà riferimento al caso in cui la dimensione sia unitaria e la politica di saturazione sia *wait*.

Esempio 4.1. Si consideri l'esempio di sistema attivo complesso riportato in figura 4.1. Esso consiste in una linea elettrica. Ogni capo della linea è protetto dai cortocircuiti per mezzo di due breaker. I componenti b sono connessi ognuno ad una protezione che rileva bassa tensione. Se uno dei due sensori rileva un imminente cortocircuito, viene comandata l'apertura del breaker ivi collegato. Se tutti e due i breaker si aprono, il cortocircuito si estingue e successivamente la chiusura dei breaker permette di ripristinare la linea. Tuttavia possono verificarsi dei guasti: o la protezione invia il comando sbagliato, oppure il breaker non si apre. Questi malfunzionamenti sono rilevati da un monitor (uno per ogni capo della linea), che in caso di non apertura del breaker ne comanda uno di emergenza, r_1 e r_2 rispettivamente per i monitor m_1 e m_2. Nel compiere questa azione, inoltre, un monitor informa l'altro in modo da effettuare la medesima operazione sull'altro breaker. Si assume che i breaker di emergenza, per ragioni di sicurezza, non vengano chiusi nuovamente. Il SAC dell'esempio possiede quattro nodi:

- l'organo di protezione W_1, che include la protezione p e il breaker b;
- l'organo di protezione W_2, che include la protezione p e il breaker b;
- l'apparato monitor M, che include i monitor m_1 e m_2, oltre ai breaker di emergenza r_1 e r_2;
- il nodo radice L, che contiene il componente l, il quale rappresenta l'intera linea elettrica.

I nodi W_1 e W_2 condividono lo stesso modello di nodo, hanno cioè gli stessi terminali, gli stessi componenti e le medesime dichiarazioni di pattern. Le frecce in figura all'interno del singolo nodo rappresentano link tra componenti o link tra terminali di input del nodo e terminali di input di componenti interni (per la gestione dei pattern event). I link tra nodi diversi rappresentano le interazioni fra sistemi attivi, che avvengono tramite la generazione di pattern event dai nodi inferiori verso i nodi superiori della gerarchia. I componenti di protezione condividono il modello in figura 3.1, mentre i breaker hanno il medesimo modello in figura 3.2. I monitor m_1 e m_2 hanno un comportamento descritto in figura 4.2, mentre la linea l è rappresentata in figura 4.3. Le transizioni relative alle protezioni e ai breaker sono fornite rispettivamente nelle tabelle 3.1 e 3.2, mentre le transizioni dei monitor e della linea sono nelle tabelle 4.1 e 4.2. Si noti che i pattern event sono indicati in grassetto. Nella tabella 4.3 sono riportate le dichiarazioni di pattern condivise dai nodi W_1 e W_2. Nella tabella 4.4 vi sono i pattern relativi all'apparato di monitor M. Si noti che non è specificato il terminale di output in corrispondenza del quale il pattern event viene generato, in quanto in questo esempio specifico si assume che ogni nodo abbia un unico terminale di uscita. Il nodo M possiede inoltre due terminale di input, sui quali riceve i pattern event dei nodi W_1 e W_2 rispettivamente. I linguaggi dei pattern sono così definiti:

- per W_1 e W_2 il linguaggio dei pattern è massimo e l'alfabeto coincide con l'intero insieme di transizioni dei componenti del nodo;
- per M il linguaggio dei pattern è minimo e l'alfabeto di ogni pattern coincide con le uniche transizioni coinvolte nella relativa espressione regolare (i tre pattern specificati hanno dunque tre linguaggi diversi).

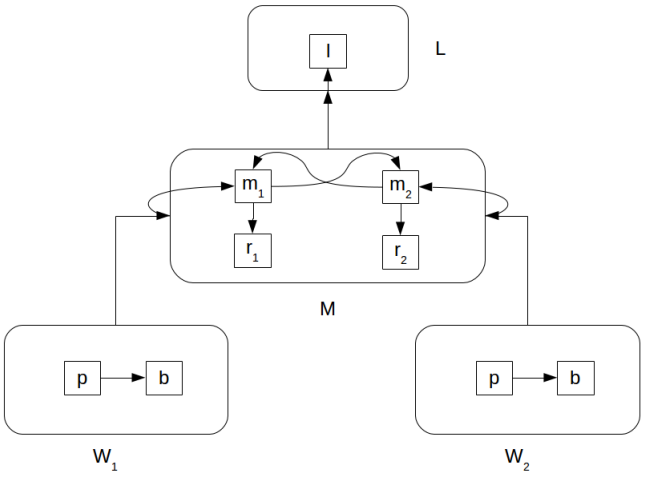
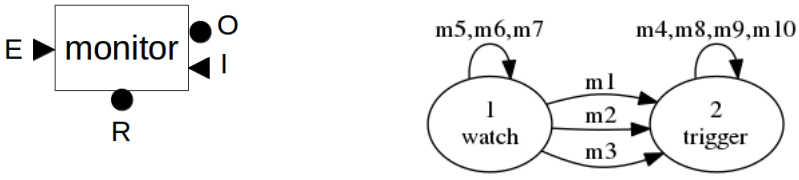
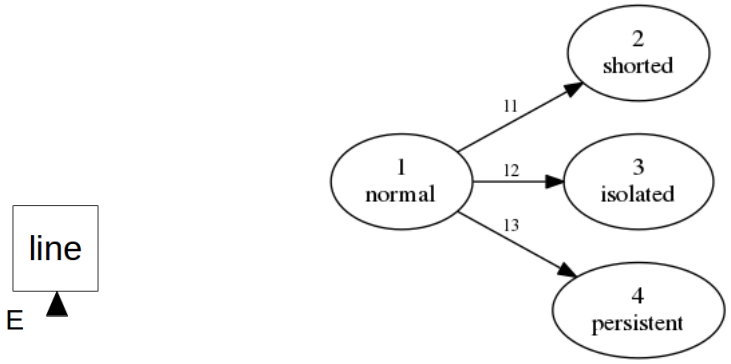


Figura 4.1: Esempio di riferimento



(a) topologia monitor (b) comportamento monitor

Figura 4.2: Modello dei componenti m_1 e m_2



(a) topologia linea (b) comportamento linea

Figura 4.3: Modello del componente l

Transizione	**Descrizione**
$m_1 = watch \xrightarrow{(\mathbf{nd},E)\Rightarrow\{(op,R),(rc,O)\}} trigger$	Rileva la mancata disconnessione del lato della linea e comanda l'apertura del breaker di emergenza, informando l'altro monitor
$m_2 = watch \xrightarrow{(\mathbf{nd},E)\Rightarrow(op,R)} trigger$	Rileva la mancata disconnessione del lato della linea e comanda l'apertura del breaker di emergenza
$m_3 = watch \xrightarrow{(rc,I)\Rightarrow(op,R)} trigger$	Riceve il messaggio dell'altro monitor e comanda l'apertura del proprio breaker
$m_4 = trigger \xrightarrow{(\mathbf{nd},E)} trigger$	Consuma il pattern event **nd**
$m_5 = watch \xrightarrow{(\mathbf{di},E)} watch$	Consuma il pattern event **di**
$m_6 = watch \xrightarrow{(\mathbf{co},E)} watch$	Consuma il pattern event **co**
$m_7 = watch \xrightarrow{(\mathbf{nc},E)} watch$	Consuma il pattern event **nc**
$m_8 = trigger \xrightarrow{(\mathbf{di},E)} trigger$	Consuma il pattern event **di**
$m_9 = trigger \xrightarrow{(\mathbf{co},E)} trigger$	Consuma il pattern event **co**
$m_{10} = trigger \xrightarrow{(\mathbf{nc},E)} trigger$	Consuma il pattern event **nc**

Tabella 4.1: Transizioni relative al monitor

Transizione	**Descrizione**
$l_1 = normal \xrightarrow{(ni,E)} shorted$	Consuma il pattern event **ni**
$l_2 = normal \xrightarrow{(nr,E)} isolated$	Consuma il pattern event **nr**
$l_3 = normal \xrightarrow{(ps,E)} persistent$	Consuma il pattern event **ps**

Tabella 4.2: Transizioni relative alla linea

Pattern event	**Espressione regolare**
di	$b_1(b)$
co	$b_2(b)$
nd	$p_3(p)\|p_1(p)b_3(b)$
nc	$p_4(p)\|p_2(p)b_4(b)$

Tabella 4.3: Dichiarazioni di pattern dei nodi W_1 e W_2

Pattern event	**Espressione regolare**
nr	$m_7(m_1)\|m_{10}(m_1)\|b_1(r_1)\|m_7(m_2)\|m_{10}(m_2)\|b_1(r_2)$
ni	$(m_1(m_1)\|m_2(m_1)\|m_4(m_1))b_3(r_1)\|b_3(r_1)m_4(m_1)\|$ $(m_1(m_2)\|m_2(m_2)\|m_4(m_2))b_3(r_2)\|b_3(r_2)m_4(m_2)$
ps	$(m_6(m_1)m_6(m_2)\|m_6(m_2)m_6(m_1))$ $(m_5(m_1)m_5(m_2)\|m_5(m_2)m_5(m_1))$

Tabella 4.4: Dichiarazioni di pattern del nodo M

4.1.4 Pattern space

In modo da individuare pattern event, deve essere mantenuto lo stato relativo al matching dei pattern coinvolti. A tal fine, per ogni nodo, è necessario generare degli automi, chiamati *pattern space*, secondo i seguenti passi:

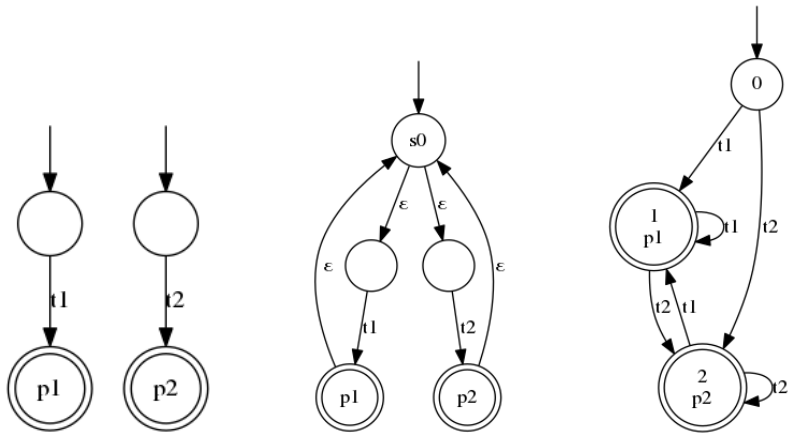
- per ogni pattern (p, l, r, o), viene generato un automa deterministico equivalente all'espressione regolare r, nel quale ogni stato finale viene marcato dal nome p del pattern event;
- per ogni linguaggio utilizzato nella definizione dei pattern, vengono identificati i pattern che utilizzano tale linguaggio, corrispondenti agli automi $[A_1, \ldots, A_k]$, e viene generato un pattern space nel seguente modo:
  1. un automa non deterministico N è creato generando il suo stato iniziale s_0 e una ϵ- transizione da s_0 a ogni stato iniziale di A_i, con $i \in [1 \ldots k]$;
  2. in modo da mantenere il matching di stringhe sovrapposte, viene aggiunta una ϵ-transizione da ogni stato non iniziale a s_0;
  3. l'automa N viene determinizzato nel risultante pattern space P, dove ogni stato finale s_f è marcato dall'unione $\mathbf{p}$ dei pattern event associati agli stati in s_f che sono finali nei corrispondenti automi di pattern generati inizialmente. Ogni stato s dell'automa deterministico P è infatti identificato da un sottoinsieme di stati dell'automa non deterministico equivalente N;
  4. l'automa P viene minimizzato, raggruppando tra loro gli stati equivalenti che sono provvisti dello stesso tag (altrimenti l'informazione relativa ai pattern event generati potrebbe andare persa) [1].

Esempio 4.2. Viene di seguito presentata la costruzione del pattern space relativo a due pattern(figura4.4). Siano

- $p_1 = t_1$
- $p_2 = t_2$

due dichiarazioni di pattern, alle quali corrispondono gli automi A_1 e A_2. L'automa non deterministico N è ottenuto inserendo uno stato iniziale fittizio s_0, collegando quest'ultimo agli stati iniziali dei due DFA per mezzo di ϵ-transizioni e collegando ogni stato non iniziale dei due DFA allo stato s_0 tramite ϵ-transizioni. La determinizzazione di tale automa risultante è il pattern space P per i pattern p_1 e p_2. Si noti che non è necessaria la minimizzazione.

[1] La minimizzazione del pattern space richiede una lieve modifica all'algoritmo di minimizzazione. Nella partizione iniziale, oltre a distinguere tra stati finali e stati non finali, si effettua una partizione in base al contenuto del tag degli stati finali.



(a) Automi A_1 e A_2 (b) NFA N (c) Pattern space

Figura 4.4: Esempio di costruzione del pattern space

Esempio 4.3. Specificato in tabella 4.5 vi è il pattern space relativo ai nodi W_1 e W_2. Gli stati finali hanno come etichetta i nomi dei relativi pattern riconosciuti. Per quanto riguarda il nodo M, dato che i pattern sono caratterizzati da linguaggi differenti, vengono invece generati tre pattern space distinti.

	b_1	b_2	b_3	b_4	p_1	p_2	p_3	p_4	
0	1	2			3	4	5	6	
1	1	2			3	4	5	6	**di**
2	1	2			3	4	5	6	**co**
3	1	2	5		3	4	5	6	
4	1	2		6	3	4	5	6	
5	1	2			3	4	5	6	**nd**
6	1	2			3	4	5	6	**nc**

Tabella 4.5: Pattern space per i nodi W_1 e W_2

4.1.5 Sistema attivo complesso

Un sistema attivo complesso è una rete di nodi interconnessi per mezzo di link. Ogni nodo ed ogni link del sistema sono caratterizzati da un modello e quindi, in generale, più elementi potrebbero avere il medesimo modello. Si assume che un solo link possa uscire da un terminale di output di un nodo e al massimo un link possa entrare in un terminale di input di un nodo.

Definizione 4.2. Un sistema attivo complesso è una coppia

$$C = (N, L_c)$$

dove N è l'insieme di nodi, mentre L_c è l'insieme dei link tra terminali di nodi appartenenti a N.

4.1.6 Traiettoria

Un SAC è una macchina che si trova in uno stato quiescente o in uno stato reattivo. Se si trova nello stato quiescente, i componenti non effettuano alcuna transizione, dal momento che nessun evento è disponibile nei terminali di ingresso. In corrispondenza del verificarsi di un determinato evento proveniente dal mondo esterno, il sistema evolve nella sua fase reattiva, fino a quando non raggiunge nuovamente uno stato di quiete. Dato che il comportamento del sistema è asincrono (non dipende dal tempo), la reazione, detta traiettoria (o storia), consiste in una sequenza di transizioni compiute da componenti presenti nel sistema. Ogni transizione di un componente porta il sistema in un nuovo stato, dove uno stato è identificato dallo stato corrente di ogni componente di ogni nodo, dalle configurazioni attuali dei link dei nodi e dallo stato attuale dei pattern space di tutti i nodi. Quando uno stato del sistema è caratterizzato da uno stato finale di un pattern space, il relativo pattern event viene inviato al terminale corrispondente. In altre parole, uno stato del sistema attivo complesso è una tripla (S, Q, P), dove S è la n-pla degli stati dei componenti dei nodi, Q è la m-pla dei contenuti dei link tra nodi, mentre P è la p-pla di pattern space di tutti i nodi. Quindi una transizione del sistema può essere scritta come:

$$T = (S, Q, P) \xrightarrow{t(c(n))} (S', Q', P'),$$

dove $t(c(n))$ identifica univocamente la transizione t appartenente al modello del componente c nel nodo n. Assumendo che $a_0 = (S_0, Q_0, P_0)$ sia lo stato iniziale del sistema, la traiettoria ottenuta partendo da a_0 è la sequenza di stati del sistema determinata dall'occorrenza di transizioni $t_1, \ldots, t_k$ attuabili da parte dei singoli componenti dei nodi:

$$h = a_0 \xrightarrow{t_1(c_1(n_1))} a1 \xrightarrow{t_2(c_2(n_2))} a2 \ldots \xrightarrow{t_k(c_k(n_k))} a_k.$$

Ogni stato (non iniziale) dipende quindi dallo stato precedente e dalla particolare transizione che lo porta allo stato corrente, ovvero la coppia $(a_{i-1}, t_i(c_i(n_i)))$. Assumendo che si parta dallo stato iniziale a_0 questo ci permette di identificare una traiettoria per mezzo delle sole transizioni dei componenti dei nodi:

$$h = [t_1(c_1(n_1)), t_2(c_2(n_2)), \ldots, t_k(c_k(n_k))].$$

4.1.7 Behavior space

Dato un SAC C ed il suo stato iniziale, possono esservi infinite traiettorie, descritte da un automa. Quest'ultimo è detto behavior space e ha come alfabeto l'intero insieme delle transizioni dei singoli componenti di ogni nodo. Il linguaggio del behavior space $Bsp(C)$

con stato iniziale a_0 coincide con l'insieme di tutte le possibili traiettorie del sistema C partendo dallo stato a_0.

Definizione 4.3. Sia $C = (N, L_c)$ un sistema attivo complesso, dove N è l'insieme di nodi, mentre L_c è l'insieme di link tra i nodi. Il behavior space di C è il DFA

$$Bsp(C) = (\Sigma, \alpha, \tau, a_0, \alpha_f)$$

dove:

1. Σ è l'alfabeto, dato dall'unione delle transizioni dei componenti di tutti i nodi in N;
2. α è l'insieme degli stati (S, Q, P), con $S = (s_1, \ldots, s_w)$ una w-pla di stati dei componenti in ogni nodo in N, $Q = (q_1, \ldots, q_m)$ una m-pla contenente gli eventi nei terminali di input di tutti i componenti dei nodi in N, e $P = (P_1, \ldots, P_p)$ una p-pla di stati dei pattern space di tutti i nodi in N;
3. $a_0 = (S_0, Q_0, P_0)$ è lo stato iniziale, con $Q_0 = (\epsilon, \ldots, \epsilon)$ e $P_0 = (P_{10}, \ldots, P_{p0})$ la p-pla degli stati iniziali dei pattern space;
4. α_f è l'insieme degli stati finali quiescenti, ovvero stati in cui $Q = (\epsilon \ldots \epsilon)$;
5. τ è la funzione di transizione deterministica, $\tau : \alpha \times \Sigma \to \alpha$, tale che

   $$(S, Q, P) \xrightarrow{t(c(n))} (S', Q', P') \in \tau,$$

   dove $S = (s_1, \ldots, s_w)$, $Q = (q_1, \ldots, q_m)$, $P = (P_1, \ldots, P_p)$, $S' = (s'_1, \ldots, s'_w)$, $Q' = (q'_1, \ldots, q'_m)$, $P' = (P'_1, \ldots, P'_p)$ e

   $$t(c(n)) = s \xrightarrow{(e,x)|\{(e_1,y_1),\ldots,(e_p,y_p)\}} s'$$

   se e solo se:

   - $x = In$, cioè l'evento è disponibile sul terminale di input virtuale sensibile agli eventi esterni al sistema di un componente, oppure e è contenuto nel terminale di input x di un componente;
   - Per ogni $i \in [1 \ldots w]$, abbiamo

     $$s'_i = \begin{cases} s' & \text{se } c_i = c \\ s_i & \text{altrimenti} \end{cases}$$

     cioè per ogni transizione del behavior space cambia lo stato relativo al singolo componente coinvolto nella transizione;
   - P' differisce da P solo negli elementi $v \in [j, \ldots, k]$ corrispondenti a pattern space del nodo n coinvolto nella transizione, secondo il seguente comportamento:

$$P'(P_v) = \begin{cases} \overline{P} & \text{se esite } P(P_v) \xrightarrow{t(c(n))} \overline{P} \\ P_{v0} & \text{altrimenti} \end{cases};$$

- Q' differisce da Q per le seguenti condizioni:
  - $Q(x) = e$, $Q'(x) = \epsilon$, cioè l'evento in input è consumato;
  - $\forall (e_j, y_j), j \in [1 \dots p], Q(y_j) = \epsilon$, $Q'(y_j) = e_j$, cioè gli eventi di uscita sono inseriti nei terminali di input connessi ai terminali di output coinvolti nella transizione;
  - se $P'(P_v) \neq P(P_v), \forall v \in [j, \dots, k]$ pattern space relativi al nodo corrente n e $P'(P_v)$ è finale e marcato dall'insieme di eventi $\mathbf{p}$, allora ogni $p_i \in \mathbf{p}$ ha un terminale destinazione $o_i \in O$ del nodo n. Ogni terminale di output di n è collegato ad uno o più terminali di input di un nodo superiore n', che a sua volta è collegato ad uno o più terminali di input i_z di componenti che gestiscono il patter event. Per ogni terminale i_z, il relativo contenuto passa da ϵ al pattern event p_i.

Esempio 4.4. Si consideri l'esempio di riferimento 4.1. Si può facilmente notare come una eventuale generazione del behavior space sia impraticabile. La dimensione dell'automa, infatti, è superiormente limitata da tutte le possibili combinazioni di stati dei nodi, contenuto dei terminali di input e stato dei pattern space dell'intero sistema. Dato che il sistema è composto da nove componenti, nove terminali di input e quattro pattern space, considerando anche solo due possibili configurazioni per ogni elemento (in realtà sono molte di più, ad esempio i pattern space dei nodi W_1 e W_2 hanno sette stati), gli stati possibili sarebbero 2^{22}, cioè più di quattro milioni. I risultati sperimentali hanno evidenziato una saturazione della memoria (4GB di RAM) per automi di dimensione inferiore ai due milioni.

4.2 Problema diagnostico

Analogamente al caso di sistemi attivi tradizionali, il problema diagnostico dei sistemi attivi complessi necessita di informazioni riguardanti l'osservabilità e i guasti delle transizioni dei componenti di ogni nodo. Si suppone sia disponibile, terminata la reazione, l'osservazione riferita ad ogni singolo sistema attivo che compone il sistema. Si ricorda che nell'ambito di questo lavoro ci si focalizza sulla cosiddetta diagnosi a posteriori, ovvero a seguito di una traiettoria completa del sistema, che parte da uno stato iniziale noto quiescente e termina in uno stato finale (sconosciuto a priori) anch'esso quiescente.

4.2.1 Viewer

Per ogni nodo un viewer locale specifica quali transizioni sono osservabili, associando ad esse una label. Il viewer globale del sistema C è la composizione dei viewer locali dei sistemi attivi appartenenti ad N, $V = (V_1, \ldots, V_n)$. Ogni viewer locale è definito come una coppia (t, l) che associa alla transizione t la label l.

Esempio 4.5. Con riferimento all'esempio 4.1, il viewer locale del nodo W_1 è così specificato:

- $p_1(p) \to awk1$, $p_2(p) \to ide1$, $p_3(p) \to awk1$, $p_4(p) \to awk1$;
- $b_1(b) \to opb1$, $b_2(b) \to clb1$, $b_5(b) \to opb1$, $b_6(b) \to opb1$.

Il viewer locale del nodo W_2 è analogo, eccetto il fatto che le relative label di osservazione terminano con 2 anziché 1. Il viewer del nodo M è riportato di seguito.

- $m_1(m_1) \to trg1, m_2(m_1) \to trg1, m_3(m_1) \to trg1$;
- $m_1(m_2) \to trg2, m_2(m_2) \to trg2, m_3(m_2) \to trg2$;
- $b_1(r_1) \to opr1, b_2(r_1) \to clr1$;
- $b_1(r_2) \to opr2, b_2(r_2) \to clr2$.

Le transizioni non appartenenti ai viewer locali non sono osservabili. Si noti che il viewer V_L, non essendo specificato, esprime il fatto che tutte le transizioni al suo interno non sono osservabili. Il viewer globale del sistema è dato da $\overline{V} = (V_{W_1}, V_{W_2}, V_M)$.

4.2.2 Osservazione temporale

L'osservazione O di un SAC è una n-pla di osservazioni locali dei singoli nodi del sistema $O = (O_1, \ldots, O_n)$. Ogni osservazione locale è una sequenza di label osservabili.

Esempio 4.6. Si consideri l'esempio di riferimento 4.1 e il viewer specificato nell'esempio 4.5. Un'osservazione per il sistema è:

$$O_{W_1} = [awk1],\ O_{W_2} = [awk2, opb2],\ O_M = [trg1, opr1].$$

L'osservazione globale è data da $\overline{O} = (O_{W_1}, O_{W_2}, O_M)$, l'osservazione locale del nodo L è nulla (non essendovi transizioni osservabili).

4.2.3 Ruler

L'informazione riguardante le transizioni di guasto è fornita, per ogni nodo del SAC, dal ruler locale. Il ruler globale del sistema C è una n-pla di ruler locali, $R = (R_1, \ldots, R_n)$. Ogni ruler locale è definito come una coppia (t, f) che associa alla transizione t la label di guasto f.

Esempio 4.7. Con riferimento all'esempio 4.1, il ruler locale R_{W_1} del nodo W_1 è così specificato:

- $p_3(p) \rightarrow fop1$, $p_4(p) \rightarrow fcp1$;
- $b_3(b) \rightarrow fob1$, $b_4(b) \rightarrow fcb1$.

Il ruler locale R_{W_2} del nodo W_2 è analogo, eccetto il fatto che le relative label di guasto terminano con 2 anziché 1. Il ruler R_M del nodo M è riportato di seguito:

- $m_2(m_1) \rightarrow fm1$;
- $m_2(m_2) \rightarrow fm2$;
- $b_3(r_1) \rightarrow for1, b_4(r_1) \rightarrow fcr1$;
- $b_3(r_2) \rightarrow for2, b_4(r_2) \rightarrow fcr2$.

Il ruler R_L del nodo L è invece:

- $l_1(l) \rightarrow fls$, $l_2(l) \rightarrow fli$, $l_3(l) \rightarrow flp$.

Il ruler globale del sistema è dato da $\overline{R} = (R_{W_1}, R_{W_2}, R_M, R_L)$.

4.2.4 Problema diagnostico

Diagnosticare il comportamento di un SAC equivale a trovare i guasti nella sua traiettoria. Quest'ultima, a causa della ridotta osservabilità del sistema, è percepita solo attraverso una traccia, a cui potrebbero però essere associate più traiettorie, anche infinite. Per questo motivo il risultato della diagnosi è un insieme diagnosi candidate, con ogni candidato che corrisponde ad un sottoinsieme di possibili traiettorie.

Definizione 4.4. Un problema diagnostico per un SAC C è una quadrupla

$$P(C) = (a_0, V, O, R),$$

dove:

- a_0 è lo stato iniziale del sistema C;
- V è il viewer globale di C, composto dall'insieme dei viewer locali $V_1, \ldots, V_n$;
- O è l'osservazione globale, formata dall'insieme di osservazioni locali $O_1, \ldots, O_n$;
- R è il ruler globale di C, contenente l'insieme dei ruler locali $R_1, \ldots, R_n$.

Esempio 4.8. Con riferimento all'esempio 4.1, al viewer $\overline{V}$ 4.5, all'osservazione $\overline{O}$ 4.6 e al ruler $\overline{R}$ 4.7, si consideri lo stato iniziale a_0 in cui tutti i breaker sono chiusi, i monitor sono nello stato *watch* è la linea in quello *normal*. Inoltre i terminali di input dello stato iniziale sono tutti vuoti (stato quiescente). Il problema diagnostico specifico degli esempi forniti è dato da $P(\overline{C}) = (a_0, \overline{V}, \overline{O}, \overline{R})$.

Concettualmente, è possibile definire la soluzione di un problema diagnostico nel seguente modo.

Definizione 4.5. Sia $P(C) = (a_0, V, O, R)$ un problema diagnostico per il SAC C, e sia $Bsp(C)$ il behavior space con stato iniziale a_0. La soluzione del problema diagnostico $\Delta(P(C))$, è l'insieme diagnostico:

$$\Delta(P(C)) = \{\delta | \delta = h_{[R]}, h \in Bsp(C), h_{[V]} = O\}$$

In altre parole, la soluzione di un problema diagnostico è l'insieme costituito dagli insiemi di guasti rilevati lungo le traiettorie la cui traccia è consistente con l'osservazione temporale.

4.3 Diagnosi greedy

Analogamente al metodo diagnostico greedy visto per i sistemi attivi tradizionali, anche per i SAC è possibile adottare un procedimento analogo. Il behavior dato dalla ricostruzione greedy racchiude tutte le possibili traiettorie del sistema che sono consistenti con le sequenze di osservazioni temporali locali dei vari nodi. Lo stato del behavior di un SAC differisce da quello visto nel capitolo precedente per il fatto di avere come informazione aggiuntiva la sequenza degli stati correnti dei pattern space. La consumazione dell'osservazione, inoltre, è espressa non più tramite un solo indice ma per mezzo di una n-pla di indici, ognuno relativo alla sequenza di osservazione locale di un nodo del sistema.

4.3.1 Ricostruzione del behavior

Il primo passo della diagnosi greedy consiste nella ricostruzione del behavior, ovvero del DFA contenente tutte le traiettorie del sistema consistenti con le osservazioni temporali.

Definizione 4.6. Sia $P(C) = (a_0, V, O, R)$ un problema diagnostico per un sistema attivo complesso $C = (N, L_c)$, dove N è un insieme di nodi, mentre L_c è un insieme di link tra i nodi. Sia I l'insieme delle sequenze di indici $[0 \dots i_{n_f}]$ che rappresentano le consumazioni delle osservazioni di lunghezza i_{n_f} di ogni nodo n del sistema, con 0 corrispondente all'indice relativo all'osservazione iniziale nulla. Il behavior spurio di $P(C)$ è l'automa deterministico

$$Bhv^s(P(C)) = (\Sigma, B^s, \tau^s, \beta_0, \beta_f)$$

dove:

- Σ è l'alfabeto, costituito dall'unione delle transizioni dei componenti di ogni nodo in N;
- B^S è l'insieme di stati (S, Q, P, I), con $S = (s_1, \dots, s_n)$ la n-pla di stati di tutti i componenti di tutti i nodi in N, $Q = (q_1, \dots, q_m)$ la m-pla del contenuto di tutti i terminali di input, $P = (P_1, \dots, P_p)$ gli stati relativi ai pattern space di tutti i nodi, e $I = (i_1, \dots, i_n)$ gli indici correnti relativi alle osservazioni locali dei nodi;
- $\beta_0 = (S_0, Q_0, P_0, I_0)$ è lo stato iniziale, dove $a_0 = (S_0, Q_0)$, $P_0 = (P_{10}, \dots, P_{p0})$ cioè la p-pla degli stati iniziali di tutti i pattern space, $I_0 = (0, \dots, 0)$ gli indici iniziali nulli degli stati delle osservazioni locali;
- β_f è l'insieme degli stati finali (S, Q, P, I_f), dove I_f è la n-pla di indici finali delle sequenze di osservazione locali;
- τ^s è la funzione di transizione, $\tau^s : B^s \times \Sigma \to B^s$, dove $(S, Q, P, I) \xrightarrow{t(c(n))} (S', Q', P', I') \in \tau^s$ e

$$t(c(n)) = s \xrightarrow{(e,x)|\{(e_1,y_1),\ldots,(e_p,y_p)\}} s'$$

con e disponibile in corrispondenza del terminale di input x, oppure nullo se rappresenta un evento esterno. Per ogni $j \in [1 \ldots n]$:

$$s'_j = \begin{cases} s' & \text{se } c_j = c \\ s_j & \text{altrimenti} \end{cases}$$

cioè per ogni transizione del behavior cambia lo stato relativo al singolo componente coinvolto nella transizione. L'inserimento degli eventi in uscita è identico a quanto avviene per il behavior space. Nel caso $t(c)$ sia osservabile, essa è presente nel viewer V associata alla label l e quest'ultima è una label contenuta nella sequenza di osservazione del nodo corrente I_n in corrispondenza dell'indice $i+1$, successivo a quello corrente. In questo caso l'indice dell'osservazione deve essere aggiornato ($i' = i+1$). Nel caso invece $t(c)$ non sia osservabile, l'indice dell'osservazione rimane immutato ($i' = i$).

Il behavior $Bhv(P(C)) = (\Sigma, B, \tau, \beta_0, \beta_f)$ è ottenuto rimuovendo dal behavior spurio tutti gli stati e tutte le transizioni che non appartengono a nessun cammino tra lo stato iniziale e uno degli stati finali. La definizione di behavior è ottenuta dalla definizione di behavior space, secondo le seguenti variazioni:

1. ogni stato del behavior include un campo aggiuntivo, la n-pla di indici I delle sequenze di osservazioni locali di ogni nodo;
2. la funzione di transizione richiede un requisito aggiuntivo di consistenza con le osservazioni temporali, nel caso la transizione sia osservabile;
3. nel behavior gli stati finali, oltre ad avere tutti i link vuoti, indicano il raggiungimento della completa consumazione della sequenza osservata per ogni nodo;
4. nel passaggio dal behavior spurio al behavior, sono mantenuti solo gli stati e le transizioni incluse in un cammino dallo stato iniziale ad uno stato finale.

Secondo le definizioni viste in precedenza, quindi, il linguaggio del behavior costituisce un sottoinsieme del linguaggio del behavior space, in quanto costituito unicamente da quelle traiettorie che sono consistenti con l'osservazione temporale.

Esempio 4.9. Si consideri il problema diagnostico visto nell'esempio 4.8. Il behavior spurio è composto da 168 stati e 274 transizioni. Rimuovendo gli stati e le transizioni spurie, l'automa risultante è rappresentato in figura 4.5.

4.3.2 Decorazione

L'algoritmo di decorazione è esattamente quello visto nel capitolo precedente (3). Si noti che in questo caso l'associazione delle etichette di guasto alle transizioni avviene considerando il ruler locale del nodo n il cui componente è coinvolto nella transizione $t(c(n))$.

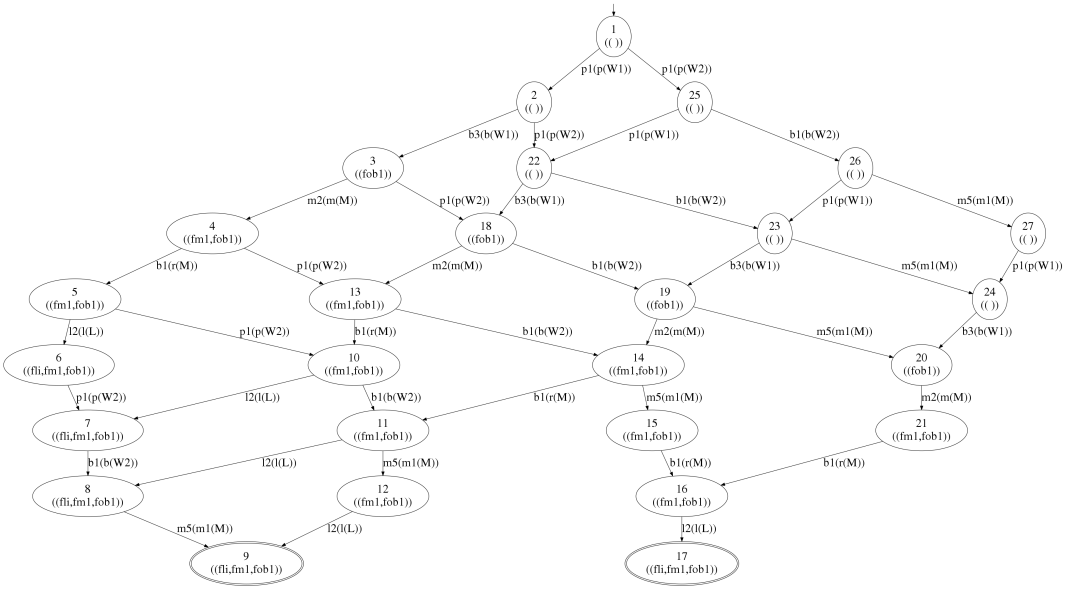


Figura 4.5: Behavior del problema di riferimento

4.3.3 Distillazione delle diagnosi

Una volta decorato il behavior, è possibile generare l'insieme delle soluzioni candidate esattamente come analizzato nel capitolo precedente.

Esempio 4.10. Si consideri il behavior decorato in figura 4.5. La soluzione del problema diagnostico è data dall'unione delle diagnosi degli stati finali 9 e 17:

$$\Delta(P(\overline{C})) = \{\{fli, fm1, fob1\}\}.$$

L'interpretazione di questo risultato è dovuto alla seguente dinamica:

- il breaker b del nodo W_1 non si apre;
- il monitor m_1 del nodo M, pur riuscendo a comandare l'apertura del proprio breaker di emergenza, fallisce nell'inviare il messaggio all'altro monitor;
- la linea l del nodo L, a fronte di un intervento di emergenza operato da un monitor, si trova isolata non potendo effettuare il ripristino.

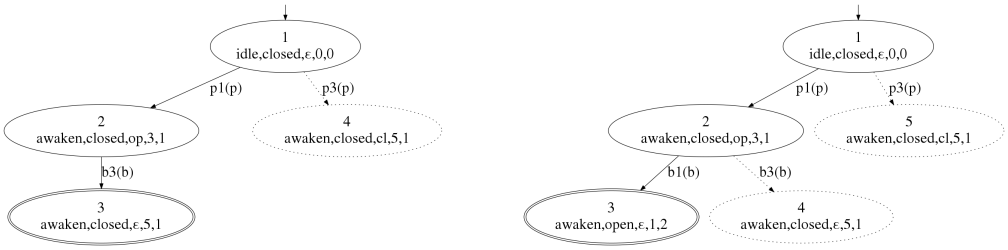
4.4 Diagnosi lazy

Il metodo di ricostruzione greedy, analizzato nei paragrafi precedenti, è inefficiente in quanto la complessità della generazione del behavior dell'intero SAC è in generale esponenziale nel numero di componenti totali, nel numero di link, nel numero dei pattern space e nel numero dei nodi (ad ognuno dei quali corrisponde una sequenza di osservazione). Sebbene la ricostruzione permetta di non generare il comportamento completo del sistema (behavior space), la ricostruzione del behavior del SAC è comunque costosa, sia in termini di tempo di esecuzione sia soprattutto di memoria. Per questo motivo, nel corso di questo lavoro di tesi, è stato sviluppato un algoritmo lazy che ricostruisce il comportamento del singolo nodo, unitamente alle informazioni necessarie scaturite dai pattern event, in modo da non dover generare il behavior del sistema nella sua interezza. Per fare questo si è seguita la topologia del sistema: trattandosi di una gerarchia, l'algoritmo opera in maniera bottom-up partendo dai nodi foglia. Il comportamento di questi ultimi, infatti, non è vincolato dal comportamento di nessun altro nodo, quindi la ricostruzione del loro behavior è esattamente quella vista nel paragrafo precedente, eccetto il fatto che si considera il singolo nodo e non l'intero sistema. Per poter ricostruire i comportamenti dei nodi nel livello superiore della gerarchia, invece, è necessario considerare i pattern event che vengono inviati dal livello base. Per fare questo viene creata, per ogni nodo foglia, una interfaccia ottenuta togliendo dal linguaggio del behavior locale tutte quelle transizioni che non portano alla generazione di un pattern event in un terminale di uscita del nodo. I comportamenti del livello superiore, quindi, sono ricostruiti tenendo conto delle interfacce dei nodi sottostanti da cui essi dipendono. Ogni passaggio di stato dell'interfaccia sottostante corrisponde alla generazione di un pattern event che può essere consumato dai componenti a livello superiore. Il procedimento viene reiterato per ogni strato della gerarchia del sistema. Una volta raggiunta la radice dell'albero, non è più necessario generare l'interfaccia di tale nodo (dato che nessun pattern event viene gestito da un livello superiore), quindi l'unico behavior corrispondente racchiude in sé tutte le diagnosi dell'intero sistema in forma sintetica. La sua decorazione permette di ricavare l'insieme di diagnosi candidate del SAC.

4.4.1 Costruzione del Behavior non vincolato

La ricostruzione dei behavior dei nodi foglia del SAC è un caso particolare della ricostruzione greedy. Le variazioni consistono nel considerare solo i pattern space relativi al nodo locale e unicamente l'osservazione locale, la cui consumazione viene monitorata per mezzo di un unico indice di sequenza. Analogamente il viewer può essere ristretto al viewer locale del nodo corrente, in quanto si considerano solo transizioni di componenti appartenenti al nodo in esame.

Esempio 4.11. In figura 4.6 sono riportati i behavior dei nodi foglia del problema diagnostico di riferimento. Le transizioni e gli stati tratteggiati, al solito, costituiscono la parte spuria degli automi che non è in un cammino dallo stato iniziale ad uno stato finale. Lo stato è descritto, in ordine, dagli stati della protezione e del breaker, dal contenuto del terminale di input del breaker, dallo stato corrente del pattern space (unico) del nodo e dall'indice della sequenza osservata del nodo.



(a) $Bhv(W1)$ (b) $Bhv(W2)$

Figura 4.6: Behavior dei nodi foglia $W1$ e $W2$

4.4.2 Generazione dell'interfaccia

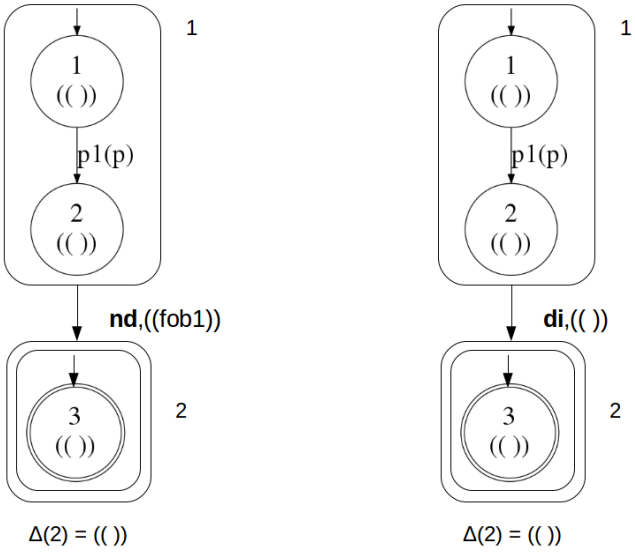
L'interfaccia $Int(N)$ di un nodo N è un automa deterministico ottenuto eliminando dal linguaggio del behavior locale tutte le transizioni che non generano un pattern event. La costruzione avviene come di seguito.

1. L'identificatore di una transizione $t(c)$ di un componente che provoca la generazione di un pattern event (p, l, r, o) viene sostituito dalla coppia (p, Δ_p), dove p è il pattern event e Δ_p la diagnosi relativa alla singola transizione che può essere:
   - l'insieme singleton $\{\emptyset\}$ se la transizione è normale;
   - l'insieme singleton $\{\{f\}\}$, se la transizione è di guasto e f è la corrispondente label del ruler.
2. Le transizioni non associate ad alcun evento di pattern vengono interpretate come ϵ-transizioni, ottenendo in questo modo un automa non deterministico (NFA). Quest'ultimo viene determinizzato attraverso un algoritmo simile alla subset construction 1: differisce da quest'ultima poiché lo stato del DFA risultante contiene una ϵ-closure che include non solo il semplice sottoinsieme di stati del NFA di partenza, ma anche la struttura del sottoautoma relativo agli stati coinvolti. Inoltre le transizioni da fattorizzare hanno come label la coppia (p, Δ_p).
3. All'interno di ogni stato d del DFA, ogni stato n del NFA è marcato applicando l'algoritmo di decorazione, partendo dallo stato radice.

4. Ogni transizione del DFA ha come diagnosi Δ_p la combinazione della diagnosi attuale con la diagnosi Δ associata allo stato del NFA da cui la transizione parte, cioè $\Delta_p = \Delta \bowtie \Delta_p$.

5. Secondo la subset construction, uno stato del DFA è finale se e solo se esso contiene almeno uno stato del corrispondente NFA che è finale. Ad ogni stato finale s_f dell'interfaccia viene associata una diagnosi $\Delta(S_f)$, corrispondente all'unione delle diagnosi degli stati interni finali per il NFA di partenza.

Esempio 4.12. In figura 4.7 sono riportate le interfacce dei nodi $W1$ e $W2$, derivate dai corrispondenti behavior.



(a) $Int(W1)$ (b) $Int(W2)$

Figura 4.7: Interfacce dei nodi $W1$ e $W2$

4.4.3 Costruzione del Behavior vincolato

La costruzione del behavior vincolato riguarda tutti i nodi che non sono dei nodi foglia nella topologia del sistema. Per questo tipi di nodi, la ricostruzione del comportamento deve tenere conto delle informazioni aggiuntive riguardanti i pattern event che giungono dal livello inferiore. Questa informazione è racchiusa nelle interfacce dei nodi inferiori da cui il nodo corrente dipende. Un nodo dipende da un nodo di un livello inferiore se esiste un link che unisce un terminale di output da quel nodo inferiore ad un terminale di input del nodo in esame. Lo stato del behavior corrente, quindi, possiede l'informazione aggiuntava che riguarda la tupla di stati relativi alle interfacce da cui il nodo corrente dipende. Una transizione del comportamento può quindi essere di due tipi:

- una transizione $t(c)$ relativa ad un componente del nodo;
- una transizione $t(Int)$ relativa ad una delle interfacce.

Si noti che anche per quest'ultima vale la condizione che i link dove vengono generati i pattern event siano liberi (secondo la politica di saturazione *wait* adottata in questo lavoro). Uno stato finale del behavior vincolato è tale se, oltre ad aver consumato completamente l'osservazione del nodo locale, ha raggiunto gli stati finali di tutte le interfacce da cui esso dipende. Quest'ultimo vincolo è necessario in quanto, proseguendo bottom-up nella ricostruzione, è indispensabile che ogni nodo abbia consumato la propria osservazione, affinché uno stato finale possa essere considerato il culmine di una traiettoria completa.

Esempio 4.13. In figura 4.8 è riportato il behavior vincolato per il nodo M. Ogni stato ha una coppia indicante lo stato, rispettivamente, delle interfacce $W1$ e $W2$ che ne vincolano il comportamento. Gli stati finali del behavior, oltre ad aver consumato l'osservazione definita in 4.6, sono caratterizzati dal fatto di aver raggiunto uno stato finale di entrambe le interfacce contemporaneamente. In figura 4.9, per completezza, viene generata l'interfaccia del nodo M, la quale vincola la ricostruzione del behavior del nodo radice L.

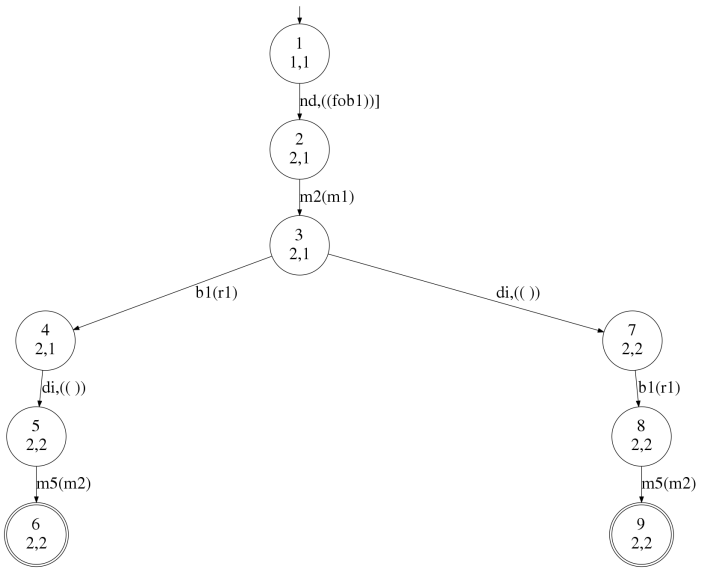


Figura 4.8: Behavior del nodo M (vincolato)

4.4.4 Decorazione del Behavior del nodo radice

Una volta che il metodo di ricostruzione lazy raggiunge il nodo radice ed è stata effettuata la ricostruzione del behavior di quest'ultimo, è necessario effettuarne la decorazione, in maniera simile a quanto visto per il caso greedy, con due variazioni:

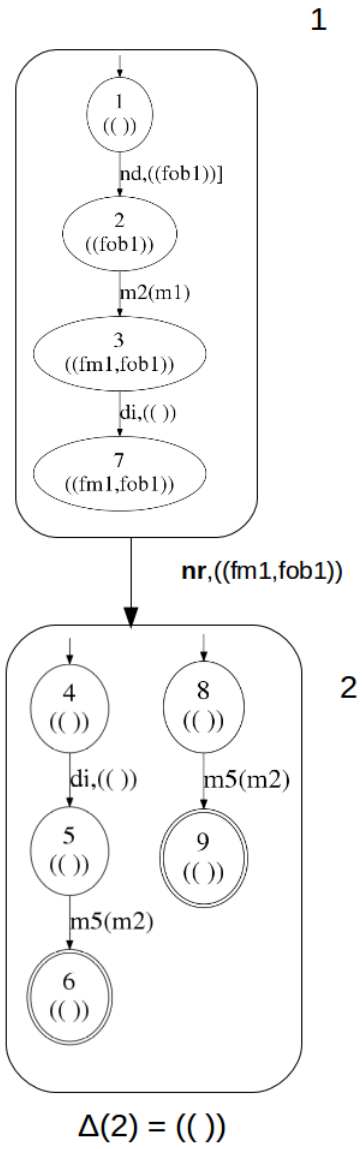


Figura 4.9: Interfaccia del nodo M

- attraversando una transizione relativa ad una interfaccia, è necessario combinare la diagnosi dello stato di partenza con la diagnosi associata alla transizione di interfaccia, mentre per le transizioni relative a componenti del nodo locale si mantiene il solito approccio, ricercando nel ruler se la transizione è presente e con quale label;
- quando la decorazione raggiunge uno stato finale, bisogna combinare la diagnosi ottenute con la diagnosi relativa agli stati finali dell'interfaccia.

Quest'ultima modifica è necessaria al fine di avere una diagnosi completa poiché, mentre negli stati non finali dell'interfaccia la presenza di una transizione uscente ne include le relative diagnosi, in corrispondenza dello stato finale mancano le diagnosi appartenenti a quello stato. La decorazione, in corrispondenza dello stato finale, deve combinare le diagnosi raccolte con le diagnosi relative a quello stato.

Esempio 4.14. La decorazione del nodo radice L del sistema di riferimento è presentata in figura. Si noti che, in corrispondenza dello stato finale, la diagnosi relativa allo stato finale dell'unica interfaccia da cui il comportamento dipende è $\Delta(2) = \{\emptyset\}$. La combinazione di quest'ultima con la diagnosi dello stato finale del behavior, quindi, non produce cambiamenti nel risultato. La soluzione, ottenuta distillando le diagnosi dello stato finale (unico) è:

$$\{\{fli, fm1, fob1\}\}.$$

Tale soluzione è, come voluto, identica a quella dell'esempio 4.10 ottenuta mediante la ricostruzione greedy.

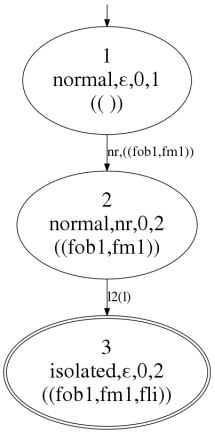


Figura 4.10: Behavior decorato del nodo radice L

Capitolo 5

Implementazione

L'architettura software del progetto sviluppato è composta da tre moduli principali:

1. il compilatore offline, che comprende un compilatore per il linguaggio di specifica e il preprocessing dei pattern space;
2. la macchina diagnostica greedy, che rappresenta il motore diagnostico greedy;
3. la macchina diagnostica lazy, che attua la tecnica di ricostruzione lazy.

La specifica dell'istanza di un sistema attivo complesso di cui si vuole determinare le diagnosi è espressa tramite un linguaggio creato ad hoc, la cui grammatica è fornita nell'appendice A. Un esempio di specifica, relativo al problema diagnostico 4.8, è riportato nell'appendice B. Durante la lettura del file di specifica, sono generate le classi opportune e sono effettuati i controlli del caso. Se questa procedura di compilazione non contiene errori (lessicali, sintattici o semantici), viene effettuato il preprocessing offline del problema, durante il quale si creano gli automi relativi ai pattern space. Tutte le informazioni ottenute vengono salvate su appositi file, in modo che non sia necessario dover effettuare nuovamente la procedura in fase di lavoro online, ovvero quando viene avviata la macchina diagnostica, sia essa in modalità greedy o lazy. La procedura diagnostica legge i file generati da una precedente compilazione della specifica e genera le soluzioni candidate. Il linguaggio di programmazione adottato è il C++, che è stato scelto in quanto buon compromesso tra le esigenze di efficienza richieste dal problema e il livello di astrazione tipico della programmazione ad oggetti.

5.1 Rappresentazione degli automi

Per la rappresentazione degli automi è stata ricercata una libreria esterna che fornisse da un lato delle strutture dati e dei metodi efficienti, dall'altro che fosse quanto più generica e personalizzabile, dal momento che gli automi necessari alle tecniche risolutive possiedono vari tipi di informazione negli stati e nelle transizioni. La libreria ASTL (*Automata*

Standard Template Library) [4] soddisfa entrambe le esigenze. Le due classi principali di questa libreria utilizzate nell'implementazione sono:

- `DFA_map<SIGMA,TAG>`, rappresenta automi deterministici ed è implementata memorizzando le transizioni di uno stato in uno standard `map`, associando i relativi simboli dell'alfabeto agli stati raggiunti. Dato che la struttura `map` costituisce un albero binario, le operazioni di accesso, inserimento e cancellazione sono effettuate in un tempo logaritmico $\mathcal{O}(\log n)$ rispetto al numero di elementi contenuti. Si tratta di un buon compromesso tra tempi richiesti dalle operazioni e memoria occupata.

- `NFA_mmap<SIGMA,TAG>`, rappresenta automi non deterministici. Le transizioni sono memorizzate per mezzo di `multimap`, le cui principali operazioni hanno complessità logaritmica.

Il template `SIGMA` permette di specificare l'alfabeto cui i simboli di transizione appartengono, mentre il template `TAG` consente di associare ad uno stato delle informazioni satelliti generiche. Per la visualizzazione degli automi, in fase di debug nonché per le illustrazioni di questa trattazione, sono stati utilizzati metodi che generano file scritti nel linguaggio `dot`, interpretati dal programma *Graphviz*[5] che ne genera la rappresentazione.

5.2 Classi condivise

I tre moduli che compongono l'architettura software ideata hanno delle classi tra loro condivise, in modo da operare sulle stesse strutture dati. Si tratta di quegli oggetti che sono creati in fase di compilazione del file di specifica e che successivamente sono utilizzati al fine di avere le informazioni necessarie nella fase diagnostica. Nel seguito vengono descritte tali classi.

- `ComponentModel`
  Rappresenta il modello di un componente. Ha come attributi la stringa del nome che lo identifica e vettori di stringhe contenenti i nomi di eventi, terminali di input, terminali di output e stati. Possiede un vettore di oggetti `Transition` che contengono le informazioni relative ad ogni transizione del modello. L'attributo `automaton` indica l'automa risultante dal modello comportamentale descritto dalle transizioni. Si noti che l'automa non è provvisto di stato iniziale, in quanto questa informazione è definitiva solo nella specifica del problema diagnostico.

- `NetComponent`
  Rappresenta un componente generico nel modello di un nodo. Ha come attributi il nome, che lo identifica nel particolare modello di nodo nel quale è contenuto, e un riferimento ad un oggetto `ComponentModel` che ne rappresenta il modello topologico e comportamentale.

- `Component`
  Rappresenta un componente concreto del problema diagnostico. É una classe derivata da `NetComponent` e possiede gli attributi aggiuntivi relativi all'automa ottenuto dal modello di componente associato, con l'informazione addizionale dello stato iniziale, e gli oggetti `Terminal` riferiti ai terminali di input e output della sua topologia.

- `NetworkModel`
  Rappresenta il modello di un nodo. Possiede gli attributi seguenti:

  - `name`: nome che identifica il modello del nodo;
  - `components`: vettore di oggetti `NetComponents` contenuti nel modello;
  - `inputs` e `outputs`: specificano i nomi dei terminali di input e dei terminali di output del modello del nodo;
  - `links`: contiene l'elenco dei link che connettono tra loro i componenti e i terminali di input del nodo ai terminali di input dei componenti che gestiscono i pattern event in ingresso;
  - `patterns`: vettore di oggetti di tipo `Pattern`, ognuno dei quali rappresenta una dichiarazione di un pattern event che è generato in corrispondenza di un terminale di output del nodo;
  - `initials`: nomi degli stati iniziali per ogni componente del nodo;
  - `viewer`: `map` tra nomi di transizioni di componenti e corrispondenti label osservabili;
  - `ruler`: `map` tra nomi di transizioni di componenti e corrispondenti label di guasto;
  - `pattern_space`: vettore di automi che costituiscono i pattern space del nodo;
  - `languages`: vettore di linguaggi, ognuno dei quali costituito da un insieme di stringhe recanti i nomi delle transizioni appartenenti al linguaggio di una o più dichiarazioni di pattern.

  Si noti che le informazioni relative agli stati iniziali, al viewer e al ruler possono non essere presenti oppure venire sovrascritte nelle successive dichiarazioni di istanza del nodo del sistema o del problema diagnostico.

- `Pattern`
  Rappresenta una dichiarazione di pattern e contiene gli attributi relativi al nome del pattern event, all'espressione regolare, al nome del terminale di destinazione del pattern event, un attributo booleano che indica se il linguaggio associato al pattern è massimo, ovvero costituito dall'insieme di tutte le transizioni di tutti i componenti del nodo, e un vettore di elementi del linguaggio.

- `Terminal`
  rappresenta un terminale di un componente o di un nodo. É caratterizzato da una stringa del nome, una relativa al valore contenuto (evento), e un vettore di riferimenti a terminali ad esso connessi.

- `Transition`
  Rappresenta la transizione di un modello di componente. Ha come attributi la stringa relativa al suo nome identificativo, una coppia di stringhe indicanti il nome dell'evento in ingresso e il terminale di input, un vettore di coppie di stringhe indicanti il nome dell'evento di uscita e il terminale di output, e la coppia del nome di stati sorgente-destinazione coinvolti nella transizione. Possiede attributi e metodi (l'operatore di confronto e di uguaglianza) in modo da permetterne l'utilizzo come alfabeto di un automa della libreria ASTL, come mostrato in figura 5.1.

- `NetTransition`
  Rappresenta la transizione di un componente del modello di un nodo. Ha come attributi il riferimento alla transizione del modello di componente del particolare componente del nodo, e un riferimento a quest'ultimo. Il nome è ottenuto, per mezzo del costruttore dell'oggetto, concatenando il nome della transizione associata e, tra parentesi, il nome del componente del nodo associato.

- `SysTransition`
  Rappresenta la transizione di un componente concreto di un nodo del problema diagnostico. É una classe derivata da `NetTransition` e presenta i seguenti attributi aggiuntivi:

  - `node`: il riferimento nodo del sistema relativo alla transizione;
  - `input_event`: una coppia formata dal nome di un evento e dall'indice della tupla di terminali di input dei componenti del sistema (utile per la diagnosi greedy);
  - `output_events`: un vettore di coppie formate dal nome di un evento e dall'indice della tupla di terminali di input dei componenti del sistema (utile per la diagnosi greedy);
  - `lazy_input_event`: una coppia formata dal nome di un evento e dall'indice della tupla di terminali di input dei componenti del nodo coinvolto (utile per la diagnosi lazy);
  - `lazy_output_events`: un vettore di coppie formate dal nome di un evento e dall'indice della tupla di terminali di input dei componenti del nodo coinvolto (utile per la diagnosi lazy).

- `System`
  Rappresenta la specifica del sistema. Possiede i seguenti attributi:
  - `name`: nome del sistema;
  - `node_list`: vettore dei nodi del sistema;
  - `emergence`: lista delle dichiarazioni di link tra nodi del sistema;
  - `dependency_graph`: grafo che rappresenta la topologia del sistema;
  - `acyclic`: attributo booleano di valore `true` se la topologia del sistema è aciclica e permette quindi la diagnosi lazy.

- `SystemNode`
  Rappresenta un nodo del sistema ed è caratterizzato da un nome identificativo e un riferimento all'oggetto di tipo `NetworkModel` che ne costituisce il modello; possiede inoltre gli attributi relativi agli stati iniziali dei suoi componenti, al viewer e al ruler locali, i quali possono essere ereditati dal modello o sovrascritti nella specifica del problema diagnostico.

- `Problem`
  Rappresenta il problema diagnostico e possiede un nome, una lista di nodi del problema e un vettore di indici che indica l'ordinamento topologico dei nodi, al fine di poter effettuare la diagnosi lazy.

- `ProblemNode`
  Rappresenta un nodo del problema diagnostico. Possiede i seguenti attributi:
  - `name`: nome del nodo, ottenuto dal nodo del sistema associato;
  - `ref_node`: riferimento al nodo del sistema;
  - `concrete_components`: vettore di componenti concreti;
  - `input_terminals`: vettore di terminali di ingresso fisici del nodo;
  - `output_terminals`: vettore di terminali di uscita fisici del nodo;
  - `observation`: sequenza di label relative all'osservazione locale;
  - `index_space`: automa lineare ottenuto dall'osservazione, utile per eventuali sviluppi futuri nei quali l'osservazione è incerta;
  - `depends`: lista di indici dei nodi dai quali il nodo corrente dipende topologicamente;
  - `patt_map`: mappa un pattern event nel terminale di uscita fisico del nodo;
  - `patt_indexes_map`: mappa un pattern event nell'indice della tupla dei terminali di input dei componenti dell'intero problema (utile per la diagnosi greedy);

  - `lazy_patt_indexes_map`: mappa un pattern event nell'indice della tupla dei terminali di input dei componenti del nodo (utile per la diagnosi lazy);

  La classe contiene inoltre le informazioni definitive riguardanti gli stati iniziali dei componenti, il viewer e il ruler locali, che possono essere ereditati dalla specifica del nodo del sistema.

- `Utils`, contiene metodi statici di utilità.

```
class Transition : public CHAR_TRAITS<Transition>
{
    ...
    //required definitions to use a Transition
    //as automata alphabet for ASTL lib
    typedef Transition char_type;
    typedef long        int_type;
    static const size_t size;
    static bool eq(const char_type &x, const char_type &y)
    { return x == y; }
    static bool lt(const char_type &x, const char_type &y)
    { return x < y; }
    bool operator<(const Transition t) const {return name<t.name;}
    bool operator==(const Transition t) const {return name == t.name;}
};
```

Figura 5.1: Classe Transition

In figura 5.2 è riportato il diagramma UML delle classi relative ai componenti del sistema, ovvero le classi `ComponentModel`, `NetComponent` e `Component`.
In figura 5.3 è riportato il diagramma UML delle classi relative alle transizioni, ovvero `Transition`, `NetTransition` e `SysTransition`.
In figura 5.4 è riportato il diagramma UML delle classi che rappresentano, nei diversi livelli di definizione, i nodi del sistema, cioè le classi `NetworkModel`, `SystemNode` e `ProblemNode`.

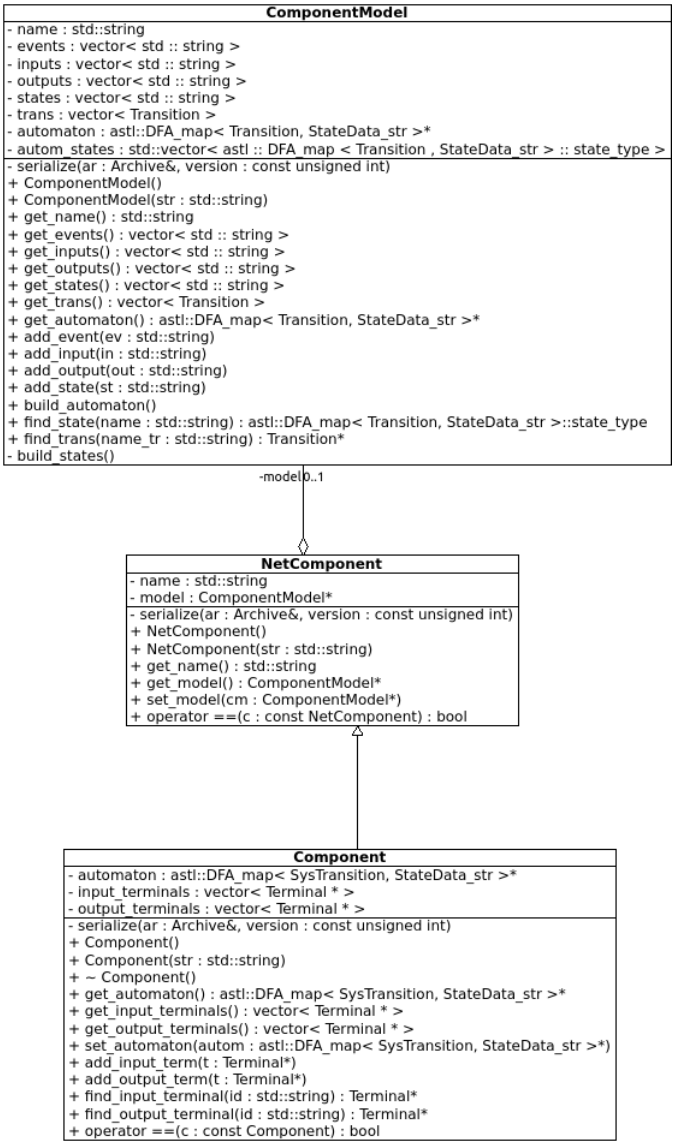


Figura 5.2: Classi dei componenti

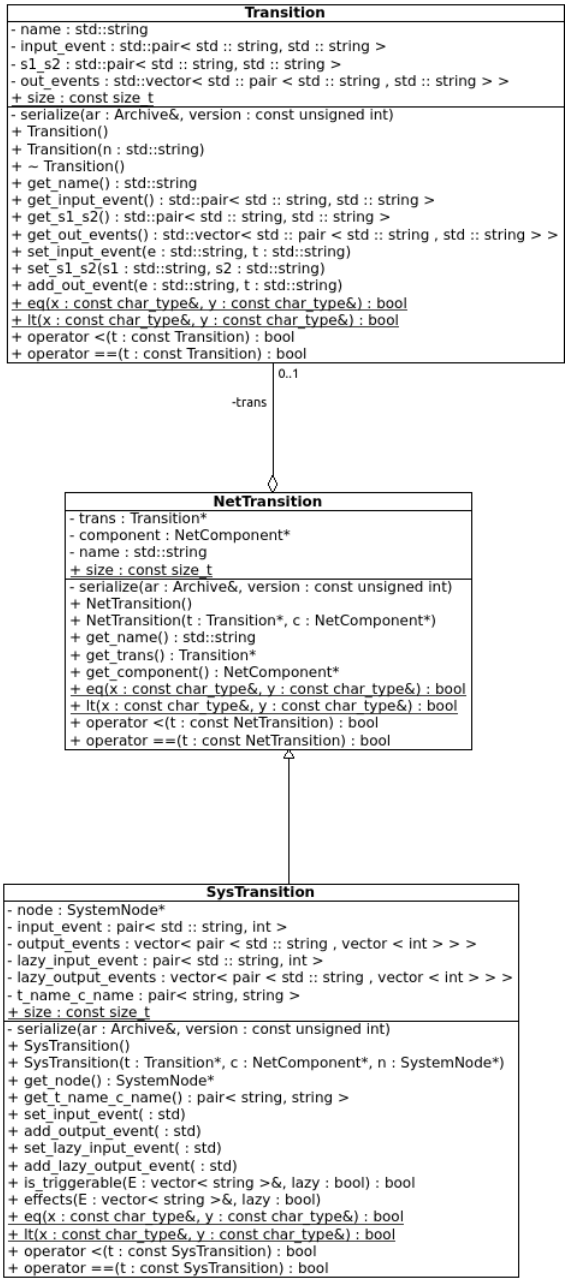


Figura 5.3: Classi delle transizioni

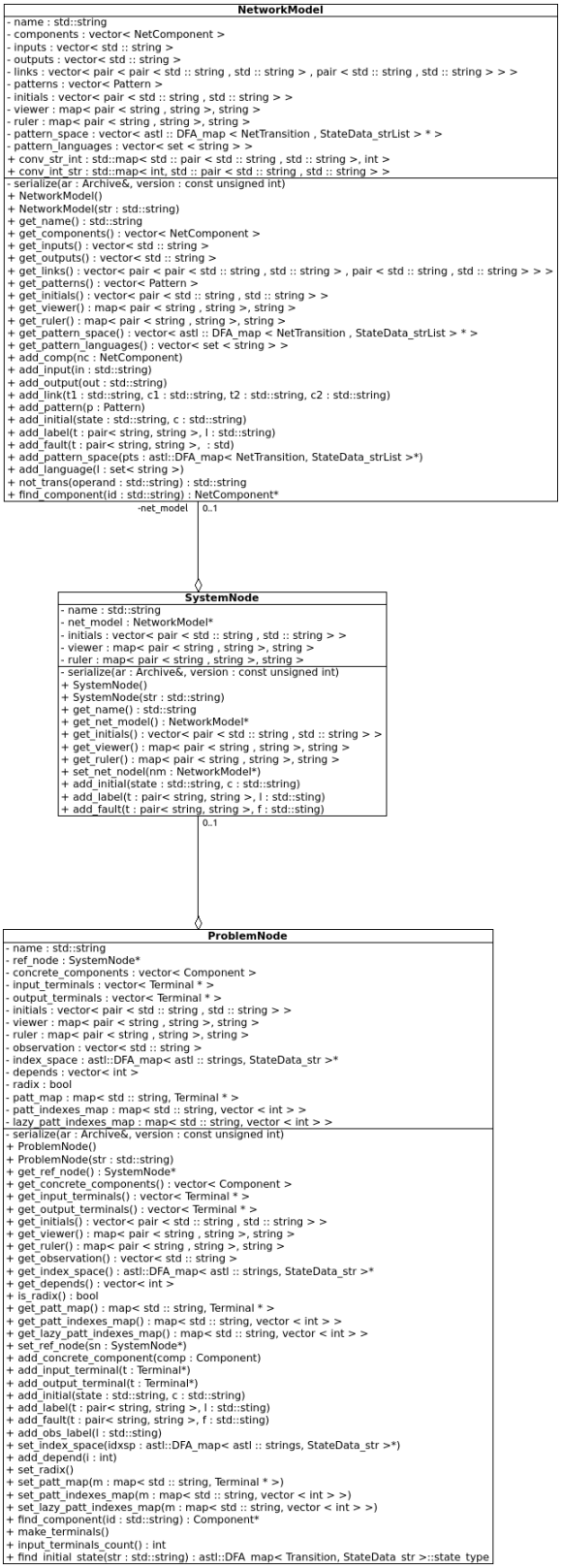


Figura 5.4: Classi dei nodi

5.3 Compilatore offline

Il linguaggio di specifica dei sistemi attivi complessi è fornito, tramite notazione BNF, in appendice A. Il file di specifica è articolato in quattro parti principali:

- modelli dei componenti, a cui i componenti concreti del sistema si riferiscono;
- modelli dei nodi, a cui i nodi concreti del sistema si riferiscono;
- sistema, che istanzia i nodi e definisce i link responsabili dell'invio di pattern event tra di essi;
- problema, che fornisce informazioni necessarie per il problema diagnostico, come le osservazioni locali dei nodi.

I viewer locali, i ruler locali e gli stati iniziali dei componenti dei singoli nodi possono essere definiti, con eventuale riscrittura, a tre diversi livelli:

- nella definizione del modello del nodo;
- nella dichiarazione di un nodo del sistema;
- nella specifica di un nodo del problema.

Questo permette una certa facilità di scrittura, non essendo costretti a dichiarare informazioni di questo tipo, quando esse sono ridondanti. Il file di specifica viene compilato seguendo le tradizionali fasi ideate per i compilatori dei linguaggi di programmazione:

1. analisi lessicale, che legge il flusso di caratteri che costituiscono il codice di specifica e raggruppa i caratteri in sequenze significative dette lessemi, quali le parole chiave del linguaggio e gli identificatori;
2. analisi sintattica, che riceve i lessemi dall'analizzatore lessicale per creare una rappresentazione ad albero del codice di specifica, detto albero sintattico;
3. analisi semantica, la quale utilizza le informazioni raccolte nell'albero sintattico in modo da verificare la consistenza del codice di specifica.

Si noti che in questo progetto è possibile articolare una specifica in più file, esplicitando delle dichiarazioni del tipo `#include` per i file che si vogliono incorporare. Questo permette il riutilizzo di specifiche già dichiarate precedentemente, aumentando inoltre la leggibilità dei file di specifica.

5.3.1 Analizzatore lessicale

L'analizzatore lessicale (o scanner) è implementato mediante lo strumento *Flex*, il quale permette di dichiarare espressioni regolari che descrivono i pattern per i token, ovvero coppie nome-attributo, ad esempio `<id,breaker>`. La struttura di un programma Flex è composta da tre parti principali:

1. dichiarazioni, che includono variabili, costanti e definizioni regolari;
2. regole di traduzione, che specificano per ogni pattern dichiarato per mezzo di una espressione regolare le azioni da compiere, attraverso frammenti di codice;
3. funzioni ausiliarie, che possono essere richiamate nelle azioni dei pattern.

Sebbene Flex abbia un'opzione per creare direttamente uno scanner C++, il codice generato spesso non funziona e contiene errori [6] . La procedura adottata per risolvere questo problema è quella di compilare il programma Flex scritto in linguaggio C per mezzo di un compilatore C++. La sezione dichiarativa del linguaggio è riportata in figura 5.5. In essa si definiscono spazi, delimitatori, commenti di linea (in stile C) e l'espressione regolare che definisce gli identificatori `id`, formata da una lettera a cui seguono altre lettere, numeri oppure il carattere `_`.

```
// The location of the current token.
yy::location loc;

%}
%option noyywrap nounput
%x incl

delimiter [ \t]
spacing {delimiter}+
letter [A-Za-z]
digit [0-9]
unscore         _
id {letter}({letter}|{digit}|{unscore})*
eol \n
comment \/\/.*

%{
  // Code run each time a pattern is matched.
  # define YY_USER_ACTION  loc.columns (yyleng);
%}
```

Figura 5.5: Dichiarazioni.

Le regole di traduzione utilizzate sono principalmente volte a generare i token relativi a parole chiave del linguaggio e agli identificatori. In corrispondenza di direttive `#include`,

il passaggio di lettura al file incluso è operato, come mostrato in figura 5.6, creando una pila di buffer.

```
#include          BEGIN(incl);
<incl>[ \t]*      /* eat the whitespace */
<incl>[^ \t\n]+
{   /* got the include file name */
    chdir(INPUT_FILE_DIR);
    FILE *file = fopen( yytext, "r" );

    if (file == NULL)
    {
       cout << "Input file error: included file does not exist" << endl;
       exit(1);
    }
    yyin = file;

    yypush_buffer_state(yy_create_buffer( yyin, YY_BUF_SIZE ));
    BEGIN(INITIAL);
}
```

Figura 5.6: Gestione delle direttive di inclusione.

Gli spazi e i commenti non invocano alcuna azione, mentre la fine del file attuale toglie dalla pila il buffer corrente, riportando il controllo alla scansione del file precedente da cui la direttiva di inclusione è stata chiamata (figura 5.7). Se il buffer corrente è l'unico elemento della pila, il processo di scanning termina ed è generato il token relativo alla fine del file.

```
{spacing} ;
{comment}   ;

{eol} {loc.lines (yyleng);loc.step();}

<<EOF>>
{
  yypop_buffer_state();
  if ( !YY_CURRENT_BUFFER )
    return yy::spec_parser::make_END_OF_FILE(loc);;

}
```

Figura 5.7: Azioni compiute alla fine del file.

Quando viene trovata una parola chiave (come ad esempio `component` e `model`), viene creato il corrispondente token che verrà utilizzato nell'analisi sintattica. Per quanto riguar-

da gli identificatori (`id`), il loro token è composto oltre che dalla posizione, dalla stringa che ne rappresenta il nome, memorizzato durante lo scanning nella variabile `yytext` (figura 5.8). La lettura di un carattere non appartenente a nessuna delle configurazioni definite genera un errore lessicale.

```
component   return yy::spec_parser::make_COMPONENT(loc);
model       return yy::spec_parser::make_MODEL(loc);
is          return yy::spec_parser::make_IS(loc);


{id}        return yy::spec_parser::make_ID(yytext,loc);
.     {cout << "Lexical error"; exit(1);}
```

Figura 5.8: Regole di traduzione per parole chiave e identificatori.

5.3.2 Analizzatore sintattico

L'analizzatore sintattico (parser) è implementato tramite lo strumento *Bison*. La generazione della grammatica è supportata dalla classe `spec_driver`, la quale avvia il parsing e ne gestisce gli eventuali errori generati. Il file della grammatica inizia richiedendo lo skeleton parser deterministico per poter utilizzare direttamente il linguaggio C++, creando il file header della grammatica e specificando il nome della classe del parser.

```
%skeleton "lalr1.cc"
%defines
%define parser_class_name {spec_parser}
```

Per poter utilizzare oggetti C++ come valori semantici della grammatica, deve essere richiesta l'interfaccia `variant`.

```
%define api.value.type variant
%define api.token.constructor
%define parse.assert
```

Successivamente sono specificate le dichiarazioni necessarie ai valori semantici. Poiché il parser utilizza la classe driver e viceversa, questa mutua dipendenza è risolta fornendo una *forward declaration*.

```
%code requires
{
...
class spec_driver;
}
```

L'oggetto driver viene passato al parser e allo scanner tramite referenza.

```
%param { spec_driver& driver}
```

Quindi viene richiesto di tenere traccia della locazione (riga e colonna del file di specifica), passando il nome del file dell'oggetto `driver`.

```
%locations
%initial-action
{
  // Initialize the initial location.
  @$.begin.filename = @$.end.filename = &driver.file;
};
```

La definizione dei token avviene come di seguito.

```
%token
COMPONENT
MODEL
IS
...
END_OF_FILE  0  "end of file"
COMMA ","
...
```

Per ogni terminale e non terminale della grammatica, deve essere specificato un tipo dell'oggetto.

```
%token <string> ID "id"
%type <ComponentModel>  comp_model_decl
%type <vector<string> > event_decl
..
```

Successivamente viene implementata la grammatica del linguaggio, assegnando i valori semantici definiti.

5.3.3 Analizzatore semantico

L'analisi semantica, a differenza di quella lessicale e sintattica, non prevede l'utilizzo di strumenti standard per la sua attuazione, ma è compito del programmatore definirne le regole, in base al significato del linguaggio specificato Nell'ambito di questo progetto, i controlli semantici sono affidati alla classe `spec_driver` (figura 5.9), la quale verifica la

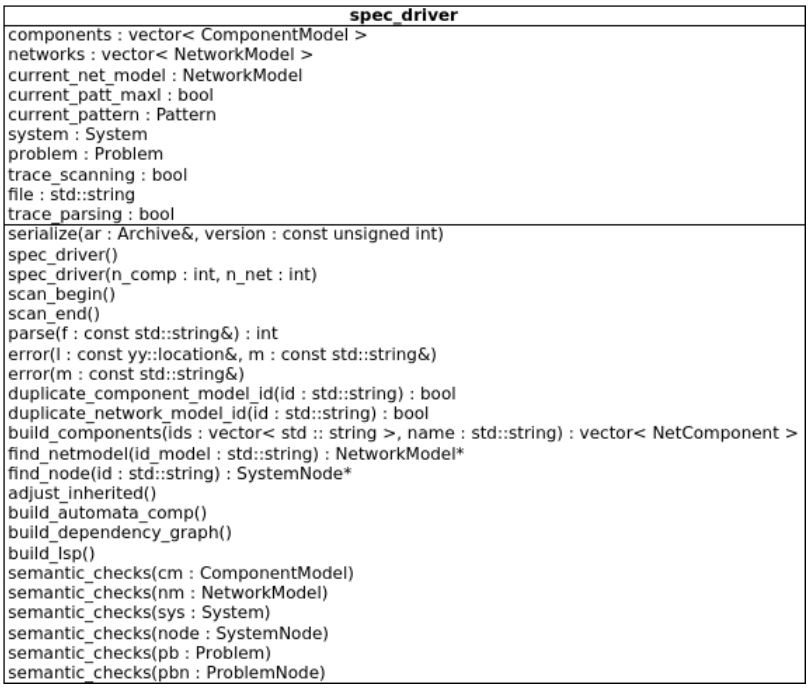


Figura 5.9: Classe spec_driver

consistenza semantica durante il parsing del file, interrompendone l'esecuzione in caso di errori.

La classe possiede, tra gli altri, i metodi `semantic_checks`, implementati in *overloading* per i seguenti tipi di parametro d'ingresso:

1. `ComponentModel`: verifica che non vi siano identificatori multipli per le liste di eventi, terminali di input, terminali di output e stati; nelle dichiarazioni delle transizioni, verifica che il nome degli eventi, dei terminali e degli stati siano stati precedentemente definiti.

2. `NetworkModel`: verifica l'unicità del nome dei componenti, dei terminali e dei nomi dei pattern; verifica che i terminali e i componenti dei link siano definiti. Verifica che non vi siano più link entranti nello stesso terminale di input. Verifica l'esistenza delle transizioni utilizzate nei pattern, nel viewer e nel ruler (se presenti), oltre all'esistenza degli eventuali stati iniziali dei componenti dichiarati.

3. `System`: verifica che nelle dichiarazioni di *emergence* i nomi dei nodi e dei relativi terminali siano stati definiti.

4. `SystemNode`: effettua i controlli sulle eventuali dichiarazioni di stati iniziali, viewer e ruler locali.

5. `Problem`: verifica che siano ivi dichiarati tutti i nodi definiti nel sistema.

6. `ProblemNode`: verifica i controlli sulle eventuali dichiarazioni di stati iniziali, viewer e ruler locali. Se esse non sono presenti, vengono ereditate dal nodo di sistema corrispondente. Viene inoltre verificata l'esistenza delle label dell'osservazione locale nel viewer corrente.

L'analisi semantica viene ultimata, attraverso il metodo `build_dependency_graph`, costruendo il grafo che rappresenta la topologia del sistema, verificando che sia connesso. La connessione è verificata nel metodo `Utils::disconnected_graph`, il quale compie un'intera traversata in ampiezza del grafo; se il numero di nodi visitati è inferiore al numero di nodi dichiarati, alcuni di essi non sono connessi. Viene inoltre verificata l'aciclicità del grafo, eliminando uno dopo l'altro i nodi foglia dal grafo, fino a quando esso non risulta essere vuoto. Se ad un certo punto della procedura non vi sono nodi foglia e il grafo non è vuoto, significa che sono presenti ciclicità e non sarà possibile avviare la diagnosi lazy. Nel caso il grafo sia aciclico, viene calcolato l'ordinamento topologico, attraverso un algoritmo simile al precedente.

5.3.4 Generazione dei pattern space

La principale azione compiuta in fase offline, ovvero in un momento antecedente la diagnosi, è la generazione dei pattern space dei modelli di nodi definiti, come descritto nel paragrafo 4.1.4. Il primo passo di questo algoritmo consiste nella costruzione di un automa equivalente all'espressione regolare definita nella dichiarazione di pattern. Un metodo noto consiste nella costruzione di Thompson descritta nel paragrafo 2.5.1. Per questo scopo, è stata utilizzata la libreria *Grail+* [7], una libreria C++ che permette di definire espressioni regolari e convertirle in automi con un linguaggio equivalente. La conversione avviene utilizzando la sintassi presentata in figura 5.10.

```
fm<int> patodfa(std::string regex)
{
    re<int> r;
    istringstream str(regex);
    str >> r;
    fm<int> dfa;
    r.retofm(dfa);
    dfa.subset();

    return dfa;
}
```

Figura 5.10: Passaggio da espressione regolare ad automa

Data una stringa in ingresso, la funzione implementata inserisce tale stringa in un oggetto della classe `re` (*regular expression*), parametrizzato per mezzo di un template che

indica l'alfabeto di appartenenza dei simboli dell'espressione. Dato che gli alfabeti possibili per questa libreria erano esclusivamente di tipo `char` o `int`, è stato scelto quest'ultimo (poiché 256 simboli possibili potevano non essere sufficienti in applicazioni di dimensioni ragguardevoli), mantenendo una corrispondenza biunivoca tra ogni intero e una transizione, coinvolta nella dichiarazione del pattern, relativa ad un componente del nodo. La conversione da espressione regolare ad automa a stati finiti avviene applicando all'oggetto che rappresenta l'espressione regolare il metodo `retofm`, che crea l'automa nell'oggetto della classe `fm` (*finite machine*) passato come parametro. A seguito della conversione, l'automa è determinizzato applicando il metodo `subset`, il quale esegue la subset construction descritta nel paragrafo 2.3. Successive manipolazioni dei diversi automi ottenuti dai pattern dichiarati permettono di unirli, aggiungere le ϵ-transizioni opportune e determinizzare il risultato. L'automa finale viene poi convertito in un automa equivalente, utilizzando le strutture dati della libreria ASTL. Infatti, la libreria Grail+, sebbene utile poiché implementa la costruzione di Thompson, non sembrava indicata ad un utilizzo negli algoritmi di diagnosi, in quanto non permette estensioni per quanto riguarda i dati satellite associati a transizioni e stati dell'automa. La minimizzazione finale del pattern space è implementata nel metodo `Utils::minimize_by_partition`, che esegue una minimizzazione tenendo conto dei tag associati (pattern event) ai nodi finali dell'automa.

5.3.5 Salvataggio dei dati compilati

Una volta che il file di specifica è stato processato, le classi opportune sono state create e i controlli semantici sono andati a buon fine, la procedura di elaborazione offline termina creando dei file binari contenenti tutte le informazioni utili alla successiva fase diagnostica, ovvero tutti gli oggetti istanziati per il problema specificato. La scrittura di questi dati, chiamata serializzazione, è una procedura piuttosto complessa se implementata manualmente in C++, in quanto deve gestire puntatori ed evitare ridondanza. Per questo è stata adottata la libreria *Serialization*, fornita dalla collezione *Boost* [8]. Essa permette di salvare un oggetto con semplici modifiche della classe a cui esso appartiene. Ad esempio, la definizione della classe `ComponentModel` è stata modificata aggiungendo le istruzioni riportate in figura 5.11. Per ogni attributo della classe, viene specificata la lettura e la scrittura in un oggetto di tipo `Archive`, appartenente alla libreria. É necessario esplicitare l'operazione per ogni attributo della classe, in modo da poter salvare lo stato dell'oggetto corrente; eventuali attributi non salvati avranno un valore impredicibile. Quando l'oggetto corrisponde ad un archivio di output, l'operatore `&` indica la scrittura, mentre nel caso di un archivio di input esso indica l'operazione di lettura. Le modifiche attuate permettono quindi sia la scrittura su file che la lettura operata durante la successiva fase diagnostica.

Il programma sviluppato salva le informazioni precompilate in quattro file distinti:

- un file contenente tutti i modelli dei componenti (`component_models.dat`);

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>

class ComponentModel{
    friend class boost::serialization::access;
    template<class Archive>
    void serialize(Archive & ar, const unsigned int version)
    {
        ar & autom_states;
        ar & name;
        ar & automaton;
        ar & events;
        ar & inputs;
        ar & outputs;
        ar & states;
        ar & trans;
    }
```

Figura 5.11: Modifiche apportate alla classe ComponentModel per la serializzazione.

- un file che comprende tutti i modelli dei nodi (`network_models.dat`);
- un file che racchiude la specifica del sistema (`system.dat`);
- un file contenente le informazioni del problema diagnostico con le osservazioni temporali (`problem.dat`).

Per eseguire la scrittura di un oggetto è sufficiente dare istruzioni simili a quelle riportate in figura 5.12, nella quale è salvato su file il `vector` dei modelli dei componenti utilizzati nel sistema.

```
{
    std::ofstream ofs("../../CompiledData/component_models.dat");
    boost::archive::text_oarchive oa(ofs);
    // write class instance to archive
    oa << driver.components;
    // archive and stream closed when destructors are called
}
```

Figura 5.12: Serializzazione.

Si noti che la libreria *Serialization* permette di salvare, senza il bisogno di istruzioni aggiuntive, oggetti appartenenti alla Standard Template Library, quali ad esempio `vector`, `map`, `list` e `set`. Per quanto riguarda il salvataggio di puntatori, al fine di evitare ridondanza, la libreria salva l'oggetto puntato un'unica volta, mantenendone i riferimenti.

5.4 Macchina diagnostica

5.4.1 Lettura dei dati precompilati

La fase diagnostica necessita degli oggetti generati dal precedente passo di compilazione dei file di specifica. La lettura delle informazioni salvate avviene in modo simile a quanto fatto per la scrittura, come si può vedere in figura 5.13.

```
vector<ComponentModel> components;
{
    // create and open an archive for input
    std::ifstream ifs("../../CompiledData/component_models.dat");
    boost::archive::text_iarchive ia(ifs);
    // read class state from archive
    ia >> components;;
    // archive and stream closed when destructors are called
}
```

Figura 5.13: Lettura dai file binari e memorizzazione degli oggetti salvati.

5.4.2 Modalità greedy

La modalità greedy implementa l'algoritmo di ricostruzione del behavior dell'intero sistema. Dopo aver acquisito gli oggetti memorizzati nei file precompilati, viene generato lo stato iniziale con i seguenti campi:

- una tupla di interi indicanti gli stati iniziali di tutti i componenti del problema;
- una tupla di stringhe recanti il valore, inizialmente vuoto, del contenuto dei terminali di input di tutti i componenti del sistema;
- una tupla di interi indicanti gli stati iniziali di tutti i pattern space di ogni nodo;
- una tupla di interi indicanti gli stati iniziali degli index space di ogni nodo (osservazione locale, inizialmente nulla).

La classe che rappresenta uno stato del behavior è riportata in figura 5.14.

L'algoritmo tiene traccia delle configurazioni di ogni stato attraverso una tabella hash: in questo modo la verifica dell'esistenza di uno stato generato da una transizione avviene in tempo costante. La tabella mappa la stringa relativa al contenuto dello stato nel valore intero associato allo stato del behavior, come mostrato di in figura 5.15 per lo stato iniziale.

La creazione del behavior avviene come descritto nella sezione 4.3. La verifica dell'esistenza di uno stato raggiunto da una transizione avviene come descritto in figura 5.16, dove `tag_s1` è un'istanza della classe `BehaviorState`.

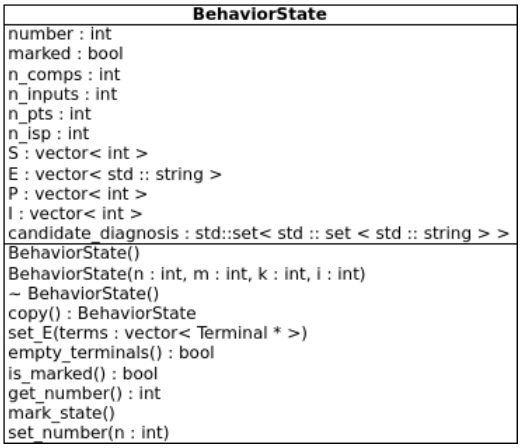


Figura 5.14: Classe BehaviorState

```
DFA_map<SysTransition,BehaviorState> behavior;
unordered_map<string,unsigned int> hash_values;
stringstream ss;
ss << tag_s0;
unsigned int s0 = behavior.new_state();
behavior.tag(s0) = tag_s0;
behavior.initial(s0);
hash_values[ss.str()] = s0;
```

Figura 5.15: Creazione tabella hash

```
stringstream s_str;
s_str << tag_s1;
string str = s_str.str();
unsigned int s1;
try{ s1 = hash_values.at(str);}
catch (const std::out_of_range&)
{
    s1 = behavior.new_state();
    behavior.tag(s1) = tag_s1;
    hash_values[str] = s1;
...
```

Figura 5.16: Ricerca nella tabella hash

Per ogni stato creato viene contestualmente verificato se si tratta di uno stato finale, ovvero caratterizzato da terminali di input vuoti e che ha consumato tutte le osservazioni locali dei vari nodi. Una volta terminato il ciclo principale della generazione del behavior, ed ottenuto quindi il behavior spurio, esso viene potato di tutte le transizioni e gli stati che non sono in un cammino tra lo stato iniziale ed uno stato finale. L'operazione avviene invocando la funzione predisposta `trim` presente nella libreria ASTL (figura 5.17).

```
DFA_map<SysTransition,BehaviorState> bhv;
unsigned int init = trim(bhv,dfirst_markc(behavior));
```

Figura 5.17: Rimozione della parte spuria del behavior

Se dopo questa operazione il behavior risulta vuoto, significa che le osservazioni date non sono consistenti con il modello specificato e l'esecuzione viene terminata con errore. Questo può avvenire in fase di sperimentazione, a causa dell'assenza di un reale sistema fisico che dia un'osservazione vera del comportamento. Altrimenti, l'algoritmo procede decorando l'automa ottenuto e distillandone le diagnosi. L'algoritmo di decorazione consiste nella semplice implementazione dello pseudocodice 3. La distillazione consiste nell'unione delle diagnosi contenute negli stati finali del behavior.

5.4.3 Modalità lazy

La diagnosi lazy avviene in maniera simile alla ricostruzione greedy, con alcune variazioni. Viene letto l'ordinamento topologico del sistema e viene applicata in successione la ricostruzione del comportamento del singolo nodo. Ogni nodo che non è l'ultimo dell'ordinamento topologico, ovvero che non costituisce un nodo radice, necessita della generazione della sua interfaccia. Durante la generazione dei behavior, si tiene conto anche delle dipendenze dalle interfacce dei nodi inferiori collegati al nodo corrente.

Determinizzazione dell'interfaccia

La classe `Determinization` fornisce dei metodi statici che permettono di ottenere l'interfaccia associata al behavior locale di un nodo. É composta dai metodi seguenti:

- `NFAtoDFA` guida il processo di determinizzazione, ricevendo il behavior del nodo;
- `eps_closure` calcola la ϵ-chiusura di uno stato o di un insieme di stati del behavior, nel quale le transizioni che non producono pattern event sono sostituite da ϵ-transizioni;
- `extract_subNFA` estrae dall'automa del behavior la parte contenente gli stati appartenenti alla ϵ-chiusura, in modo che tale sottoautoma possa essere memorizzato nello stato dell'interfaccia;

- `move` calcola gli stati raggiungibili attraverso transizioni, applicate a stati interni allo stato dell'interfaccia, che generano pattern event.

La decorazione del sottoautoma contenuto negli stati dell'interfaccia avviene richiamando un metodo della classe `Decoration` che implementa la classica decorazione come nell'algoritmo 3. La classe che rappresenta lo stato dell'interfaccia è visualizzata in figura 5.18. Le transizioni dell'interfaccia (figura 5.19) possiedono invece i seguenti attributi:

- `trans`: transizione di un componente del nodo;
- `delta`: diagnosi associata alla transizione;
- `pattern_events`: elenco dei patter event associati alla transizione dell'interfaccia.

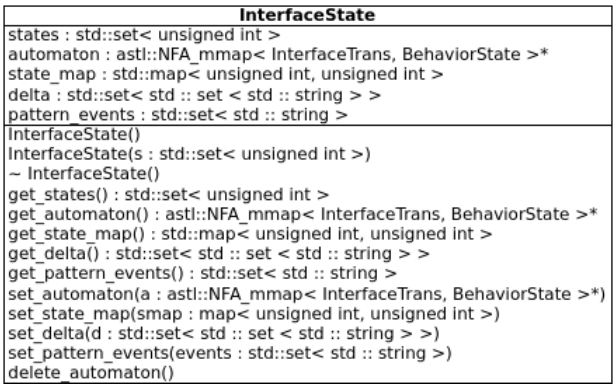


Figura 5.18: Classe InterfaceState

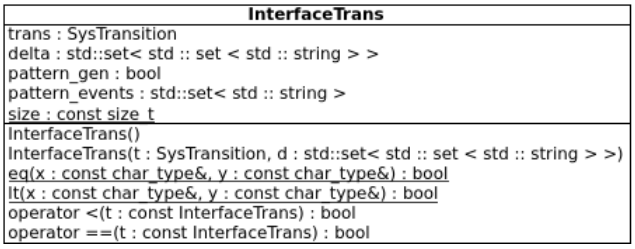


Figura 5.19: Classe InterfaceTrans

Decorazione del behavior del nodo radice

La decorazione del behavior del nodo radice avviene attraverso il metodo `decorate_lazy_bhv` della classe `Decoration`. La procedura consiste in alcune variazioni dell'algoritmo di decorazione della diagnosi greedy:

- una transizione del behavior, se relativa ad una interfaccia, contiene una diagnosi che deve essere propagata agli stati successivi;

- in corrispondenza di uno stato finale del behavior, la decorazione deve tenere conto delle diagnosi relative agli stati finali delle interfacce da cui il nodo corrente dipende.

Capitolo 6

Sperimentazione

La fase di sperimentazione è stata orientata in due principali direzioni:

- confrontare i metodi diagnostici greedy e lazy in termini di risorse computazionali utilizzate;
- analizzare i tempi di esecuzione e la memoria allocata al crescere delle dimensioni delle istanze utilizzate.

L'aspettativa riguardante l'inefficienza del metodo greedy è stata immediatamente confermata, in quanto problemi di diagnosi relativi a sistemi attivi complessi caratterizzati da un numero limitato di componenti esauriscono con estrema facilità la memoria del calcolatore. Dal momento che, come previsto, il metodo lazy è molto più efficiente e permette il calcolo delle diagnosi di sistemi molto più grandi, un'analisi di scalabilità è stata compiuta ristrettamente alla macchina diagnostica lazy. I risultati ottenuti sono estremamente soddisfacenti, in quanto hanno permesso di catturare un andamento lineare nel numero di componenti sia in termini di tempo computazionale, sia in termini di memoria occupata. Gli esperimenti svolti sono stati condotti utilizzando un calcolatore con processore *Intel Core i5* da 2.40GHz, 4GB di RAM, su sistema operativo *Linux Ubuntu 15.10*. Le istanze adottate sono caratterizzate da componenti che possiedono i modelli descritti nelle figure 3.1, 3.2 e 4.3 rispettivamente per la protezione, il breaker e la linea. Di fatto i SAC utilizzati negli esperimenti sono composizioni e variazioni di insiemi di reti elettriche connesse.

6.1 Confronto dei metodi greedy e lazy

Il confronto tra il metodo greedy e quello lazy ha prodotto i risultati riportati in tabella 6.1. Sono state utilizzate 22 istanze di sistemi attivi complessi di dimensioni che vanno da 2 a 20 componenti totali, e un numero di nodi compreso tra 1 e 9. Per le istanze piccole, come immaginabile, la differenza tra i metodi non è significativa. Il comportamento

esponenziale del metodo greedy, tuttavia, emerge chiaramente con la crescita del numero dei componenti, portando alla saturazione della memoria con 20 componenti. I risultati relativi all'ultima istanza del metodo greedy non sono infatti riportati: la saturazione della memoria causa lo swap compiuto da parte del sistema operativo e il calcolo non sembra poter terminare in un tempo accettabile. Il metodo lazy, dal canto suo, si mantiene contenuto sia nel tempo che nella memoria allocata. Le variazioni di prestazioni nell'ambito di quest'ultimo metodo sono dovute alla variazione di dimensione dei nodi del sistema.

		Greedy		Lazy	
Componenti	Nodi	Tempo	Memoria	Tempo	Memoria
2	1	1,39 ms	4,2 MB	1,03 ms	4,3 MB
3	1	4,95 ms	4,2 MB	4,64 ms	4,4 MB
3	2	1,98 ms	4,3 MB	2,45 ms	4,4 MB
4	2	4,41 ms	4,3 MB	4,69 ms	4,4 MB
5	3	4,93 ms	4,8 MB	6,45 ms	4,6 MB
6	3	19,84 ms	4,9 MB	8,90 ms	4,9 MB
7	4	23,98 ms	5,5 MB	6,38 ms	5,2 MB
8	3	81,16 ms	5,8 MB	10,82 ms	5,1 MB
9	4	96,82 ms	6,9 MB	16,52 ms	5,9 MB
9	4	0,13 s	6,6 MB	12,89 ms	5,0 MB
10	5	0,19 s	11,9 MB	19,35 ms	9,4 MB
11	6	0,45 s	12,9 MB	9,02 ms	5,8 MB
12	5	1,23 s	25,1 MB	49,63 ms	5,9 MB
13	5	2,04 s	40,8 MB	96,69 ms	11,0 MB
13	6	1,77 s	55,5 MB	76,29 ms	28,5 MB
14	7	4,78 s	82,6 MB	13,14 ms	5,8 MB
15	5	24,18 s	395,4 MB	0,33 s	11,9 MB
16	5	26,10 s	417,8 MB	0,27 s	11,0 MB
17	6	39,19 s	616,2 MB	1,56 s	38,1 MB
18	7	1,90 min	1,7 GB	30,67 ms	6,2 MB
19	8	4,50 min	3,3 GB	32,82 ms	10,5 MB
20	9			68,53 ms	9,8 MB

Tabella 6.1: Risultati dei test riguardanti il confronto dei metodi greedy e lazy

6.1.1 Tempo di esecuzione

In figura 6.1 viene analizzata, tramite un diagramma cartesiano, la differenza tra i due metodi diagnostici per quanto riguarda il tempo di calcolo. L'asse delle ascisse indica il numero di componenti dell'istanza, mentre l'asse verticale indica, in scala logaritmica, il tempo di esecuzione della diagnosi. Il metodo lazy, pur con delle variazioni in base all'istanza utilizzata, non mostra l'andamento esponenziale del metodo greedy. Ad esempio, per l'ultima istanza confrontabile di 19 componenti, il metodo greedy impiega più di 4 minuti e mezzo, mentre l'algoritmo lazy giunge a termine in soli 32.82ms. L'istanza

successiva di 20 componenti satura la memoria nel caso greedy, quindi i tempi non sono calcolabili. Per quanto riguarda il metodo lazy, invece, sono sufficienti 68.53ms.

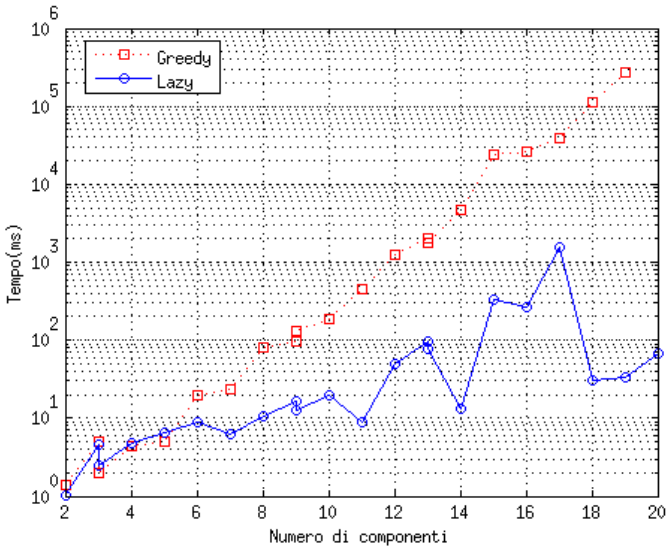


Figura 6.1: Confronto dei tempi di esecuzione tra la diagnosi greedy e lazy

6.1.2 Memoria

In figura 6.2 viene analizzata, tramite un diagramma cartesiano, la differenza tra i due metodi diagnostici per quanto riguarda la memoria allocata. L'asse delle ascisse indica il numero di componenti dell'istanza, mentre l'asse verticale indica, in scala logaritmica, la memoria. Anche in questo caso, il metodo lazy non segue l'andamento esponenziale della diagnosi greedy. Ad esempio, per l'ultima istanza confrontabile di 19 componenti, il metodo greedy occupa 3.3GB, mentre l'algoritmo lazy utilizza solamente 10.5MB. L'istanza successiva di 20 componenti satura la memoria nel caso greedy, mentre con il metodo lazy sono sufficienti meno di 10MB.

6.2 Approfondimento del metodo lazy

Una volta verificata l'impraticabilità della diagnosi greedy, è stata effettuata una serie di esperimenti volti a catturare la complessità dell'algoritmo lazy. In tabella 6.2 sono riportati i tempi di esecuzione e la memoria allocata per istanze di SAC con un numero di componenti che varia da 9 a 638, e un numero di nodi del sistema compreso tra 4 e 255. Si noti che quasi tutte le istanze utilizzate, eccetto le prime due di 9 e 18 componenti, non siano calcolabili attraverso il metodo greedy. Uno stato del behavior del SAC nella sua interezza infatti, potrebbe nel caso peggiore aver un numero di stati, considerando ad esempio l'ultima istanza, esponenziale rispetto ai 638 componenti. Per mezzo del metodo

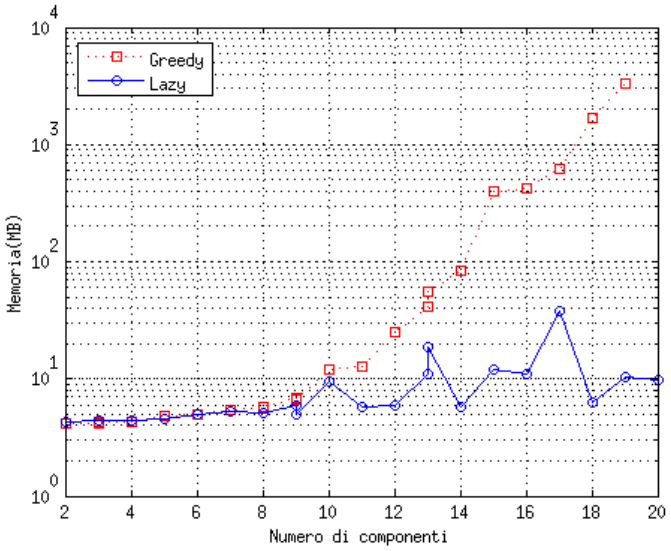
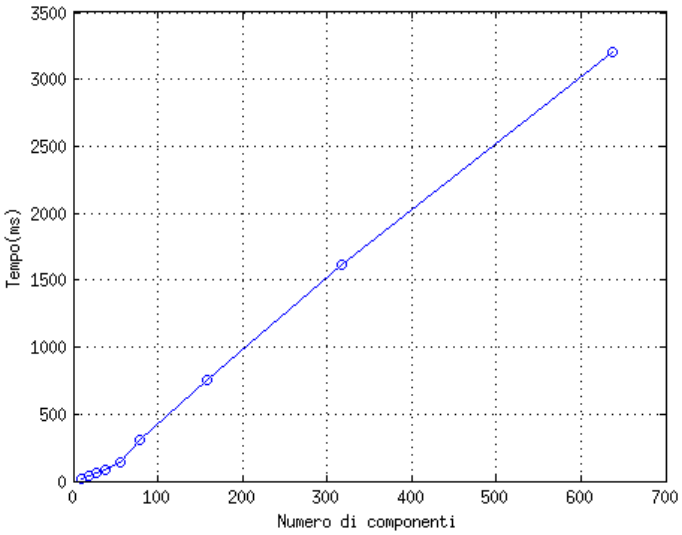


Figura 6.2: Confronto della memoria allocata tra la diagnosi greedy e lazy

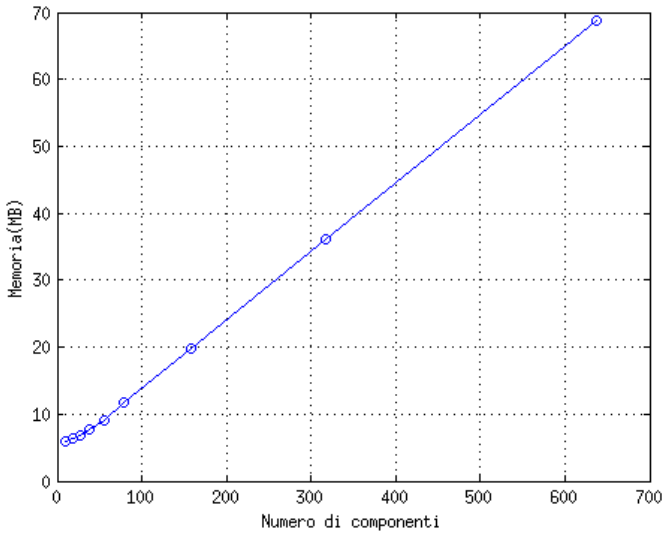
lazy, i risultati per l'istanza più grande utilizzata (638 componenti e 255 nodi) sono di 3.209s di calcolo e 68.87MB di memoria allocata. Sebbene sia chiaro che la complessità dell'algoritmo di ricostruzione del behavior sia esponenziale all'interno del singolo nodo che compone il SAC, è risultato che l'aggiunta di nodi al sistema non comporta una crescita esponenziale di risorse. Al contrario, i tempi di esecuzione si mantengono lineari nel numero di nodi del sistema e, per nodi di dimensione costante, l'andamento è lineare anche rispetto al numero di componenti. Questo andamento è facilmente osservabile nella figura 6.3(a) e 6.3(b), rispettivamente per quanto riguarda il tempo di esecuzione e la RAM utilizzata.

Componenti	Nodi	Tempo(ms)	Memoria(MB)
9	4	16.68	5.88
18	7	37.27	6.28
27	10	59.38	6.86
38	15	87.79	7.82
56	21	142	9.08
78	31	302	11.84
158	63	751	19.87
318	127	1615	36.18
638	255	3209	68.87

Tabella 6.2: Risultati dei test riguardanti il metodo lazy



(a) Tempo di esecuzione



(b) Memoria allocata

Figura 6.3: Risultati dei test riguardanti il metodo lazy

Capitolo 7

Conclusioni

Il lavoro svolto ha raggiunto gli obiettivi che erano stati prefissati. Da un lato è stato definito un nuovo insieme di sistemi a eventi discreti, chiamati sistemi attivi complessi (SAC) e ne è stata fornita una specifica formale. I SAC sono stati concepiti come estensione dei sistemi attivi tradizionali, in quanto formati da una rete di questi ultimi, secondo una topologia gerarchica. Questo modello è stato suggerito dal comportamento di molti sistemi reali che possono essere descritti a diversi livelli di astrazione. D'altro canto è stato sviluppato un software in grado di calcolare le diagnosi di questi nuovi sistemi, secondo due metodi risolutivi. Il metodo greedy è stato ottenuto come naturale ampliamento dell'algoritmo diagnostico dei sistemi attivi tradizionali, mentre il metodo lazy è stato concepito al fine di migliorarne l'efficienza. Il confronto sperimentale tra le due tecniche, infatti, ha mostrato come a fronte di una crescita delle dimensioni del sistema, l'algoritmo greedy fosse inefficiente, consumando una quantità di memoria e un tempo di esecuzione esponenziali nel numero di componenti totali. L'algoritmo lazy, invece, non possiede un comportamento siffatto, ma trae vantaggio dalla topologia del sistema e produce la diagnosi in un unico passo bottom-up lungo la gerarchia del sistema. La complessità di questo algoritmo si stabilizza alla linearità per nodi di dimensione costante, potendo risolvere problemi che attraverso il metodo greedy non sarebbero affrontabili nella pratica. A fronte di questi risultati positivi, il lavoro svolto può essere esteso in molteplici direzioni.

7.1 Sviluppi futuri

7.1.1 Parallelizzazione della diagnosi lazy

Il metodo di diagnosi lazy ricostruisce in successione i comportamenti dei singoli nodi che compongono il sistema, seguendo un ordinamento topologico dell'albero che caratterizza l'interazione dei nodi. Un ordinamento topologico totale, tuttavia, non è strettamente necessario, in quanto è sufficiente fornire un ordinamento parziale. Il calcolo dei behavior non vincolati è eseguito uno indipendentemente dall'altro; questo suggerisce in modo naturale una parallelizzazione in fase diagnostica. Una volta ricostruiti i comportamenti dei nodi foglia, allo stesso modo il livello superiore della gerarchia è composto da nodi che, sebbene siano vincolati dal comportamento dei nodi sottostanti, non sono vincolati tra loro. Anche la ricostruzione del loro comportamento, quindi, può avvenire in parallelo. Procedendo in questo modo, è possibile effettuare una diagnosi in parallelo sui diversi nodi appartenenti allo stesso livello dell'albero. Questo metodo se implementato, causerebbe un ulteriore miglioramento nelle prestazioni del calcolo, rendendo l'algoritmo linearmente dipendente dal numero di livelli nella topologia del sistema, piuttosto che dal numero di nodi totale.

7.1.2 Aumento del preprocessing

La fase di preprocessing permette di effettuare dei calcoli in fase di modellazione, evitando di spendere risorse nella successiva fase di diagnosi. L'aumento del calcolo in fase precedente potrebbe consistere nella generazione delle interfacce, la quale può richiedere delle risorse computazionali in alcuni casi costose, trattandosi di un algoritmo analogo alla subset construction. Attualmente le interfacce sono generate a partire dai behavior dei nodi, ma nulla vieta che tali interfacce possano essere generate partendo dai behavior space, che descrivono il comportamento prescindendo dall'osservazione, nota soltanto nella successiva diagnosi. Sebbene la generazione del behavior space è stata scartata nel metodo greedy poiché non praticabile nemmeno offline, la costruzione del behavior space del singolo nodo non dovrebbe essere altrettanto estenuante, assumendo una dimensione del singolo nodo limitata. In fase diagnostica le transizioni di interfaccia attuabili dovranno però essere selezionate in base all'osservazione data: parte delle interfacce costituirà una evoluzione inconsistente con l'osservazione temporale e andrà scartata. Si noti che in letteratura sono presenti metodi diagnostici che calcolano un automa, detto diagnosticatore, derivato dal behavior space dell'intero sistema [9, 10]. Sebbene questo renda la successiva diagnosi lineare, la generazione di tale automa offline non è praticabile per problemi di dimensione reale.

7.1.3 Variazioni nella topologia del sistema

Nell'ambito di questo lavoro di tesi è stato analizzato il caso di sistemi attivi connessi tra loro in maniera gerarchica, formando un albero. Questa topologia particolare, tuttavia, può essere facilmente estesa al caso più generale di grafo aciclico. Anche in questo contesto è possibile ricostruire il comportamento dei nodi non vincolati, proseguendo in modo analogo a quanto descritto nella trattazione. Un'attenzione particolare deve essere prestata, tuttavia, nel caso in cui un nodo abbia più di un genitore, cioè influenzi più di un altro nodo del sistema. In questo caso la ricostruzione dei behavior dei nodi genitori deve essere effettuata in maniera sincronizzata, nel senso che un cambiamento di stato dell'interfaccia relativa al nodo figlio deve valere contemporaneamente per tutti i nodi padre, altrimenti la diagnosi potrebbe avere delle soluzioni aggiuntive non sound. La ricostruzione prosegue allo stesso modo del caso di topologia ad albero: se vi sono più nodi radice, il behavior di ognuno di questi deve essere decorato, e le rispettive diagnosi combinate tra loro.
Un problema più complicato si verifica quando il sistema costituisce un generico grafo ciclico. In questo caso non è possibile effettuare semplicemente un processo bottom-up. Il procedimento consiste nella scelta di un nodo di partenza, ricostruendone il comportamento relativo e la corrispondente interfaccia; successivamente, in caso esso sia influenzato da un altro nodo, il comportamento e l'interfaccia verranno potati opportunamente, in modo da essere consistenti con i pattern event ricevuti. Il processo quindi si sviluppa percorrendo i link che collegano i nodi del sistema un numero finito di volte, fino a quando il pruning non raggiunge una situazione di stabilità.

7.1.4 Osservazioni incerte

In questo lavoro, le osservazioni dei nodi sono assunte essere sequenze lineari di label osservabili. In sistemi reali, tuttavia, un'osservazione potrebbe consistere in un ordinamento parziale delle label ricevute, anziché un ordinamento totale. In questo caso l'osservazione sarebbe percepita come un grafo orientato aciclico, invece di una semplice sequenza. Altri tipi di incertezze potrebbero riguardare il contenuto dell'osservazione: una label incerta potrebbe essere presente o meno nell'osservazione. Una volta delineato il grafo aciclico relativo all'osservazione, la consumazione di essa può essere monitorata generando un corrispondente automa, chiamato *index space* [2, 11], nelle cui transizioni sono presenti le label dell'osservazione.

7.1.5 Monitoring

Nell'ambito di questo lavoro il focus è sulla cosiddetta diagnosi a posteriori, la quale avviene dopo un'osservazione completa del sistema, a seguito della quale viene calcolato l'insieme di diagnosi candidate. In un sistema reale come ad esempio una centrale nucleare o una rete elettrica nazionale, il sistema difficilmente può essere pensato in uno stato

quiescente a seguito del quale operare la diagnosi. L'osservazione in questi casi verrà ricevuta passo dopo passo e il compito della diagnosi (detta in questo ambito *monitoring*) consiste nel generare le diagnosi candidate in tempo reale durante l'evoluzione del sistema, a seguito di una singola label di osservazione. Estendendo la ricerca in tale direzione si potrebbe dare un metodo utile in applicazioni reali di questo tipo.

Appendice A

BNF del linguaggio di specifica

É di seguito riportata la grammatica, in notazione BNF, del linguaggio di specifica adottato per i sistemi attivi complessi. Si noti che un file di specifica può avere delle direttive `#include` (non coinvolte nella definizione della grammatica), gestite da un preprocessore, in modo da incorporare delle parti di specifica contenute in altri file.

```
specification → spec_list
spec_list → spec . spec_list | spec .
spec → model_decl | system_decl | problem_decl
model_decl → comp_model_decl | net_model_decl

comp_model_decl → component model id is
                                    event_decl
                                    input_decl
                                    output_decl
                                    state_decl
                                    transition_decl
                                end id
event_decl → event id_list ;
id_list → id , id_list | id
input_decl → input id_list ; | ε
output_decl → output id_list ; | ε
state_decl → state id_list ;
transition_decl → transition trans_decl_list ;
trans_decl_list→ trans_decl , trans_decl_list | trans_decl
trans_decl → id = event , id -> id , { opt_ref_list }
event → ref | ( )
ref → id ( id )
opt_ref_list → ref_list | ε
```

```
ref_list → ref , ref_list | ref

net_model_decl → network model id is
                        component_section
                        input_decl
                        output_decl
                        link_section
                        pattern_section
                        initial_section
                        viewer_section
                        ruler_section
                end id
component_section → component decl_list ;
decl_list → decl , decl_list | decl
decl → id_list : id
link_section → link link_list ; | ε
link_list→ link_decl , link_list | link_decl
link_decl → ref -> ref
pattern_section → pattern pattern_list ; | ε
pattern_list→ pattern_decl , pattern_list | pattern_decl
pattern_decl → ref pattern_op expr
pattern_op → = | ==
expr → expr '|' term | term
term → term factor | term & factor | factor
factor → factor * | factor + | factor ? | ( expr ) | ~ ref | ref
initial_section → initial ref_list ; | ε
viewer_section → viewer map_list ; | ε
map_list → map_decl , map_list | map_decl
map_decl → ref -> id
ruler_section → ruler map_list ; | ε

system_decl → system id is
                system_node_
                list emergence_section
        end id
system_node_list → system_node, system_node_list | system_node ;
system_node → node id: id is
                        initial_section
                        viewer_section
```

```
                        ruler_section
                end id
emergence_section → emergence link_list ; | ε

problem_decl → problem id is problem_node_list end id
problem_node_list → problem_node, problem_node_list
                | problem_node ;
problem_node → node id is
                        initial_section
                        viewer_section
                        obs_section
                        ruler_section
                end id
obs_section → obs [ opt_id_list ] ;
opt_id_list → id_list | ε
```

Appendice B

Esempio di specifica

In questa appendice è riportata la specifica del problema diagnostico dell'esempio 4.8.

B.1 Modelli dei componenti

```
component model Breaker is
        event op, cl;
        input I;
        state closed, open;
        transition b1 = op(I), closed -> open, {},
                   b2 = cl(I), open -> closed, {},
                   b3 = op(I), closed -> closed, {},
                   b4 = cl(I), open -> open, {},
                   b5 = cl(I), closed -> closed, {},
                   b6 = op(I), open -> open, {};
end Breaker.

component model Protection is
        event op, cl;
        output O;
        state idle, awaken;
        transition p1 = (), idle -> awaken, {op(O)},
                   p2 = (), awaken -> idle, {cl(O)},
                   p3 = (), idle -> awaken, {cl(O)},
                   p4 = (), awaken -> idle, {op(O)};
end Protection.
```

```
component model Monitor is
        event di, co, nd, nc, rc, op;
        input E, I;
        output O, R;
        state watch, trigger;
        transition
                m1  = nd(E), watch -> trigger, {op(R), rc(O)},
                m2  = nd(E), watch -> trigger, {op(R)},
                m3  = rc(I), watch -> trigger, {op(R)},
                m4  = nd(E), trigger -> trigger, {},
                m5  = di(E), watch -> watch, {},
                m6  = co(E), watch -> watch, {},
                m7  = nc(E), watch -> watch, {},
                m8  = di(E), trigger -> trigger, {},
                m9  = co(E), trigger -> trigger, {},
                m10 = nc(E), trigger -> trigger, {};
end Monitor.

component model Line is
        event ni, nr, ps;
        input E;
        state normal, shorted, isolated, persistent;
        transition l1 = ni(E), normal -> shorted,    {},
                   l2 = nr(E), normal -> isolated,   {},
                   l3 = ps(E), normal -> persistent, {};
end Line.
```

B.2 Modelli dei nodi

```
network model ProtectionHardware is
        component b: Breaker, p: Protection;
        output O;
        link O(p)->I(b);
        pattern di(O) == b1(b),
                co(O) == b2(b),
                nd(O) == p3(p) | p1(p)b3(b),
                nc(O) == p4(p) | p2(p)b4(b);
        initial closed(b), idle(p);
end ProtectionHardware.
```

```
network model MonitorApparatus is
        component m1, m2: Monitor, r1, r2: Breaker;
        input Im1, Im2;
        output O;
        link Im1(this)->E(m1), Im2(this)->E(m2),
             O(m1)->I(m2), O(m2)->I(m1),
             R(m1)->I(r1), R(m2)->I(r2);
        pattern nr(O) = m7(m1)|m10(m1)|b1(r1)|
                        m7(m2)|m10(m2)|b1(r2),
                        ni(O)=(m1(m1)|m2(m1)|m4(m1))b3(r1)|b3(r1)m4(m1)
                        |(m1(m2)|m2(m2)|m4(m2))b3(r2)|b3(r2)m4(m2),
                        ps(O) = (m6(m1)m6(m2)|m6(m2)m6(m1))
                        (m5(m1)m5(m2)|m5(m2)m5(m1));
        initial closed(r1), closed(r2), watch(m1), watch(m2);
        viewer m1(m1)->trg1, m2(m1)->trg1, m3(m1)->trg1,
               b1(r1)->opr1, b2(r1)->clr1,
               m1(m2)->trg2, m2(m2)->trg2, m3(m2)->trg2,
               b1(r2)->opr2, b2(r2)->clr2;
        ruler  m2(m1)->fm1, m2(m2)->fm2,
               b3(r1)->for1, b4(r1)->fcr1,
               b3(r2)->for2, b4(r2)->fcr2;
end MonitorApparatus.

network model LineNet is
        component l: Line;
        input I;
        link I(this)->E(l);
        initial normal(l);
        ruler l1(l)->fls, l2(l)->fli, l3(l)->flp;
end LineNet.
```

B.3 Sistema

```
system PowerTransmission is
        node L: LineNet is
        end L,
        node M: MonitorApparatus is
        end M,
```

```
        node W1: ProtectionHardware is
                viewer b1(b)->opb1, b2(b)->clb1,
                       b5(b)->alr1, b6(b)->alr1,
                       p1(p)->awk1, p2(p)->ide1,
                       p3(p)->awk1, p4(p)->ide1;
                ruler  b3(b)->fob1, b4(b)->fcb1,
                       p3(p)->fos1, p4(p)->fcs1;
        end W1,
        node W2: ProtectionHardware is
                viewer b1(b)->opb2, b2(b)->clb2,
                       b5(b)->alr2, b6(b)->alr2,
                       p1(p)->awk2, p2(p)->ide2,
                       p3(p)->awk2, p4(p)->ide2;
                ruler  b3(b)->fob2, b4(b)->fcb2,
                       p3(p)->fos2, p4(p)->fcs2;
        end W2;
        emergence O(W1)->Im1(M), O(W2)->Im2(M), O(M)->I(L);
end PowerTransmission.
```

B.4 Problema

```
problem Example is
        node L is
                obs [ ];
        end L,
        node M is
                obs [ trg1, opr1 ];
        end M,
        node W1 is
                obs [ awk1 ];
        end W1,
        node W2 is
                obs [ awk2, opb2 ];
        end W2;
end Example.
```

Appendice C

Istruzioni di installazione del software

Il software sviluppato è distribuito nell'archivio `CASdiagnosis.tar.gz`. Nell'appendice corrente è descritto come compilare ed eseguire il codice relativo, in ambiente Linux. Per poter compilare il codice, una volta copiato l'archivio nella cartella desiderata, digitare da terminale:

```
$ tar -zxvf CASdiagnosis.tar.gz
```

Quindi, per accedere alla cartella creata, dare il comando:

```
$ cd CASdiagnosis/
```

All'interno della cartella, dare il comando:

```
$ make
```

per compilare l'intero programma. Una volta portata a termine la procedura, la cartella principale del programma conterrà le seguenti sottocartelle:

- `SpecificationLanguage`, contenente il codice del compilatore offline;
- `GreedyDiagnosis`, contenente il codice della macchina diagnostica greedy;
- `LazyDiagnosis`, contenente il codice del metodo lazy;
- `bin`, contenente i file eseguibili;
- `doc`, con la documentazione generata da *doxygen*;
- `Graphs`, contenente le rappresentazioni degli automi generati in fase diagnostica;
- `CompiledData`, contenente le informazioni del file di specifica compilate in fase offline;

- `Libraries`, contenente le librerie esterne utilizzate dal software;
- `InputFiles`, dove sono collocati alcuni file di specifica di esempio.

Lo script `compareLazyGreedy.sh` (presente nella cartella principale) permette, passando un parametro indicante il nome del file di specifica, di effettuare un confronto tra i due metodi diagnostici. Per ulteriori informazioni, visionare i file `README` presenti nelle cartelle.

Elenco delle figure

Bibliografia

[1] Aho A., Lam M., Sethi R., and Ullman J. *Compilers. Principles, Techniques and Tools*. Pearson, 2006.

[2] Lamperti G. and Zanella M. *Diagnosis of active systems. Principles and Techniques*. Kluwer Academic Publishers, 2003.

[3] Cassandras C. and Lafortune S. *Introduction to Discrete Event Systems*. Springer, 2008.

[4] Le Maout V. Astl, the automata standard template library. `http://astl.sourceforge.net/`, 2015.

[5] AT&T Research Labs. Graphviz - graph visualization software. `http://www.graphviz.org/`, 2015.

[6] Levine J. *Flex & Bison*. O'Really Media, 2009.

[7] Department of Computer Science University of Western Ontario(Canada). The grail+ project. `http://ftp.csd.uwo.ca/Research/grail/index.html`, 2002.

[8] Dawes B., Abrahams D., and Rivera R. Boost c++ libraries. `http://www.boost.org/`, 2015.

[9] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D.C. Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, 1995.

[10] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D.C. Teneketzis. Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology*, 4(2):105–124, 1996.

[11] G. Lamperti and M. Zanella. Diagnosis of discrete-event systems from uncertain temporal observations. *Artificial Intelligence*, 137(1–2):91–163, 2002.

[12] Lamperti G. and Zhao X. Diagnosis of active systems by semantic patterns. In *IEEE Transactions on Systems, Man, and Cybernetics: systems, vol.44, no.8*, 2014.

[13] Lamperti G. and Zhao X. Diagnosis of higher-order discrete-event systems. In *A. Cuzzocrea et al.(Eds.):CD-ARES 2013, LNCS 8127, pp.162-177*, 2013.

[14] Lamperti G. Compilers. `http://gianfranco-lamperti.unibs.it/co/co.html`, 2015.