POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di laurea specialistica in Ingegneria Informatica

Algoritmi Efficienti per Model Checking Parametrico di Modelli Markoviani Discreti

Relatore: **Prof. Carlo Ghezzi**
Correlatore: **Ing. Antonio Filieri**

Tesi di laurea di
Marco BUSSI
matr. n° 735359

Anno Accademico 2010/2011

Non perdere mai di vista ciò che intendi raggiungere, non perdere mai di vista chi sei tu in realtà..
(Sergio Bambarén)

Ai miei genitori che mi hanno sostenuto in questo percorso,
A Rachele che mi sostiene,
A Michele che mi ha aiutato nella realizzazione di questo progetto,
A tutti i miei amici.

Ringrazio Antonio, per le idee, i confronti ed i consigli che mi hanno permesso di concludere questo cammino iniziato insieme....

Indice

Elenco delle figure

Elenco dei Grafici

Sommario

I cambiamenti imprevedibili affliggono continuamente i sistemi software e possono causare gravi conseguenze, compromettendo in modo irrimediabile la qualità del servizio, paralizzando il sistema, causando in certi casi la violazione dei requisiti del software stesso. Questi cambiamenti possono coinvolgere componenti critici del sistema, profili operazionali e requisiti arrivando talvolta agli ambienti di sviluppo. L'adozione di modelli software e tecniche di Model Checking a run-time, potrebbe essere utile per supportare sistemi automatici di reasoning nella valutazione di questi cambiamenti: la rilevazione di configurazioni dannose permetterebbe così di reagire, attivando appropriate contromisure automatiche. Le tecniche di MC tradizionali e i tools a loro associati non sono adatti, per come sono progettatti attualmente, ad essere utilizzati a run-time, poiché non sono in grado di soddisfare i vincoli rigidi imposti dall'esecuzione on-the-fly (al volo) in termini di memoria e tempo di esecuzione. In questo elaborato verranno proposti metodi formali di Model Checking per valutare vincoli di affidabilità su sistemi modelizzabili come Catene di Markov a Tempo Discreto Parametriche (PMC). E' stato sviluppato un software basato sul calcolo algebrico matriciale che, dato un vincolo di affidabilità ed un modello del sistema, restituisce un insieme di espressioni che possono essere utilizzate per una efficiente verifica dei requisiti a run-time. Verrà inoltre effettuato un confronto dell'approccio qui proposto con i Model Checker probabilistici esistenti mostrando come ognuno di questi sia utilizzabile nel campo della verifica a run-time.

Capitolo 1

Introduzione

Ai giorni nostri la società è sempre più dipendente da computer e sistemi software dedicati che ormai coinvolgono quasi ogni aspetto della vita di tutti i giorni; molte funzioni di controllo nelle automobili moderne: airbag, ABS, cruise control per esempio; dispositivi audio video, medicali contengono spesso al loro interno sistemi software embedded dedicati, con l'obiettivo di semplificare l'utilizzo dei dispositivi guadagnando in efficienza e riducendo i costi. Un elemento comune alle soluzioni informatiche odierne è senza ombra di dubbio l'aumento della complessità, accelerata anche dall'impiego di soluzioni di networking (wired e wireless). Queste architetture vengono impiegate universalmente anche in sistemi safety-critical; una delle principali sfide per il settore dell'informatica è quella di fornire formalismi, tecniche, strumenti che permettano una progettazione corretta, efficace e ben funzionante in modo del tutto indipendente dalla complessità del sistema stesso. In questi sistemi diventa evidente come la progettazione e la gestione della totalità dei componenti sia più complessa della gestione delle singole parti che lo compongono.

Le principali motivazioni per impiegare queste metodologie di verifica sono, come spesso accade, di natura economica. E' importante cercare e correggere i difetti di progettazione il prima possibile poiché i costi di riparazione di un difetto presente nel software, rilevato durante la fase di manutenzione, so-

no circa 500 volte maggiori rispetto a una correzione in fase di progettazione iniziale (Figura 1.1). Nella progettazione di sistemi sia software che hardware complessi, lo sforzo ed il tempo impiegati per la verifica sono maggiori di quelli spesi per la realizzazione; impiegare tecniche in grado di rilevare errori in modo automatizzato attraverso metodi formali, riduce gli sforzi necessari per la verifica aumentando, allo stesso tempo, la copertura, così dà ottenere un miglior risultato con un impiego minore di risorse grazie alle potenzialità ed al rigore della matematica.

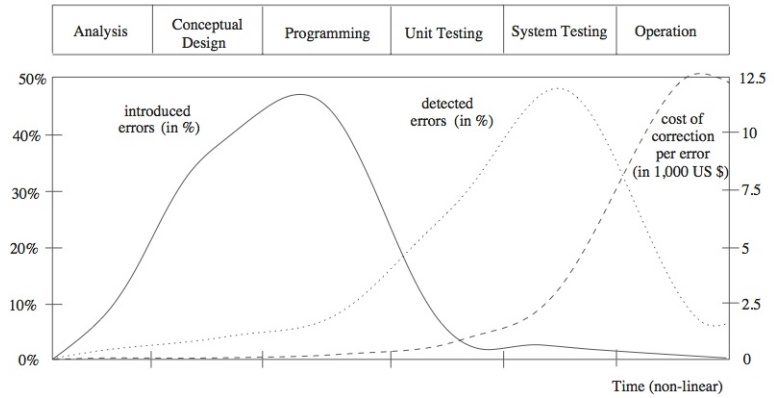

Figura 1.1: Ciclo di vita del SW: introduzione di errori, rilvamento e costi di riparazione.

Negli ultimi due decenni un approccio molto interessante per la verifica della correttezza di software basati su sistemi di controllo ha preso piede: il Model Checking. Il Model Checking (MC) è una tecnica di validazione formale che consente di verificare rigorosamente una proprietà comportamentale, su un dato sistema, attraverso l'ispezione sistematica di tutti gli stati che il modello può assumere; più formalmente il Model Checking è una tecnica automatizzata che, dato un modello a stati finiti di un sistema e una proprietà formale, controlla sistematicamente se questa proprietà vale per (un determinato stato in) quel modello. La principale peculiarità del MC è insita nella sua natura: è completamente automatizzato e fornisce anche dei con-

troesempi nel caso in cui il modello non sia in grado di soddisfare una data proprietà, informazione utilissima in fase di debugging.

Il MC si divide in tre fasi distinte: Modellazione, Esecuzione ed Analisi. La fase di Modellazione prevede di definire il sistema mediante il linguaggio proprio del Model Checker, con la relativa formalizzazione delle proprietà che si intende verificare. La definizione di un modello permette di fornire una descrizione delle caratteristiche salienti del sistema senza occuparsi degli aspetti implementativi; in fase di progetto è utile poter riflettere su tali caratteristiche, per valutare se soddisfano i requisiti richiesti o se è necessario modificarle. Il vantaggio è quello di poter manipolare un modello astratto anziché dover implementare un sistema che potrebbe contenere errori tali da richiedere la riprogettazione dell'intero sistema o di alcune sue parti. Durante la fase di esecuzione il MC verifica che il modello soddisfi le proprietà. L'analisi è la fase di valutazione: La proprietà è soddisfatta? In caso affermativo si valuta la successiva (se presente), in caso contrario si analizza il controesempio, si modifica il modello e si riprende l'intera procedura da capo. A questo punto una domanda nasce spontanea: Che tipo di proprietà è possibile verificare attraverso il MC? Qualsiasi proprietà esprimibile mediante logica, come ad esempio, deadlock, mutua e esclusione. Esistono vari tipi di logiche che si prestano a questo scopo: Linear Temporal Logic (LTL), Computation Tree Logic (CTL), Probabilistic CTL (PCTL), CTL*, Dinamic Logic (DL) e molte altre.

La logica PCTL è basata su un' estensione della semantica di Kripke; oltre ad avere un insieme di stati ed una relazione di raggiungibilità, ad ogni arco appartenente a tale relazione viene associata una probabilità. Una rete di questo tipo è detta Discrete Time Markov Chain, acronimo DTMC. Tale rappresentazione risulta essere estremamente utile poiché permette di avere una visione semplificata del sistema, in cui ogni stato della catena di Markov rappresenta uno stato del sistema ed ogni arco tra stati definisce la funzione di transizione. Inoltre i DTMC godono di interessanti proprietà come ad esempio l'assenza di memoria. In questo elaborato verrà proposto un

Model Checker per verifiche a run-time basato su un'estensione del linguaggio *PRISM* [8], estensione che prevede l'introduzione di parametri in sostituzione ai soli valori numerici associati agli archi di transizione, con l'obiettivo di fornire un metodo di risoluzione esatto basato su numeri razionali, alternativo a quello adottato da *Param V1.8* [3], attualmente unico Model Checker in grado di verificare proprietà su DTMC parametriche.

Il documento viene organizzato come segue. Il secondo capitolo tratta le DTMC, focalizzando l'attenzione sulla loro definizione e sulle loro proprietà, introduce la logica PCTL: sintassi e proprietà costitutive e fornisce una breve introduzione a *Maple* [7]. Il terzo capitolo descrive il problema del run-time Model Checking, discute lo stato dell'arte, arrivando alla descrizione di Param con i suoi punti di forza debolezza. Il quarto capitolo è centrato sul problema del *Run-Time efficient Model Checking* [1], è composto da una breve introduzione fino ad arrivare a descrivere in modo formale il concetto di WorkingMom. Nel quinto capitolo vengono presentati gli algoritmi di verifica implementati in Maple arrivando a definire la loro collocazione all'interno del run-time Model Checker. Il sesto capitolo contiene al suo interno una descrizione dei case studies eseguiti per la validazione ed un confronto diretto con Param sull'analisi delle performance. Il documento si conclude esprimendo conclusioni ed eventuali sviluppi futuri.

Capitolo 2

Background

In questo capitolo verranno prima introdotte le *Catene di Markov a tempo discreto* (DTMC), formalismo molto utile per la descrizione di eventi probabilistici, e saranno presentate le proprietà tipiche che caratterizzano questo tipo di modello stocastico. Nella seconda parte si parlerà di *Probabilistic Computation Tree Logic*, una logica probabilistica, che permette di fare inferenza su sistemi modelizzabili attraverso DTMC; il capitolo verrà concluso da una breve introduzione a Maple, linguaggio di programmazione orientato al calcolo numerico che verrà largamente impiegato in questo elaborato per la risoluzione di problemi di algebra lineare.

2.1 Catene di Markov a tempo discreto

Una catena di Markov è un processo stocastico contraddistinto dalla proprietà di assenza di memoria, in altre parole, lo stato futuro in cui il processo evolverà dipende solo ed esclusivamente dallo stato corrente, indipendentemente da quello che è successo nel passato. Un processo stocastico è un modello matematico per lo sviluppo casuale nel tempo. Da un punto di vista formale, un processo stocastico a tempo discreto è una sequenza $\{X_n : n = 0, 1, 2, ...\}$ di variabili casuali. Il valore della variabile casuale X_n è interpretato come lo stato di uno sviluppo casuale dopo n step. Una catena

di Markov, inoltre, è un processo casuale discreto che gode della proprietà di Markov. Un processo di questo tipo è un sistema che si trova in un determinato stato ad ogni step temporale in maniera tale per cui lo stato cambi casualmente da uno step all'altro. Gli step possono essere definiti con diverse unità di misura discrete (intervalli temporali prefissati, numeri interi o naturali, ...). I processi di Markov rappresentano senza alcun dubbio la classe più nota di sistemi dinamici di tipo stocastici.

2.1.1 Definizione e Proprietà di base

Distribuzione

Supponiamo I come un insieme discreto di dimensione finita o per lo meno costituito da una infinità di elementi numerabili. Ciascun $i \in I$ è definito col nome di *Stato* mentre I è chiamato *Spazio degli Stati*. Sia $\lambda = (\lambda_i : i \in I)$ una misura su I se $0 \leq \lambda_i \leq \infty$ per tutti gli $i \in I$. Se in aggiunta si verifica che $\sum_{i \in I} \lambda_i = 1$, allora λ prende il nome di *distribuzione*. Per ipotesi si lavora nello spazio di probabilità $(\Omega, \mathcal{F}, P)$. Ovviamente, la *variabile casuale* X con valori in I è una funzione del tipo $X : \Omega \to I$.
Supponiamo di definire

$$\lambda_i = P(X = i) = P(\omega : X(\omega) = i).$$

Allora è possibile definire λ come una *distribuzione di* X ed è possibile pensare a X come un modello di uno stato casuale che prende il valore i con probabilità λ_i.

Matrice di Transizione

Per semplicità, si consideri inizialmente la matrice di transizione ad un singolo passo. Tale matrice può essere così rappresentata:

$$P = \begin{pmatrix} p_{0,0} & \cdots & p_{0,n} \\ p_{1,0} & \cdots & p_{1,n} \\ & \vdots & \\ p_{n,0} & \cdots & p_{n,n} \end{pmatrix}$$

Per convenzione si definisce ogni singolo elemento della matrice di transizione P come segue:

$$P = (p_{i,j}) : i, j \in I.$$

Tale matrice consente di definire per ogni elemento $p_{i,j}$ la probabilità (pari a $p_{i,j}$) che ha il sistema di pervenire nello stato successivo j, essendo partito dallo stato attuale i. La matrice P deve essere *stocastica*, quindi deve valere quanto segue:

$$\forall i \in I, \sum_{j \in I} p_{i,j} = 1.$$

Molto più semplicemente è necessario che ogni riga della matrice di transizione P debba essere una distribuzione.

A puro titolo informativo, si fornisce un esempio tramite il quale è possibile capire come realizzare una matrice di transizione a partire da un semplice sistema.

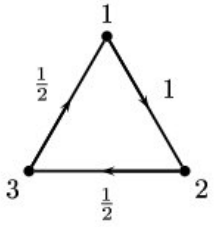

Figura 2.1: Esempio di Catena di Markov.

Dalla figura 2.1 si evince che gli stati vengono rappresentati coi numeri interi 1, 2 e 3, mentre le probabilità di passare da uno stato ad un altro sono

rappresentate sugli archi direzionati che collegano gli stati. Dalla Figura 1.1 si può ricavare facilmente la matrice di transizione P, come segue:

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \end{pmatrix}.$$

Ogni elemento $p_{i,j}$ della matrice di transizione P è ricavato considerando come i lo stato di partenza, j lo stato di arrivo e $p_{i,j}$ come valore di probabilità associata all'arco che collega gli stati i e j. Qualora non vi siano archi che colleghino due stati, il valore $p_{i,j}$ associato è pari a 0, in quanto non vi è alcuna possibilità di passare dallo stato i allo stato j. Infine, resta da considerare il caso in cui vi sia la possibilità di rimanere nello stato attuale e la probabilità risulta essere calcolata nel seguente modo:

$$p_{i,i} = 1 - \sum_{j \in I, j \neq i} p_{i,j}, \forall i \in I.$$

Questa scelta è stata fatta in maniera tale per cui ciascuna riga della matrice di transizione P possa essere una distribuzione e, quindi, fare in modo che la matrice stessa sia stocastica.

2.1.2 Definizione della Catena di Markov

È ora possibile definire in maniera più formale una *Catena di Markov* tramite una definizione in termini di corrispondente matrice di transizione P. Si può affermare che $(X_n)_{n \geq 0}$ è una catena di Markov con *distribuzione* λ e *matrice di transizione* P se

1. X_0 ha distribuzione λ;
2. per $n \geq 0$, dato che lo stato attuale è $X_n = i$ e quello successivo sarà $X_{n+1} = j$, si deve avere una distribuzione $p_{i,j}$ che sia indipendentemente da i precedenti $X_0, X_1, ..., X_{n-1}$.

Queste condizioni, però, possono esprimere molto più esplicitamente aspetti di notevole rilevanza. Considerando $n \geq 0$ e $i_0, i_1, ..., i_{n+1} \in I$, si possono interpretare quelle condizioni come segue:

1. $P(X_0 = i_0) = \lambda_{i_0}$;

2. $P(X_{n+1} = i_{n+1}|X_0 = i_0, ..., X_0 = i_0) = p_{i_n} p_{i_{n+1}}$.

È possibile affermare, in presenza di tali condizioni, che $(X_n)_{n\geq 0}$ è *catena di Markov* (λ, P). Inoltre, considerando un numero N finito di variabili casuali che soddisfino sempre le condizioni (i) e (ii), allora si può sempre affermare che $(X_n)_{0\leq n\leq N}$ è una *catena di Markov* (λ, P). A questo punto si possiedono tutti gli strumenti per formalizzare il seguente teorema:

Teorema 2.1.1.
Un processo casuale a tempo discreto $(X_n)_{0\leq n\leq N}$ *è una catena di Markov* (λ, P) *se e solo se per tutti gli* $i_0, i_1, ..., i_N \in I$ *si ha che*

$$P(X_0 = i_0, X_1 = i_1, ..., X_N = i_N) = \lambda_{i_0} p_{i_0 i_1} p_{i_1 i_2} ... p_{i_{N-1} i_N}$$

Del teorema appena descritto è possibile anche citare la sua dimostrazione in quanto consente di capire il meccanismo di costruzione di una *catena di Markov*.

Dimostrazione. Se si suppone che $(X_n)_{0\leq n\leq N}$ sia una *Markov* (λ, P), allora è possibile verificare che
$P(X_0 = i_0, X_1 = i_1, ..., X_N = i_N) =$
$= P(X_0 = i_0)P(X_1 = i_1|X_0 = i_0)...P(X_N = i_N|X_0 = i_0, X_1 = i_1, ...$
$X_{N-1} = i_{N-1}) = \lambda_{i_0} p_{i_0 i_1} ... p_{i_{N-1} i_N}$

Analogamente, se la dimostrazione vale per un numero N finito, fissato $i_N \in I$ e sfruttando sempre che $\sum_{j\in I} p_{i,j} = 1$, si può intuire che la dimostrazione vale anche per un numero finito $N - 1$ e allo stesso tempo, per induzione, la proprietà

$$P(X_0 = i_0, X_1 = i_1, ..., X_n = i_n) = \lambda_{i_0} p_{i_0 i_1} ... p_{i_{n-1} i_n}$$

per ogni $n = 0, 1, ..., N$. In particolare, dato che $P(X_0 = i_0) = \lambda_{i_0}$ e per $n = 0, 1, ..., N - 1$, si può affermare che

$P(X_{n+1} = i_{n+1}|X_0 = i_0, ..., X_n = i_n) =$
$= P(X_0 = i_0, ..., X_n = i_n, X_{n+1} = i_{n+1})/P(X_0 = i_0, ..., X_n = i_n) =$
$= p_{i_n} p_{i_{n+1}}$

Allora è possibile affermare che $(X_n)_{0 \leq n \leq N}$ è *Markov* (λ, P). □

La dimostrazione mostra chiaramente quanto sia importante la proprietà di assenza di memoria nella catene di Markov. Infatti, è possibile verificare come lo stato futuro sia influenzato solo ed esclusivamente dallo stato attuale e non in alcun modo dalla storia passata del sistema.

2.1.3 Proprietà di Markov

Il prossimo passaggio è quello di rafforzare l'idea di assenza di memoria che le catene di Markov possiedono con l'ausilio del seguente teorema. Definendo ad esempio $\delta_i = (\delta_{i,j} : j \in I)$ per l'unità di massa su i dove

$$\delta_{i,j} = \begin{cases} 1 & se \quad i = j \\ 0 & altrimenti \end{cases}$$

È possibile allora definire il teorema che segue.

Teorema 2.1.2.
Si consideri $(X_n)_{n \geq 0}$ *essere una Markov* (λ, P). *Data la condizione* $X_m = i$, $(X_{n+m})_{n \geq 0}$ *è una catena di Markov* (δ_i, P) *ed è indipendente dalle variabili casuali* $X_0, X_1, ..., X_m$.

È possibile avvalersi della seguente dimostrazione per capire meglio quanto affermato dal teorema appena espresso.

Dimostrazione. Occorre dimostrare che per ogni evento A determinato da $X_0, ..., X_m$ si ha che

$$P(\{X_m = i_m, ..., X_{n+m} = i_{n+m}\} \cap A | X_m = i) =$$

$$= \delta_{ii_m} p_{i_m i_{m+1}} ... p_{i_{m+n-1} i_{m+n}} P(A | X_m = i)$$

Prima di tutto si consideri il caso di eventi elementari

$$A = \{X_0 = i_0, ..., X_m = i_m\}$$

In quel caso si deve mostrare che

$$P(X_0 = i_0, ..., X_{m+n} = i_{m+n} \, and \, i = i_m)/P(X_m = i) =$$
$$= \delta_{ii_m} p_{i_m i_{m+1}} ... p_{i_{m+n-1} i_{m+n}} P(X_0 = i_0, ...X_{m+n} = i_{m+n} \, and \, i = i_m)/P(X_m = i)$$

che è valido per il teorema 1.2.1. In generale, un qualsiasi evento A determinato da $X_0, ...X_m$ può essere scritto come una unione disgiunta di eventi elementari che possono essere contati:

$$A = \bigcup_{k=1}^{\infty} A_k.$$

Ne consegue che l'identità desiderata per A si ottiene sommando le corrispondenti identità per A_k. □

Anche in questo caso, la dimostrazione permette di rafforzare ancor di più il concetto di assenza di memoria.

2.1.4 Struttura di Classe

In alcuni casi può essere possibile rompere una catena di Markov in parti più piccole, ciascuno dei quali risulta essere di semplice comprensione e la

somma delle quali permette di avere una veduta complessiva della catena stessa. Questo è fatto essenzialmente per trovare le classi di comunicazione della catena di Markov.

Solitamente si afferma che i *porta a* j e si scrive $i \rightarrow j$ se

$$P_i(X_n = j \text{ per qualche } n \geq 0) > 0$$

Si dice inoltre che i *comunica con* j e si scrive $i \leftrightarrow j$ se e solo se $i \rightarrow j$ e $j \rightarrow i$. Da notare che $i \leftrightarrow j$ sta ad indicare che partendo da i è possibile pervenire in j, ma allo stesso tempo esiste anche un percorso che in un numero n finito di step, a partire da j, porta la catena di Markov nello stato i. E' allora plausibile effettuare la seguente considerazione: è perfettamente possibile che dallo stato i si possa arrivare nello stato j da cui ripartire per tornare nello stato di partenza i (infatti come detto precedentemente, esistono questi due percorsi). Per definizione un percorso che connette uno stato della catena di Markov a se stesso prende il nome di *ciclo*.

Tutto questo è esprimibile in maniera molto più formale attraverso l'ausilio di un semplice teorema.

Teorema 2.1.3.
Dati due stati distinti i e j, le affermazioni che seguono sono tutte equivalenti:

1. $i \rightarrow j$;
2. $p_{i_0 i_1} p_{i_1 i_2} ... p_{i_{n-1} i_n} > 0$ *per alcuni stati* $i_0, i_2, ..., in$ *con* $i_0 = i$ *e* $i_n = j$;
3. $p_{ij}^{(n)} > 0$ *per alcuni* $n > 0$.

Il teorema definito è dimostrabile nel modo seguente:

Dimostrazione. Si osservi che

$$p_{ij}^{(n)} \leq P_i(X_n = j \text{ per qualche } n \geq 0) \leq \sum_{n=0}^{\infty} p_{ij}^{(n)}$$

che dimostra l'equivalenza tra (i) e (iii). Inoltre

$$p_{ij}^{(n)} = \sum_{i_1, ..., i_{n-1}} p_{ii_1} p_{i_1 i_2} ... p_{i_{n-1} j}$$

in maniera tale per cui sia possibile dimostrare l'equivalenza tra (ii) e (iii) e per proprietà transitiva dell'equivalenza, si ha che (i) è equivalente a (ii). □

Per quanto definito grazie al teorema 1.4.1, risulta essere evidente che se $i \rightarrow j$ e $j \rightarrow k$ allora tutto ciò implica che $i \rightarrow k$. Inoltre, per ogni stato i, vale $i \rightarrow i$. Il simbolo $\leftrightarrow$ soddisfa una relazione di equivalenza su I ed ha lo scopo di partizionare I in *classi di comunicazione*. Si dice che una classe C è *chiusa* se

$$i \in C,\ i \rightarrow j \quad implica\ j \in C$$

Una classe chiusa diventa allora un qualcosa dal quale non è più possibile uscire, infatti qualsiasi successore della stato attuale apparterrà sempre alla classe di appartenenza dello stato di partenza. Si dice che uno stato i è *assorbente* se $\{i\}$ è una classe chiusa. Infine, qualora una catena o una matrice di transizione P possiedano I come una singola classe, allora si dice che la catena o la matrice è *irriducibile*. Oppure una catena di Markov è irriducibile se partendo da ogni stato i esiste sempre una probabilità $p_{i,j} > 0$ di raggiungere ogni altro stato j. Da un punto di vista formale, tutto ciò è così esprimibile:

$$\forall i, j \in I, \forall m \in \mathbb{N}, \exists n \in \mathbb{N} : P(X_{m+n} = j | X_m = i) > 0.$$

Infatti, qualora esistesse uno stato che non possa essere raggiunto, chiaramente questo stato non è raggiungibile e deve essere eliminato e quindi la catena di Markov è riducibile.

Infine, si considerano le catene di Markov *aperiodiche*. Il periodo di uno stato $i \in I$ di una catena di Markov è definito come il massimo comune divisore $d(i)$ della lunghezza di tutti i cicli del grafo associato contenenti i. Se $d(i) = 1$, si dice che lo stato i è *aperiodico*. Formalmente, tale concetto è così esprimibile:

$$d(i) = MCD\left\{n, m \in \mathbb{N}, n > 0 : P(X_{m+n} = i | X_m = i) > 0\right\}$$

. Una catena di Markov è *aperiodica* se tutti i suoi stati sono aperiodici, altrimenti è detta *periodica*.

2.1.5 Matrice di Transizione a n-step

Si è arrivati a definire che $(X_n)_{n\geq 0}$ è una catena di Markov (λ, P) su uno spazio degli stati I e possiede la proprietà di Markov che permette di affermare che, avendo la matrice P di transizione a uno step:

$$P\{X_{n+1} = j | X_n = i\} = p_{i,j}.$$

Sarebbe molto interessante poter avere una matrice di transizione che possa permettere di capire quale sarà l'evoluzione del sistema dopo n-step generici. A tal scopo, per semplicità, si può partire considerando l'esempio della matrice di transizione a due step. Tale matrice esprime le probabilità di passare da uno stato di partenza i ad uno stato finale j, ma passando per uno stato intermedio. Consideriamo allora la matrice di transizione a due step:

$$\begin{aligned} &P(X_{n+2} = j | X_n = i) = \textstyle\sum_{k\in I} P(X_{n+2} = j, X_{n+1} = k | X_n = i) = \\ &= \textstyle\sum_{k\in I} P(X_{n+1} = k | X_n = i) P(X_{n+2} = j | X_{n+1} = k, X_n = i) = \\ &= \textstyle\sum_{k\in I} p_{i,k} p_{k,j} = (P^2)_{i,j}, \end{aligned}$$

dove P^2 è il prodotto della matrice P con se stessa. Ancor più in generale è allora possibile affermare quanto segue

$$P(X_{n+k} = j | X_k = i) = (P^n)_{i,j}.$$

Soprattutto, se il vettore $(\delta_i : i \in I)$ è la distribuzione iniziale, si ottiene

$$\begin{aligned} &P(X_n = j) = \textstyle\sum_{k\in I} P(X_0 = k) P(X_n = j | X_0 = k) = \\ &= \textstyle\sum_{k\in I} \delta_k (P^n)_{k,j}. \end{aligned}$$

E allora si ottiene

$$P(X_n = j) = (\delta P^n)_j.$$

Ora, dopo aver effettuato il calcolo della matrice P^n e trovato la distribuzione di X_n, è possibile calcolare le probabilità delle matrici di transizione

a n-step, con n generico. Ad esempio, sfruttando tale procedura è possibile conoscere la probabilità di essere nello stato j al tempo $n + k$ se lo stato di partenza era lo stato i al tempo n. È altrettanto ovvio che, questo metodo è molto importante e semplice da utilizzare, ma ha un grosso limite dovuto essenzialmente al calcolo della matrice P^n. Infatti, per un valore di n molto elevato, il calcolo di tale matrice diventa particolarmente oneroso in termini di complessità spaziale e temporale.

2.2 Probabilistic Computation Tree Logic

La logica *Probabilistic Computation Tree Logic* (PCTL), sebbene abbia una diversa espressività rispetto a CTL, sintatticamente differisce soltanto per l'introduzione dell'operatore probabilistico $\mathcal{P}_J(\varphi)$ e per la presenza del Bounded Until ($\Phi_1 \; \boldsymbol{U}^{\leq n} \; \Phi_2$); per introdurla risulta quindi comodo partire dalla definizione di CTL.

2.2.1 Computation Tree Logic

Computation Tree Logic CTL fa parte della famiglia delle logiche temporali, è basata sulle classiche formule della logica proposizionale, utilizza una nozione di tempo discreto, e la sua relazione di raggiungibilità è definita solo nel futuro. CTL è un'importante logica con la quale è possibile formulare un vasto insieme di proprietà. Va però sottolineato che l'espressività delle *linear temporal logic* LTL, altra famiglia di formalismi utilizzati nel MC è incomparabile con la Computation Tree Logic. Questo significa che alcune proprietà che sono esprimibili in LTL ma non possono essere espresse in CTL e viceversa. CTL ha una sintassi a due stadi, i quali classificano le sue formule: di stato e di cammino. Le prime sono affermazioni riguardanti le proposizioni atomiche negli stati, mentre seconde esprimono le proprietà temporali dei percorsi.

Definizione 2.1. Sintassi CTL

Le formule di stato partendo dall'insieme delle proposizioni atomiche AP sono definite dalla seguente grammatica:

$$\Phi ::= true \mid a \mid \ \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \ \forall\varphi$$

dove $a \in AP$ è una formula di cammino generata dalla seguente grammatica:

$$\varphi ::= \bigcirc \Phi \mid \Phi_1 \ \boldsymbol{U} \ \Phi_2$$

dove Φ , Φ_1 e Φ_2 sono formule di stato.

Con le lettere greche maiuscole denoteremo le formule di stato, mentre con le minuscole indicheremo formule di cammino. Il significato degli operatori Next ed Until è il seguente: $\bigcirc \Phi$ è vera se Φ risulta verificata nello stato successivo e $\Phi_1 \ \boldsymbol{U} \ \Phi_2$ vale in un percorso p se esiste uno stato (appartenente a p) in cui è vera Φ_2 e per tutti gli altri Φ_1 è verificata. Una formula di cammino può essere trasformata in una di stato anteponendole o l'operatore $\exists\varphi$ (esiste un cammino in cui φ è vera) o $\forall\varphi$ (per ogni percorso in cui è vera φ). Da notare che per gli operatori $\bigcirc$ e $\boldsymbol{U}$ è richiesto che siano necessariamente preceduti da $\exists$ o $\forall$ per ottenere una formula di stato valida. Infine la formula $\exists\varphi$ è vera in uno stato s_i se esiste un percorso in cui è vera φ che inizia in s_i; dualmente $\forall\varphi$ è verificata in s_j se tutti i cammini che partono da s_j soddisfano φ.

Semantica di CTL

Le formule CTL sono interpretate negli stati e attraverso i percorsi di un sistema di transizione TS. Formalmente dato TS, la semantica di formule CTL è definita da due relazioni di verità (indicate con $\models$) una per le formule di stato e l'altra per le formule di cammino. Per le formule di stato, $\models$ è una relazione tra gli stati di TS e le relative formule. Si scrive $s \models \Phi$ e si interpreta come Φ è un teorema in s se e solo se la formula Φ appartiene

ad s. Invece per le formule di cammino, $\models$ è una relazione tra frammenti di percorsi massimali in TS e formule di cammino. Si scrive $\pi \models \varphi$ il suo significato è il seguente: φ è un teorema di un cammino π se e solo se il cammino soddisfa la formula.

Definizione 2.2. Relazione di soddisfacibilità per CTL

Presa una proposizione atomica $a \in AP$, $TS = (S, Act, \rightarrow, I, AP, L)$ un sistema di transizioni senza stati finali, uno stato $s \in S$, Φ e Ψ formule di stato e φ una formula di cammino. La relazione di soddisfacibilità $\models$ è definita in uno stato da:

$$s \models a \quad sse \quad a \in L(s)$$

$$s \models \neg\Phi \quad sse\ non \quad s \models \Phi$$

$$s \models \Phi \wedge \Psi \quad sse \quad (s \models \Phi)\ e\ (s \models \Psi)$$

$$s \models \exists\varphi \quad sse \quad \pi \models \varphi \quad per\ almeno\ un \quad \pi \in Paths\ (s)$$

$$s \models \forall\varphi \quad sse \quad \pi \models \varphi \quad per\ tutti\ i \quad \pi \in Paths\ (s)$$

Per un percorso π la relazione di soddisfacibilità $\models$ per una formula di cammino è definita da:

$$\pi \models \bigcirc\Phi \quad sse \quad \pi[1] \models \Phi$$

$$\pi \models \Phi\ \boldsymbol{U}\ \Psi \quad sse \quad \exists j \geq 0.\, (\pi[j] \models \Psi \wedge (\forall\ 0 \leq k \leq j,\ \pi[k] \models \Phi))$$

dove per percorso si intende $\pi = s_0 s_1 s_2 .....$ Preso un intero $i \geq 0$, $\pi[i]$ indica l' $(i+1) - esimo$ stato di π. Di seguito verrà proposto il medesimo cammino per definire la logica PCTL; questa sezione infatti serviva soltanto a rendere più semplice la definizione di PCTL.

2.2.2 Introduzione a PCTL

Probabilistic computation tree logic (acronimo PCTL) è una logica temporale branching-time. Una formula PCTL esprime inferenza basata su uno stato appartenente ad una catena di Markov. La valutazione è di tipo booleano, questo significa che uno stato o soddisfa o viola una data formula PCTL. La logica PCTL è definita in modo simile a CTL dalla quale differisce per la mancanza dei quantificatori di cammino esistenziale ($\exists\varphi$) e universale ($\forall\varphi$) e per l'introduzione dell'operatore probabilistico $\mathcal{P}_J(\varphi)$ (dove φ è una formula di cammino e J un sottointervallo di $[0,1]$). La formula di cammino φ impone una condizione nell'insieme dei cammini, J indica invece un lower/upper bound sulla probabilità. Intuitivamente, il significato di una formula PCTL $\mathcal{P}_J(\varphi)$ vera in uno stato s è il seguente: la probabilità per un insieme di percorsi i quali soddisfano φ e partono da s è compresa nell'insieme di valori imposto da J. L'operatore probabilistico può essere considerato come la controparte quantitativa dei quantificatori di cammino utilizzati in CTL, i quali potevano solamente esprimere o l'esistenza di un percorso con una certa proprietà ($\exists\varphi$) o imporre che la totalità dei percorsi fosse in grado di soddisfarla ($\forall\varphi$). PCTL come accennato in precedenza differisce anche per l'introduzione del Bounded Until $\Phi\ \boldsymbol{U}^{\leq n}\ \Psi$, la semantica di questo nuovo operatore è la seguente: preso un numero naturale n se in uno stato è valida Ψ e quest'ultimo è raggiunto da un cammino composto da un numero di transizioni $j \leq n$, le quali attraversano stati che soddisfano Φ, $\Phi\ \boldsymbol{U}^{\leq n}\ \Psi$ è soddisfatta.

2.2.3 Sintassi

Definizione 2.3. Sintassi di PCTL

Come in CTL sintatticamente le formule di PCTL si dividono in formule di stato e di cammino. Una formula di stato PCTL basata su un set AP di preposizioni atomiche è definita sulla base della seguente grammatica:

$$\Phi ::= true \mid a \mid \ \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathbb{P}_J(\varphi)$$

dove $a \in AP$, φ è una formula di cammino e $J \subseteq [0,1]$ è un intervallo con limiti strettamente razionali.

Le formule di cammino in PCTL invece sono definite come segue:

$$\varphi ::= \bigcirc \Phi \mid \Phi_1 \ \boldsymbol{U} \ \Phi_2 \mid \Phi_1 \ \boldsymbol{U}^{\leq n} \ \Phi_2$$

dove Φ, Φ_1 e Φ_2 sono fomrule di stato e $n \in \mathbb{N}$.

In modo del tutto analogo a PCTL, gli operatori temporali $\bigcirc$, $\boldsymbol{U}$ e $\boldsymbol{U}^{\leq n}$ devono essere assolutamente preceduti dall'operatore probabilistico $\mathbb{P}$. Scritture alternative a quella esplicita dell'intervallo sono concesse ad esempio: $\mathcal{P}_{\leq 0.75}(\varphi)$, $\mathcal{P}_{>0}(\varphi)$ e $\mathcal{P}_{=0.75}(\varphi)$ sono formule PCTL valide. Al solito l'operatore ($\Diamond$) può essere ricavato come al solito: $\Diamond\Phi = true \ \boldsymbol{U} \ \Phi$, lo stesso vale per l'operatore di passo quantificato. Le formule PCTL sono da interpretare sugli stati e i cammini di una Catena di Markov $\mathcal{M}$. Per le formule di stato, la relazione di verità $\models$ è una relazione tra gli stati di $\mathcal{M}$ e formule di stato. Scriveremo $s \models \Phi$ come al solito per indicare che una formula Φ è soddisfatta in uno stato s. $\models$ è una relazione tra infiniti percorsi in $\mathcal{M}$ e le formule di cammino.

Definizione 2.4. Relazione di soddisfacibilità per PCTL

Presa $a \in AP$ una proposizione atomica, $\mathcal{M} = (S,\boldsymbol{P},\iota_{init},AP,L)$ una catena di Markov, uno stato $s \in S$ *e* Φ,Ψ due formule PCTL di stato. La relazione di soddisfacibilità $\models$ è definita per le formule di stato da

$$s \models a \quad sse \quad a \in L(s)$$

$$s \models \neg\Phi \quad sse\ non \quad s \models \Phi$$

$$s \models \Phi \wedge \Psi \quad sse \quad (s \models \Phi)\ e\ (s \models \Psi)$$

$$s \models \mathbb{P}_J(\varphi) \quad sse \quad Pr(s \models \varphi) \in J$$

dove $Pr(s \models \varphi) = Pr_s\{\pi \in Paths(s) \mid \pi \models \varphi\}$. Dato un cammino π in $\mathcal{M}$, la relazione di soddisfacibilità è definita:

$$\pi \models \bigcirc\Phi\ sse\ \pi[1] \models \Phi$$

$$\pi \models \Phi\ \boldsymbol{U}\ \Psi\ sse\ \exists j \geq 0.(\pi[j] \models \Psi \wedge (\forall\ 0 \leq k \leq j.\pi[k] \models \Phi))$$

$$\pi \models \Phi\ \boldsymbol{U}^{\leq n}\ \Psi\ sse\ \exists 0 \leq j \leq n.(\pi[j] \models \Psi \wedge (\forall\ 0 \leq k \leq j.\pi[k] \models \Phi))$$

dove per un path $\pi = s_0 s_1 s_2.....$ e un intero $i \geq 0$, $\pi[i]$ indica l'$(i+1) - esimo$ stato di π ovvero $\pi[i] = s_i$. La semantica dell'operatore temporale $\mathbb{P}$, fa riferimento alla probabilità per i set dei cammini nei quali la formula di cammino è valida. Per assicurare che questa definizione sia valida, è necessario che gli eventi specificati con formule PCTL siano misurabili.

Lemma 2.2.1. *Misurabilità di eventi PCTL*

Per ogni formula di cammino PCTL φ e per uno stato s di una catena di Markov $\mathcal{M}$, il set $\{\pi \in Paths(s) \mid \pi \models \varphi\}$ è misurabile.

L'equivalenza tra formule PCTL si definisce in modo simile rispetto alle altre logiche, due formule di stato sono equivalenti quando le loro semantiche sono uguali. Formalmente , per PCTL prese due formule Φ e Ψ:

$\Phi \equiv \Psi$ sse per tutte le catene di Markov $\mathcal{M}$ e gli stati s di $\mathcal{M}$ vale la seguente:

$s \models \Phi \Leftrightarrow s \models \Psi$

sse $Sat_{\mathcal{M}}(\Phi) = Sat_{\mathcal{M}}(\Psi)$ per tutte le catene di Markov $\mathcal{M}$.

Inoltre per l'equivalenza di regole appartenenti alla logica proposizionale, abbiamo:

$$\mathcal{P}_{<\ p}(\varphi) \equiv \neg\mathcal{P}_{\geq p}(\varphi)$$

dove $p \in]0, 1]$ è un numero razionale e φ una qualsiasi formula di cammino PCTL. Questa equivalenza deriva direttamente dalla semantica di PCTL. Va notato che l'espressività della logica non cambia quando viene permesso soltanto uno degli operatori di confronto sulle probabilità di percorso e uno dei bound qualitativi $= 0$ o $= 1$; per esempio:

$$\mathcal{P}_{]0.3,0.7]}(\varphi) \equiv \neg\mathcal{P}_{\leq 0.3} \wedge \neg\mathcal{P}_{>0.7}$$

2.2.4 Semantica di PCTL

L'operatore probabilistico $\mathcal{P}_J$ dal punto di vista semantico si interpreta come segue: una formula $\mathcal{P}_J(\varphi)$, è vera se il valore probabilistico atteso associata ai cammini che verificano φ cade all'interno dell'intervallo J. Le fomule di cammino $\bigcirc\ \Phi$ e $\Phi_1\ \boldsymbol{U}\ \Phi_2$ si valutano allo stesso modo di PCTL; invece il Bunded Until $\Phi_1\ \boldsymbol{U}^{\leq n}\ \Phi_2$ si interpreta come variante di $\Phi_1\ \boldsymbol{U}\ \Phi_2$, ovvero la semantica di questo nuovo operatore è così definita: preso un numero naturale n se in questo stato è valida Φ_2 questo stato è raggiunto da un cammino composto da n transizioni, le quali attraversano stati che soddisfano Φ_1.

2.3 Maple

Il software di Model Checking proposto in questo elaborato si occuperà di eseguire inferenza costruita sulla logica modale PCTL, basandosi su Catene di Markov finite a tempo discreto, nelle quali i valori delle transizioni da uno stato all'altro possono essere anche definite con parametri; ovvero la verità della formula viene espressa mediante un polinomio in funzione dei parametri. La sua valutazione finale avverrà soltanto a parametri noti; inoltre attraverso una formulazione di questo tipo è possibile studiare il comportamento del sistema in funzione dei valori assunti (dai parametri). Durante la ricerca di algoritmi validi, efficaci e performanti adatti a risolvere questo problema, abbiamo provato varie alternative ma nessuna è risultata abbastanza efficace per risolvere problemi di dimensione sufficienti tali da essere considerati di interesse pratico. E' stato quindi preso in considerazione un linguaggio di programmazione orientato al calcolo matematico con prestazioni elevate: Maple [7].

Maple [7] è un software a pagamento questo è l'unico punto a sfavore nella sua scelta, ma non esiste allo stato dell'arte un software distribuito sotto licenza GNU/GPL in grado di competere per quanto riguarda le prestazioni.

Nella prossima sezione viene presentato il software con una breve storia la quale sottolinea le sue origini accademiche, seguita da una presentazione dell'ambiente e delle sue caratteristiche salienti.

2.3.1 Introduzione

Maple[7] è un pacchetto applicativo matematico commerciale multipiattaforma ad uso generico. Fu sviluppato per la prima volta nel 1981 dal Symbolic Computation Group all'Università di Waterloo, Canada. Dal 1988, è stato sviluppato e venduto dalla Waterloo Maple (conosciuta anche come Maplesoft), una compagnia canadese che ha sede in Waterloo, Ontario. Maple combina un linguaggio di programmazione con un'interfaccia che consente agli utenti di scrivere formule matematiche usando la notazione matematica

tradizionale. La maggior parte delle funzioni di Maple sono scritte nel linguaggio Maple, che è interpretato dal kernel di Maple, il quale è scritto in C. Il linguaggio di programmazione Maple è un linguaggio di programmazione interpretato con una sintassi basata sul linguaggio Pascal, e caratterizzato dalla tipizzazione dinamica; questa permette di lavorare in modo comodo e intuitivo con polinomi ed espressioni complesse lasciando che sia il software a scegliere l'esecuzione più performante. Attualmente l'ultima versione rilasciata è Maple 15.

2.3.2 Ambiente

In questo paragrafo verrà descritto l'applicativo: presentando l'interfaccia, il linguaggio e le sue capacità espressive, le librerie di calcolo di nostro interesse e concludendo mostrando le possibilità di interfacciamento con gli altri linguaggi di programmazione. Le descrizioni saranno basate sulla current release Maple 15 [7].

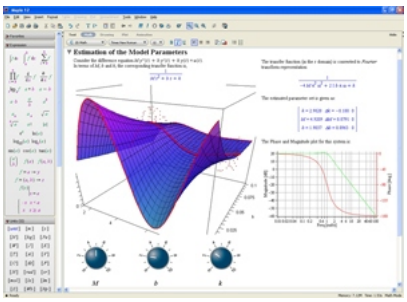

Figura 2.2: Interfaccia di Maple.

L'interfaccia si presenta simile a qualsiasi altro programma orientato al calcolo numerico: tutto è incentrato attorno ad un foglio di programmazione dinamico, dove ogni comando viene eseguito istantaneamente, rendendo così possibile la verifica del codice scritto in tempo reale o di ottenere immediatamente i risultati.

```
rewrite_t := proc(f, select_t)
  local tv, v;
  if tv={} then return f end if;
  v := `if`(nops(tv)=1, op(1,tv), seq(set(Float(abs(coeff(f,v)),0), v)));
end proc:
```

Esempio di codice Maple.

Il linguaggio utilizzato per la programmazione si basa sulla sintassi Pascal, nota per la sua verbosità, ma caratterizzata da una facile lettura. Il paradigma adottato è prettamente strutturale, non è presente nessun costrutto inerente la programmazione ad oggetti. In Maple [7] per raggruppare funzioni e procedure si utilizza il costrutto modulo, quest'ultimo permette sia di avere visibilità delle variabili all'interno dello stesso, sia di mantenere raggruppati i metodi. La caratteristica principale di questo linguaggio di programmazione è la tipizzazione dinamica ovvero. La tipizzazione dinamica è una particolare politica di casting, ovvero di assegnazione di tipi alle variabili. Nei linguaggi a tipizzazione dinamica, le variabili hanno dei tipi ben definiti, che però possono cambiare dinamicamente in seguito a manipolazioni esterne. Per la loro natura fortemente imprevedibile, i linguaggi sono spesso interpretati, in quanto l'interprete costituisce un ambiente di esecuzione sicuro, in grado di assecondare tutti i cambiamenti di tipo delle variabili. Anche Maple segue questa filosofia. Nonostante in teoria la tipizzazione dinamica dovrebbe ridurre in modo significativo le prestazioni, Maple rimane una validissima scelta.

Come in tutti i linguaggi di programmazione orientati al calcolo numerico, la parte che richiede la maggiore versatilità, completezza e potenza di calcolo è quella riguardante l'algebra lineare ovvero il calcolo matriciale. Maple possiede un modulo LinearAlgebra robusto ed efficiente, che offre funzioni per costruire, manipolare Matrici e Vettori, eseguire operazioni standard, gestire le interrogazioni per i dati di output e risolvere problema di algebra lineare in genere. Linear Algebra supporta completamente matrici sparsi, a bande, triangolari.

Per utilizzare codice Maple richiamandolo da un altro applicativo basta utilizzare OpanMaple, un'interfaccia che permette al programmatore di aver accesso al motore computazionale di Maple facendo riferimento alle sue librerie dinamiche; il supporto è multipiattaforma (Linux, OsX, Win). Mediante OpenMaple è possibile da un programma esterno (C, Java, Fortran, Java, C++, C# ...):

- Inviare comandi che inizializzano l'engine
- Chiamare API per eseguire calcoli con Maple
- Definire funzioni, passandogli parametri direttamente dal codice del chiamante.
- Gestire e richiamare l'output derivante dall'esecuzione del codice.

In conclusione Maple risulta un'ottima alternativa per realizzare il codice necessario alla risoluzione del problema che stiamo andando a formalizzare; è necessario avere delle routine efficienti nel caso di sistemi lineari sparsi polinomiali, Maple risulta possedere le caratteristiche necessarie per risolvere questi tipi di problemi.

Capitolo 3

Stato dell'arte

I cambiamenti imprevedibili affliggono continuamente i sistemi software e possono causare gravi conseguenze, compromettendo in modo irrimediabile la qualità del servizio, paralizzando il sistema, causando in certi casi la violazione dei requisiti del software stesso. Questi cambiamenti possono coinvolgere componenti critici del sistema, profili operazionali e requisiti arrivando talvolta agli ambienti di sviluppo. L'adozione di modelli software e tecniche di Model Checking a run-time, potrebbe essere utile per supportare sistemi automatici di reasoning nella valutazione di questi cambiamenti: la rilevazione di configurazioni dannose permetterebbe così di reagire, attivando appropriate contromisure automatiche. Le tecniche di MC tradizionali e i tools a loro associati non sono adatti, per come sono progettatti attualmente, ad essere utilizzati a run-time, poiché non sono in grado di soddisfare i vincoli rigidi imposti dall'esecuzione on-the-fly (al volo) in termini di memoria e tempo di esecuzione. Diventa quindi necessario impiegare un nuovo paradigma in grado di adattarsi a questi requisiti; per arrivare a definirlo, è necessaria prima, un'introduzione ai sistemi di Model Checking attuali fino ad arrivare alle alternative esistenti nel ramo del Run-time MC.

3.1 Introduzione

Spesso i sistemi software sono progettati, sviluppati ed implementati per operare in un ambiente immutabile e completamente noto, con profili operazionali e requisiti costanti, che non cambiano al trascorrere del tempo e al mutare delle condizioni. In un ambiente di questo tipo, come introdotto in precedenza, qualsiasi cambiamento è imprevedibile ed in grado di paralizzare l'intero sistema, compromettendo la sua capacità di soddisfare i requisiti per i quali è stato progettato. Ogni qual volta un software deve essere adattato a dei cambiamenti, un completo ciclo di vita, design, progettazione, sviluppo e manutenzione, deve essere programmato. In uno scenario di questo tipo i cambiamenti sono considerati dannosi poiché inevitabilmente introducono costi elevati per le attività di manutenzione.

La crescita, sia delle dimensioni che degli ambiti di impiego dei software, ha inevitabilmente introdotto variabili e scenari imprevedibili difficili da valutare completamente a design-time; i cambiamenti accadono sempre più spesso e costituiscono uno dei fattori dominanti dei sistemi software del giorno d'oggi. I software attuali sono spesso costruiti attraverso la composizione di moduli sviluppati da diverse organizzazioni indipendenti tra loro: questo può causare profili operazionali e ambienti di sviluppo inattesi ed in continua evoluzione. Diventa quindi necessario per ovviare a questo tipo di problemi introdurre nell'ambito dell'Ingegneria del SW, specifici applicativi in grado di realizzare software come sistemi adattativi in grado di rilevare e reagire ai cambiamenti.

Esistono attualmente molte ricerche che propongono e descrivono tecniche per il design di sistemi adattativi; questo elaborato si focalizza su sistemi software che provano ad adattare loro stessi, con lo scopo di preservare i *requisiti di affidabilità* in presenza di cambiamenti. La soluzione più promettente è costituita da due tecniche complementari: *monitoring* e *models*. La prima si propone di interpretare i dati estrapolati a run-time dalle istanze del sistema: i dati vengono catalogati dal monitor e analizzati continuamente per aggiornare i parametri del modello (esempio: Probabilità di Failure), con

l'obiettivo di mantenerlo consistente nel tempo, durante i cambiamenti di comportamento dell'ambiente. La seconda aggiornato il modello analizzato dal *Model Checker*, verifica la conformità tra il comportamento corrente del sistema ed i requisiti desiderati. Partendo dal presupposto che la tipologia dei requisiti di nostro interesse è costituita solamente dalla classe dei vincoli di affidabilità, la tipologia di modello sulla quale noi ci focalizzeremo sarà basata su *Catene di Markov a Tempo Discreto* (DTMC).

Le attuali tecniche di Model Checking e i relativi tools sono concepiti per un uso a design-time e, come detto in precedenza, difficilmente sono in grado di soddisfare i vincoli di tempo e occupazione della memoria imposti da un'analisi a run-time, primo problema tra tutti l'esplosione dello spazio degli stati, inevitabile durante l'analisi del modello. Per ottenere un sistema in grado di soddisfare suddetti vincoli, bisogna modificare la struttura logica di esecuzione del MC. In particolare una tecnica tradizionale prende come input un modello formale che descrive il sistema reale e una proprietà espresso con un formalismo appropriato (il requisito che stiamo valutando) e verifica se il primo è coerente rispetto al secondo (se il requisito è valido nel modello).

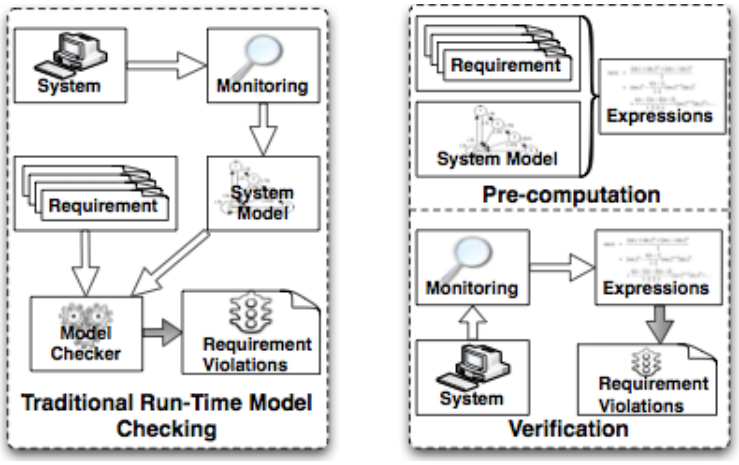

Figura 3.1: Tecniche di Run-Time Model Checking.

Riepilogando, il monitoraggio eseguito per aggiornare il modello del sistema a run-time avviene in modo continuativo, mentre il processo di Model

Checking è attivato periodicamente. Una procedura a run-time di questo tipo è computazionalmente costosa e richiede l'esplorazione completa dello spazio degli stati del modello, il quale potrebbe essere enorme e l'analisi della formula potrebbe essere arbitrariamente complessa. La figura 3.1 riassume questo approccio; i dettagli della complessità di un MC tradizionale probabilistico può essere trovata in [4]. In questo elaborato ci si concentrerà sulla valutazione efficiente dei requisiti di affidabilità a run-time. Il *concetto chiave* della soluzione proposta sta nel *dividere* l'attività di *Model Checking* in *due fasi distinte*: una eseguita a design-time e l'altra a run-time. D'ora in poi si farà riferimento a *design-time* come ad un passo di *precomputazione* e *run-rime* come alla fase di *verifica*. La fase di precomputazione prende come input:

1. Il modello del sistema definito come un DTMC o con un formalismo equivalente, dove alcune transizioni sono definite con parametri.

2. I requisiti di affidabilità del sistema, definiti con un apposito formalismo.

Le transizioni definite con parametri sono parzialmente incognite: il loro valore sarà noto solamente a run-time e potrà cambiare durante il trascorrere del tempo. Prendiamo, come esempio, una transizione che connette uno stato, che modellizza l'invocazione di un servizio, ad un'altro che rappresenta una situazione di failure; la variabile associata alla transizione rappresenta il possibile cambiamento del rate di falliemento del servizio. L'output prodotto dal passo di precomputazione è costituito da un insieme di espressioni simboliche che rappresentano la soddisfacibilità del requisito. La verifica, eseguita a run-time, valuta semplicemente la formula sostituendo alle variabili il corrispondente valore ottenuto dal monitoraggio del sistema reale. Riprendendo l'esempio precedente, la combinazione tra il valore reale del rate di fallimento del servizio e la formula rappresentante il requisito permette di ottenere la sua valutazione: vera o falsa.

Il vantaggio principale di questo approccio è quello di spostare il costo computazionale dell'analisi del modello il più possibile a design-time (precomputazione); portando l'esplorazione dello spazio degli stati a design-time si riduce la fase di run-time alla semplice sostituzione delle variabili con i valori. Il concetto chiave, che sta dietro a questo approccio, è quello di pagare un passo di trasformazione molto costoso durante la fase di design-time, per ottenere una fase di analisi a run-time efficiente ed in grado di soddisfare i requisiti stringenti dei processi on-line. In questo modo lo speed up ottenuto a run-time rispetto ai Model Checker esistenti, PRISM [8] e MRMC [9], risulta notevole. L'approccio proposto parte dall'assunzione, come verrà definito in seguito, che i potenziali cambiamenti posso essere previsti, ovvero il modello deve contenere al suo interno tutti gli scenari possibili di evoluzione che successivamente attraverso i valori stimati a run-time si portano il modello nella corrispondente situazione reale. In teoria si potrebbe arrivare al caso limite assumendo tutte le transizioni variabili: questo renderebbe l'approccio totalmente inutilizzabile ad eccezione di modelli con uno spazio degli stati di modeste dimensioni.

Questa introduzione ripercorre il paper Run-time Efficient Model Checking [1] cercando inoltre un confronto diretto con l'unica alternativa ad oggi esistente: Param [3]. Param [3] è l'unico MC attualmente presente in campo internazionale che segue il metodo proposto in questo elaborato, ovvero separare la fase di precomputazione con quella di verifica a run-time; è stato sviluppato da Ernst Moritz Hahn, Holger Hermanns, Lijun Zhang, Björn Wachter attraverso la collaborazione della Saarland University (Germania) e INRIA di Grenoble (Francia). D'ora in avanti si farà sempre riferimento a questo MC che risulta essere molto valido e performante e si cercherà di superare i suoi limiti, partendo da una metodologia di approccio al calcolo totalmente diversa. Gli obiettivi principali di questo elaborato sono:

- Confrontarsi direttamente con Param
- Implementare e risolvere tutti i vincoli di affidabilità che PCTL permette di creare.

Di seguito è riportata una breve sezione che introduce Param descrivendolo nella sua completezza, focalizzandosi sui suoi punti di forza e debolezza.

3.2 Param

La modellizzazione di sistemi attraverso Catene di Markov è stata applicata con successo in un gran numero di settori come: informatica, biologia, ingegneria; in questa classe di formalismi per ottenere un modello valido e forte rispetto ai cambiamenti è necessario lasciare libertà a certi aspetti cruciali. Param, per formalizzare problemi basati su DTMC, utilizza un'estensione del linguaggio PROMELA [10] dove le probabilità di transizione da un'azione ad un'altra possono essere espresse, oltre che da un classico valore numerico, anche da un parametro dichiarato all'interno del modello stesso.

```
// Model taken from Daws04
// This version by Ernst Moritz Hahn (emh@cs.uni-sb.de)
probabilistic

param double p;
param double q;
const int n = 10;
module main
  s: [-2..n+1];
  [b] (s=-1) -> (s'=-2);
  [a] (s=0) -> 1-q : (s'=-1) + q : (s'=1);
  [a] (s>0) & (s<n+1) -> 1-p : (s'=0) + p : (s'=s+1);
endmodule
init
  s = 0
endinit
```

Esempio di codice PARAM in grado di supportare transizioni parametriche.

In questo contesto lo scopo del Model Checker Param è quello di determinare in forma parametrica proprietà di reachability. Presa un'asserzione priva di operatori di cammino ed un Bound B, la reachability risulta verificata in un cammino $s_0..s_i$ se quest'ultimo è in grado di raggiungere uno stato nel quale l'asserzione risulta valida la probabilità di percorrerlo risulta contenuta in B. La probabilità così ottenuta è una *funzione razionale* in *n incognite* ognuna costituite da un *parametro*. Per le Catene di Markov a tempo Discreto, Daws ha concepito un approccio teorico per risolvere questo tipo di problemi. L'approccio di Param segue questo algoritmo, dove le probabilità delle transizioni sono considerate come lettere di un alfabeto. Il modello può così essere visto come un automa a stati finiti; il principio di base è quello di *eliminazione* degli *stati*; si usano espressioni regolari per descrivere il linguaggio sul quale avvengono i calcoli. In un passo di postprocessing, questa espressione regolare viene valutata ricorsivamente fino ad ottenere come risultato una *funzione razionale* basata sui *parametri* del modello. Di recente Gruber e Johannsen hanno mostrato come la dimensione dell'espressione regolare di un automa a stati finiti esploda: la complessità è asintotica con $n^{\Theta log(n)}$ dove ad n corrisponde il numero degli stati.

Il metodo principale adottato da questo MC è costituito essenzialmente dall'algoritmo di *eliminazione* degli *stati*; la differenza principale rispetto al metodo proposto da Daws sta nel rimuovere il postprocessing dell'espressione regolare, computando la *funzione razionale* durante la fase di eliminazione degli stati. Più precisamente, in un passo di eliminazione i margini non vengono etichettati mediante espressioni regolari, ma vengono direttamente marcati con l'appropriata funzione razionale rappresentante il flusso delle probabilità.

Come risultato si ottiene una funzione razionale attraverso una variante dell'algoritmo di Daws, la quale utilizza anche semplificazioni matematiche derivanti da simmetrie, cancellazioni, e semplificazioni con l'obiettivo di non espandere inutilmente la funzione risultante. Param per ridurre le dimensioni dello spazio degli stati, il quale cresce asintoticamente con 2^n dove n è il

numero di sottoformule della proprietà da verificare, impiega tecniche dette di lumping come weak e strong bisimulation [8], che sostanzialmente calcolano l'insieme quoziente a partire dal modello dato.

3.2.1 Catene di Markov Parametriche

In questa sezione verrà presentato e formalizzato il modello parametrico che verrà utilizzato nella restante parte dell'elaborato: le catene di Markov parametriche. Come prima cosa vengono riprese alcune notazioni generali:

- S un insieme finito $s_0, s_1, .., s_n$ di stati.
- Una relazione $R \in S \times S$ e $R(s) = \{s' \mid (s, s') \in R\}$ e $R^{-1} = \{s' \mid (s, s') \in R\}$.
- $V = \{x_1, ..., x_n\}$ costituisce un set di variabili con dominio $\mathbb{R}$.
- Una funziona parziale di *valutazione* $u : V \rightharpoonup \mathbb{R}$.
- $Dom(u)$ assegna ad u il proprio dominio; definiamo u totale se $Dom(u) = V$.
- Un polinomio g su V è definito come somma di monomi:

$$g(x_1, ..., x_i) = \sum_{i_1,...,i_n} a_{i_1,....,i_n} x_i^{i_1} ... x_n^{i_n}$$

  dove $i_j \in \mathbb{N}_0$ e ogni $a_{i_1,....,i_n} \in \mathbb{R}$.

- Una *funzione razionale* f su un set di variabili V è una frazione definita come:

$$f(x_1, ..., x_n) = \frac{f_1(x_1, ..., x_n)}{f_1(x_1, ..., x_n)}$$

  ovvero rapporto di due polinomi f_1, f_2 su V.

$\mathcal{F}_V$ rappresenta l'insieme delle *funzioni razionali* da V a $\mathbb{R}$. Preso $f \in \mathcal{F}_V$, un set di variabili $X \in V$ e una valutazione u, $f[X/u]$ è la funzione razionale ottenuta sostituendo ogni occorrenza di $x \in X \cap Dom(u)$ con $u(x)$.

Definizione 3.1. Una catena di Markov parametrica (PMC) è una tupla $\mathcal{D} = (S, s_0, \mathbf{P}, V)$ dove: S è un insieme finito di stati, s_0 è lo stato iniziale, $V = \{v_1, ..., v_n\}$ un insieme finito di parametri e $\mathbf{P}$ matrice di probabilità $\mathbf{P} : S \times S \to \mathcal{F}_V$.

Le catene di Markov parametriche sono introdotte e definite in [5, 6]. Di seguito è riportata la definizione di Grafo di una PMC.

Definizione 3.2. Il grafo sotteso $\mathcal{G}_\mathcal{D}$ di una PMC $\mathcal{D} = (S, s_0, \mathbf{P}, V)$ è definita come $\mathcal{G}_\mathcal{D} = (S, E_\mathbf{D})$ dove $E_\mathbf{D} = \left\{ \left(s, s^{'}\right) \mid \mathbf{P}\left(s, s^{'}\right) \neq 0 \right\}$

Partendo dalle definizioni appena enunciate, vengono ora introdotte le seguenti notazioni:

- Per ogni $s \in S$ si definisce $pre\,(s) = R^{-1}\,(s)$ l'insieme dei predecessori di s.
- Per ogni $s \in S$ si definisce $post\,(s) = R\,(s)$ l'insieme dei successori di s.
- $s^{'}$ è raggiungibile da s e si scrive $reach^{\mathcal{G}_\mathcal{D}}$, se $s^{'}$ è raggiungibile da s attraverso il grafo sotteso $\mathcal{G}_\mathcal{D}$.
- Preso un sottoinsieme A tale che $A \subseteq S$ si scrive $reach^{\mathcal{G}_\mathcal{D}}\,(s, A)$ se $reach^{\mathcal{G}_\mathcal{D}}\left(s, s^{'}\right)$ per ogni $s^{'} \in A$.

Ora viene definito il PMC indotto rispetto ad una valutazione:

Definizione 3.3. Dato un PMC $\mathcal{D} = (S, s_0, \mathbf{P}, V)$ e una valutazione u il PMC indotto $\mathcal{D}_u$ è definito come una quadrupla $(S, s_0, \mathbf{P}_u, V/Dom\,(u))$, dove la matrice di transizione $\mathbf{P}_u : S \times S \to \mathcal{F}_V$ è definita da: $\mathbf{P}_u\left(s, s^{'}\right) = \mathbf{P}\left(s, s^{'}\right)[Dom\,(u)\,/u]$.

Viene introdotta ora la nozione di valutazione *ben definita*. Una valutazione *totale* u è *ben definita* per $\mathcal{D}$ se $\mathbf{P}_u\left(s, s^{'}\right) \in [0, 1]$ per tutti gli $s, s^{'} \in S$ e $\mathbf{P}_u\,(s, S) \in [0, 1]$ e $\forall s \in S$ dove $\mathbf{P}_u\,(s, S)$ esprime la somma $\sum_{s^{'} \in S} \mathbf{P}_u\,(s, S)$. In modo non formale u è *ben definito* se e solo se la PMC $\mathcal{D}_u$ è un classico DTMC privo di parametri; per una valutazione ben definita, uno stato s è

chiamato *stocastico* se $\mathbf{P}_u\left(s, s^{'}\right) = 1$ e *sub-stocastico* se $\mathbf{P}_u\left(s, s^{'}\right) < 1$. Se $\mathbf{P}_u\left(s, s^{'}\right) = 0$, s è chiamato *assorbente*. Presa u una valutazione ben definita e considerando i grafi sottesi $\mathcal{G}_{\mathcal{D}} = (S, E_{\mathcal{D}})$ e $\mathcal{G}_{\mathcal{D}_u} = (S, E_{\mathcal{D}_u})$, risulta vero che $E_{\mathcal{D}_u} \subseteq E_{\mathcal{D}}$; la funzione di valutazione totale u è *ben definita* in *senso stretto* se $E_{\mathcal{D}_u} = E_{\mathcal{D}}$. Una valutazione strettamente ben definita *non distrugge* le proprietà di reachability del grafo sotteso, questo implica che qualche margine con funzione f valuta una probabilità diversa da zero. Come ad una probabilità spesso corrisponde un indice di failure, altrettanto spesso non è necessario considerare valutazioni ben definite ma non stringenti; poiché non è di gran interesse valutare probabilità di successo o di failure pari a 0.

Un path infinito è una sequenza illimitata $\sigma = s_0 s_1 s_2....$ di stati, invece un path finito è una sequenza limitata $\sigma = s_0 s_1 s_2 .... s_n$; un percorso finito σ ha lunghezza $\mid \sigma \mid = n$, $first\,(\sigma) = s_0$ restituisce il primo stato del path, in modo duale $last\,(\sigma) = s_n$ fornisce, in caso di percorso finito, come risultato l'ultimo stato. Un path σ (finito o infinito) è detto *massimale* se $last\,(\sigma)$ è uno stato assorbente. Con $\sigma\,[i]$ si indica l'$i-esimo$ stato appartenente a σ. $Path^{\mathcal{D}}$ definisce l'insieme dei percorsi massimali di $\mathcal{D}$ e $Path_{fin}$ l'insieme dei cammini finiti. $Path^{\mathcal{D}}\,(s)$ e $Path^{\mathcal{D}}_{fin}\,(s)$ sono rispettivamente l'insieme dei path massimali, finiti o infiniti, che iniziano in s. Per un path finito σ, viene definita una funzione razionale $Pr^{\mathcal{D}}\,(\sigma) \in \mathcal{F}_V$ come $Pr^{\mathcal{D}}\,(\sigma) = \prod_{i=0}^{|\sigma|-1} Pr\,(\sigma\,[i]\,, \sigma\,[i+i])$. Per un insieme di percorsi $C \subseteq Path^{\mathcal{D}}_{fin}$ nel quale non esistono $\sigma, \sigma^{'} \in C$, dove σ è un prefisso di $\sigma^{'}$, $Pr^{\mathcal{D}}\,(C) = \sum_{\sigma \in C} Pr^{\mathcal{D}}\,(\sigma)$. La funzione $Pr^{\mathcal{D}}$ può essere estesa unicamente rispetto all'insieme dei percorsi $Path^{\mathcal{D}}$; per una valutazione ben definita u, $Pr^{\mathcal{D}}\,(\sigma)\,[V/u]$ è esattamente la probabilità del percorso σ in $\mathcal{D}_u$, e $Pr^{\mathcal{D}_u}$ è l'unica misura definita di probabilità sull'insieme dei percorsi $Path^{\mathcal{D}}$ dato uno stato iniziale s_0 fissato.

La *Bisimulation* è una relazione di equivalenza sugli stati, i quali vengono raggruppati in classi con stessa probabilità di reachability, con lo scopo di ridurre lo spazio degli stati. Param utilizza tecniche di Bisimulation applicate alle PMC, le quali verranno impiegate anche nell'algoritmo proposto in questo elaborato; per le definizioni e le proprietà si rimanda al capitolo 5.

3.2.2 Architettura

L'architettura e i componenti di PARAM sono mostrati in Figura 3.2. Come prima cosa l'esploratore dello spazio degli stati genera un'esplicita rappresentazione grafica dalla descrizione simbolica del modello e della proprietà; in questo modo vengono sfruttate le ottimizzazioni derivanti dalle proprietà per rendere i passi seguenti più veloci. I metodi di lumping sfruttano le classi di equivalenza per minimizzare il modello; il modulo di analisi, valuta e risolve la rete applica la variante dell'algoritmo di Daws definita nella precedente sezione; il risultato finale del processo è una *funzione razionale*. Il componente che analizza e risolve la rete applica una variante dell'algoritmo di Daws, definito nella sezione precedente.

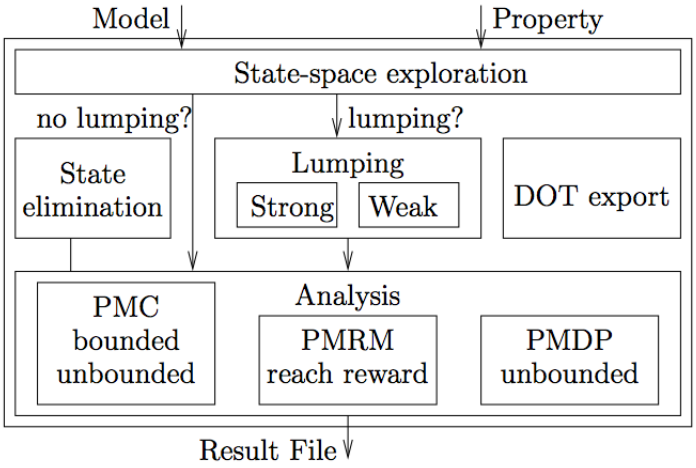

Figura 3.2: Architettura di Param.

Ora viene presentato l'algoritmo per il calcolo della probabilità relativa alla proprietà di reachability, unica classe di proprietà PCTL implementata in Param, concludendo infine la sezione definendo la complessità dell'algoritmo.

Presi $\mathcal{D}$, una PMC e $\mathbf{B}$, un insieme di stati obiettivo, l'output ottenuto è la funzione che rappresenta la *probabilità di reachability parametrica*; questa funzione esprime la probabilità di raggiungere un insieme di stati obiettivo

B da s_0, partendo da una valutazione ben definita. La funzione è ottenuta dal calcolo di $Pr^{\mathcal{D}}(\{\sigma \mid \sigma[0] = s_0 \wedge \exists i.\sigma[i] \in \mathbf{B}\})$.

Algorithm 1 Calcolo della Probabilità di Reachability Parametrica su PMC

Require: PMC $\mathcal{D} = (S, s_0, \mathbf{P}, V)$, l'insieme degli stati obiettivo **B**. Uno stato $s \in \mathbf{B}$ è assorbente; Per tutti gli $s \in S$, valgono $reach(s_0, s)$ e $reach(s, \mathbf{B})$.

1: **for all** $s \in S \setminus (\{s_0\} \cup \mathbf{B})$ **do**
2: **for all** $(s_1, s_2) \in pre(s) \times post(s)$ **do**
3: $\mathbf{P}(s_1, s_2) = \mathbf{P}(s_1, s_2) + \mathbf{P}(s_1, s) \frac{1}{1-\mathbf{P}(s,s)} \mathbf{P}(s, s_2)$
4: **end for**
5: $eliminate(s)$
6: **end for**
7: **return** $\frac{1}{1-\mathbf{P}(s,s)} \mathbf{P}(s_0, \mathbf{B})$

Daws ha già proposto una soluzione a questo problema: Come prima cosa, la PMC viene trasformata in un automa a stati finiti, con stesso stato iniziale s_0 e **B** come insieme di stati finali. Le probabilità relative alle transizioni sono descritte attraverso simboli di un alfabeto dell'automa nelle seguenti forme:

- $\frac{p}{q}$ ovvero un numero razionale $\mathbb{Q}$ espresso come rapporto di due numeri interi p e q.
- x una variabile anch'essa con dominio razionale $D \subset \mathbb{Q}$ nell'intervallo $[0, 1]$.

Attraverso il metodo di *eliminazione* degli *stati*, l'espressione regolare che descrive l'automa viene calcolata; le espressioni vengono poi trasformate in funzioni razionali rappresentanti la probabilità di raggiungere uno stato obiettivo. Questo approccio può diventare molto costoso poiché la lunghezza dell'espressione regolare ottenuta è nell'ordine di $n^{\Theta(logn)}$.

Per ovviare a questo problema ed ottenere un algoritmo utilizzabile in campo pratico, anche se è impossibile migliorare la complessità teorica, è

molto utile impiegare di tecniche di semplificazione e cancellazione durante i passi intermedi, per evitare l'esplosione dell'espressione regolare.

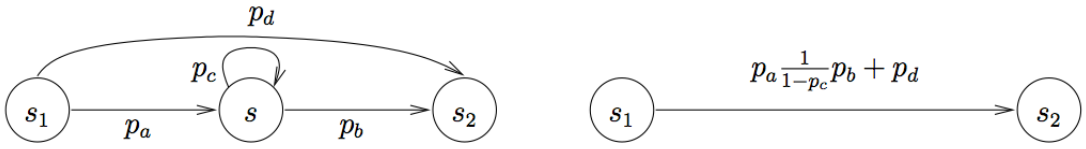

Figura 3.3: Esecuzione dell'algoritmo per PMC.

Nella figura 3.3 è mostrata graficamente l'esecuzione dell'algoritmo 1. L'input è costituito da: una PMC $\mathcal{D}$ e un insieme di stati obiettivo **B**. Dal momento che Param risolve solamente probabilità riferite a proprietà di reachability, è possibile trasformare gli stati obiettivo in assorbenti e rimuovere tutti gli stati non raggiungibili da s_0 o che non possono raggiungere **B**. L'impiego di un'*aritmetica esatta* applicata alle funzione razionali è un punto chiave, anche se rallenta la computazione; evitare problemi derivanti dall'approssimazione numerica, quando si lavora con un vasto spazio degli stati e con valori di probabilità piccoli, risulta fondamentale per ottenere un output corretto. L'idea base di questo algoritmo è di eliminare gli stati uno ad uno, aggiornando ogni volta l'espressione della probabilità di reachability; la funzione $eliminate\,(s)$ rimuove s dall'insieme $\mathcal{D}$. Durante questa fase vengono considerate tutte le coppie $(s_1, s_2) \in pre\,(s) \times post\,(s)$. Prima dell'eliminazione di s, la nuova probabilità di transizione da s_1 a s_2 viene aggiornata in questo modo $f\,(s_1, s_2) := p_d + \frac{p_a pb}{1-p_c}$; il secondo termine $\frac{p_a pb}{1-p_c}$ corrisponde al valore di convergenza della serie geometrica $\sum_{i=0}^{\infty} p_a p_c^i p_b$; il significato di questo valore è raggiungere s_2 da s_1 passando per s. Per valutare la correttezza dell'algoritmo, viene utilizzata la PMC mostrata in Figura 3.3. Preso $V = \{p_a, p_b, p_c, p_d\}$, per una valutazione ben definita, la funzionale razionale calcolata $f\,(s_1, s_2)$ è corretta, come mostrato di seguito. Se u è definita in modo *stringente* si ottiene che $E_{\mathcal{D}} = E_{\mathcal{D}_u}$, che implica $u\,(p_c) > 0$, $u\,(p_b) > 0$ e $u\,(p_c) + u\,(p_b) \leq 1$. Questo indica che il denominatore $1 - u\,(p_c)$ è sempre diverso da 0. Nel caso in cui $u\,(p_c) = 1$ il risultato di $f\,(s_1, s_2)$

non è completamente definito: il problema sta nel fatto che lo stato s non può raggiungere ancora s_2 in $\mathcal{G}_{\mathcal{D}_u}$. Viene considerata ancora una valutazione ben definita (ma non in modo stringente) u la quale soddisfa $u(p_c) = 0$ e $u(p_b) = 1$; in questo caso $f(s_1, s_2)$ restituisce il risultato corretto. Ora viene introdotto il concetto di valutazione *massimale* e *ben definita*:

Definizione 3.4. Assumendo che la PMC $\mathcal{D}$ e l'insieme degli stati $\mathbf{B}$ soddisfino i requisiti dell'Algoritmo 1 e la valutazione totale u è massimale ben definita se è ben definita e per ogni $s \in S$ risulta vero che $reach^{\mathcal{D}_u}(s, \mathbf{B})$

Questo significa che per una valutazione massimale ben definita si può ancora raggiungere l'insieme degli stati obiettivo da tutti gli stati del modello; ciò non implica che la probabilità di reachability sia 1 poiché esistono sottofrazioni del modello che nn raggiungono stati obiettivo. Di seguito è definita la correttezza dell'Algoritmo 1:

Lemma 3.2.1. *Assumendo che la PMC $\mathcal{D}$ e l'insieme degli stati $\mathbf{B}$ soddisfano i requisiti dell'Algoritmo 1 e che questo algoritmo restituisca una $f \in \mathcal{F}_V$, allora, per una valutazione massimale ben definita u, vale che $Pr^{\mathcal{D}_u}(s_0, u) = f[V/u]$.*

Siccome l'algoritmo è basato su numeri e funzioni razionali, è necessario introdurre brevemente la complessità per polinomi basati su questa classe numerica. Preso un polinomio f, definiamo $mon(f)$ una funzione che restituisce il numero di monomi. L'addizione e la sottrazione tra due polinomi f e g sono effettuate attraverso la sottrazione o la somma dei rispettivi monomi, questa operazione richiede una complessità temporale pari a $mon(f) + mon(g)$. La moltiplicazione invece è eseguita attraverso il prodotto incrociato di ogni monomio. Questa calcolo ha complessità asintotica pari a $\vartheta(mon(f) \cdot mon(g))$. La divisione tra due polinomi risulta essere una funzione razionale, che viene semplificata sfruttando il calcolo del *massimo comun divisore*, quest'ultimo può essere ottenuto in modo efficiente attraverso una variante dell'algoritmo di Euclide. L'aritmetica nel caso delle funzioni razionali si riduce alla manipolazione di polinomi, per esempio $\frac{f_1}{f_2} + \frac{g_1}{g_2} = \frac{f_1 g_2 + f_2 g_1}{f_2 g_2}$. Per controllare se due

funzioni razionali $\frac{f_1}{f_2}$ e $\frac{g_1}{g_2}$ sono uguali equivale a verificare che $\frac{f_1}{f_2} - \frac{g_1}{g_2}$ sia un polinomio nullo.

Per valutare la complessità dell'algoritmo proposto da Param, prima di tutto, va analizzato il passo di eliminazione e aggiornamento delle funzioni di transizione, il quale richiede $\vartheta(n^2)$ operazioni polinomiali nel caso peggiore. Ne consegue che l'algoritmo completo richieda un numero di operazioni asintotico con $\vartheta(n^3)$ per ottenere la funzione razionale desiderata. La complessità aritmetica dell'esecuzione in caso di polinomi però dipende dal grado del sistema stesso; presa n la dimensione del sistema, la dimensione della funzione razionale finale è nel caso peggiore di lunghezza $n^{\Theta(logn)}$. Per le complessità delle tecniche di bisimulation si rimanda al capitolo 6.

Capitolo 4

Run-Time Efficient Model Checking

Spesso i sistemi software sono progettati, sviluppati ed implementati per operare in un ambiente immutabile e completamente noto, con profili operazionali e requisiti costanti, che non cambiano al trascorrete del tempo e al variare delle condizioni. In un ambiente di questo tipo, come introdotto in precedenza, qualsiasi cambiamento è imprevedibile ed in grado di paralizzare l'intero sistema, compromettendo la sua capacità di soddisfare i requisiti per i quali è stato progettato. La crescita sia delle dimensioni che degli ambiti di impiego dei software, ha inevitabilmente introdotto variabili e scenari imprevedibili da valutare completamente a design-time; i cambiamenti accadono sempre più spesso e costituiscono uno dei fattori dominanti dei sistemi software al giorno d'oggi. Le attuali tecniche di Model Checking e i relativi tools sono concepiti per un uso a design-time e difficilmente sono in grado di soddisfare i vincoli di tempo e occupazione della memoria imposti da un'analisi a run-time, primo problema tra tutti l'esplosione dello spazio degli stati, inevitabile durante l'analisi del modello. Per ottenere un sistema in grado di soddisfare suddetti vincoli, bisogna modificare la struttura logica di esecuzione del MC dei tools attuali. L'idea base è quella di ottenere sistemi software che provano ad adattare loro stessi con lo scopo di preservare i *requisiti di*

affidabilità in presenza di cambiamenti. Si ripropone la tecnica di *monitoring* e *models* per la valutazione del sistema reale e aggiornamento del modello.

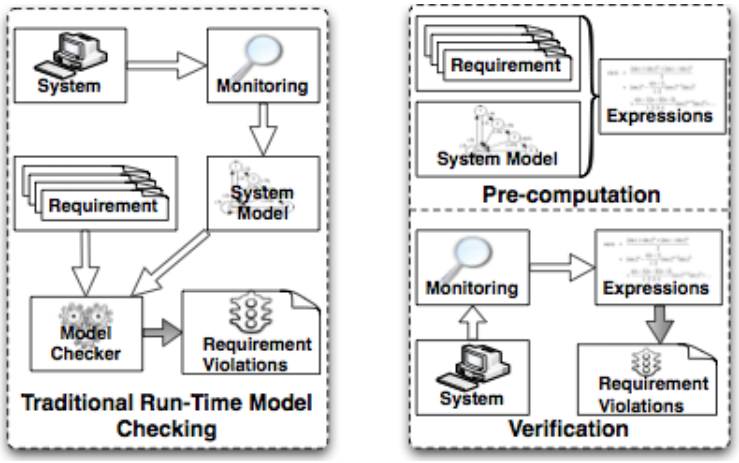

Figura 4.1: Tecniche di Run-Time Model Checking.

La tipologia dei requisiti qui trattati è costituita solamente dalla classe dei vincoli di affidabilità, la classe di modelli sulla quale verrà basata l'inferenza sarà quella delle *Catene di Markov Parametriche a Tempo Discreto* (PMC). La figura 3.1 riassume l'approccio proposto per risolvere problemi di *Run-Time efficient Model Checking* (schema di destra). In questo capitolo verra formalizzato un metodo teorico per ottenere una valutazione efficiente dei requisiti di affidabilità a run-time.

4.1 L'approccio WorkingMom

La prima assunzione che risulta necessaria riguarda il sistema in fase di sviluppo, il quale deve essere modellizzato come una Catena di Markov a Tempo Discreto (DTMC). Le DTMC sono un formalismo largamente accettato nel campo dell'affidabilità di componenti e nei sistemi service-based; in particolare risultano essere molto utili per una prima valutazione o previsione di affidabilità. L'adozione di un formalismo come le DTMC richiede che il sistema modellizzato rispetti, in prima approssimazione, le proprietà di Mar-

kov descritte nel capitolo 2; per ulteriori dettagli si fa riferimento a [4]. Come accade nella maggior parte degli approcci di design basati sulle DTMC, in questo elaborato viene fatta l'assunzione che il modello descrive comportamenti che dipendono: da un profilo di interazione e da probabilità di failure, che sono usate per etichettare le transizioni della catena. Per alcuni di questi valori risulta difficile, praticamente impossibile, effettuare una previsione a design-time; in pratica, un software designer può fare affidamento su stime relative alle probabilità di failure, ottenute dall'esecuzione di istanze di sistemi simili, ma in alcuni casi questi valori possono variare al trascorrere del tempo dopo che il sistema è stato sviluppato e distribuito.

Un'altra assunzione necessaria da fare riguarda la libertà a run-time che, attraverso un'attenta analisi a design-time, deve essere ristretta ad un sottoinsieme di parametri d'ambiente. Più precisamente è necessario che:

- Le possibili variazioni del modello a run-time siano tutte prevedibili durante la fase di design.
- Il numero di queste transizioni variabili, sia una frazione di quelle presenti in tutto il modello.

Queste assunzioni risultano comunque valide nella maggior parte dei casi pratici; nel caso in cui non siano verificate, si può continuare ad applicare questo approccio ma in questo modo si perdono i benefici in termini di speed-up durante la verifica a run-time. Riprendendo velocemente le DTMC, esse vengono definite come modelli stato transizione alla quale è associata una probabilità. Gli *stati* rappresentano tutte le possibili configurazioni del sistema, le *transizioni* invece sono relazioni tra stati alle quali è associato un valore di probabilità. Le DTMC sono definite come processi stocastici discreti che godono delle proprietà di Markov, le quali affermano che la distribuzione probabilistica degli stati successivi è caratterizzata solamente dallo stato attuale. Formalmente una DTMC è una tupla $(s, S_0, \mathbf{P}, V)$ dove: S è un insieme di stati, S_0 è il set degli stati iniziali (solitamente uno), P è un'applicazione da $S \times S$ a $[0, 1]$ che gode della proprietà $\sum_{s' \in S} \mathbf{P}\left(s, s'\right) = 1$

$\forall s \in S$ e $V : S \to 2^{AP}$ è la funzione di valutazione che assegna ad ogni stato s l'insieme delle *proposizioni atomiche* che sono vere in s. Se P viene estesa per assumere anche valori parametrici, si ottiene una Catena di Markov Parametrica PMC; d'ora in poi si farà riferimento solamente a PMC. Se una PMC ha almeno uno stato assorbente, ovvero vale $\mathbf{P}(s, s) = 1$, la catena si dice *assorbente*. In una PMC con r stati assorbenti e t stati transienti le colonne e le righe della matrice si possono ordinare in modo da ottenere la seguente *forma canonica*:

$$P = \begin{bmatrix} Q & R \\ 0 & I \end{bmatrix}$$

La matrice Q contiene le probabilità associate alle transizioni tra due stati non assorbenti, R quelle tra uno stato transiente ed uno assorbente, 0 e I compongono la sottomatrice relativa alle transizioni degli stati assorbenti (hanno come unico successore loro stessi); le dimensione e le caratteristiche di queste matrici sono:

- I è una matrice identità di dimensione r per r.
- 0 è una matrice nulla r per t.
- R è una matrice non nulla r per t.
- Q è una matrice non nulla t per t.

Presi due stati transienti distinti s_i e s_j, la probabilità di raggiungere s_j da s_i in esattamente due passi è $\sum_{s_x \in S} \mathbf{P}(s_i, s_x) \cdot \mathbf{P}(s_x, s_j)$. Generalizzando, per un percorso composto da k passi, risulta necessario richiamare la definizione di prodotto matriciale; si ottiene che la probabilità di passare da uno stato transiente s_i ad un altro s_j in esattamente k passi corrisponde al valore (s_i, s_j) della matrice Q^k. Attraverso questo principio possiamo definire Q^0, nella quale gli elementi associati ad una coppia (s_i, s_j) di stati assumono il valore 1 nel caso in cui $i = j$, 0 altrimenti; si ottiene così una matrice identità di dimensione t per t. Grazie al fatto che R è una matrice non nulla e P

è una matrice stocastica, Q risulta una sottomatrice con norma uniforme strettamente minore di 1, si ottiene così il

Teorema 4.1.1. *Convergenza di Q^n:*
Presa una matrice Q con norma uniforme strettamente minore di 1, implica che $Q^n \to 0$ al tendere di $n \to \infty$.

Questo implica che l'intero processo p rappresentato dalla matrice P verrà assorbito con probabilità = 1; in modo meno formale l'implicazione asserisce che: la probabilità per un cammino di lunghezza n di non finire in uno stato assorbente al tendere di $n \to \infty$ è 0.

Di seguito è presentato un semplice esempio basato sul protocollo *zero-conf* dell'IPv4, dove è possibile applicare con successo una modellizzazione attraverso DTMC; il protocollo zerofonf opera come segue: un host H ha bisogno di una configurazione e sceglie casualmente un indirizzo IP U, H manda in broadcast un messaggio nel quale chiede se esiste qualche altro host che usa l'indirizzo U; se il messaggio è ricevuto da un host che sta già utilizzando quell'indirizzo, quest'ultimo replica comunicando in broadcast che U è già in uso. Dopo aver ricevuto questo messaggio, H ricomincia l'esecuzione dell'algoritmo: nuovo indirizzo, broadcast, verifica.... A causa di fattori imprevedibili come la perdita di un messaggio o l'host ricevente occupato, la verifica o la risposta potrebbero non arrivare agli altri host; Per cercare di incrementare l'affidabilità del protocollo, ad ogni host è richiesto di inviare n messaggi di verifica, ognuno seguito da un periodo di ascolto di r unità di tempo. L'host può quindi iniziare ad usare l'indirizzo IP scelto soltanto dopo n prove inviate ed aver atteso $n \cdot r$ unità di tempo senza aver ricevuto nessuna risposta. Va notato che, dopo l'esecuzione dell'algoritmo, l'host H può ottenere un indirizzo IP già in uso da un altro host; ciò accade quando tutte le verifiche vengono perse. Questa situazione, chiamata *address collision*, è molto indesiderata poiché forza un host ad uccidere tutte le sue connessioni TCP/IP attive.

Il comportamento dell'esecuzione del protocollo relativa ad un singolo host è modellizzabile da una Catena di Markov costituita da $n+5$ stati (Rif.

Figura 4.2 per $n = 4$) dove n è il numero massimo di prove necessarie. Lo stato s_0 etichettato come *start* corrisponde allo stato iniziale, in s_{n+4} (*ok*) l'host termina l'esecuzione dell'algoritmo con un indirizzo IP non utilizzato, nello stato s_{n+2} (etichettato come *errore*) invece l'host alla fine si trova con un indirizzo già in uso, ovvero in una situazione di address collision.

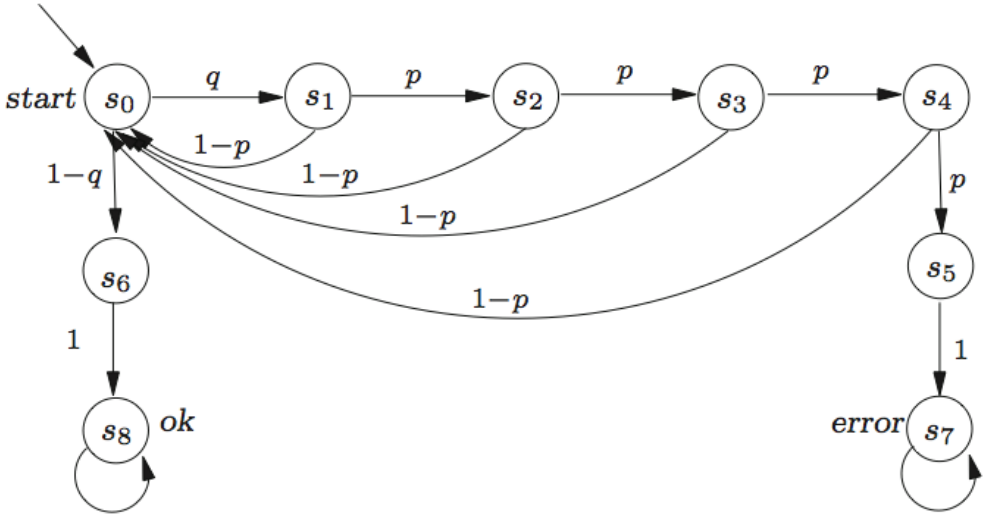

Figura 4.2: PMC rappresentante il protocollo zeroconf dell'IPv4 (per $n = 4$ prove).

Ogni stato s_i $(0 < i \leq n)$ viene raggiungo dopo aver eseguito l'i-esima prova. Nello stato iniziale s_0, l'host sceglie in modo casuale un indirizzo IP; la probabilità di collisione è pari a $q = m/65024$ dove m è il numero degli host nella rete, mentre, con probabilità $1 - q$, l'host sceglie un indirizzo non utilizzato e conclude nello stato s_{n+3}, ma deve inviare prima $n - 1$ prove ed aspettare $n \cdot n$ unità di tempo per poter utilizzare questo indirizzo (Questo viene astratto dalla PMC). Se invece l'indirizzo Ip scelto è già in uso, viene raggiunto lo stato s_1 e da qui si prospettano due situazioni differenti. Con probabilità p nessuna risposta viene ricevuta durante r unità di tempo (potrebbe essere andata persa sia la prova che la risposta) e un nuovo messaggio viene inviato portando il sistema nello stato s_2; se invece una risposta è arrivata in tempo, l'host ritorna nello stato iniziale e ricomincia l'esecuzione del

protocollo. Il comportamento negli stati s_i con $(2 \leq i < n)$ è simile a quello appena descritto: se nello stato s_n, dopo aver inviato l'n-sima richiesta e trascorse r unità di tempo, non viene ricevuta alcuna risposta, potrebbe essere accaduta una collisione tra indirizzi.

Le transizioni etichettate con variabili, ad esempio p, q, $1-p$, danno libertà al modello di adattarsi a run-time, questo perché i valori sono difficili da stabilire durante la fase di design e possono variare durante l'esecuzione. Il modello PMC proposto può essere anche espresso in forma matriciale.

$$Q = \begin{bmatrix} 0 & q & 0 & 0 & 0 & 0 & (1-q) \\ (1-p) & 0 & p & 0 & 0 & 0 & 0 \\ (1-p) & 0 & 0 & p & 0 & 0 & 0 \\ (1-p) & 0 & 0 & 0 & p & 0 & 0 \\ (1-p) & 0 & 0 & 0 & 0 & p & 0 \end{bmatrix} \quad R = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Questo esempio giocattolo è stato usato per introdurre l'approccio Working-Mom e far comprendere le sue potenzialità; infatti PMC che descrivono concetti applicabili a sistemi reali possono avere migliaia di stati e proprietà complesse da verificare, per questo è necessario introdurre soluzioni efficienti ed efficaci.

Per esprimere proprietà formali ed effettuare inferenza su sistemi modellati sia con DTMC che PMC esiste una specifica famiglia di logiche chiamate temporali. In questo elaborato ci si focalizzerà sulla logica PCTL, introdotta nel capitolo 2, con la quale è possibile formulare una considerevole quantità di proprietà relative alla reliability. PCTL è una logica ispirata a CTL ed è costituita da due tipi di formule: di *stato* e di *cammino*. La definizione formale di come si calcola $\mathbf{P}(s \models \varphi)$ è stata definita in precedenza, più semplicemente il suo valore corrisponde alla frazione dei path originati in s che soddisfano φ rispetto all'intero insieme di path originati in s. La proprietà più importante esprimibile in PCTL è senza ombra di dubbio la *reachability*. Una proprietà di reachability afferma che uno stato, dove risulta vera una certa proprietà caratteristica, è raggiunto in qualche modo da un dato stato

iniziale; nella maggior parte dei casi lo stato che si desidera raggiungere è uno stato assorbente. Lo stato raggiunto può rappresentare una *situazione* di *failure*, dove il sistema modellizzato dalla PMC potrebbe trovarsi bloccato o semplicemente in una situazione indesiderata; dualmente si potrebbe raggiungere uno *stato* di *success*. Le proprietà di *reachability* sono espresse in PCTL nella forma $\mathcal{P}_{\bowtie p}\left(true\ \boldsymbol{U}\ \Phi\right)$ (espressa anche come $\mathcal{P}_{\bowtie p}\left(\boldsymbol{F}\left(\Phi\right)\right)$), la quale descrive in modo rigoroso il concetto di raggiungere uno stato qualsiasi che soddisfa Φ e che la probabilità di raggiungerlo sia all'interno dell'intervallo definito da $\bowtie p$. E' necessario che Φ sia una formula di stato semplice, ovvero che non includa sottoformule di cammino al suo interno; nella maggior parte dei casi Φ è una semplice preposizione atomica che è valida solo in uno stato assorbente della PMC. Nel caso di uno stato di *failure*, il bound probabilistico sarà nella forma $\leq p$, dove p rappresenta l'upper bound per la probabilità di failure, per uno stato di *success* accade il duale il vincolo verrà espresso nella forma $\geq p$,dove p è il lower bound per il success. Una cosa importante da notare è che la rete viene adattata in base alla proprietà da verificare poiché lo stato iniziale della PMC non è necessariamente lo stato iniziale del path π dove la formula è verificata. Lo stato iniziale della rete è intuitivamente la partenza del cammino π nel caso in cui la proprietà da verificare sia di reachability. Ora vengono presentate metodologie dal punto di vista prettamente matematico per calcolare la formula analitica relativa alle proprietà da verificare a run-time. Si inizierà definendo una soluzione per analizzare proprietà di reachability e progressivamente fornendo metodi per coprire tutte le formule PCTL.

In questa sezione viene illustrato come una formula PCTL possa essere pre-processata a design time. La precomputazione genera una formula per ogni proprietà PCTL; la formula è costituita da una espressione analitica contenente variabili che diventeranno note a run-time. Le variabili corrispondono a probabilità delle transizioni che sono sconosciute o incerte a design time. E' importante sottolineare che nel caso in cui una transizione parametrica sia uscente da uno stato, è necessario che ne esista almeno un'altra e

che la somma tra le variabili sia sempre e strettamente uguale a 1. Un'assunzione importante da fare è che se esiste una transizione variabile uscente da uno stato, tutte le altre uscenti saranno dello stesso tipo. Dal punto di vista teorico le variabili sono di natura atomica, ma nella realtà è possibile esprimerle sotto forma di espressioni, ad esempio: preso uno stato dal quale escono 3 transizioni, una possibile assegnazione parametrica potrebbe essere p, q, $(1-p-q)$. Questo permette di ridurre il numero dei parametri all'interno del modello. Stati con transizioni parametriche uscenti sono definiti *stati variabili*. Ora verranno introdotte le tecniche per la valutazione delle possibili formule esprimibili in PCTL basandosi su PMC, iniziando con quelle che definiscono proprietà di reachability

4.1.1 Proprietà di Reachability

La reachability è, senza ombra di dubbio, la proprietà più studiata nel campo dell'analisi dell'affidabilità poiché esprime il concetto di raggiungere un certo stato, che tipicamente rappresenta il raggiungimento nel sistema di una situazione di *success* (desiderata) oppure di *failure* (indesiderata). Entrambe le situazioni failure e success sono modellizzate come stati assorbenti. Una formula di reachability è espressa nella forma $\mathcal{P}_{\bowtie p}\boldsymbol{F}l$ dove l è una formula atomica o appartenete alla logica proposizionale dello stato obiettivo. Viene mostrato ora come precomputare a design time una formula relativa alla proprietà di reachability per uno stato assorbente di una PMC; nel caso in cui lo stato sia transiente e la proprietà l sia verificata, lo stato viene tramutato in assorbente. Per calcolare la probabilità associata alla reachability è necessario calcolare la probabilità tra i transienti di raggiungersi attraverso un path π in $0, 1, 2, ...$ passi, il che corrisponde a calcolare la matrice Q^i e sommarla a tutte le quelle con potenza inferiore a i ovvero $N = I + Q^1 + +Q^2 + ... = \sum_{i=0}^{\infty} Q^i$. N si può calcolare come matrice di convergenza di questa serie come $I - Q$; il valore $n_{i,j}$ di N rappresenta la probabilità attesa di raggiungere uno stato s_j della PMC, partendo da uno

stato iniziale s_i. Invece il valore assunto da $q_{i,j}$ corrisponde alla probabilità di passare da un transiente s_i ad un altro s_j in esattamente un passo.

E' importante sottolineare la proprietà di convergenza di Q ovvero $Q^n \to 0$ quando $n \to \infty$, questo implica che il processo verrà sempre assorbito con probabilità 1 dopo un numero sufficientemente grande di passi. La fase di maggior interesse per la valutazione della proprietà di reachability è sicuramente il calcolo della distribuzione della probabilità rispetto agli stati assorbenti. Questa distribuzione può essere calcolata in forma matriciale come segue:

$$B = N \times R$$

dove $r_{i,k}$ rappresenta la probabilità che, in un passo a partire da s_i, il sistema finisca in uno stato assorbente s_k; B è una matrice di dimensione $t \times r$ che rappresenta la probabilità di ogni condizione di essere verificata partendo da qualsiasi stato della PMC, preso come iniziale. In dettaglio un elemento $b_{i,j}$ della matrice B rappresenta la probabilità di finire in uno stato assorbente s_j partendo da s_i. La computazione di un elemento $b_{i,j}$, generalmente, può essere eseguita soltanto in modo simbolico, poiché alcune transizioni variabili possono essere attraversate nel raggiungere s_j. Definita $W = I - Q$, gli elementi della sua inversa N sono definiti come segue:

$$n_{i,j} = \frac{1}{det\,(W)} \cdot \alpha_{ji}\,(W)$$

dove α_{ji} rappresenta il complementare algebrico dei w_{ji}. Si ottiene che:

$$b_{ik} = \sum_{x \in 0..t-1} n_{ix} \cdot r_{xj} = \frac{1}{det\,(W)} \sum_{x \in 0..t-1} \alpha_{xi}\,(W) \cdot r_{xj}$$

Nel campo dell'analisi numerica, esistono diversi modi per eseguire il calcolo di b_{ik}: partendo dalla definizione matematica di determinante, metodo di eliminazione Gaussiana, metodo di Gauss-Jordan, fino ad arrivare a tecniche e soluzioni specifiche per matrici sparse. L'impiego di tecniche specifiche per matrici sparse, come si vedrà in seguito, è fondamentale in questo tipo

di problemi, poiché la matrice che rappresenta una rete PMC è di natura intrinsecamente sparsa, ovvero il numero di transizioni in uscita da uno stato (valori sulla riga diversi da zero) è molto minore della cardinalità degli stati stessi (dimensione della matrice).

In conclusione è stato presentato l'approccio matriciale scelto per la valutazione delle proprietà nella forma $\mathcal{P}_{\bowtie p}(\mathbf{F} s_k)$; Param si basa su un algoritmo diverso che opera sfruttando la topologia della rete (variante dell'algoritmo di Daws). Attraverso la procedura appena proposta, è possibile ottenere formule chiuse per un numero interessante di proprietà di reliability.

Viene preso ora in considerazione il calcolo della probabilità di raggiungere con successo uno stato s_j che non sia uno stato assorbente. Il valore da stimare è f_{ij}, che corrisponde alla probabilità di effettuare sempre una transizione in uno stato s_j, sapendo che il processo è partito da s_i. Formalmente, preso f_{ij}^n che rappresenta la probabilità di spostarsi da uno stato transiente s_i ad un altro s_j, del medesimo tipo, raggiungendolo per la prima volta in esattamente n passi, definiamo:

$$\begin{cases} f_{ij}^0 = 0 \ \ \forall i \neq j; \quad f_{ij}^0 = 1 \ \ \forall i = j \\ f_{ij}^n = Pr\left\{X_n = s_j \wedge X_k \neq s_j \ \ \forall (1 \leq k \leq (n-1)) \mid X_0 = s_i\right\} \end{cases}$$

dove X_k rappresenta lo stato del sistema al passo di esecuzione k, si ottiene

$$f_{ij} = \sum_{n=0}^{\infty} f_{ij}^n$$

E' possibile calcolare il valore f_{ij} dalla matrice N attraverso la modifica dell'elemento n_{ij}, quest'ultimo rappresenta numero atteso di volte che la PMC si trova nello stato s_j:

$$n_{ij} = n_{jj} fij$$

si ottiene che:

$$f_i j = \frac{n_{ij}}{n_{jj}} = \frac{\alpha ji\,(W)}{\alpha jj\,(W)}$$

Dal punto di vista teorico è quindi possibile calcolare la probabilità di spostarsi tra due transienti riducendo il calcolo al rapporto tra due determinanti di due matrici con dimensione $t - 1$. In campo pratico, manipolare a design time la rete non richiede un costo computazionale eccessivo; quindi, quando uno stato s_j, per esempio, verifica una certa proprietà di reachability, risulta più efficace trasformare lo stato in assorbente, nel caso in cui non lo sia. Iterando questa filosofia per tutti gli stati che verificano o meno la proprietà, in molti casi si ottiene una drastica riduzione della dimensione della rete con un conseguente aumento delle performance a design-time. Esiste, tuttavia, un'altro metodo in grado di risolvere proprietà di reachability, basato sull'assegnazione di un valore agli stati assorbenti, che verrà poi propagato a tutti gli altri stati mediante la risoluzione di un sistema lineare; questa procedura verrà illustrata nel prossimo capitolo.

4.1.2 Estensione a full PCTL

Nella sezione precedente sono state presentate soluzioni limitate alla proprietà di reachability. Anche se quest'ultima rappresenta il modello principale nella descrizione di requisiti nel campo dell'affidabilità, non è in grado di coprire tutte le proprietà ed i vincoli di cui gli ingegneri necessitano nelle applicazione reali. Ora verranno proposte soluzioni in grado di estendere questo approccio a tutte le proprietà esprimibili in PCTL. Di seguito vengono vengono proposti metodi per gestire prima formule di tipo *flat* e poi formule contenenti operatori probabilistici annidati.

La procedura per valutare i requisiti nella forma $\Phi_1 \, \boldsymbol{U} \, \Phi_2$ si divide in due fasi:

- Modificare la PMC per poter ricondurre la proprietà in forma di reachability.
- Applicare le tecniche mostrate nella sezione precedente per risolvere l'espressione associata alla probabilità.

La procedura di riduzione inizia con la costruzione di una PMC $\overline{P}$ definita come segue. Di seguito faremo riferimento all'insieme degli stati di $\overline{P}$ indicandolo con $S_{\overline{P}}$. Questo insieme è costituito dall'unione di tre sottoinsiemi disgiunti: S_{goal}, $S_{\neg goal}$ e $S_{transient}$ definiti rispettivamente:

- L'insieme degli stati assorbenti della PMC originaria P con l'aggiunta degli stati in cui Φ_2 risulta vera.
- L'insieme di tutti gli stati della PMC originaria P aggiungendo gli stati in cui Φ_1 è falsa.
- La restante parte degli stati originari della PMC.

Ora rimane solo da definire come saranno le transizioni di $\overline{P}$, partendo da come sono formulate in P. Di seguito è proposto un metodo valido per completare questa trasformazione costituito da due fasi distinte:

- L'eliminazione di tutte le transizioni uscenti da ogni stato transiente s_i della PMC originale nel quale è verificata $(\neg\Phi_1 \wedge \neg\Phi_2)$, la trasformazione di questi stati in assorbenti ed infine l'aggiunta degli stessi nel set $s_{\neg goal}$.
- L'inserimento nell'insieme s_{goal} di tutti gli stati s_j della PMC originaria dove è verificata Φ_2, trasformandoli così in stati finali.

Gli stati in s_{goal} rappresentano le situazioni del sistema nelle quali la formula è soddisfatta. Infatti tutte le formule di cammino nella forma $\Phi_1 \, \boldsymbol{U} \, \Phi_2$ sono verificate in ciascuno gli stati in cui Φ_2 risulti vera; dopodiché ogni stato predecessore (di un cammino qualsiasi) deve avere Φ_1 vera e Φ_2 falsa. Nel caso in cui Φ_2 risultasse vera, verrebbe applicato il primo passo dell'algoritmo proposto trasformando lo stato in assorbente. Invece tutti gli stati nei quali è vera $(\neg\Phi_1 \wedge \neg\Phi_2)$ vengono trasformati in assorbenti e inseriti nell'insieme $s_{\neg goal}$. Dopo l'esecuzione di questi due passi, la proprietà Until viene ridotta alla semplice valutazione di reachability: $\mathcal{P}_{\bowtie p}\left(F\left(s \in S_{goal}\right)\right)$.

Viene preso ora in considerazione l'ultima parte del sottoinsieme *flat* (privo di operatori probabilistici annidati) di PCTL; Questo sottoinsieme è costituito da formule contenenti gli operatori: Bounded Until $\left(\Phi_1 \ \boldsymbol{U}^{\leq n} \ \Phi_2\right)$ e Next ($X\left(\Phi\right)$). Questi due operatori si differenziano da F e U ovviamente per la semantica associata, ma anche per la tipologia del cammino sul quale la proprietà viene verificata . L'insieme dei cammini da prendere in considerazione, per valutare una formula nella forma $X\left(\Phi\right)$ in uno stato s_i, è composto da tutti i percorsi di lunghezza 1 che partono da s_i. La dimensione massima di questo insieme di stati è, nel caso peggiore, n (cardinalità del modello). La probabilità che in uno stato s_i della PMC sia verificata $s_i \models X\Phi$ è pari a:

$$Pr\left(X\Phi\right) = \sum_{s_j:\ s_j \models \Phi \ \wedge \ (s_i,s_j)\in R} \left(p_{ij}\right)$$

il che equivale a sommare le probabilità associate alle transizioni nello stato s_i che sono entranti in stati s_j dove è verificata Φ ovvero $s_j \models \Phi$.

Un approccio similare è applicabile nel caso di formule del tipo Bounded Until. Un percorso π con origine in s_i soddisfa la formula $\Phi_1 \ \boldsymbol{U}^{\leq n} \ \Phi_2$ se in un certo stato $s_k \in \pi$ è verificata $s_k \models \Phi_2$ con $k \leq n$ e se in ogni stato s_i con $i < k$ è verificata Φ_1. Per verificare il Bounded Until diventa quindi necessario considerare tutti i path con lunghezza $l \leq n$; anche in questo caso è necessario modificare la rete P trasformandola in $\overline{P}$ seguendo l'algoritmo proposto per la proprietà Until. Ottenuta $\overline{P}$, rimane da calcolare la probabilità di spostarsi, in $n+1$ passi, da s_i a s_k, il che corrisponde a calcolare l'elemento p_{ij}^n dell'n-sima potenza della matrice di transizione di $\overline{P}$. In formule si ottiene che:

$$Pr\left(\Phi_1 \ \boldsymbol{U}^{\leq n} \ \Phi_2\right) = \sum_{j:j\in S_{goal}, i\in S_{init}} p_{ij}^n$$

Rimane ora da valutare il sottoinsieme che completa la casistica delle proprietà formulabili in PCTL. Viene in questa classificazione definito sottoinsieme ma in realtà questo metodo è applicabile a qualsiasi formula esprimibile in PCTL. L'approccio proposto risulta nettamente meno efficiente nel caso in cui la formula sia riconducibile ad uno dei casi precedenti. Quando

una formula di cammino $\mathcal{P}_{\bowtie p}(\varphi)$ contiene al suo interno almeno una sottoformula che è, a sua volta, una path-formula, per la valutazione a run-time servono ulteriori informazioni che non sono disponibili a run-time. Come esempio prendiamo una formula del tipo $\mathcal{P}_{\bowtie p_1}(true\ \boldsymbol{U}\ (\mathcal{P}_{\bowtie p_2}(true\ \boldsymbol{U}\ \Phi_2)))$. Per ottenere una valutazione degli stati di P in cui è vera la proprietà di reachability interna, è necessario conoscere i valori associati alle transizioni variabili. Queste informazioni, come detto in precedenza, saranno disponibili solamente a run-time. Ne consegue quindi, che per valutare una formula composta da operatori $\mathcal{P}$ annidati, dobbiamo prima conoscere i valori di verità delle sue sottoformule. Lo stesso ragionamento è applicabile alle sottoformule delle sottoformule, ricorsivamente fino ad ottenere espressioni prive di operatori $\mathcal{P}$ (formula flat). L'idea base è quindi quella di generare un albero composto da tutte le sottoformule della formula data, avendo come nodi foglia tutte e sole le espressioni di tipo *flat*. In questo modo verranno valutate per prime le sottoespressioni prive di annidamenti (nodi foglia) per poi ripercorrere all'indietro l'albero valutando ogni volta tutte le altre fino ad arrivare a quella che etichetta il nodo radice (la più esterna). Per sviluppare una soluzione è necessario trovare un modo per ritardare la valutazione della formula a run-time, momento in cui tutte le sottoformule saranno valutate fornendo le informazioni mancanti. Vengono quindi introdotti parametri aggiuntivi a design time, per poter tener conto in seguito della mancanza di informazioni relative alla soddisfacibilità delle sottoformule; questi parametri costituiscono una struttura, introdotta a design time, che verrà totalmente rimossa durante la valutazione a run time.

Ora verranno analizzate le metodologie di valutazione, per ognuna delle tipologie di sottoformule che possono presentarsi sull'albero. Vengono innanzitutto prese in considerazione le formule di tipo Until ($\Phi_1\ \boldsymbol{U}\ \Phi_2$). La procedura per risolvere questo tipo di formule richiama quella proposta per la risoluzione di proprietà del flat Until, ovvero la costruzione di una rete $\overline{P}$ attraverso due operazioni basilari:

- La parametrizzazione di tutte le transizioni contenute nella matrice Q e R.
- L'introduzione delle transizioni parametriche verso gli stati assorbenti s_{goal} e $s_{\neg goal}$ in R.

Da un punto di vista matematico, eliminare una transizione è equivalente ad etichettare l'elemento della matrice con probabilità 0. Moltiplicando ogni transizione non nulla p_{ij} della PMC originale con un parametri $a_j \in \{0, 1\}$ è possibile ritardare la decisione di eliminare o mantenere la transizione assegnando 0 o 1 al corrispondente coefficiente a_j. Per completare la costruzione di $\overline{P}$ è necessario collegare ogni stato transiente con gli assorbenti s_{goal} e $s_{\neg goal}$; per fare questo completiamo la struttura di parametri introducendo due transizioni $b_{is\,goal}$ e $b_{is\neg goal}$, atte a collegare ogni stato s_i con s_{goal} e $s_{\neg goal}$. I valori dei parametri a_j e b_{ij} vengono assegnati durante la valutazione a run time. Facendo riferimento all'esempio del protocollo zeroconf si ottengono le seguenti matrici Q e R:

$$Q = \begin{bmatrix} 0 & a_0 q & 0 & 0 & 0 & 0 & \alpha_0 (1-q) \\ \alpha_1 (1-p) & 0 & \alpha_1 p & 0 & 0 & 0 & 0 \\ \alpha_2 (1-p) & 0 & 0 & \alpha_2 p & 0 & 0 & 0 \\ \alpha_3 (1-p) & 0 & 0 & 0 & \alpha_3 p & 0 & 0 \\ \alpha_4 (1-p) & 0 & 0 & 0 & 0 & \alpha_4 p & 0 \end{bmatrix} \quad R = \begin{bmatrix} 0 & 0 & \beta_{0,2} & \beta_{0,3} \\ 0 & 0 & \beta_{1,2} & \beta_{1,3} \\ 0 & 0 & \beta_{2,2} & \beta_{2,3} \\ 0 & 0 & \beta_{3,2} & \beta_{3,3} \\ 0 & 0 & \beta_{4,2} & \beta_{4,3} \end{bmatrix}$$

L'ultimo passo nella procedura di decisione è costituito dalla somma delle probabilità di raggiungere ogni stato contenuto in s_{goal}; questo insieme sarà noto solamente a run time, quando le sottoformule Φ_1 e Φ_2 verranno valutate. A design time verrà calcolata la matrice B relativa alla PMC originale con l'aggiunta dei parametri, secondo il metodo appena proposto. A run-time, quando, a seguito delle valutazioni delle sottoformule, l'insieme s_{goal} diventerà noto, verranno assegnati i valori a a_j e b_{ij}, ottenendo così l'espressione associata alla soddisfacibilità della sottoformula. Per quanto riguarda

gli operatori Next e Bounded Until, i metodi proposti per le formule flat valgono anche in presenza di sottoformule annidate; tuttavia in questo caso rimane da valutare quanto sia conveniente adottare questa tecnica e se esistono alternative valide.

L'ultima osservazione riguarda la valutazione eseguita a run-time nel caso in questione, poiché la valutazione della formula PCTL non può più avvenire in un singolo passo come accadeva per le formule di tipo flat. Comunque il numero di espressioni da valutare cresce in modo *lineare* rispetto al numero di operatori di cammino presenti nella formula oggetto della valutazione. Nel prossimo capitolo verranno affrontate le stesse proprietà proponendo la soluzione adottata dal punto di vista algoritmico, ovvero verranno descritte le tecniche e le soluzioni adottate dal punto di vista delle prestazioni e verranno discusse le semplificazioni introdotte per cercare di migliorare l'esecuzione del codice ed ottenere il massimo dei risultati.

Capitolo 5

Algoritmi di Verifica in Maple

In questo capitolo verranno analizzate tecniche e soluzioni per ottenere in modo efficiente le espressioni parametriche relative alla verità di proprietà (espresse in PCTL) a partire da un dato modello, con l'obiettivo principale di ottenere prestazioni massime per quanto riguarda il tempo di esecuzione. Per fare ciò è necessario impiegare tecniche di analisi numerica.

Questa disciplina ingegneristica, si occupa della risoluzione algoritmica di calcoli matematici, utilizza i principi dell'algebra lineare e delle successioni convergenti per risolvere la maggior parte dei problemi di cui si occupa. Negli ultimi anni l'impatto sul mondo reale dell'analisi numerica è stato decisivo ed ha permesso evoluzioni fino a prima impensabili in svariati campi dell'ingegneria. Il punto di forza degli algoritmi impiegati in questo campo è indubbiamente l'efficienza; infatti affiancando euristiche, basate su ipotesi statistiche a metodi rigorosi supportati da una teoria matematica, si possono ottenere attraverso un impiego mirato risultati reali migliori, anche se non esistono teoremi o dimostrazioni matematici in grado di provarlo.

Quando un metodo viene applicato ad una situazione reale, l'impiego di euristiche efficaci può determinare la differenza tra una totale impraticabilità nell'utilizzo ed un impiego pratico e veloce poiché gli speed-up, derivati dall'utilizzo di queste tecniche, possono raggiungere diversi ordini di grandezza. In generale è possibile affermare che l'analisi numerica è una scienza sia teo-

rica che sperimentale: infatti utilizza assiomi teoremi e dimostrazioni come il resto della matematica, affiancandogli però i risultati empirici delle elaborazioni eseguite, con l'obiettivo di incrementare il più possibile le prestazioni dei propri metodi. Nella parte restante del capitolo verranno descritti gli algoritmi utilizzati per il calcolo delle espressioni relative alle proprietà da verificare (nei modelli basati su PMC).

Gli algoritmi impiegati nell'ambito del calcolo numerico risolvono problemi completamente noti, ovvero costituiti soltanto da valori numerici; dato che il problema per il quale si sta cercando una soluzione contiene parametri, si cercherà di applicare i metodi attuali, adattandoli all'elaborazioni di polinomi. Prima però verranno descritti dal punto di vista formale gli algoritmi risolutivi specifici utilizzati poi per il calcolo delle varie espressioni riferite alle proprietà PCTL.

5.1 Soluzione di sistemi lineari parametrici

Come vedremo in seguito risolvere un sistema lineare parametrico sarà utile per il calcolo dell'espressione associata alle proprietà . La risoluzione di questi sistemi è da sempre oggetto di studi nell'ambito dell'analisi numerica che cerca di migliorarli per offrire algoritmi efficiente.

In matematica, e più precisamente in algebra lineare, un sistema di equazioni lineari, anche detto sistema lineare, è un insieme di equazioni (lineari) che devono essere verificate contemporaneamente. Una soluzione del sistema è un vettore i cui elementi sono le soluzioni delle equazioni che compongono il sistema, ovvero espressioni tali che, se sostituite alle incognite rendono le equazioni delle identità.

Definizione 5.1. Un sistema di equazioni lineari è un insieme di m equazioni lineari in n incognite, che può essere scritto nel modo seguente:

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 + \cdots + a_{1,n}x_n = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 + \cdots + a_{2,n}x_n = b_2 \\ \vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + a_{m,3}x_3 + \cdots + a_{m,n}x_n = b_m \end{cases}$$

Il numero n delle incognite è detto anche ordine del sistema. Il sistema può essere descritto usando la matrice:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & \cdots & a_{m,n} \end{bmatrix}$$

e un'ulteriore colonna:

$$b = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ \vdots \\ b_m \end{bmatrix}$$

detta colonna dei termini noti; la rappresentazione con equazioni e quella matriciale sono del tutto equivalenti. La matrice A è costituita dai coefficienti $a_{1,1}, \cdots, a_{1,n}, \cdots\cdots, a_{m,n}$. Ogni coefficiente $a_{i,j}$, nell'algoritmo proposto può assumere un *valore numerico razionale* ovvero $a_{i,j} \in \mathbb{Q}$, oppure può essere un'espressione contenete *parametri* affiancati da termini e coefficienti razionali del tipo $1 - x - y$. Nella forma matriciale le incognite $x_1, \cdots, x_n$ sono definite in modo implicito in base alla posizione del coefficiente associato: $a_{i,j}$ è il coefficiente di x_i nell'equazione j. Due sistemi si dicono equivalenti se ogni equazione di uno è combinazione lineare delle equazioni dell'altro

ed avranno di conseguenza soluzioni uguali. In forma matriciale estesa, il sistema è definito nel seguente modo:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Per evitare queste scritture prolisse d'ora in poi faremo riferimento quando, non strettamente necessario, al sistema lineare utilizzando la notazione compatta $A\mathbf{x} = \mathbf{b}$ dove A è la matrice $m \times n$ dei coefficienti, $\mathbf{x}$ è il vettore delle n incognite e $\mathbf{b}$ è il vettore degli m termini noti. In generale, un sistema può essere:

- Determinato: il sistema ammette un'unica soluzione.
- Indeterminato: il sistema ammette $1, 2, \cdots, j$ infinità di soluzioni.
- Impossibile: il sistema non ammette alcuna soluzione.
- Omogeneo: il vettore dei termini noti è nullo $\mathbf{b} = \overline{0}$.
- Numerico: le soluzioni sono numeriche.
- Parametrico: nelle soluzioni appaiono espressioni contenenti letterali.

Per risolvere i sistemi, i calcolatori non utilizzano i metodi classici come sostituzione e Kramer, ma sfruttano le enormi potenzialità di calcolo applicando, in continua successione, gli stessi passi fino ad ottenere la soluzione del sistema. I metodi per la risoluzione di sistemi di equazioni lineari possono essere divisi in due categorie. La prima categoria è quella dei metodi diretti ed ad essa appartengono, per esempio, il metodo di eliminazione Gaussiana e la fattorizzazione LU. I metodi diretti costruiscono l'esatta soluzione, a meno di errori di arrotondamento, in un numero finito di passi. In sostanza, utilizzano l'idea della fattorizzazione della matrice dei coefficienti del sistema nel prodotto di due matrici più semplici (solitamente triangolari o

ortogonali). Sono generalmente i metodi più efficienti, soprattutto quando operano su matrici dense; purtroppo su sistemi di grosse dimensioni e/o con matrice dei coefficienti sparse, tendono ad essere troppo onerosi in termini di consumo di memoria. In queste situazioni sono di norma preferibili i metodi appartenenti alla seconda categoria. La seconda categoria è quella dei metodi iterativi. Appartengono a questa categoria, per esempio, il metodo di Jacobi, il metodo di Gauss-Seidel e il metodo del gradiente coniugato. I metodi iterativi conducono alla soluzione del sistema in un numero teoricamente infinito di passi: partendo da un'approssimazione iniziale della soluzione, forniscono una serie di approssimazioni che, affiancate da opportune ipotesi, convergono verso la soluzione esatta. Il processo iterativo viene arrestato non appena la precisione desiderata è raggiunta. Il metodo applicato risulterà efficiente se la precisione desiderata sarà stata raggiunta in un numero accettabile di iterazioni. In matematica, il metodo di eliminazione di Gauss (MEG), che prende il nome dal matematico tedesco Carl Friedrich Gauss, è un algoritmo usato in algebra lineare per determinare le soluzioni di un sistema di equazioni lineari, per calcolare il rango o l'inversa di una matrice. L'algoritmo, attraverso l'applicazione di operazioni elementari, dette mosse di Gauss, riduce la matrice in una forma detta a scalini.

5.1.1 Metodo di Eliminazione Gaussiana

Il metodo di eliminazione Gaussiana mira alla riduzione di un sistema nella forma $A\mathbf{x} = \mathbf{b}$ in un sistema equivalente (ovvero con le medesime soluzioni) nella forma $U\mathbf{x} = \hat{\mathbf{b}}$, dove U è una matrice triangolare alta e $\hat{\mathbf{b}}$ è il vettore dei termini noti aggiornato. Quest'ultimo sistema può essere risolto tramite backsubstitution (definita in seguito). Durante la procedura di riduzione viene sfruttata la proprietà dei sistemi lineari secondo la quale sostituendo una delle equazioni, con la differenza tra se stessa ed un'altra moltiplicata per un coefficiente costante non nullo, permette di ottenere un sistema equivalente (con stessa soluzione). Consideriamo ora una matrice non singolare (determinante diverso da zero) A, è necessario che la matrice

conservi questa caratteristica almeno fino all'assegnazione dei valori ai parametri. Supponendo ora che in A l'elemento sulla diagonale $a_{1,1}$ non sia nullo, è possibile definire il *fattore moltiplicativo*

$$m_{i,1} = \frac{a_{i,1}^{(1)}}{a_{1,1}^{(1)}}, \quad i = 2, 3, \cdots, n$$

dove $a_{i,j}^{(1)}$ denota l'elemento in posizione (i, j) di $A^{(1)}$. E' quindi possibile eliminare l'incognita x_1 dalla altre equazioni, ad eccezione che dalla prima, semplicemente sottraendo alla riga i, con $i = 2, \cdots, n$, la prima riga moltiplicata per $m_{i,1}$, reiterando questo procedimento per le restanti colonne di i si ottiene:

$$\begin{aligned} a_{i,j}^{(2)} &= a_{i,j}^{(1)} - m_{i,1} a_{1,j}^{(1)}, \quad i, j = 2, \cdots, n \\ b_i^{(2)} &= b_i^{(1)} - m_{i,1} b_1^{(1)}, \quad i = 2, \cdots, n \end{aligned}$$

dove $b_i^{(1)}$ definisce i componenti di $\mathbf{b}^{(1)}$. Il risultato è un nuovo sistema nella forma

$$\begin{bmatrix} a_{1,1}^{(1)} & a_{1,2}^{(1)} & \cdots & a_{1,n}^{(1)} \\ 0 & a_{2,2}^{(2)} & \cdots & a_{2,n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{m,2}^{(2)} & \cdots & a_{m,n}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_m^{(2)} \end{bmatrix}$$

indicato con $A^{(2)}\mathbf{x} = \mathbf{b}^{(2)}$ il quale è equivalente al sistema iniziale. Allo stesso modo è possibile trasformare il sistema in modo che l'incognita x_2 sia eliminata dalle righe $3, \cdots, n$. In generale, è possibile concludere in un insieme finito di passi la triangolarizzazione fino ad arrivare ad un sistema nella forma

$$A^{(k)}\mathbf{x} = \mathbf{b}^{(k)}, \quad 1 \leq k \leq n, \tag{5.1}$$

dove per $k \leq 2$. La matrice $A^{(k)}$ assume al passo $k-esimo$ la seguente forma:

$$\begin{bmatrix} a_{1,1}^{(1)} & a_{1,2}^{(1)} & \cdots & \cdots & \cdots & a_{1,n}^{(1)} \\ 0 & a_{2,2}^{(2)} & & & & a_{2,n}^{(2)} \\ \vdots & & \ddots & & & \vdots \\ 0 & \cdots & 0 & a_{k,k}^{(k)} & \cdots & a_{k,n}^{(k)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{m,k}^{(k)} & \cdots & a_{m,n}^{(k)} \end{bmatrix}$$

Assumendo che tutti gli $a_{i,i}^{(i)} \neq 0 \ \forall i = 1, \cdots, k-1$, per $k = m$ si ottiene così il sistema triangolare alto $A^{(n)}\mathbf{x} = \mathbf{b}^{(n)}$

$$\begin{bmatrix} a_{1,1}^{(1)} & a_{1,2}^{(1)} & \cdots & \cdots & a_{1,n}^{(1)} \\ 0 & a_{2,2}^{(2)} & & & a_{2,n}^{(2)} \\ \vdots & & \ddots & & \vdots \\ 0 & & & \ddots & \vdots \\ 0 & & & & a_{m,n}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ \vdots \\ b_m^{(n)} \end{bmatrix}$$

U viene definita come la matrice triangolare alta $A^{(n)}$. Ottenuto un sistema in questa forma , per calcolare la soluzione del sistema, basta semplicemente sostituire, partendo dal basso, all'equazione $i-esima$ le soluzioni (già ottenute) delle incognite che la compongono. Gli elementi $a_{k,k}^{(k)}$ sono detti *pivot* e devono necessariamente essere non nulli per $k = 1, \cdots, n-1$. Un passo di esecuzione k, che trasforma il sistema attuale ($k-esimo$) in quello successivo ($(k+1)-esimo$), per $k = 1, \cdots, n-1$ è composto da due operazioni basilari. La prima, assumendo che $a_{i,i}^{(i)} \neq 0$, è costituita dal calcolo del fattore moltiplicativo

$$m_{i,k} = \frac{a_{i,k}^{(k)}}{a_{k,k}^{(k)}}, \quad i = k+1, \cdots, n$$

La seconda aggiorna di conseguenza i valori $a_{i,j}^{(k+1)}$ e $b_i^{(k+1)}$ nel modo seguente:

$$a_{i,j}^{(k+1)} = a_{i,j}^{(k)} - m_{i,k} a_{k,j}^{(k)}, \quad i,j = k+1, \cdots, n$$
$$b_i^{(k+1)} = b_i^{(k)} - m_{i,k} b_k^{(k)}, \quad i = k+1, \cdots, n$$

Per completare l'algoritmo di eliminazione Gaussiana sono necessarie $2\left(n-1\right)n\left(n+1\right)/3 + n\left(n-1\right)$ operazioni, ne sono invece richieste n^2 per l'operazione di backsolve del sistema $U\mathbf{x} = \mathbf{b}^{(n)}$. Ignorando i termini di ordine inferiore è possibile asserire che per eseguire il metodo di eliminazione Gaussiana sono necessarie circa $2n^3/3$ operazioni. In questo caso la risoluzione viene applicata a polinomi, le operazioni eseguite su questo tipo di dato hanno un costo che varia in funzione della lunghezza dello stesso; definita n come la cardinalità del sistema, nel caso pessimo la complessità diventa pari a $n^{\Theta(log(n))}$.

5.1.2 Matrici Sparse

Nel caso in cui la matrice sia di natura sparsa, ovvero il numero di coefficienti $a_{i,j}$ per riga sia molto minore rispetto alla cardinalità del sistema, è possibile impiegare tecniche specifiche per la risoluzione di questa tipologia di sistemi. Sfruttando la sparsità intrinseca di molte classi di sistemi (di grandi dimensioni), è possibile ottenere un incremento drastico delle performance. Gli algoritmi che operano su matrici sparse hanno, prima di tutto, l'obiettivo di evitare la memorizzazione degli zeri, gestendo l'archiviazione della matrice in memoria con opportuni metodi e cercando di evitare il più possibile operazioni che coinvolgeranno prima o poi termini nulli (somme, moltiplicazioni). Riconoscere una matrice sparsa, avvantaggiandosi poi dei metodi specifici permette a volte di risolvere problemi, che senza l'impiego di queste tecniche sarebbero impossibili da risolvere. Ogni matrice, con un numero abbastanza significante di elementi nulli, può essere classificata come sparsa. Non esiste comunque un metodo matematico rigoroso in grado di definire se una matrice

è considerabile sparsa; la definizione di conseguenza rimane piuttosto vaga. E' importante scegliere con attenzione se impiegare o meno metodi per le matrici sparse; poiché essi richiedono un overhead computazionale aggiuntivo. Nel caso in cui lo speed-up derivato dal grado di sparsità, compensa l'overhead introdotto, risulta conveniente impiegare tecniche specifiche; altrimenti rimane una scelta migliore applicare metodi per matrici dense. Ora verrà descritta una tecnica specifica per la risoluzione di *sistemi* di *grandi dimensioni*, caratterizzati da una forte sparsità . Un approccio comune in questi casi è dividere il metodo di risoluzione in 3 fasi distinte:

1. *Analyze*: determinare un ordinamento delle equazioni per preservare il più possibile la sparsità. Questo problema è stato dimostrato essere N-P completo, ma esiste un numero soddisfacente di euristiche in grado di avvicinare la soluzione ottima. La fase di analisi del metodo di risoluzione parte dalla definizione del sistema, al quale vengono poi effettuate delle eliminazione Gaussiana strutturata [12] ove possibile; questo metodo cerca di rendere implicite il maggior numero di variabili, queste verranno poi esplicitate nuovamente nella fase di Backsubstitute. Questo viene fatto per diminuire il più possibile la cardinalità del sistema *denso*, che verrà risolto nella fase seguente.

2. *Solve*: in questa fase viene risolto il sistema *denso*. Si cerca di ottenere un sistema il più denso possibile, poiché durante la risoluzione, anche se vengono impiegati algoritmi ottimizzati, questi tendono inevitabilmente a modificare la natura della matrice facendole perdere la caratteristica di sparsità. Il sistema viene risolto con metodi classici di analisi numerica come per esempio il Metodo di Eliminazione Gaussiana.

3. *Backward substitution*: risolto il sistema denso si procede alla fase di sostituzione, Backsubstitution, che esplicita la soluzione del sistema di equazioni.

Nell'algoritmo proposto si cerca di ridurre al massimo la dimensione del sistema denso: questo permette di avere una fase di solve molto veloce pa-

gando, a volte, una leggera perdita di prestazioni nella fase di Backsubstitution. Nella descrizione della fase di analisi si è parlato di ordinamento delle equazioni e di come determinarlo fosse un problema N-P; per risolvere questo problema vengono impiegate delle euristiche che non offrono la soluzione ottimale ma permettono di ottenere ottimi risultati.

5.1.3 Markovitz Pivoting

Le strategie di pivot sono molto importanti poiché preservano la sparsità, questo perché, sia durante l'esecuzione degli algoritmi di risoluzione diretti, sia nella fase di eliminazione Gaussiana strutturata, si tende ad avere un effetto detto *fill-in* ovvero la matrice tende a riempirsi di nuovi valori non nulli, perdendo così pian piano la sua sparsità. Diviene quindi un obiettivo importante trovare una strategia di pivot in grado di minimizzare il numero di riempimenti nella matrice. Il pivoting si divide essenzialmente in: *parziale* e *completo*; le differenze sono mostrate in figura 5.1. Un'altra differenza importante tra tecniche di pivot, è costituita sostanzialmente dal tipo di euristica impiegato; la sua scelta è determinante perché una buona euristica porta ad ottenere una soluzione valida, viceversa una inadatta porta a risultati che si discostano molto dall'ottimo teorico.

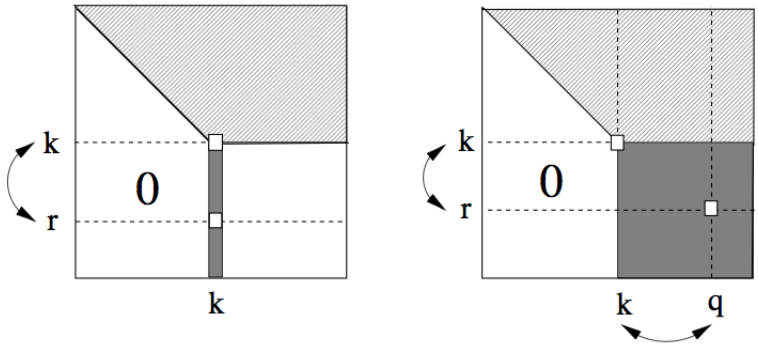

Figura 5.1: Esempio di pivoting parziale (a sinistra) e di pivoting completo (a destra).

La maggior parte delle euristiche in grado di ottenere buoni risultati in questa campo, sono fondate sul lavoro di Markovitz. Una strategia di pivot di Markovitz si basa sulla scelta di un elemento $a_{i,j}$ che minimizza una quantità, detta Markovitz Count, per ogni passo di eliminazione. Il conteggio di Markovitz è definito come il prodotto

$$(r_i - 1)\,(c_j - 1)$$

dove

- r_i è il numero di elementi nella riga i della matrice ridotta
- c_j è il numero di elementi nella colonna j della matrice ridotta.

Le equazioni contenenti i pivot vengono triangolarizzate e un blocco di variabili viene eliminato dal sistema. E' necessario fare delle considerazioni di stabilità per il pivoting di Markovitz, definendo una soglia che può essere applicata ai pivot candidati. In effetti questa soglia viene introdotta per dare un indice statistico per la definizione di un bound per il concetto di buon candidato per il pivot, il quale deve soddisfare la seguente formula

$$\mid a_{i,j}^{(k)} \mid \geq u \max_{l \geq k} \mid a_{i,j}^{(k)} \mid$$

dove u è un valore compreso nell'intervallo $0 < u \leq 1$.

5.1.4 Algoritmo risolutivo

L'algoritmo proposto, denominato d'ora in poi RationalSparse (RatSparse), sfrutta le soluzioni descritte in precedenza per cercare di ottenere la soluzione del sistema in modo efficiente; il linguaggio di programmazione scelto è Maple [7], specifico per la risoluzione di problemi matematici ad alte prestazioni. Il codice realizzato segue la filosofia classica dell'approccio per matrici sparse, ovvero è diviso fasi di Analisi, Solve e Backsubstitution; sfrutta inoltre il pivoting di Markovitz per preservare la sparsità del sistema. Questo algoritmo è basato sul modello proposto nell'articolo "Solving Sparse

Linear Systems in Maple"[11]. A differenza del codice al quale è ispirato questo metodo permette la risoluzione di sistemi lineari del primo ordine parametrici, ovvero i coefficienti associati alle incognite e ai termini noti, sono caratterizzati da polinomi contenenti letterali $(p_1, p_2, \cdots, p_n)$. La soluzione del sistema non sarà quindi prettamente numerica ma sarà un insieme di funzioni razionali

$$sol = \begin{bmatrix} f_1(p_1, p_2, \cdots, p_n) \\ f_2(p_1, p_2, \cdots, p_n) \\ \vdots \\ f_n(p_1, p_2, \cdots, p_n) \end{bmatrix}$$

Qui di seguito è riportata la parte del codice relativo della fase di sostituzione iniziale, relativa al pivoting di Markovitz ed alla Eliminazione Gaussiana Strutturata; la funzione si conclude poi effettuando le dovute sostituzioni (dovute all'eliminazione Gaussiana) ed aggiornando il sistema e la relativa lista delle variabili.

Algorithm 2 Procedura di Analisi (Eliminazione Gaussiana Strutturata)

```
MarkowitzPivot := proc()
  local c1, c2, C, R, W, M, S, i, j, s, n, d, TIMER, coeffM;
  if nrows=0 or ncols=0 or sn < 0 then return NULL; end if;
  s := 0;
  eqns := op(eqns) minus 0;
  d := evalf(density(cpoly,nrows,ncols));
  while type(cpoly, `+`) and mkz_dense_cont(d) do
  # compute number of other variables in each column and row
  R := [seq(numterms(i) - 1, i=eqns)];
  c1, c2 := selectremove(type, cpoly, 'name');
  C := table([seq(i=0, i=`if`(type(c1,`+`), c1,
      `if`(c1=0, [], [c1]))), seq(op(2,i)=op(1,i)-1, i=`if`(type(c2,`+`),
```

```
  c2, `if`(c2=0, [], [c2]))]);

  // select minimum degree pivot for each row

  W := [seq(min(seq(setattribute(Float(R[i]*C[j],0), [i,j]),
      j=GetVars(indets(eqns[i]),vars))), i=1..nops(eqns))];
  M, W := selectremove(type, W, poszero);
  n := max(mkz_block_size(nrows)-nops(M),0);

  // remove duplicates
  W := sort(W, `>`);
  M := map(attributes, [op(W[-n..-1]), op(M)]);
  M := [op(op(table([seq(i[2]=i[1], i=M)])))];
  S := [seq(rewrite(eqns[op(2,i)], op(1,i)), i=M)];

  eqns := subsop(seq(op(2,i)=NULL, i=M), eqns);
```

Ora viene proposta invece la procedure relativa alla fase di Backward Substitution.

Algorithm 3 Procedura di Backsubstitution

```
BackSubstitute := proc()
  local S, V, i, j, n;
  if sn < 0 then return  end if;
  n := 0;

  for i from sn to 1 by -1 do
    S := eval(sol[i]);
    n := n + nops(S);
```

```
    V[i] := S;
    userinfo(4, 'solve', nprintf("%d variables solved", n));
    for j in S do xassign(lhs(j),rhs(j)) end do:
  end do;
  userinfo(4, 'solve', nprintf("replacing variables"));
  for i in rsub do xassign(lhs(i), rhs(i)) end do;
  S := seq(op(eval(V[i])), i=1..sn);
  for i in rsub do xassign(lhs(i), lhs(i)) end do;
  for i to N do unassign(x[i]) end do:
  S
end proc:
```

5.2 Inversione di matrici

In casi in cui il numero dei parametri, all'interno di una matrice $A \quad n \times n$ quadrata, sia abbastanza piccolo e distribuito su un numero sufficientemente ristretto di righe o colonne, può risultare conveniente calcolare l'inversa di A attraverso la definizione algebrica, anche se la teoria dell'analisi numerica sconsiglia di eseguire il calcolo in questo modo poiché, nel caso pessimo, l'algoritmo ottenuto è notoriamente N-P completo. Prima di tutto viene data la definizione di inversa.

Definizione 5.2. Una matrice quadrata A $n \times n$ si dice invertibile se esiste una matrice B $n \times n$ tale che $A \cdot B = I_n = B \cdot A = I_n$.

Se B esiste è unica, di conseguenza se la matrice A possiede un'inversa, è anch'essa unica. Per ammettere inversa una matrice A deve inoltre soddisfare il seguente teorema:

Teorema 5.2.1. *Una matrice $A \quad n \times n$ è invertibile se e solo se il rango della matrice è uguale alla sua dimensione ovvero: $rank\,(A) = n$.*

Esistono vari sistemi per calcolare l'inversa di una matrice e quello che viene formalizzato ora (derivato direttamente dalla definizione algebrica) è detto *Metodo dei Cofattori*. Il metodo della matrice dei cofattori risulta particolarmente rapido quando non interessa calcolare tutti gli elementi della matrice inversa. Inoltre, la presenza di variabili letterali tra gli elementi non aumenta di molto la complessità del calcolo. Presa una matrice A $n \times n$ quadrata e invertibile

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{i,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$$

la sua inversa è definita come segue:

$$A^{-1} = \frac{1}{det\,(A)} \begin{bmatrix} cof\,(A, a_{1,1}) & \cdots & cof\,(A, a_{i,n}) \\ \vdots & \ddots & \vdots \\ cof\,(A, a_{n,1}) & \cdots & cof\,(A, a_{n,n}) \end{bmatrix}^T$$

dove $det\,(A)$ è il determinante della matrice A e il cofattore $cof\,(A, a_{i,j})$ è definito come

$$cof\,(A, a_{i,j}) = (-1)^{i+j} \cdot det\,(minor\,(A, i, j))$$

e $minor\,(A, i, j)$ rappresenta il *minore* della matrice A il quale restituisce una sottomatrice $A^{'}$, ottenuta partendo da A privandola della $i - esima$ riga e della $j - esima$ colonna.

L'approccio qui proposto, per il calcolo della matrice inversa è di natura ibrida, ovvero non si basa solamente sull'applicazione del metodo dei cofattori, ma affianca a quest'ultimo un metodo di risoluzione diretto.

La filosofia di base è di iniziare utilizzando il metodo dei cofattori per calcolare una sola riga della matrice inversa, sfruttando la definizione di determinante, seguendo i seguenti passi:

1. il cofattore $cof\,(A, a_{i,j})$ viene calcolato per ogni elemento della matrice $a_{i,j}$, ottenendo così il determinante della sottomatrice minorata; per la valutazione di $det\,(minor\,(A, i, j))$ si possono distinguere due casi:

   (a) Nel caso in cui la sottomatrice sia completamente numerica ovvero priva di parametri, viene calcolato il determinante attraverso il metodo di eliminazione Gaussiana, moltiplicando gli elementi sulla diagonale della matrice triangolare U.

   (b) In caso contrario, ovvero quando sono presenti parametri all'interno della sottomatice viene applicata ricorsivamente la definizione di cofattore, l'euristica impiegata espande per prime le righe/colonne contenenti il maggior numero di elementi parametrici; le sottomatrici generate vengono a loro volta prese in considerazione e valutate riapplicando l'algoritmo dal punto 1.

2. Il determinante di A viene calcolato semplicemente sfruttando i cofattori ottenuti in precedenza, ovvero sommando tutti i valori $a_{i,j}$, della riga A_i oggetto del calcolo, moltiplicati per i rispettivi cofattori. Formalmente fissato l'indice i della riga A_i da invertire il determinante si ottiene come $\sum_{j \in A_i} (a_{i,j} \cdot cof\,(A, a_{i,j}))$ dove i $cof\,(A, a_{i,j})$ sono già stati calcolati nel passo 1 dell'algoritmo.

L'obiettivo di questo metodo è di espandere prima possibile la parte parametrica della matrice mediante il metodo dei cofattori, permettendo così l'impiego di metodi diretti su sottomatrici puramente numeriche. Il calcolo di una riga della matrice inversa A richiede la risoluzione di t determinanti di sottomatrici di dimensione $t - 1$. Definendo τ come il valore medio di valori su una riga (assumendo $\tau << n$), ognuno dei determinanti può essere ottenuto attraverso l'ulteriore applicazione del metodo dei cofattori.

Espandendo le prime c righe contenenti parametri, è necessario calcolare al massimo τ^c determinanti per poi combinarli tra loro attraverso applicazioni lineari. Le sottomatrici di dimensione $t - c$ non contengono al loro interno alcun parametro, il loro determinante può essere quindi ottenuto con circa $(t-c)^3$ operazioni semplici impiegando metodi di analisi numerica (Eliminazione Gaussiana, Lu-Decomposition...). Combinando questi metodi si ottiene una complessità finale di:

$$\tau^c \cdot (t-c)^3 \sim \tau^3 \cdot t^3$$

Nel caso reale, in cui i parametri siano pochi e concentrati, l'impiego di questa tecnica anche se dal punto di vista teorico appare inefficiente, può portare buoni risultati.

5.3 Valutazione formule PCTL

Lo scopo principale di questo elaborato è fornire una soluzione efficiente per valutare requisiti di affidabilità. Verranno ora definiti metodi ottimizzati per la valutazione di alcune classi di formule esprimibili in linguaggio PCTL, fino ad arrivare a definire un metodo di risoluzione generale. Le proprietà verranno verificate su catene di Markov parametriche (PMC), con l'obiettivo di ottenere valutazioni efficienti a run-time. L'idea di base è di cercare di fornire oltre ad una soluzione generale, metodi specifici per la risoluzione di particolari sottoclassi di PCTL. Vengono a questo scopo distinti i seguenti 4 casi:

- Valutazione di una formula flat nella forma: $X\,(\Phi)$.
- Valutazione di una formula flat nella forma: $\Phi_1\ \boldsymbol{U}\ \Phi_2$.
- Valutazione di una formula flat nella forma: $\Phi_1\ \boldsymbol{U}^{\leq n}\ \Phi_2$.

- Valutazione di una formula nella forma: $\Phi_1 \; \boldsymbol{U} \; \Phi_2$ oppure $\Phi_1 \; \boldsymbol{U}^{\leq n} \; \Phi_2$ oppure $X\,(\Phi)$, dove Φ_1 e Φ_2 sono formule di stato, che contengono al loro interno un operatore probabilistico.

Tutte le tipologie di formule precedentemente elencate hanno in comune, il primo passo di valutazione a design-time: l'espansione dello spazio degli stati. Viene presa la definizione del modello in linguaggio PRISM e, partendo dallo stato iniziale, vengono espansi sistematicamente tutti i possibili scenari. Le soluzioni fornite in seguito partiranno sempre dalla rete PMC già generata.

5.3.1 Until

La prima tipologia da prendere in considerazione è costituita da formule del tipo $\mathcal{P}_{\bowtie p}\,(\Phi_1 \; \boldsymbol{U} \; \Phi_2)$; per valutare questa classe di formule è necessario di tutto individuare a design-time prima gli stati iniziali e finali, applicare tecniche di lumpin' se richiesto, ed infine ricondurre la proprietà ad una appartenente alla classe di reachability. Viene innanzitutto presa in considerazione la PMC. In base alla valutazione delle proprie formule uno stato è oggetto di classificazione, questo per individuare, lo stato iniziale (unico), poiché la sua definizione assegna un valore di verità a tutte le proposizioni atomiche, ed i due sottoinsiemi degli stati finali costruiti come segue:

- il primo S_{goal} al quale appartengono tutti gli stati s_i del sistema nei quali Φ_2 risulta verificata;
- il secondo $S_{\neg goal}$ nel quale sono contenuti gli stati che non soddisfano la formula di cammino, ovvero dove vale $\neg\Phi_1 \wedge \neg\Phi_2$.

A questo punto si ottiene una PMC P' nella quale sono stati eliminati tutti i cammini nei quali $\Phi_1 \; \boldsymbol{U} \; \Phi_2$ non è verificata. Non rimane altro che applicare un algoritmo di risoluzione allo scopo di ottenere l'espressione parametrica; quest'ultima rappresenta il valore probabilistico di soddisfacibilità della formula di cammino a partire dallo stato iniziale. Per il calcolo dell'espressione vengono presentati due metodi complementari, uno permette

di ottenere risultati con un maggior livello di dettaglio (informazioni meno aggregate), l'altro si adatta meglio a condizioni differenti. Il primo prevede il calcolo della formula, vista nel capitolo precedente, $(I-Q)^{-1} \cdot R$ dove Q contiene le probabilità associate alle transizioni tra due stati non assorbenti e R rappresenta la frequenza attesa di passaggio tra gli stati transienti e gli assorbenti assorbenti (finali). In realtà, visto che lo stato iniziale s_0 è unico, tutti i possibili cammini avranno origine in s_0; di conseguenza l'unica parte della matrice $(I-Q)^{-1}$ utile al calcolo dell'espressione è la riga relativa ad s_0. A questo punto non rimane altro che calcolare questa riga attraverso il metodo dei cofattori oppure sfruttando un'interessante proprietà matematica.

Definizione 5.3. Presa una matrice A quadrata $n \times n$, il sistema $A\mathbf{x} = \mathbf{b}_{\overline{j}}$ dove fissato $\overline{j} \in [0, n]$, $\mathbf{b}_{\overline{\mathbf{j}}}$ viene così definito $b_{\overline{j}} = 1$ e $\forall\,(j \leq n)\,, j \neq \overline{j} : b_j = 0$, permette di ottenere la colonna $\overline{j}$ della matrice A^{-1}.

L'espressione probabilistica associata al vincolo $\mathcal{P}_{\bowtie p}\,(\Phi_1\ \boldsymbol{U}\ \Phi_2)$ si ottiene attraverso due passi distinti: nel primo viene risolto il sistema $(I-Q)^T\,\mathbf{x} = \mathbf{b}_{\overline{\mathbf{j}}}$ dove $\overline{j}$ corrisponde alla riga dello stato iniziale s_{init} (se sono presenti più stati all'interno di S_{init} è necessario reiterare l'operazione); col secondo il vettore riga $\mathbf{x}$ viene moltiplicato per la matrice degli assorbenti R: $\mathbf{x} \cdot R$. E' importante sottolineare che nel caso in cui Q rappresenti una PMC, l'inversa della matrice $(I-Q)$ esiste sempre; ne consegue che il sistema $(I-Q)^T\,\mathbf{x} = \mathbf{b}_{\overline{\mathbf{j}}}$ ammette sempre soluzione. Applicando questo metodo si ottiene un'espressione che, è funzione sia dei parametri, sia di tutti gli stati assorbenti s_k; assegnando a questi ultimi il valore 1 nel caso in cui $s_k \in S_{goal}$ e 0 altrimenti ($s_k \in S_{\neg goal}$), si ricava l'espressione finale.

Il secondo, a differenza del primo sfrutta la risoluzione un sistema lineare utilizzando un concetto diverso basato sulle caratteristiche topologiche della rete. Questo metodo viene applicato direttamente sull'intera matrice di transizione P di dimensione $p \times p$:

$$B = \begin{bmatrix} I-Q & -R \\ 0 & I \end{bmatrix}$$

L'idea è di risolvere il sistema nella forma $B\mathbf{x} = \mathbf{b}$, dove il vettore $\mathbf{b}$ (vettore colonna), viene costruito nel modo seguente: $\forall j, (j \leq p)$ se $s_j \in S_{goal}$ allora $\mathbf{b}_j = 1$ altrimenti $\mathbf{b}_j = 0$. Questo sistema è dotato di importanti caratteristiche: è sempre determinato (ammette sempre soluzione) e la sua soluzione è sempre unica. Ora viene presentata una spiegazione formale del metodo appena proposto. Il sistema così definito è equivalente alla scrittura in sequenza di tutte le equazioni, una per ogni stato; le equazioni degli stati non assorbenti sono così definite

$$\begin{cases} x_0 = b_{1,0}x_1 + b_{2,0}x_2 + \cdots + b_{b,0} \\ \quad \vdots \\ x_q = b_{1,q}x_1 + b_{2,q}x_2 + \cdots + b_{b,q} \end{cases}$$

una per ogni stato transiente del sistema. Per quanto riguarda gli stati assorbenti, semplicemente viene assegnato un valore 0 o 1 rispettivamente nei casi in cui lo stato assorbente sia in S_{goal} o in $S_{\neg goal}$. Questa formalizzazione è del tutto equivalente al sistema $B\mathbf{x} = \mathbf{b}$ definito in precedenza. A questo punto, propagando i valori associati alle variabili x_i degli stati assorbenti alle equazioni relative ai transienti, si ottengono espressioni in funzione dei soli stati non assorbenti. Risolvere ognuna di queste equazioni, rispetto ad uno specifico stato transiente, equivale a calcolare il valore probabilistico atteso di attraversare un percorso composto da $1, 2, \cdots, \infty$ stati che verificano Φ_1, arrivando in uno stato assorbente nel quale è verificata Φ_2. Il risultato finale è del tutto equivalente al primo metodo; esistono tuttavia alcune importanti considerazioni da fare sulla differenza tra i due. Il primo metodo ritarda la risoluzione rispetto agli stati assorbenti; viene prima calcolata $(I - Q)^{-1}$ per poi moltiplicarla con R. Questo permette di mantenere separate le probabilità attese rispetto ad ognuno degli stati assorbenti (informazione molto dettagliata), ottenendo però un minor numero di informazioni a parità di complessità computazionale. Applicando il secondo metodo invece, si ottiene un'espressione meno dettagliata: infatti l'espressione di verifica della proprietà esprime il concetto di raggiungere attraverso un cammino qualsia-

si, uno degli stati assorbenti. Il tipo di informazione ottenuto è di carattere più generale ma questo secondo metodo permette, impiegando le medesime risorse, di ottenere, rispetto al primo, una quantità maggiore di informazioni (ovvero una valutazione per ogni stato) sulla soddisfacibilità della proprietà. I due metodi sono essenzialmente uno il duale dell'altro: il primo parte dallo stato iniziale per arrivare ad ottenere la formula in funzione di ognuno degli stati assorbenti (propagazione in avanti), il secondo propaga i valori, assegnati ad ognuno degli stati assorbenti, verso tutti gli stati appartenenti alla PMC (propagazione all'indietro). Sono stati forniti due metodi differenti per la verifica su PMC di proprietà PCTL $\Phi_1 \ \boldsymbol{U} \ \Phi_2$, con Φ_1, Φ_2 formule prive dell'operatore probabilistico $\mathcal{P}$. Questa tipologia di vincoli è sicuramente la più importante tra le formulabili in PCTL poiché la proprietà di reachability viene formulata attraverso l'operatore $\boldsymbol{U}$. Ora vengono proposti metodi di risoluzione per formule semplici contenenti gli operatori Next o Bounded Until, fino a concludere mostrando un approccio, di risoluzione a design-time e di calcolo a run-time, per formule nelle quali è annidato (una o più volte) l'operatore probabilistico $\mathcal{P}$.

5.3.2 Bounded Until

Ora verrà proposta una strategia per la risoluzione di formule appartenenti alla classe Bounded Until $\mathcal{P}_{\bowtie p}\left(\Phi_1 \ \boldsymbol{U}^{\leq n} \ \Phi_2\right)$ dove Φ_1, Φ_2 sono formule semplici, ovvero non contengono operatori probabilistici annidati. Le formule nella classe Flat Bounded Until hanno una semantica del tutto simile alle proprietà Until, ma formalizzano il concetto di valore atteso della situazione in cui si raggiungere uno stato assorbente di goal (è vera Φ_2) partendo da uno stato iniziale, dove il cammino compreso tra i due ha una lunghezza inferiore o pari a n. Per ottenere la valutazione relativa all'espressione della probabilità per questa tipologia di formule non esiste nessuna alternativa all'esplorazione di tutti i possibili cammini di dimensione $0, 1, \cdots n$. L'obiettivo, con questa classe di formule rimane quello di cercare il massimo delle prestazioni per ottenere l'espressione associata. Anche in questo caso risulta

molto utile l'impiego del calcolo matriciale, poiché la probabilità che presi due stati s_i, s_j entrambi transienti, s_j sia raggiungibile das_i in zero passi vale 1 se $i = j$, 0 altrimenti; in un passo vale $q_{i,j}$ con $q_{i,j} \in Q$, in due passi è esattamente $q^2_{i,j}$ dove $q^2_{i,j} \in Q^2$, così via fino ad arrivare ad n. Per calcolare il valore atteso di una proprietà in questa forma (essendo sempre di tipo Until) sarà necessario modificare le matrici di transizione Q ed R in modo tale da distinguere se uno stato diventa obiettivo (S_{goal}) oppure semplicemente assorbente ($S_{\neg goal}$). Vengono ora definite $Q^{'}$ e $R^{'}$, che verranno utilizzate per il calcolo dell'espressione finale. Ottenute direttamente da Q ed R, $Q^{'}$ ed $R^{'}$ vengono adattate in caso sia verificata una delle due seguenti condizioni:

- se uno stato s_i è contenuto in S_{goal} tutti le transizioni per cui vale $\forall j : \ p_{j,i} \neq 0$ vengono inserite rispettando l'ordine di riga nella matrice $Q^{'}$. s_i diventa uno stato assorbente
- se uno stato s_i è contenuto in $S_{\neg goal}$ allora non comparirà ne in $Q^{'}$ ne in $R^{'}$ ed inoltre tutte le transizioni tali che $\forall i : \ p_{j,i} \neq 0$ non verranno inseriti in nessuna delle due matrici
- tutti gli elementi rimasti tali che $\forall i, j : \ p_{i,j} \neq 0$ e $j \notin S_{goal} \cup S_{\neg goal}$ compariranno in $Q^{'}$
- tutti gli elementi rimasti tali che $\forall i, j : \ p_{i,j} \neq 0$ e $j \in S_{goal} \cup S_{\neg goal}$ compariranno in $R^{'}$

L'espressione associata al calcolo del valore atteso di soddisfacibilità per la classe di formule Bounded Until è quindi

$$\sum_{i=0}^{n} \left(Q^{'}\right)^{i} \times R^{'}$$

Il calcolo può essere quindi suddiviso in due fasi: nella prima viene calcolata $\left(Q^{'}\right)^{i+1}$ come $\left(Q^{'}\right)^{i} \times Q^{'}$ e sono sommate tra loro le varie potenze $S = S + \left(Q^{'}\right)^{i}$, ovvero vengono addizionata $\left(Q^{'}\right)^{i}$ a tutte le precedenti potenze di $\left(Q^{'}\right)$, nella seconda viene eseguito il prodotto tra S ed R, $S \times R$. In

realtà non è necessario, per ottenere l'espressione finale, eseguire ogni volta il prodotto dell'intera matrice $\left(Q'\right)^i$: è sufficiente prendere le righe di $\left(Q'\right)$ che corrispondono agli elementi inclusi nell'insieme S_{init} degli stati iniziali (solitamente costituito da un solo elemento s_{init}). Nel caso in cui sia presente un unico stato iniziale, la computazione totale viene ridotta di un numero di volte pari alla cardinalità dell'insieme degli stati. Questo è il metodo proposto per formule appartenenti alla tipologia Bounded Until. Valutando la complessità, la prima cosa da considerare è la crescita del tempo di esecuzione, che è direttamente proporzionale al bound n. Ogni prodotto tra il vettore corrispondente alla riga $q'_{s_{init}}$ e la matrice $Q_{'}$ richiede in caso di polinomi, all'interno del vettore e della matrice, circa $q^{\Theta(log(q))}$ operazioni dove, q è la cardinalità della matrice Q'; la somma richiede un numero di operazioni minore di un ordine di grandezza rispetto al prodotto quindi è possibile trascurarla. La complessità totale che otteniamo, considerando che il prodotto $S \times R'$ è anch'esso trascurabile, è pari a $n \cdot q^{\Theta(log(n))}$ dove n corrisponde al valore di Bound della proprietà.

5.3.3 Next

L'operatore Next è il più semplice tra quelli contenuti in PCTL; prevede di valutare la formula al suo interno negli stati successivi rispetto a quello in cui viene formulata. La classe delle formule Next $\mathcal{P}_{\bowtie p}\left(X\left(\Phi\right)\right)$ ha un metodo di risoluzione molto semplice poiché il cammino, in grado di soddisfare o meno questa categoria di proprietà, è noto a priori ed ha una lunghezza finita pari a 1.

La tecnica di risoluzione prevede di valutare tutti i successori degli stati contenuti in S_{init} (nel caso delle PMC è unico) e suddividere i suoi successori in due sottoinsiemi:

- preso i l'indice corrispondente allo stato iniziale s_{init}, vengono inseriti in S_{goal} tutti gli stati s_j nei quali sono verificate: $\forall j:\ p_{i,j} \neq 0$ e $s_j \models \Phi$.

- viceversa preso i l'indice corrispondente allo stato iniziale s_init, vengono inseriti in $S_{\neg goal}$ tutti gli stati s_j nei quali non sono verificate: $\forall j : \ p_{i,j} \neq 0$ o $s_j \models \Phi$.

A questo punto l'espressione associata alla soddisfacibilità della classe delle formule $\mathcal{P}_{\bowtie p}(X(\Phi))$ si ottiene sommando tutte le $p_{i,j}$ dove i corrisponde all'indice nella matrice P di s_{init} e j a tutti gli indici associati agli stati appartenenti a S_{goal}. Questo metodo è intrinsecamente costruito basandosi sulla topologia della rete. La complessità è in funzione del numero dei successori dello stato iniziale s_{init}, definendo k come il numero dei successori di s_{init} in cui è verificata Φ, l'espressione finale è ottenuta attraverso k somme tra polinomi. La complessità totale è quindi $k^{\Theta(log(k))}$.

5.3.4 Annidamento dell'operatore $\mathcal{P}$

Quando siamo in presenza di una formula ben formata in PCT,L che non è riconducile ai casi precedentemente descritti, è necessario applicare un metodo generale di risoluzione. Per ottenere un metodo generale bisognerà pagare un prezzo in termini di velocità di elaborazione. Una formula nella forma $\mathcal{P}_{\bowtie p}(\Phi_1 \ \boldsymbol{U} \ \Phi_2)$, $\mathcal{P}_{\bowtie p}\left(\Phi_1 \ \boldsymbol{U}^{\leq n} \ \Phi_2\right)$ o $\mathcal{P}_{\bowtie p}(X(\Phi))$, dove Φ, Φ_1 e Φ_2 sono a loro volta formule contenenti l'operatore probabilistico $\mathcal{P}_{\bowtie p}(\varphi)$, non è riconducibile ad uno dei casi descritti in precedenza. Il problema principale da affrontare, di fronte ad una formula appartenente a questa classe, è legato all'impossibilità di stabilire a design-time l'insieme di verità delle sottoformule composte da operatori probabilistici annidati; in altre parole è impossibile, durante la fase di precomputazione, conoscere i valori di verità della formula poiché questi sono in funzione anche dei valori associati ai parametri, i quali saranno noti solamente nella fase di valutazione a run-time. L'idea di base è cercare di eseguire la maggior parte possibile di elaborazione durante la precomputazione (design-time), per lasciare alla fase di run-time la parte di valutazione che deve essere necessariamente eseguita a parametri noti. Prima di tutto è necessario dividere la formula in sottoformule per poi prepararle,

attraverso la precomputazione a design-time, per la verifica a run-time. Di seguito verranno forniti metodi, per risolvere tutte le possibili tipologie di sottoespressioni, adattati per ottenere una valutazione a run-time veloce ed efficiente. Until è la prima classe di formule oggetto di valutazione: l'approccio riprende il secondo metodo mostrato per la risoluzione di Formule Until di tipo Flat e consiste nella risoluzione di un sistema rappresentante la rete PMC, al quale vengono introdotti $2 * b$ parametri $\alpha_1, \cdots, \alpha_b$ e $\beta_1, \cdots \beta_b$, dove b è il numero degli stati del sistema. Riprendendo la definizione di B

$$B = \begin{bmatrix} I - Q & -R \\ 0 & I \end{bmatrix}$$

Si ottiene un sistema parametrico di equazioni nella forma:

$$\begin{cases} x_0 = \alpha_1 \left(b_{1,0}x_1 + b_{2,0}x_2 + \cdots + b_{b,0}\right) + \beta_1 \\ \quad \vdots \\ x_q = \alpha_2 \left(b_{1,q}x_1 + b_{2,q}x_2 + \cdots + b_{b,q}\right) + \beta_2 \\ \quad \vdots \end{cases}$$

Risolvendo il sistema appena definito, si ottiene un insieme di espressioni in funzione dei parametri propri del sistema e di quelli appena introdotti:

$$\begin{cases} x_0 = f\left(\alpha_1, \cdots, \alpha_b, \beta_1, \cdots, \beta_b\right) \\ \quad \vdots \\ x_q = f\left(\alpha_1, \cdots, \alpha_b, \beta_1, \cdots \beta_b\right) \\ \quad \vdots \end{cases}$$

Per le sottoformule Until si conclude qui la fase di precomputazione. A run-time oltre a sostituire i valori dei parametri del sistema verranno assegnati i valori ai termini $\alpha_1, \cdots, \alpha_b$ e $\beta_1, \cdots \beta_b$ nel seguente modo:

- nel caso in cui la valutazione a run-time di Φ_2 risulti vera nello stato $i - esimo$, vengono assegnati i valori $\beta_i = 1$ e $\alpha_i = 0$.

- nel caso in cui la valutazione a run-time di Φ_1 risulti vera ma Φ_2 risulti falsa nello stato $i - esimo$, vengono assegnati i valori $\beta_i = 0$ e $\alpha_i = 1$.

- nel caso in cui le valutazione a run-time di Φ_1 e Φ_2 risultino entrambe false nello stato $i - esimo$, vengono assegnati i valori $\beta_i = 0$ e $\alpha_i = 0$.

Dopo aver effettuato le sostituzioni, si ottiene il valore atteso di soddisfacibilità della sottoformula Until valutata. Anche se la complessità teorica dell'algoritmo rimane la stessa, in campo pratico l'introduzione dei parametri α e β rallenta notevolmente la risoluzione del sistema.

Nel caso in cui una sottoformula appartenga alla classe Bounded Until, è indispensabile eseguire il calcolo $\sum_{i=0}^{n} \left(Q'\right)^i \times R'$, dove Q' ed R', sono le solite sottomatrici di P modificate come descritto in precedenza. La valutazione a runtime è necessaria poiché Q' e R' vengono costruite in base alla valutazione delle sottoformule Φ_1 e Φ_2, le quali saranno note solo durante la fase di esecuzione. La complessità, trattandosi di una matrice interamente numerica, è $n \cdot p^3$ dove p è la cardinalità del matrice di transizione e n il valore del bound; purtroppo è un costo da attribuire alla valutazione a run-time.

L'ultimo caso da prendere in considerazione è composto dalle formule della tipologia Next la quale prevede la semplice applicazione del metodo per le formule Flat in ognuno degli stati che compongono il sistema. La complessità diventa quindi $p \cdot \overline{k}$, dove p è la cardinalità del sistema e $\overline{k}$ il fattore medio di ramificazione della rete; non lavorando su polinomi (si è a run-time) il costo computazionale di ogni somma è considerato unitario.

Questo è il metodo proposto per la valutazione di formule complesse. Di seguito è riportata una breve descrizione delle tecniche di bisimulation, che possono essere applicate alle reti sulle quali viene fatta inferenza basata sulle tipologie flat. E' invece impossibile utilizzarle con il metodo appena proposto, poiché a run time è necessario avere l'intera struttura della rete per valutare correttamente le formule.

5.4 Bisimulation

La bisimulation è una relazione di equivalenza tra gli stati che punta ad identificare sistemi di transizione (come le PMC) con la medesima braching structure, per maggiori informazioni riguardo a queste tecniche fare riferimento a [4]. Ora verranno estese le definizioni standard di *strong* e *weak bisimulation*, per essere valide anche nel caso di Catene di Markov Parametriche (PMC).

Definizione 5.4. Prese $\mathcal{D} = (S, s_0, \mathbf{P}, V)$ una catena di Markov Parametrica (PMC) ed R una relazione di equivalenza su S, R è una *strong bisimulation* su $\mathcal{D}$ rispetto a $\mathbf{B}$ (insieme degli stati obiettivo) *se, per tutte* le coppie $s_1 R s_2$, vale $s_1 \in \mathbf{B} \quad sse \quad s_2 \in \mathbf{B}$ e per tutti i $C \in S/R$ vale $\mathbf{P}(s_1, C) = \mathbf{P}(s_2, C)$

Gli stati s_1 e s_2 sono definiti strong bisimilar, $s_1 \sim_{\mathcal{D}} s_2$, se solo se esiste una strong bisimulation R su D. Va notato, che nel caso di PMC vengono svolte operazioni su polinomi invece che su numeri. Nel caso delle PMC, la condizione strong bisimulation è ottenuta aggiungendo il requisito secondo il quale: $r(s_1) = r(s_2)$ e $r(s_1, C) = r(s_2, C)$ per tutti i $C \in S/R$ se $s_1 R s_2$. Ora viene data una definizione per la weak bisimulation:

Definizione 5.5. Prese $\mathcal{D} = (S, s_0, \mathbf{P}, V)$ una catena di Markov Parametrica (PMC) ed R una relazione di equivalenza su S, R è una *weak bisimulation* su $\mathcal{D}$ rispetto a $\mathbf{B}$ (insieme degli stati obiettivo) *se, per tutte* le coppie $s_1 R s_2$, vale $s_1 \in \mathbf{B}$ se $s_2 \in \mathbf{B}$ e

1. Se $\mathbf{P}(s_i, [s_i]_R) \neq 1$ *for* $i = 1, 2$ *allora per tutti i* $C \in S/R$, $C \neq [s_1]_R = [s_2]_R$: $\frac{\mathbf{P}(s_1, C)}{1 - \mathbf{P}(s_1, [s_1]_R)} = \frac{\mathbf{P}(s_2, C)}{1 - \mathbf{P}(s_2, [s_2]_R)}$
2. s_1 *può raggiungere uno stato al di fuori dell'insieme* $[s_1]_R$ *se e solo se* s_2 *può raggiungere uno stato al di fuori di* $[s_2]_R$.

Diciamo che due stati s_1 e s_2 sono weak bisimilar, $s_1 \approx s_2$, se esiste una weak bisimulation R su D. La weak Bisimulation è più grossolana della strong bisimulation. L'equivalenza di bisimulation (strong o weak) permette

di ottenere il *quoziente*, che è il più piccolo modello equivalente (nel senso della bisimulazione) a quello originario. Tutte le proprietà di tipologia Flat (Until, Bounded, Next) sono preservate dalla bisimulation, quindi è possibile lavorare col modello *quoziente* al posto del modello originale; la differenza sostanziale tra i due sta nella dimensione dello spazio degli stati che viene ridotto, anche di molto, grazie alla bisimulation.

Capitolo 6

Validazione

Lo scopo principale di questo progetto è trovare un modo efficiente per valutare vincoli di affidabilità, in ambienti instabili ed in continua evoluzione. E' stata presentata una soluzione computazionalmente costosa nella fase di design-time, con l'obiettivo di ottenere una verifica a run-time veloce ed efficiente (quando i parametri diverranno noti). Questa soluzione si adatta molto bene a situazioni in cui il costo computazionale a design-time non risulta importante, ma la valutazione delle formule a run-time è soggetta a stringenti vincoli temporali. Verranno ora proposti alcuni modelli per mostrare il corretto funzionamento dell'algoritmo e, per ognuno di questi, saranno verificate: tutte le possibili tipologie di formule Flat e alcune combinazioni per quando riguarda le formule annidate. Successivamente verranno proposti test su modelli più grandi, concludendo il capitolo presentando i dati relativi all'analisi delle performance.

6.1 Case studies

Per valutare il corretto funzionamento del codice verranno confrontate prima le espressioni ottenute da questo applicativo con quelle restituite da Param, infine verranno sostituiti i parametri con dei valori permettendo così il confronto anche con PRISM. Iniziamo la sezione dei case studies proponendo,

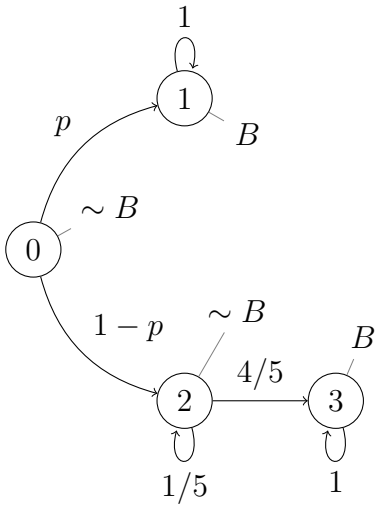

Figura 6.1: Modello giocattolo basato su PMC.

un esempio giocattolo, che permette di verificare direttamente se le proprietà siano vere o meno. L'esempio è composto da 4 stati, due terminali s_1, s_3 e due transienti s_0, s_2. Le transizioni sono in totale 6, di cui due parametriche (quelle uscenti da s_0). Il modello appena descritto è rappresentato in figura 6.1.

Verificheremo ora in ordine proprietà di tipo: Next, Until e Bounded Until semplici. La prima classe di formule flat presa in considerazione è Next. La proprietà $\mathcal{P} =? [X\,(b = 1)]$, nello stato iniziale s_0, ha associato un valore di probabilità esattamente pari a p, questo perchè $b = 1$ è verificata in s_1. Per la classe Until verifichiamo la proprietà $\mathcal{P} =? [true\ U\ \ (b = 1)]$; il valore di questa espressione, non dipende dal valore del parametro p ed assume sempre valore pari a 1. Per l'ultima classe di proprietà flat, costituita dal Bounded Until, è stata valutata la proprietà $\mathcal{P} =? [true\ U\ \ (b = 1)]$; il valore restituito sia da Param che da questo MC è pari a $\frac{1}{5}p + \frac{4}{5}$, ottenuto dalla somma dei cammini $(0, 1) = p$ e $(0, 2, 3) = (1 - p)\,\frac{4}{5}$. In tutti e tre i casi il risultato dei tre Model Checker è stato equivalente; per il confronto delle espressioni con PRISM è stato scelto come valore del parametro p, $\frac{1}{2}$.

I test sul modello base vengono conclusi con una serie di proprietà, com-

poste da operatori $\mathcal{P}$ annidati. Questa serie è costituita da 6 vincoli, uno per ogni possibile combinazione di operatori $(X, \mathbf{U}, \mathbf{U}^{\leq n})$ ottenibile con un livello di annidamento pari a 2. Prima di tutto viene effettuata la valutazione a design-time per l'intero spazio degli stati. Si ottiene la seguente soluzione:
$x[0] = \frac{2*a0*p*b1-5*a0*p*b1-5*b0+5*a0*p*b2-4*a0*a2*b3+a2*b0+4*a0*p*a2*b3-5*a0*b2}{-5+a2}$;
$x[1] = b1$;
$x[2] = \frac{-(4*a2*b3+5*b2)}{-5+a2}$;
$x[3] = b3$;

Di seguito è riportato un elenco contenente le proprietà valutate, ognuna affiancata da una breve spiegazione riguardante il valore atteso associato alla sua soddisfacilità. Il valore scelto per p rimane ancora $\frac{1}{2}$.

1. $\mathcal{P} =? \left[X \left(\mathcal{P} > 0.5 \left[true\ \boldsymbol{U}^{\leq 2}\ (b = 1)\right]\right) \& (b = 0)\right]$ intuitivamente la formula interna è vera in 2, ma non in 1; a questo punto la probabilità attesa di soddisfare tutta la formula è pari a $\frac{1}{2}$

2. $\mathcal{P} =? [X (\mathcal{P} > 0.5 [true\ \boldsymbol{U}\ (b = 1)]) \& (b = 0)]$, per questa formula vale il ragionamento precedente

3. $\mathcal{P} =?[(b = 0)\ \boldsymbol{U}\ \mathcal{P} \geq 0.5[X(b = 1)]]$ la sottoformula interna è verificata sia in 0 che in 2 e, dato che $b = 0$ è valida sia in 0 che in 2, la probabilità attesa associata alla formula è pari a 1

4. $\mathcal{P} =?[(b = 0)\ \boldsymbol{U}^{\leq 2}\ \mathcal{P} \geq 0.5[X(b = 1)]]$ per questa formula vale il ragionamento precedente

5. $\mathcal{P} =? \left[(b = 0)\ \boldsymbol{U}\ \mathcal{P} \geq 0.5 \left[true\ \boldsymbol{U}^{\leq 1}\ (b = 1)\right]\right]$ la sottoformula interna è verificata in 1, 2 e 3; dato che in 0 vale $b = 0$, ne consegue che la probabilità attesa è pari a 1

6. $\mathcal{P} =? \left[(b = 0)\ \boldsymbol{U}^{\leq 1}\ \mathcal{P} \leq 0.5 [true\ \boldsymbol{U}\ (b = 1)]\right]$ la formula interna è verificata in tutti gli stati del sistema, di conseguenza la proprietà completa avrà valore atteso pari a 1

Il MC qui proposto ha restituito, in tutti questi casi, il valore probabilistico atteso corretto. Per effettuare un confronto sia sui tempi che sui risultati, verrà utilizzato un modello più complesso, che rappresenta il protocollo di Crowds (descritto in precedenza), composto da 1198 stati. Il modello è scaricabile da [3]. In questi test le espressioni ottenute verranno verificate soltanto con quelle fornite da Param.

La prima proprietà è relativa alla reachability di uno stato di falso positivo:

```
Pmax=?[true U (newInstance & runCount=0 & observe0 <=1 &
     (observe1 > 1 | observe2 > 1 |  observe3 > 1 |
     observe4 > 1 | observe5 > 1 |  observe6 > 1 |
     observe7 > 1 |  observe8 > 1 |  observe9 > 1 |
     observe10 > 1 |  observe11 > 1 |  observe12 > 1|
     observe13 > 1 |  observe14 > 1 | observe15 > 1 |
     observe16 > 1 |  observe17 > 1 | observe18 > 1 |
          observe19 > 1))]
```

dove $observe-i$ è il contatore delle prove andate a buon fine riferito all'iesimo stadio del protocollo di Crowd, *runCount* è il contatore delle istanze del protocollo e *newInstance* indica se l'istanza del protocollo è nuova. La seconda è ottenuta direttamente dalla precedente, ma l'operatore di cammino Until è sostituito dalla sua versione Bounded; si ottiene così la formula:

```
Pmax=?[true U<=34 (newInstance & runCount=0 & observe0 <=1 &
     (observe1 > 1 | observe2 > 1 |  observe3 > 1 |
     observe4 > 1 | observe5 > 1 |  observe6 > 1 |
     observe7 > 1 |  observe8 > 1 |  observe9 > 1 |
     observe10 > 1 |  observe11 > 1 |  observe12 > 1|
     observe13 > 1 |  observe14 > 1 | observe15 > 1 |
     observe16 > 1 |  observe17 > 1 | observe18 > 1 |
          observe19 > 1))]
```

I Model Checker hanno restituito, per entrambe le formule, espressioni equivalenti con o senza l'impiego di bisismulation. Utilizzando quest'ultima i tempi di precomputazione impiegati dai due tools, per la verifica delle due proprietà, sono pressoché identici. Senza l'impiego di questa tecnica, invece, Param risulta più veloce nella verifica della seconda formula (Bounded Until), mentre l'approccio in esame è decisamente più veloce nella valutazione della prima (Unbounded Until). Nella prossima sezione è riportata un'analisi delle performance basata su test generati casualmente. Questa serie di test ha, come primo obiettivo, quello di valutare il comportamento dell'algoritmo in casi reali, ovvero in presenza di centinaia di parametri.

6.2 Analisi dei tempi di esecuzione

In questa sezione ci si concentrerà sull'analisi di proprietà di reachability ($\mathcal{P}_{\bowtie p}\,(true\ \boldsymbol{U}\ \Phi)$), poiché questa tipologia di formule è tra quelle di maggior interesse nelle applicazioni reali. La valutazione dei tempi di esecuzione è stata eseguita soltanto per la parte di risoluzione del MC, ovvero per la risoluzione del sistema implementata in Maple. Tutti i test cases sono stati generati in modo casuale; ogni test suite è identificata dal random seed utilizzato per inizializzare il generatore casuale, questo per poter rendere replicabili tutti i test cases.

E' stata studiata la crescita della complessità rispetto a due dimensioni del problema: la prima rappresentata dal numero degli stati e la seconda dal numero di parametri all'interno del sistema. La prima serve per valutare come si comporta l'algoritmo al crescere della dimensione del modello, la seconda per valutare l'impatto della computazione simbolica. Tutti i test cases sono composti da due stati assorbenti: uno corrispondente ad una situazione di *successo* (S) e l'altro ad una di *failure irreparabile* (F). Ognuno degli stati transienti ha associate 5 transizione uscenti. La proprietà oggetto di valutazione è $\mathcal{P}_{\bowtie p}\,(true\ \boldsymbol{U}\ \ (state = S))$ per tutti i modelli.

L'ambiente di esecuzione è composto da un calcolatore Dual Intel(R)

Xeon(R) CPU E5530 @ 2.40Ghz con 8Gb di RAM, equipaggiato con Linux Ubuntu Server 11.04 64bit. Tutti i test eseguiti non hanno mai richiesto paginazione, ovvero non hanno occupato, durante la loro esecuzione, tutta la memoria RAM disponibile. I grafici presentati in seguito, rappresentano il tempo di esecuzione medio con una sottile linea (nera) e mostrano il tempo massimo di esecuzione, all'interno di una test suite, attraverso una linea tratteggiata (in azzuro). Di seguito sono presentati confronti tra il metodo Rational Sparse, proposto in questo elaborato elaborato, e Linear, il miglior risolutore per sistemi lineari disponibile tra le librerie Maple. Ognuna delle test suite è composta da modelli che differiscono soltanto per uno dei seguenti fattori: cardinalità del modello, quantità dei parametri o numero delle transizioni uscenti da uno stato.

Le figure 6.1 e 6.2 mostrano, per i metodi Rational Sparse e Linear (Risolutore a basso livello di Maple), i tempi di esecuzione necessari alla risoluzione del sistema in funzione del numero degli stati. In ogni modello sono presenti 15 transizioni parametriche. Il numero totale di test cases è 4132. Anche per il sistema più grande, il risolutore non ha impiegato più di 50s per la risoluzione.

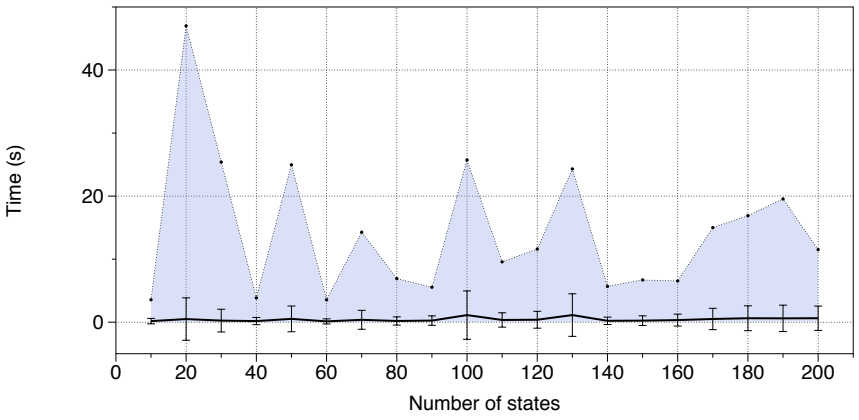

Grafico 6.1: Precomputazione a Design-Time in funzione del Numero degli Stati (metodo RatSparse)

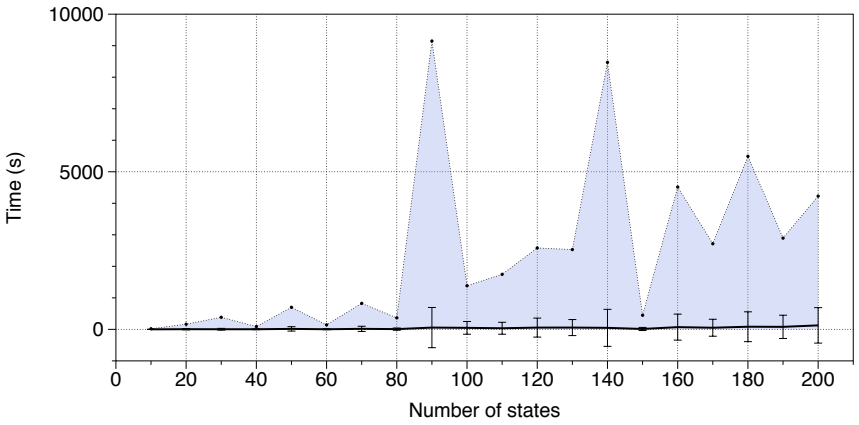

Grafico 6.2: Precomputazione a Design-Time in funzione del Numero degli Stati (metodo Linear)

Le figure 6.3 e 6.4 mostrano come la precomputazione a design-time cresca in funzione del numero dei parametri contenuti nel modello. In questi test il numero degli stati del modello è fissato a 100. La crescita dei parametri all'interno del sistema peggiora le performance a design-time; comunque l'algoritmo proposto è stato in grado di completare le valutazioni, nel caso peggiore, in un tempo inferiore a 5 minuti.

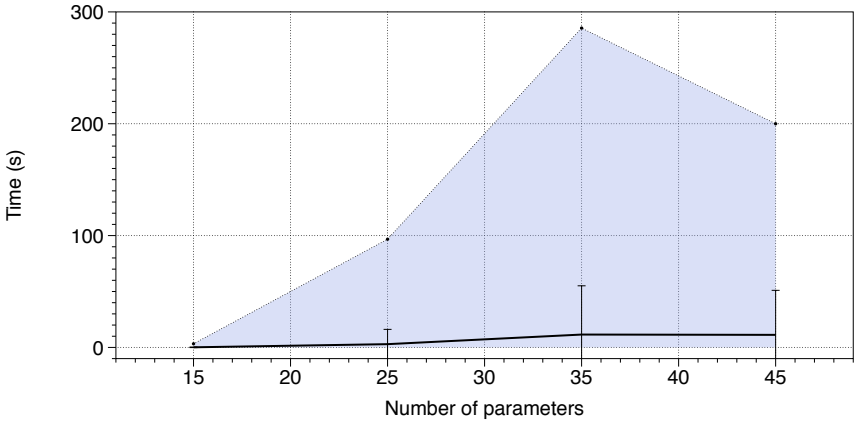

Grafico 6.3: Precomputazione a Design-Time in funzione del Numero degli Stati (metodo RatSparse)

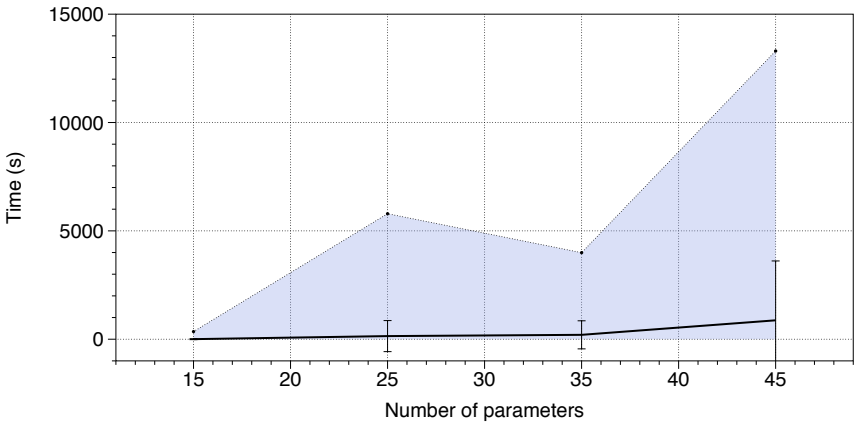

Grafico 6.4: Precomputazione a Design-Time in funzione del Numero degli Stati (metodo Linear)

L'ultima serie di test, composta da 140 modelli, confronta i tempi di esecuzione delle due soluzioni, in funzione del numero di transizioni uscenti da uno stato. Di seguito nelle figure 6.5 e 6.6 sono mostrati i tempi di esecuzione al variare del numero di transizioni uscenti dagli stati del modello. Rational Sparse in questi test è stato in media 10 volte più veloce di Linear.

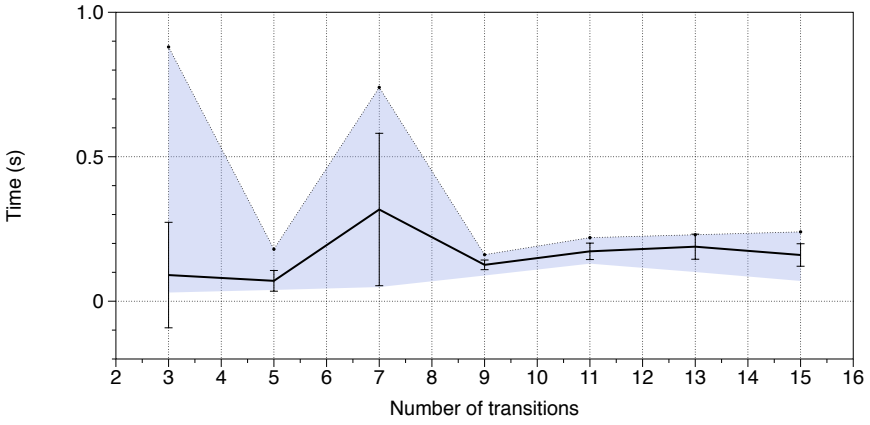

Grafico 6.5: Precomputazione a Design-Time in funzione del Numero di Transizioni (metodo RatSparse)

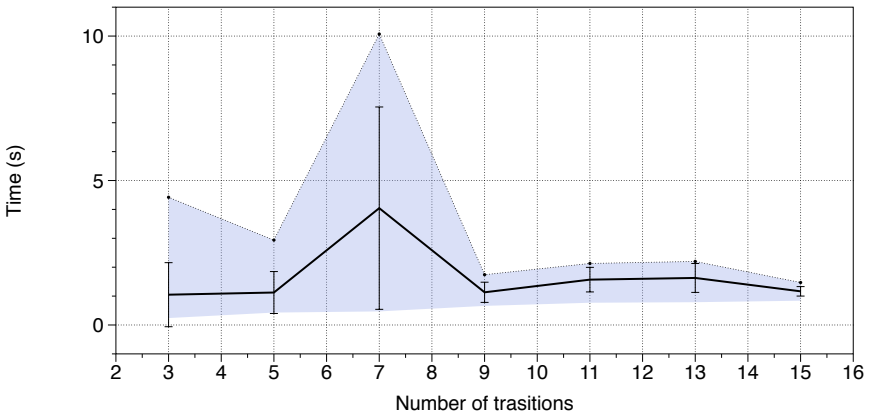

Grafico 6.6: Precomputazione a Design-Time in funzione del Numero di Transizioni (metodo Linear)

Ora verrà proposto un breve confronto con Param (versione 1.8). In questa test suite, composta da 5 modelli, è stata disabilitata la funzione di bisimulation; questo per ottenere una verifica delle sole capacità di calcolo parametrico per entrambi i tool. L'ambiente di esecuzione è ancora quello definito in precedenza; per Param è stata usata la versione compilata per LINUX a 64bit. La figura 6.7 mostra i risultati preliminari, i quali evidenziano che la risoluzione mediante Rational Sparse è migliore in confronto a quella applicata da Param (algoritmo di Daws modificato) in termini di tempo di esecuzione.

		States	
		50	100
Pars	13	24030.4	0.081887
	23	37602.8	—

		States	
		50	100
Pars	13	0.11	0.09
	23	1.01	96.77

Grafico 6.7: Rational Sparse (dx) Vs Param (sx)

D'altra parte Param è molto più efficiente in termini di utilizzo di memoria: RatSparse arriva ad occupare fino a 10 volte più memoria di Param. In entrambi i casi l'algoritmo qui proposto risulta, nel caso pessimo, molto più veloce di Param. I risultati ottenuti dimostrano l'efficacia del metodo

in esame, ma analizzando i dati è facile notare come, all'interno di alcune test suite, il tempo necessario per la risoluzione di qualche modello sia molto maggiore rispetto alla media. Questo fatto dimostra che esistono alcune tipologie di modelli che, a parità dei fattori descritti in precedenza (numero di stati, transizioni uscenti e parametri), sono caratterizzate da una risoluzione più onerosa; sarebbe interessante effettuare degli studi più approfonditi cercando di comprendere come siano costituite queste classi, con l'obiettivo di individuare i fattori principali che le caratterizzano.

Capitolo 7

Conclusioni e Sviluppi Futuri

7.1 Conclusioni

Questo studio ha introdotto il problema del *run-time efficient model checking* descrivendo come esso sia particolarmente adatto alla verifica di proprietà di reliability. E' stato fornito un approccio matematico, basato su Catene di Markov Parametriche, diviso in due passi rispettivamente da eseguire durante la fase di design e quella di run-time, con l'obiettivo di migliorare il più possibile le performance dell'analisi a run-time.

Il Model Checker è in grado di prendere in ingresso una PMC, definita attraverso una variante della sintassi PRISM, una proprietà (formulata in PCTL) e di fornire come output un'espressione, che rappresenta in forma parametrica il valore atteso di soddisfacibilità. Partendo dalla definizione del modello, viene generato lo spazio degli stati associato, sfruttando il linguaggio di scripting LUA [13]; successivamente la rete (PMC) viene modificata in modo tale da renderla adatta per la fase seguente. La risoluzione è la fase più importante di tutto il processo di MC; è a questo punto che l'impiego di tecniche ottimizzate permette di ottenere, in termini di utilizzo delle risorse, buoni risultati. In questo caso la scelta è caduta sull'utilizzo di tecniche di analisi numerica basate sull'algebra lineare, le quali sono oggetto di studi continui e sono particolarmente adatte ad essere parallelizzate. E' vero che

durante l'elaborato si è sottolineato, più e più volte, che i vincoli stringenti erano riferiti all'analisi durante la fase di run-time ma è altrettanto vero che la valutazione a design-time deve essere il più veloce possibile ed in grado di trattare correttamente problemi di grandi dimensioni. A run-time verrà applicata la filosofia di monitoring e models, con lo scopo di rilevare i cambiamenti all'interno del sistema, per poi aggiornare lo stato del modello e ottenere la valutazione aggiornata delle proprietà, relative all'affidabilità da monitorare.

Questo software è stato realizzato prevalentemente in C++ con alcune dovute eccezioni come: la computazione delle espressioni finali, le quali sono ottenute attraverso Maple, l'espansione dello spazio degli stati e la valutazione delle formule proposizionali entrambe realizzate in LUA. Inoltre sono state impiegate librerie avanzate tra cui: CVC3 per la gestione della parte proposizionale della logica, LibBoost per la parte di calcolo esterna a Maple ed il parsing delle formule durante la fase di valutazione a run-time. Durante lo sviluppo di questo progetto, la realizzazione del risolutore di sistemi lineari parametrici è, senza ombra di dubbio, la parte che ha richiesto l'impiego maggiore di risorse nelle fasi di ricerca e sviluppo.

Per tutto l'elaborato è stato preso come punto di riferimento Param, attualmente unico Model Checker adatto alle applicazioni run-time; questa soluzione è in grado di comportarsi meglio nei casi in cui non viene applicata bisimulation e di essere altrettanto valida, nei confronti di Param, quando invece viene applicate. Nel capitolo precedente è stato effettuato anche un piccolo confronto. Si è cercato poi di andare oltre, fornendo tecniche in grado di trattare qualsiasi proprietà esprimibile in PCTL e offrendo anche metodi di valutazione specifici per opportune classi di formule (formule Flat); tutto questo è stato realizzato cercando di non perdere mai di vista la ricerca delle prestazioni.

In conclusione è stato realizzato un MC in grado di trattare modelli parametrici di grandi dimensioni, sfruttando tecniche di calcolo matriciale algebrico e di valutare, in modo efficiente, proprietà espresse in PCTL su Catene

di Markov Parametriche (PMC); questo permette inoltre una veloce valutazione delle proprietà a run-time, rendendola possibile anche su dispositivi mobile.

7.2 Sviluppi Futuri

I possibili sviluppi futuri sono innumerevoli, partendo dall'espansione classe delle proprietà da valutare, arrivando fino alla modifica dei modelli sui quali è fatta l'inferenza. Prendendo in considerazione la possibilità di modificare la classe logica, sulla quale sono basate le proprietà da valutare, l'introduzione del supporto a formule appartenenti alla logica PCTL* potrebbe essere molto interessante. PCTL* è un'estensione di PCTL, che si ottiene da quest'ultima rimuovendo il vincolo secondo il quale ogni operatore temporale deve essere preceduto da una formula stato, e consente combinazioni booleane di formule percorso. Questo permette di formulare anche proprietà appartenenti ad un'altra famiglia di logiche dette LTL (Linear Temporal Logic). La logica PCTL* è strettamente più espressiva sia di PCTL, sia di LTL (impiegando bound probabilistici).

Una seconda modifica possibile potrebbe essere quella di cambiare, o meglio ampliare, la classe di modelli supportata dal MC. Un formalismo interessante è costituito dai Markov Decision Process (MDP). Le catene di Markov non permettono di esprimere un concetto importante nella modellazione di sistemi: il nondeterminismo. Gli MDP possono essere visti come una variante delle catene di Markov che permettono scelte sia di tipo probabilistico, sia di tipo nondeterministico. Come nelle DTMC, è possibile definire la funzione di stato successore attraverso una distribuzione di probabilità. Nei casi in cui una situazione non sia prevedibile in alcun modo o dove le proprietà di sistema non siano modellizzabili, è utile introdurre una scelta non deterministica.

E' possibile, tuttavia ricondurre un MDP ad una Catena di Markov Parametrica, introducendo opportuni parametri al posto delle scelte non determi-

nistiche. Infine è opportuno ricordare che tutti questi approcci sono validi, essendo modelli Markoviani, solamente se viene rispettata la proprietà di Markov, come definito nella sezione di Background. E' evidente che, per questa proprietà, il comportamento di un sistema è indipendente dalla sua storia passata (serve solamente conoscere lo stato iniziale). Questa è una limitazione importante e, in ottica futura potrebbe essere estremamente utile inserire nel sistema funzionalità che permettano di capire se, effettivamente, ci possono esistere delle dipendenze dalla storia passata del sistema. In realtà il concetto di condizione di Markov può essere esteso ad un numero finito di passi n, ovvero la storia del sistema non dipende soltanto dallo stato attuale, ma anche dagli n precedenti. E' però importante sottolineare che un modello di questo tipo può essere sempre e comunque ricondotto ad un altro equivalente che soddisfa la proprietà di Markov ad 1 passo.

Bibliografia

[1] Antonio Filieri, Carlo Ghezzi, Giordano Tamburelli *Run-Time Efficient Probabilistic Model Checking*

[2] Antonio Filieri, Carlo Ghezzi *Further Steps Toward Runtime Verification: Handling Probabilistic Cost Models*

[3] Ernst Moritz Hahn, Holger Hermanns, Lijun Zhang, Björn Wachter *Param TOOL*: http://depend.cs.uni-sb.de/tools/param/

[4] C. Bayer, J.P. Katoen *Principles of Model Checking.*

[5] Daws C. *Symbolic and Parametric Model Checking of Discrete Time Markov Chains*

[6] Lanotte R., Maggiolo-Schettini A., Troina A. *Parametric probabilistic transition systems for system design and analysis*

[7] *Maple*: http://www.maplesoft.com/products/Maple/index.aspx

[8] *Prism Model Checker*: http://www.prismmodelchecker.org/

[9] *Markov Remarked Model Checker*: http://www.prismmodelchecker.org/

[10] *Promela Language (Spin Model Checker)*: http://www.spinroot.com/

[11] R. Pearce *Solving Sparse Linear System in Maple*: http://www.mapleprimes.com/posts/41191-Solving-Sparse-Linear-Systems-In-Maple

[12] *Structured Gaussian Elimination*: http://www.farcaster.com/papers/crypto-solve/node5.html

[13] *Lua Scripting Language*: http://www.lua.org/