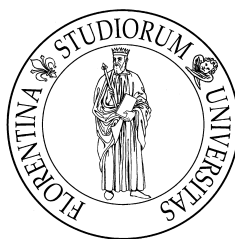


UNIVERSITÀ DEGLI STUDI DI FIRENZE  
Facoltà di Scienze Matematiche, Fisiche e Naturali  
Corso di Laurea Magistrale in Informatica



Tesi di Laurea

MODEL CHECKING COME SUPPORTO PER LE  
SCELTE DI SISTEMI ADATTIVI

MARCO TINACCI

Relatore: *Rocco De Nicola*

Correlatore: *Michele Loreti*

Anno Accademico 2012-2013



## SOMMARIO

---

Il problema affrontato è la ricerca di strategie per agenti adattivi, agenti che devono prendere decisioni che portino al raggiungimento dei loro obiettivi avendo a disposizione una conoscenza parziale o nulla degli elementi che compongono l'ambiente in cui agiscono. L'idea di base è quella di utilizzare un model checker su un modello di ambiente costruito su delle ipotesi e verificando una formula che formalizza l'obiettivo. Si fa quindi un utilizzo non convenzionale del model checker che viene impiegato per fare previsioni anziché verifiche. Presentiamo poi *Language for Population of Self-adaptive Agents*, un linguaggio specifico per modellare agenti adattivi, e la rispettiva implementazione in *Xtext*. Il model checker utilizzato nell'implementazione proposta è *PRISM*. Viene infine mostrato il funzionamento in un caso di studio dove è necessario fornire a un agente mobile uno scheduler che minimizzi gli scontri con altri agenti presenti nelle vicinanze.



*The key to any successful cooperative test is trust.  
And as our data clearly shows, humans cannot be trusted.  
The solution: robots! [...]  
Creating a foundation of mutual respect,  
reinforced by the simulated bonds of artificial friendship. [...]  
And finally, we put that trust to the test.  
Bam! Robots gave us six extra seconds of cooperation.  
Good job, robots.*

— Cave Johnson - Portal 2

## RINGRAZIAMENTI

---

Desidero ringraziare il Professor De Nicola per avermi seguito durante questo lavoro di tesi, per avermi indicato le direzioni su cui concentrarmi e per aver messo a confronto le mie idee con le sue. Ringrazio il Professor Loreti per il suo aiuto costante e per le ore passate a disegnare schemi sulla lavagna del suo ufficio.

Voglio ringraziare i miei genitori per il loro affetto che in questi anni non è mai mancato e per il sostegno economico. Ringrazio mio fratello che col suo carattere non mi permette mai di mostrare debolezza e che mi sprona a fare sempre meglio.

Ringrazio Valentina che è stata al mio fianco negli ultimi mesi e che ha condiviso con me le soddisfazioni e la fatica.

Infine voglio ringraziare gli informatici, sia scienziati che ingegneri, che hanno vissuto con me questi anni all'università rendendoli memorabili, spero che le nostre strade non smettano di incrociarsi. Un grazie anche alla compagnia dell'ostello, al novello sposo e alle altre due componenti del trio paloma per il tempo passato a giocare insieme senza sentire il bisogno di crescere.



## INDICE

---

1	INTRODUZIONE	1
<b>I</b>	<b>BACKGROUND</b>	<b>5</b>
2	RICHIAMI DI BASE	7
2.1	Variabili casuali	9
2.2	Processi stocastici	9
2.3	Discrete-Time Markov Chains	10
2.4	Markov Decision Processes	12
3	MODEL CHECKING	15
3.1	Probabilità elementari	16
3.2	Probabilistic Computation Tree Logic	18
3.3	PRISM model checker	20
<b>II</b>	<b>LAPSA: UN LINGUAGGIO PER AGENTI ADATTIVI</b>	<b>25</b>
4	LAPSA: LANGUAGE FOR POPULATION OF SELF-ADAPTIVE AGENTS	27
4.1	Sintassi	28
4.2	Zucchero sintattico	30
4.3	Semantica	32
4.4	Esempi	36
4.5	Da LAPSA a PRISM	37
<b>III</b>	<b>CASO DI STUDIO</b>	<b>41</b>
5	CASO DI STUDIO	43
5.1	Analisi	44
5.2	Approcci proposti	45
5.3	Simulazioni	48
<b>IV</b>	<b>CONCLUSIONI</b>	<b>55</b>
6	CONCLUSIONI	57
A	IMPLEMENTAZIONE DI LAPSA IN XTEXT	59

## ELENCO DELLE FIGURE

---

Figura 1	Rappresentazione grafica di una <i>DTMC</i> a quattro stati	11
Figura 2	Rappresentazione grafica di una <i>MDP</i> a quattro stati	13
Figura 3	Esempio di <i>DTMC</i> derivata dalla <i>MDP</i> di figura 2 tramite lo scheduler $\Sigma_1$	14
Figura 4	Schema del funzionamento di un model checker	16
Figura 5	Schema del funzionamento di un model checker probabilistico	17
Figura 6	Schema di come viene modellato uno scenario attorno all'agente che lo analizza	27
Figura 7	Schema del compilatore <i>LAPSA</i>	38
Figura 8	Utilizzo dei risultati nell'agente adattivo	39
Figura 9	Rappresentazione grafica di un possibile scenario del caso di studio	43
Figura 10	Alcuni passi di una simulazione. L'agente blu si muove secondo uno scheduler basato sul model checker che considera due altri agenti nell'arena. Gli agenti due agenti sono rossi e si muovono con uno scheduler casuale. Le frecce indicano la direzione verso la quale gli agenti si vogliono spostare al passo successivo.	51
Figura 11	Grafico dei risultati delle simulazioni: sull'ascissa variano gli agenti che sono realmente nell'area assieme all'agente principale, sull'ordinata viene indicato il numero di collisioni e i diversi colori rappresentano di diversi scheduler utilizzati dall'agente principale.	52

## ELENCO DELLE TABELLE

---

Tabella 1	Operatori derivati di <i>PCTL</i>	21
Tabella 2	Sintassi di <i>LAPSA</i>	31
Tabella 3	Operatori derivati di <i>LAPSA</i>	33
Tabella 4	Semantica di <i>LAPSA</i>	35
Tabella 5	Semantica del target del modulo principale	36



## LISTATI

---

3.1	Esempio di definizione dei moduli in <i>PRISM</i> . . . . .	22
3.2	Scelta nondeterministica in <i>PRISM</i> . . . . .	22
4.1	Esempio di random walk in <i>LAPSA</i> . . . . .	37
4.2	Versione nondeterministica della random walk in <i>LAPSA</i> . . . . .	37
5.1	Implementazione <i>LAPSA</i> dello scheduler basato su model checking	46
A.1	Sintassi di <i>LAPSA</i> in <i>Xtext</i> . . . . .	59
A.2	Implementazione della funzione di generazione in <i>Xtend</i> . . . . .	62
A.3	Generazione dei moduli in <i>Xtend</i> . . . . .	62
A.4	Traduzione della dichiarazione di variabili da <i>LAPSA</i> a <i>PRISM</i> . . . .	63
A.5	Traduzione delle transizioni da <i>LAPSA</i> a <i>PRISM</i> . . . . .	64

## ACRONIMI

---

CTL	<i>Computation Tree Logic</i>
CSL	<i>Continuous Stochastic Logic</i>
CSP	<i>Communicating Sequential Processes</i>
CTMC	<i>Continuous-Time Markov Chain</i>
DTMC	<i>Discrete-Time Markov Chain</i>
LAPSA	<i>Language for Population of Self-adaptive Agents</i>
LTL	<i>Linear Time Logic</i>
MDP	<i>Markov Decision Process</i>
PCTL	<i>Probabilistic Computation Tree Logic</i>
PTA	<i>Probabilistic Timed Automata</i>



## INTRODUZIONE

---

In molti scenari si stanno presentando problematiche inerenti a sistemi dove sono presenti agenti con capacità adattive. In tali sistemi un agente ha, generalmente, una conoscenza parziale del sistema in cui si muove e deve essere in grado di adattarsi alle circostanze modificando opportunamente la propria strategia per raggiungere l'obiettivo. Nel progetto ASCENS vengono proposti dei casi di studio dove sono coinvolti tre tipi diversi di agenti: robot, risorse e e-vehicles [SRA<sup>+</sup>11]. Pur sembrando tre scenari molto diversi vengono accomunati dalla necessità di interazione tra agenti al fine di raggiungere determinati obiettivi, del singolo o della collettività. Ogni agente è tipicamente a conoscenza delle sue caratteristiche interne ma può non avere a disposizione tutte le informazioni delle altre entità e dell'ambiente circostante.

Al fine di evitare ambiguità e di definire con maggior precisione gli obiettivi cerchiamo di rispondere alla domanda *"in che caso un sistema software si può dire adattivo?"* [BCL<sup>+</sup>12]. Diciamo che un sistema software è adattivo quando il suo comportamento e le sue scelte dipendono direttamente da un insieme di *dati di controllo* che possono variare a tempo di esecuzione. Un semplice esempio è un robot che deve arrivare a destinazione senza scontrarsi con altri robot o con ostacoli. L'area circostante il robot viene analizzata dai sensori di prossimità dai quali si potrà ricavare se e dove sono presenti ostacoli e sulla base di queste informazioni dovrà stabilire quale sarà la direzione migliore da prendere.

Gli scenari che coinvolgono robot sono tipicamente orientati sul comportamento di sciame: obiettivi come l'attraversamento di una buca [BCG<sup>+</sup>12] assemblandosi in gruppi richiedono una collaborazione esplicita mentre per la raccolta di risorse [GLPT11] o il raggiungimento di una posizione di arrivo comune è sufficiente minimizzare i casi in cui gli agenti si ostacolano a vicenda. Nel caso del cloud computing sono le risorse ad essere viste come gli agenti in gioco e gli obiettivi di interesse possono riguardare la disponibilità e gli aspetti legati alla qualità del servizio. Se consideriamo un insieme di veicoli elettrici (e-vehicles) in grado di comunicare entro un raggio limitato si può pensare a problemi legati all'ottimizzazione del trasporto di persone o oggetti, al traffico, alle disponibilità di parcheggio e alle stazioni di ricarica.

I due aspetti fondamentali comuni in questi scenari sono la comunicazione e la conoscenza limitata dell'ambiente, dove per ambiente si considera tutto quello che è esterno all'agente che osserva. Sono parte dell'ambiente anche gli altri agenti e quindi anche la conoscenza su di essi può essere parziale o nulla.

Quello che si propone è un metodo di risoluzione delle scelte di un agente basato sul *model checking*. L'uso convenzionale dei model checker è mirato alla

verifica di proprietà su modelli dei sistemi interessati. Generalmente si ha quindi la conoscenza completa del modello, cosa che non è garantita nel nostro caso. L'idea consiste nel formulare una strategia tramite la quale ipotizzare come si comporterà l'ambiente ed effettuare la verifica su proprietà di interesse. Quando si pone una scelta locale questa può essere risolta valutando la probabilità di raggiungere il nostro obiettivo dallo scenario in cui si arriva compiendo una determinata azione. Quello che viene fatto è quindi una previsione tramite la verifica della proprietà obiettivo sul modello ipotizzato.

Il model checker utilizzato è *PRISM* [KNP11], in quanto in grado di gestire ed analizzare processi stocastici come i *Markov Decision Process (MDP)* che sono il principale modello a cui si fa riferimento. *PRISM* esegue model checking probabilistico ed è quindi in grado di restituire le probabilità con cui una certa formula viene soddisfatta.

Se prendiamo in considerazione un altro esempio con protagonista un robot che deve rimanere in movimento minimizzando gli scontri con altri robot o ostacoli, possiamo immaginare il sistema composto dal soggetto in parallelo all'ambiente. L'ambiente sarà composto da tutti gli altri robot e ostacoli che il robot adattivo riesce a percepire o di cui ipotizza la presenza. Supponendo che la scelta da prendere riguardi il punto cardinale verso quale muoversi, quello che può essere fatto è utilizzare il model checker per ricavare le probabilità di non scontrarsi con nessuno entro dieci passi nel caso in cui si faccia un passo in una delle quattro direzioni. Avremo così a disposizione una probabilità di successo finale per ogni scelta e sarà sufficiente propendere per quella più alta.

Per realizzare questo approccio è necessario un lavoro di formalizzazione, introduciamo quindi un linguaggio col quale modellare il comportamento dell'agente adattivo. Introduciamo la specifica di *Language for Population of Self-adaptive Agents (LAPSA)*, un linguaggio per agenti adattivi, dove i comportamenti vengono modellati da moduli descritti come modelli reattivi e vengono offerte primitive per la gestione della percezione dell'ambiente. L'utilizzatore di *LAPSA* può limitarsi alla descrizione del comportamento del soggetto e di come viene gestita la visione dell'ambiente. La compilazione del codice *LAPSA* comporrà un modello *PRISM* e un file di formule *Probabilistic Computation Tree Logic (PCTL)* sui quali potrà essere eseguito il model checking. A seconda delle potenzialità dell'agente si potrà inserire dei richiami al model checker da valutare sul momento oppure, in caso di un numero sufficientemente basso di scenari considerati, fornire un codice precompilato contenente solamente la migliore scelta da fare a seconda dell'ambiente percepito.

Viene fornita un'implementazione di *LAPSA* in *Xtext* [xteb], un plugin di *ECLIPSE* che permette lo sviluppo di compilatori per linguaggi completi di un ambiente di sviluppo a supporto. A partire dalla grammatica del linguaggio *Xtext* genera automaticamente funzionalità accessorie legate all'ambiente di sviluppo come auto-completamento e colorazione del codice. Inoltre aggiungendo istruzioni sulla traduzione del codice tramite il linguaggio di template *Xtend* [xtea] vengono

costruiti automaticamente lexer e parser basati su *ANTLR* [PPQ95]. Raccogliendo tutte queste funzionalità si ottiene un tool installabile come plugin direttamente su *ECLIPSE*.

Si mostreranno infine i risultati di un semplice caso di studio confrontando i risultati di simulazioni basate su ipotesi di complessità crescente e quelli di approcci naïve.



## Parte I

### BACKGROUND





## RICHIAMI DI BASE

---

In questo capitolo verranno richiamate alcune definizioni base che risulteranno fondamentali per la comprensione delle sezioni successive. Cominciando dalle basi di probabilità [Bil12] si arriverà a formulare le definizioni dei principali modelli utilizzati: *Discrete-Time Markov Chain* (DTMC) e *Markov Decision Process* (MDP) [KNP11].

**Definizione 1 (Esperimento casuale  $\mathcal{C}$ )** Per esperimento casuale si intende un qualsiasi avvenimento, provocato più o meno direttamente dall'uomo, suscettibile di manifestarsi secondo una pluralità di eventi elementari.

**Definizione 2 (Spazio fondamentale  $\Omega$ )** Lo spazio fondamentale  $\Omega$  di  $\mathcal{C}$  è l'insieme di tutti i suoi eventi elementari. Indichiamo tali eventi elementari come gli elementi  $\omega \in \Omega$ .

**Definizione 3 (Eventi casuali  $\mathcal{E}$ )** Un evento casuale  $A \in \mathcal{E}$  è una proposizione relativa all'esito di un evento casuale  $\mathcal{C}$  che, prima del compimento di  $\mathcal{C}$ , è in qualche modo incerto.

**Osservazione 1**  $\mathcal{E}$  contiene sottoinsiemi di  $\Omega$

$$A \in \mathcal{E} \Rightarrow A \subseteq \Omega$$

**Definizione 4 ( $\sigma$ -algebra)** Sia  $\Omega$  lo spazio fondamentale dell'evento casuale  $\mathcal{C}$ .  $\mathcal{F} \subseteq 2^\Omega$  è una  $\sigma$ -algebra se e solo se

- $\Omega \in \mathcal{F}$ ,
- $A \in \mathcal{F} \Rightarrow \bar{A} \in \mathcal{F}$ ,
- $\bigwedge_{i=1}^{\infty} A_i \in \mathcal{F} \Rightarrow \bigcup_{i=1}^{\infty} A_i \in \mathcal{F}$   
oppure  $A_i \in \mathcal{F} (i \in I) \Rightarrow \bigcup_{i \in I} A_i \in \mathcal{F} \wedge \bigcap_{i \in I} A_i \in \mathcal{F}$ .

Gli elementi di una  $\sigma$ -algebra sono chiamati *insiemi misurabili*. Chiamiamo *spazio misurabile* uno spazio fondamentale su cui è definita una  $\sigma$ -algebra e quindi lo identifichiamo con la coppia  $(\Omega, \mathcal{F})$ .

**Definizione 5 (Insieme dei rettangoli)** Sia  $\Omega = \mathbb{R}$ , l'insieme dei rettangoli è definito come

$$\mathcal{I} = \{(a, b] \mid a, b \in \mathbb{R} \cup \{-\infty, \infty\}\}.$$

**Definizione 6 (Insieme di Borel)** Un insieme di Borel  $\mathcal{B}(\mathbb{R})$  è la più piccola  $\sigma$ -algebra che contiene l'insieme dei rettangoli  $\mathcal{I}$ .

**Definizione 7 (Spazio di Borel)** Uno spazio di Borel su  $\mathbb{R}$  è lo spazio misurabile  $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$ .

**Definizione 8 (Assiomi di Kolmogoroff)** Dato lo spazio misurabile  $(\Omega, \mathcal{F})$ , una misura di probabilità su di esso è una funzione  $\mathbb{P} : \mathcal{F} \rightarrow \mathbb{R}_{\geq 0}$  tale che

$$\mathbb{P}(\emptyset) = 0$$

$$\mathbb{P}(\Omega) = 1$$

e, per qualsiasi famiglia  $\{A_i \mid A_i \in \mathcal{F}, i \in \mathbb{N}\}$  tale che  $k \neq h \Rightarrow A_k \cap A_h = \emptyset$ , vale:

$$\mathbb{P}\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \mathbb{P}(A_i)$$

Chiamiamo *spazio di probabilità* dell'esperimento casuale  $\mathcal{C}$  la tripla  $(\Omega, \mathcal{F}, \mathbb{P})$ , dove  $\Omega$  è lo spazio fondamentale,  $(\Omega, \mathcal{F})$  lo spazio misurabile e  $\mathbb{P}$  la misura di probabilità su  $\mathcal{F}$ . Se esiste l'insieme numerabile  $A \subseteq \Omega$  tale che  $\sum_{a \in A} \mathbb{P}\{a\} = 1$  allora diciamo che  $\mathbb{P}$  è una *misura di probabilità discreta* e  $(\Omega, \mathcal{F}, \mathbb{P})$  è uno *spazio di probabilità discreto*.

**Proposizione 1 (Proprietà di  $\mathbb{P}$ )** Dato lo spazio di probabilità  $(\Omega, \mathcal{F}, \mathbb{P})$ :

1.  $\forall A \in \mathcal{F} : \mathbb{P}A + \mathbb{P}\bar{A} = 1,$
2.  $\forall A, B \in \mathcal{F} : A \subseteq B \Rightarrow \mathbb{P}A \leq \mathbb{P}B,$
3.  $\forall A \in \mathcal{F} : \mathbb{P}A \leq 1,$
4.  $\forall A, B \in \mathcal{F} : \mathbb{P}(A \cup B) \geq \max\{\mathbb{P}A, \mathbb{P}B\},$
5.  $\forall A, B \in \mathcal{F} : \mathbb{P}A \cap B \leq \min\{\mathbb{P}A, \mathbb{P}B\},$
6.  $\forall A, B \in \mathcal{F} : \mathbb{P}(A \cup B) = \mathbb{P}A + \mathbb{P}B - \mathbb{P}(A \cap B),$
7.  $\forall A, B \in \mathcal{F} : A \subseteq B \Rightarrow \mathbb{P}(B \setminus A) = \mathbb{P}B - \mathbb{P}A,$
8.  $\forall A_i \in \mathcal{F} : \mathbb{P}\left(\bigcup_{i=0}^{\infty} A_i\right) \leq \sum_{i=0}^{\infty} \mathbb{P}A_i.$

**Definizione 9 (Probabilità condizionale)** Dato lo spazio di probabilità  $(\Omega, \mathcal{F}, \mathbb{P})$  e  $A, B \in \mathcal{F}$  tali che  $\mathbb{P}B > 0$  si definisce *probabilità condizionale*

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}B}$$

o *alternativamente*

$$\mathbb{P}(A|B) \cdot \mathbb{P}B = \mathbb{P}(A \cap B) = \mathbb{P}(B|A) \cdot \mathbb{P}A$$

**Definizione 10 (Eventi stocasticamente indipendenti)** Due eventi  $A$  e  $B$  sono stocasticamente indipendenti se e solo se

$$\mathbb{P}(A \cap B) = \mathbb{P}A \cdot \mathbb{P}B$$

**Proposizione 2 (Proprietà di eventi stocasticamente indipendenti)** Se  $A$  e  $B$  sono eventi stocasticamente indipendenti allora valgono le seguenti proprietà:

1.  $\bar{A}$  e  $B$  sono stocasticamente indipendenti,
2.  $A$  e  $\bar{B}$  sono stocasticamente indipendenti,
3.  $\bar{A}$  e  $\bar{B}$  sono stocasticamente indipendenti,
4.  $\mathbb{P}(A \cup B) = 1 - \mathbb{P}\bar{A} \cdot \mathbb{P}\bar{B}$ .

## 2.1 VARIABILI CASUALI

Una variabile casuale è definita da una funzione che assegna un valore a ogni elemento dello spazio fondamentale  $\Omega$ .

**Definizione 11 (Funzione misurabile)** Dati gli spazi misurabili  $(\Omega_1, \mathcal{F}_1)$  e  $(\Omega_2, \mathcal{F}_2)$ ,  $f : \Omega_1 \rightarrow \Omega_2$  è una funzione misurabile se e solo se

$$\forall A \in \mathcal{F}_2 : f^{\text{orig}}(A) \triangleq \{\omega \in \Omega_1 \mid f(\omega) \in A\} \in \mathcal{F}_1$$

**Definizione 12 (Variabile casuale)** Una variabile casuale è definita da una funzione misurabile

$$X : \Omega \rightarrow \mathbb{R}$$

dove  $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$  è lo spazio di Borel su  $\mathbb{R}$ .

**Proposizione 3** Dato  $(\Omega_1, \mathcal{F}_1, \mathbb{P})$  spazio di probabilità,  $(\Omega_2, \mathcal{F}_2)$  spazio misurabile e  $f : \Omega_1 \rightarrow \Omega_2$  funzione misurabile, allora:

$$(\Omega_2, \mathcal{F}_2, \mathbb{P} \circ f^{\text{orig}})$$

è uno spazio di probabilità.

## 2.2 PROCESSI STOCASTICI

**Definizione 13 (Processo stocastico)** Un processo stocastico è una famiglia  $T$ -indicizzata di variabili casuali, dove  $T \subseteq \mathbb{R}^+$  è lo spazio della variabile temporale

$$\{X_i \mid i \in T\}$$

L'insieme  $T$  può essere considerato sia continuo che discreto e, a seconda del caso, si può formulare nelle due seguenti definizioni:

**Definizione 14 (Processo stocastico discreto)** Un processo stocastico discreto è un processo stocastico con  $T = \mathbb{N}$

$$\{X_n \mid n \in \mathbb{N}\}$$

**Definizione 15 (Processo stocastico continuo)** Un processo stocastico continuo è un processo stocastico con  $T = \mathbb{R}$

$$\{X_t \mid t \in \mathbb{R}\}$$

### 2.3 DISCRETE-TIME MARKOV CHAINS

Le DTMC sono processi stocastici discreti  $\{X_n \mid n \in \mathbb{N}\}$  aventi uno spazio degli stati discreto e che godono della proprietà di *assenza di memoria* di Markov

$$\mathbb{P}\{X_n = i_n \mid X_{n-1} = i_{n-1}, \dots, X_0 = i_0\} = \mathbb{P}\{X_n = i_n \mid X_{n-1} = i_{n-1}\}.$$

Definiamo una DTMC come segue

**Definizione 16 (DTMC)** Una DTMC è una tupla  $\mathcal{D} = (S, \bar{s}, \mathbb{P})$  dove:

- $S$  è un insieme finito di stati;
- $\bar{s} \in S$  è lo stato iniziale;
- $\mathbb{P} : S \times S \rightarrow [0, 1]$  è la matrice di probabilità delle transizioni, tale che:

$$\sum_{s' \in S} \mathbb{P}(s, s') = 1$$

per ogni stato  $s \in S$ .

**Definizione 17 (DTMC path)** Un cammino  $\pi$  di una DTMC  $\mathcal{D} = (S, \bar{s}, \mathbb{P})$  che comincia da  $s \in S$  è una sequenza non vuota  $s_0, s_1, s_2, \dots$  di stati tali che:

1.  $s = s_0$
2.  $\forall i : \mathbb{P}(s_i, s_{i+1}) > 0$

**Osservazione 2** Un cammino  $\pi$  di una DTMC può essere finito o infinito.

**Definizione 18 (DTMC path up to  $n$ )** Sia  $\pi$  un cammino, il cammino up to  $n$ , o  $\pi \upharpoonright n$ , è il prefisso  $s_0, \dots, s_n$  di  $\pi$ .

**Definizione 19 (DTMC paths)**  $\text{Paths}_{s_0}^{\mathcal{D}}$  o  $\text{Paths}(\mathcal{D}, s_0)$ , è l'insieme di tutti i cammini di una DTMC  $\mathcal{D}$  che cominciano in  $s_0$ .

Utilizziamo inoltre le seguenti notazioni riguardanti i cammini su DTMC:

- $\pi(i)$  indica l' $i$ -esimo stato del cammino  $\pi$ ,
- $|\pi|$  indica la lunghezza del cammino  $\pi$ ,
- $\text{last}(\pi)$  indica l'ultimo stato del cammino finito  $\pi$  e
- $\text{Paths}_s^{\text{fin}}$  indica l'insieme di tutti i cammini finiti che cominciano in  $s$ .

Illustriamo con un esempio la struttura appena definita: in figura 1 viene raffigurata con una notazione grafica una *DTMC* a quattro stati. Nella rappresentazione gli stati sono indicati come cerchi e le transizioni come frecce etichettate con la probabilità associata. Lo stato iniziale viene indicato con una freccia entrante. La matrice di probabilità associata a questa *DTMC* avrà la seguente

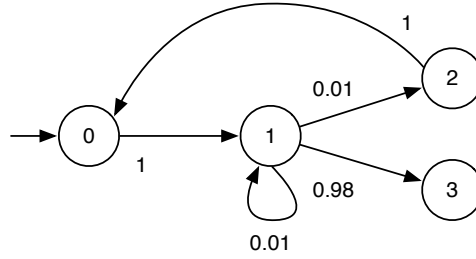


Figura 1.: Rappresentazione grafica di una *DTMC* a quattro stati

forma:

$$\mathbb{P} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0.01 & 0.01 & 0.98 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

L'esempio può rappresentare un processo che tenta di spedire un messaggio: dallo stato iniziale 0 si entra nello stato 1 dove con probabilità 0.01 si aspetta l'istante successivo, con probabilità 0.98 si consegna il messaggio con successo e con probabilità 0.01 la consegna fallisce. In caso di fallimento il processo ricomincia dallo stato iniziale mentre in caso di successo si entra permanentemente nello stato 3.

Definiamo la misura di probabilità di un singolo cammino.

**Definizione 20** Sia  $\pi_{\text{fin}} \in \text{Paths}_s^{\text{fin}}$  un qualsiasi cammino finito, la probabilità di un cammino finito si definisce come segue

$$\mathbb{P}_s(\pi_{\text{fin}}) \triangleq \begin{cases} 1 & \text{se } n = 0 \\ \mathbb{P}(\pi(0), \pi(1)) \dots \mathbb{P}(\pi(n-1), \pi(n)) & \text{altrimenti} \end{cases}$$

dove  $n = |\pi_{\text{fin}}|$ .

**Definizione 21 (Insieme dei cilindri)** Sia  $n \in \mathbb{N}_0$  e  $\mathcal{D}$  una DTMC, definiamo un insieme dei cilindri come

$$C(\pi_{\text{fin}}) \triangleq \{\pi \in \text{Paths}_s^{\mathcal{D}} \mid \pi_{\text{fin}} \text{ è un prefisso di } \pi\}$$

L'insieme dei cilindri quindi rappresenta tutti i cammini che hanno come prefisso comune il cammino finito specificato. Definiamo poi la misura di probabilità di un singolo cammino

Forniamo infine la misura di probabilità di un insieme di cammini di una DTMC

**Definizione 22** Sia  $\pi_{\text{fin}} \in \text{Paths}_s^{\text{fin}}$  un qualsiasi cammino finito, la probabilità di un insieme di cammini è definita come segue

$$\text{Prob}_s(C(\pi_{\text{fin}})) = \mathbb{P}_s(\pi_{\text{fin}}).$$

Di seguito riportiamo alcuni esempi di probabilità su insiemi di cammini della DTMC di figura 1:

$$\text{Prob}_0(C(0, 1, 1, 1)) = 1 \cdot 0.01 \cdot 0.01 = 0.00001$$

$$\text{Prob}_0(C(0, 1, 1, 2)) = 1 \cdot 0.01 \cdot 0.01 = 0.00001$$

$$\text{Prob}_0(C(0, 1, 1, 3)) = 1 \cdot 0.01 \cdot 0.98 = 0.00098$$

$$\text{Prob}_0(C(0, 1, 2, 0)) = 1 \cdot 0.01 \cdot 1 = 0.01$$

$$\text{Prob}_0(C(0, 1, 3, 3)) = 1 \cdot 0.98 \cdot 1 = 0.98$$

## 2.4 MARKOV DECISION PROCESSES

Le MDP estendono le DTMC aggiungendo la possibilità di esprimere *nondeterminismo*. Il nondeterminismo è un meccanismo di astrazione che permette di nascondere qualsiasi aspetto quantitativo che caratterizza una scelta, come le probabilità. Non si può quindi fare alcuna assunzione su come verrà presa una scelta deterministica ma sono note le scelte possibili.

**Definizione 23** Un MDP è una tupla  $\mathcal{M} = (S, \bar{s}, \text{Act}, \text{Steps})$  dove:

- $S$  è un insieme finito di stati;
- $\bar{s} \in S$  è lo stato iniziale;
- $\text{Act}$  è un insieme di azioni;
- $\text{Steps} : S \rightarrow 2^{\text{Act} \times \text{Dist}(S)}$  è la funzione di transizione probabilistica.

**Definizione 24 (MDP path)** Un cammino  $\pi$  di una MDP  $\mathcal{M} = (S, \bar{s}, \text{Act}, \text{Steps})$  che comincia da  $s_0$  è una sequenza non vuota

$$s_0, a_1, \mu_1, s_1, a_2, \mu_2, s_3, \dots$$

dove  $s_i \in S$ ,  $(a_{i+1}, \mu_{i+1}) \in \text{Steps}(s_i)$  e  $\mu_{i+1}(s_{i+1}) > 0$  per ogni  $i \geq 0$ .

**Osservazione 3** Un cammino  $\pi$  di una MDP può essere finito o infinito.

**Definizione 25 (MDP path up to  $n$ )** Sia  $\pi$  un cammino, il cammino up to  $n$ , o  $\pi \uparrow n$ , è il prefisso  $s_0, \dots, s_n$  di  $\pi$ .

**Definizione 26 (MDP paths)**  $\text{paths}_{s_0}^{\mathcal{M}}$  o  $\text{paths}(\mathcal{M}, s_0)$ , è l'insieme di tutti i cammini della MDP  $\mathcal{M}$  che cominciano in  $s_0$ .

Analogamente alle DTMC, anche per le MDP utilizziamo le seguenti notazioni:

- $\pi(i)$  indica l' $i$ -esimo stato del cammino  $\pi$ ,
- $|\pi|$  indica la lunghezza del cammino  $\pi$ ,
- $\text{last}(\pi)$  indica l'ultimo stato del cammino finito  $\pi$  e
- $\text{Paths}_s^{\text{fin}}$  indica l'insieme di tutti i cammini finiti che iniziano in  $s$ .

Riportiamo un esempio di MDP in figura 2. La rappresentazione grafica estende quella utilizzata nell'esempio di DTMC in figura 2 aggiungendo delle etichette che riportano il nome dell'azione eseguita dalle transizioni. Nello stato iniziale 0 ci si muove sempre nello stato 1 compiendo l'azione  $a$  dove abbiamo una scelta nondeterministica tra le azioni  $b$  e  $c$ . Scegliendo  $b$  avremo una probabilità 0.7 di tornare allo stato iniziale e 0.3 di restare nello stato 1. Se invece viene scelta l'azione  $c$  gli stati di arrivo 2 e 3 avranno ciascuno probabilità 0.5.

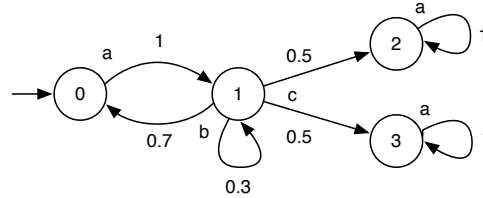


Figura 2.: Rappresentazione grafica di una MDP a quattro stati

**Definizione 27 (Scheduler)** Uno scheduler  $\Sigma$  di una MDP  $\mathcal{M}$  è una funzione che mappa tutti gli elementi di  $\pi_{\text{fin}}$  di  $\mathcal{M}$  in un elemento dell'insieme  $\text{Steps}(\text{last}(\pi_{\text{fin}}))$  indicato con  $S(\pi_{\text{fin}})$ . Con  $\text{Scheduler}_{\mathcal{M}}$  denotiamo l'insieme di tutti i possibili scheduler di  $\mathcal{M}$  e, per ogni scheduler  $\Sigma$ , indichiamo con  $\text{path}_s^{\Sigma}$  il sotto insieme di  $\text{path}_s$  che corrisponde a  $\Sigma$ .

Facendo riferimento all'esempio di figura 2 descriviamo dei possibili scheduler  $\Sigma_1$  e  $\Sigma_2$  di quella MDP. Useremo le notazioni  $\mu_b$  e  $\mu_c$  per rappresentare rispettivamente le distribuzioni derivanti dalla scelta delle azioni  $b$  e  $c$  presenti nello stato 1.

- $\Sigma_1 : \Sigma_1(0, a, 1) = \mu_c$
- $\Sigma_2 : \Sigma_2(0, a, 1) = \mu_b, \Sigma_2(0, a, 1, b, 1) = \mu_c, \Sigma_2(0, a, 1, b, 0, a, 1) = \mu_c$

Entrambi gli scheduler risolvono tutte le scelte nondeterministiche che si possono trovare durante l'esecuzione riducendo la *MDP* a una *DTMC*. Una volta giunti agli stati 2 e 3, infatti, non sarà più possibile tornare agli stati precedenti o eseguire qualsiasi tipo di scelta.

Formalizziamo il passaggio da una *MDP*  $\mathcal{M} = (S, \bar{s}, \text{Act}, \text{Steps})$  alla *DTMC*  $\mathcal{D}^\Sigma = (S^\Sigma, \bar{s}^\Sigma, \mathbb{P}^\Sigma)$  applicando lo scheduler  $\Sigma$ . Il comportamento di uno stato  $s \in S$  può essere descritto infatti con la *DTMC* a stati infiniti la cui struttura dipenderà da quella della *MDP* nel seguente modo:

- $S^\Sigma = \text{Paths}_s^{\text{fin}},$
- $\bar{s}^\Sigma = s$
- per due cammini finiti  $\pi, \pi' \in S^\Sigma$ :

$$\mathbb{P}^\Sigma(\pi, \pi') = \begin{cases} \mu(s') & \text{se } \pi' \text{ è della forma } \pi, \Sigma(\pi), s' \text{ con } \Sigma(\pi) = (a, \mu) \\ 0 & \text{altrimenti} \end{cases}$$

In figura 3 viene riportata la *DTMC*  $\mathcal{D}^{\Sigma_1}$  ottenuta dall'esempio di scheduler  $\Sigma_1$ . La misura di probabilità definita per le *DTMC* può quindi essere utilizzata per

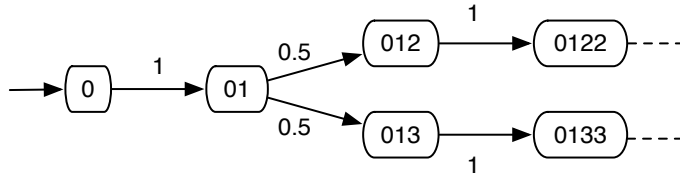


Figura 3.: Esempio di *DTMC* derivata dalla *MDP* di figura 2 tramite lo scheduler  $\Sigma_1$

una *DTMC* derivata da una *MDP* tramite uno scheduler. Dato che l'utilizzo di un singolo scheduler risulta eccessivamente restrittivo possiamo generalizzarla per ricercare la massima o la minima probabilità su tutti i possibili scheduler.



## MODEL CHECKING

---

Garantire l'assenza di errori in sistemi complessi come i software è un problema di grande interesse sia nell'ambiente dell'industria che in quello della ricerca. Nel primo settore si sono diffusi svariati strumenti tra cui il *testing*, le *simulazioni* e le *peer review* al fine di far fronte al problema. Il testing è una tecnica dinamica che spesso si appoggia a componenti di terzi e ad attività di *mock-up*, cioè anteprime parziali a scopo esplicativo dei requisiti richiesti. Si sono diffuse anche tecniche di sviluppo orientate al testing: partendo dai requisiti si ricercano i test che il sistema (ancora inesistente) dovrà superare, quindi si passa allo sviluppo supervisionato da verifiche costanti. Il principale problema del testing è che fornisce una copertura solamente parziale di quello che viene richiesto e in sistemi complessi come ad esempio i *software multithread* gli interleaving che si creano sono un numero impossibile da gestire. Anche per le simulazioni valgono gli stessi punti elencati per il testing: possono garantire che il caso simulato sia corretto ma non che l'intero sistema lo sia.

La tecnica di peer review consiste nello scambio di codice tra programmatori. Anche in questo caso però diventa impossibile gestire sistemi concorrenti per l'elevato numero di interleaving.

Il *model checking* è un metodo formale che affronta il problema della correttezza di un sistema. La tecnica si basa sulla costruzione di un modello astratto che rappresenti il sistema e di una formula che rappresenti il requisito da soddisfare. Entrambi gli elementi devono essere espressi in modo formale secondo una struttura nota per poter rendere questo metodo automatizzabile. Il *model checker* (figura 4) è quindi uno strumento che prende in ingresso la rappresentazione del sistema e la formula e risponde con un esito positivo se il sistema la soddisfa, negativo altrimenti, possibilmente fornendo un controesempio.

In un contesto reale è spesso impossibile pensare di ottenere un sistema complesso totalmente privo di errori o imperfezioni, si rilassano quindi le richieste introducendo dei gradi di tolleranza degli errori. Un caso concreto può essere il gestore di una compagnia telefonica che permette di effettuare un'alta percentuale di chiamate senza problemi di interferenze o interruzioni, ad esempio il 98%. Oltre al modello e alla formula viene quindi introdotto nel model checker il parametro dell'*accuratezza* che rende una formula verificata se il grado di fallimento rientra nella tolleranza espressa. Un model checker a cui si aggiunge un parametro di accuratezza espresso in probabilità viene chiamato *probabilistic model checker* (figura 5).

Dalla formalizzazione dei requisiti si può ottenere un insieme di proprietà che dovranno essere soddisfatte. Più precisamente, dai requisiti *funzionali* possiamo

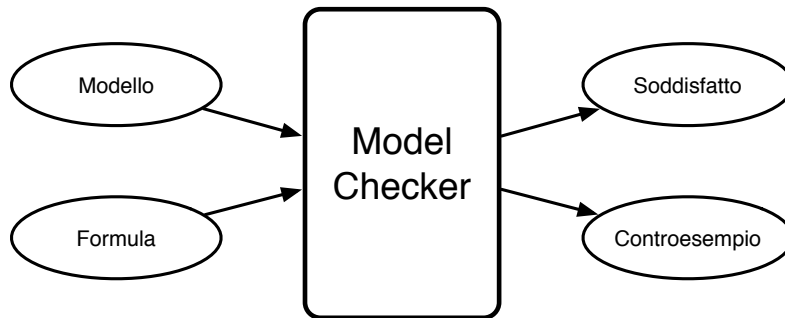


Figura 4.: Schema del funzionamento di un model checker

ottenere le formule di proprietà da soddisfare, mentre quelli *non funzionali* forniscono l'accuratezza da utilizzare. Dopo aver fornito al model checker il modello del sistema, la proprietà da verificare e l'accuratezza, questo può produrre tre tipi di risposte:

- la proprietà viene soddisfatta dal modello nei limiti richiesti,
- la memoria non è sufficiente per portare a termine la verifica oppure
- la proprietà non viene soddisfatta dal modello, fornendo un controesempio.

Se non si riceve esito positivo si può intervenire a seconda di che tipo di problema è stato riscontrato. Nel secondo caso si verifica il fenomeno conosciuto come *esplosione degli stati* che ha origine quando si vuole modellare sistemi molto complessi, come i già citati software multithread. La conseguenza è una richiesta di memoria e tempo di calcolo proibitivi. Per far fronte a questo problema si ricorre spesso alla scomposizione del sistema in sotto sistemi e ad astrazioni di modelli troppo complessi. Ovviamente se invece non viene soddisfatta la formula è necessario intervenire sul modello stesso cercando di correggere i comportamenti erronei evidenziati.

Il model checking viene utilizzato in modo diverso se ci troviamo prima o dopo la fase di sviluppo: nel primo caso è necessario costruire il modello astratto del sistema, nel secondo, invece, è possibile utilizzare degli strumenti (i.e. *java path finder*) per effettuare le verifiche direttamente sul codice.

### 3.1 PROBABILITÀ ELEMENTARI

Introduciamo le misure principali utilizzate dei model checker probabilistici, assumendo di utilizzare come strutture dei modelli le *DTMC* e le *MDP*. Esistono due tipi di misure di probabilità elementari delle *DTMC*:

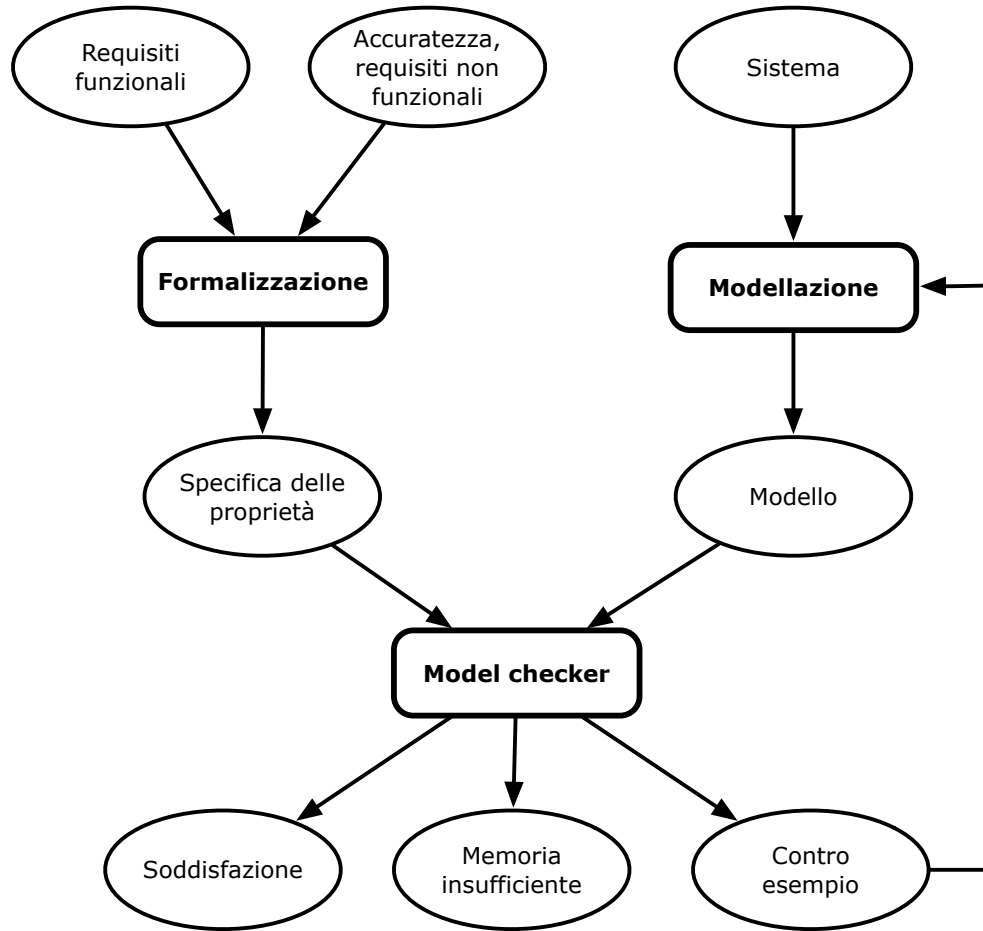


Figura 5.: Schema del funzionamento di un model checker probabilistico

- probabilità *transiente* e
- probabilità *a regime*, o *steady state*.

**Definizione 28 (Probabilità transiente  $\pi_j(n)$ )** La probabilità transiente  $\pi_j(n) = \mathbb{P}\{X_n = j\}$  è la probabilità che la DTMC sia nello stato  $j$  al passo  $n$

Possiamo quindi associare alla DTMC  $\mathcal{D}$  un vettore che al passo  $n$  descriva la probabilità di trovarsi in ogni stato  $s \in S$

$$\underline{\pi}(n) \triangleq (\pi_1(n), \dots, \pi_{|S|}(n)).$$

Si indica con  $\underline{\pi}(0)$  la distribuzione di probabilità iniziale mentre con  $\underline{\pi}(n)$  la distribuzione di probabilità al passo  $n$ . Considerando che moltiplicare il vettore di distribuzione di probabilità per la matrice  $\mathbb{P}$  rappresenta un avanzamento del

sistema che aggiorna la distribuzione al passo successivo, allora vale la seguente *relazione di ricorrenza*

$$\underline{\pi}(n) = \underline{\pi}(n-1) \cdot \mathbb{P}$$

da cui si ricava immediatamente la seguente forma dipendente solo dalla distribuzione iniziale e dalla matrice di transizione

$$\underline{\pi}(n) = \underline{\pi}(0) \cdot \mathbb{P}^n$$

**Definizione 29 (Probabilità steady state  $\pi_j$ )** *La probabilità steady state*

$$\pi_j = \lim_{n \rightarrow \infty} \mathbb{P}\{X_n = j\}$$

*è la probabilità che la DTMC sia a regime nello stato  $j$ .*

L'esistenza di questo limite è garantita solo sotto determinate condizioni di *ergodicità* della catena. Supponendo che il limite sia indipendente dalla distribuzione iniziale, per calcolare questa distribuzione è sufficiente risolvere il seguente sistema di equazioni lineari

$$\begin{cases} \underline{\pi} \cdot \mathbb{P} = \underline{\pi} \\ \sum_{i=1}^{|S|} \pi_i = 1 \end{cases}$$

dove  $0 \leq \pi_i \leq 1$  e  $1 \leq i \leq |S|$ .

Una volta scelto uno scheduler che risolva le scelte nondeterministiche della *MDP* trasformandola in una *DTMC* sarà possibile applicare la valutazione delle probabilità sopra descritte. Il risultato però sarà valido solo in presenza di quello specifico scheduler che potrebbe avere un peso poco rilevante nell'analisi della *MDP*. Quello che si fa quindi è calcolare il range di probabilità in cui si muove la misura interessata *per ogni* possibile scheduler in modo da poter fare inferenza su *lower* e *upper bounds*.

### 3.2 PROBABILISTIC COMPUTATION TREE LOGIC

Al fine di poter effettuare model checking su strutture come *DTMC* e *MDP* utilizziamo *PCTL*, un'estensione probabilistica della logica temporale *Computation Tree Logic* (*CTL*). Il linguaggio *PCTL* è uno strumento che permette di esprimere specifiche di interesse sulla struttura che stiamo considerando. Può essere quindi utilizzato sia sulle *DTMC* che sulle *MDP* adattando la struttura alla potenza espressiva del modello. In generale le formule che esprimono specifiche in modo formale hanno un ruolo fondamentale all'interno del model checking in quanto permettono di rendere automatico il procedimento di verifica.

In seguito riportiamo la sintassi e la semantica di *PCTL* definita per le *MDP*, più generale rispetto a quella per le *DTMC* ed effettivamente impiegata all'interno di questo lavoro.

**Definizione 30 (Sintassi PCTL)** *La sintassi PCTL viene definita come segue:*

$$\begin{aligned}\phi &::= \text{true} \mid a \mid \phi_1 \wedge \phi_2 \mid \neg \phi \mid \mathcal{P}_{\bowtie p}[\psi] \\ \psi &::= \mathcal{X}\phi \mid \phi_1 \mathcal{U}^{\leq k} \phi_2 \mid \phi_1 \mathcal{U} \phi_2\end{aligned}$$

dove  $a$  è una proposizione atomica,  $\bowtie \in \{\leq, <, \geq, >\}$ ,  $p \in [0, 1]$  e  $k \in \mathbb{N}$ .

Dalla sintassi distinguiamo due tipi di formula: le formule di stato  $\phi$  e le formule di cammino  $\psi$ . Per definire formalmente la semantica è necessario specificare come gli elementi dell'insieme AP delle proposizioni atomiche sono gestiti in una MDP.

**Definizione 31 (MDP etichettata)** *Una MDP etichettata è una tupla  $(\mathcal{M}, L)$  dove:*

- $\mathcal{M}$  è una MDP;
- $L : S \rightarrow 2^{\text{AP}}$  è una funzione di etichettatura.

Estendiamo quindi la struttura delle MDP con una funzione  $L$  che associa ad ogni stato un certo insieme di proposizioni atomiche. A questo punto abbiamo tutti gli strumenti per definire la semantica di PCTL secondo una relazione di soddisfacibilità.

**Definizione 32 (Relazione di soddisfacibilità)** *Sia  $\mathcal{M} = (S, \bar{s}, \text{Act}, \text{Steps}, L)$  una MDP etichettata. Per ogni stato  $s \in S$ , la relazione di soddisfacibilità di stato  $\models$  è definita per induzione come segue:*

$$\begin{aligned}\mathcal{M}, s &\models \phi && \Leftrightarrow \mathcal{M}, s \models_{\text{st}} \phi \\ \mathcal{M}, s &\models_{\text{st}} \text{true} && \forall s \in S, \\ \mathcal{M}, s &\models_{\text{st}} a && \Leftrightarrow a \in L(s), \\ \mathcal{M}, s &\models_{\text{st}} \neg \phi && \Leftrightarrow s \not\models_{\text{st}} \phi, \\ \mathcal{M}, s &\models_{\text{st}} \phi_1 \wedge \phi_2 && \Leftrightarrow s \models_{\text{st}} \phi_1 \text{ e } s \models_{\text{st}} \phi_2, \\ \mathcal{M}, s &\models_{\text{st}} \mathbb{P}\{\mathcal{P}_{\bowtie p}[\psi]\} && \Leftrightarrow p_s^\Sigma(\psi) \bowtie p, \forall \Sigma \in \text{Scheduler}_{\mathcal{M}},\end{aligned}$$

dove per ogni scheduler  $\Sigma \in \text{Scheduler}_{\mathcal{M}}$ :

$$p_s^\Sigma \triangleq \text{Prob}_s^\Sigma(\{\pi \in \text{Path}_s^\Sigma \mid \pi \models_{\text{pt}} \psi\})$$

e per ogni cammino  $\pi \in \text{Path}$ :

$$\begin{aligned}\mathcal{M}, \pi &\models_{\text{pt}} \mathcal{X}\phi && \Leftrightarrow \mathcal{M}, \pi(1) \models_{\text{pt}} \phi, \\ \mathcal{M}, \pi &\models_{\text{pt}} \phi_1 \mathcal{U}^{\leq k} \phi_2 && \Leftrightarrow \exists i \leq k. (\mathcal{M}, \pi(i) \models_{\text{pt}} \phi_2 \text{ e } \mathcal{M}, \pi(j) \models_{\text{pt}} \phi_1 \forall j < i), \\ \mathcal{M}, \pi &\models_{\text{pt}} \phi_1 \mathcal{U} \phi_2 && \Leftrightarrow \exists k \geq 0. \mathcal{M}, \pi \models_{\text{pt}} \phi_1 \mathcal{U}^{\leq k} \phi_2.\end{aligned}$$

Per specificare una formula *PCTL* si utilizza sempre una formula di stato che al suo interno potrà utilizzare formule di cammino. Intuitivamente gli operatori logici possono essere utilizzati per indagare sulle proposizioni atomiche contenute in un determinato stato, mentre l'operatore  $\mathcal{P}_{\bowtie p}[\psi]$  viene soddisfatto dagli stati che soddisfano la formula di cammino  $\psi$  con una probabilità nell'intervallo specificato da  $\bowtie p$ . Questo operatore si comporta sempre nel modo appena descritto se i cammini non incontrano scelte nondeterministiche (nelle *DTMC* è sempre vero) ma come deve comportarsi in caso contrario? Dato che non si può assumere niente sulle suddette scelte e sfruttando il fatto che l'applicazione di uno scheduler ad una *MDP* produce una *DTMC*, l'operatore è considerato soddisfatto se la condizione è valida *per qualsiasi scheduler*.

Per quanto riguarda le formule di cammino, l'operatore *next*  $\mathcal{X}\phi$  richiede che  $\phi$  venga soddisfatta dallo stato successivo, l'operatore *bounded until*  $\phi_1 \mathcal{U}^{\leq k} \phi_2$  richiede che  $\phi_2$  venga soddisfatto entro  $k$  passi e che  $\phi_1$  resti sempre soddisfatto fino a quel punto, mentre l'operatore *unbounded until*  $\phi_1 \mathcal{U} \phi_2$  richiede che prima o poi  $\phi_2$  venga soddisfatto e che  $\phi_1$  sia sempre soddisfatto fino a quel punto.

Possiamo rielaborare i costrutti principali definiti finora per estendere il linguaggio con degli operatori derivati (tabella 1). Mentre gli operatori logici false,  $\vee$  e  $\rightarrow$  possono essere derivati facilmente, vi sono alcuni operatori meno banali. Gli operatori  $\diamond$  e  $\square$  sono molto comuni nelle logiche temporali e servono a specificare rispettivamente proprietà che hanno speranza di avverarsi in futuro e proprietà che sicuramente non si verificheranno mai. Delle applicazioni interessanti di questi due concetti sono le proprietà di *liveness* e di *safety*. Una proprietà di *liveness* esprime la possibilità che prima o poi accada qualcosa di positivo mentre la duale *safety* indica che qualcosa di negativo non potrà mai accadere. I due operatori possono essere usati per descrivere queste due tipologie proprietà ma sono più generali. Le varianti *bounded*  $\diamond^{\leq k}$  e  $\square^{\leq k}$  stabiliscono, tramite il numero di passi  $k \in \mathbb{N}$ , il tempo entro il quale la proprietà  $\phi$  deve rimanere soddisfatta a partire dall'istante iniziale. La proprietà di cammino  $\diamond^{\leq k} \phi$  sarà quindi soddisfatta se entro  $k$  passi  $\phi$  si verificherà almeno una volta, mentre la proprietà di cammino  $\square^{\leq k} \phi$  sarà soddisfatta se  $\phi$  rimarrà sempre soddisfatta per tutti e  $k$  i passi. Per definire gli operatori  $\square$  e  $\square^{\leq k}$  viene introdotta la notazione  $\boxtimes$  che rappresenta l'inversione degli operatori secondo le seguenti equivalenze:  $\overline{\leq} \equiv \geq$ ,  $\overline{<} \equiv >$ ,  $\overline{\geq} \equiv \leq$  e  $\overline{>} \equiv <$ . Un ultimo operatore interessante è il quantificatore esistenziale  $\exists$  che *ricerca l'esistenza di uno scheduler* che soddisfa una certa formula, contrariamente all'approccio generale basato sulla soddisfacibilità di tutti i possibili scheduler.

### 3.3 PRISM MODEL CHECKER

Il model checker probabilistico che utilizzeremo è *PRISM* [KNP11]. Si tratta di un tool con il quale è possibile fare modellazione e analisi di sistemi che pre-

$$\begin{aligned}
\text{false} &\equiv \neg \text{true} \\
\phi_1 \vee \phi_2 &\equiv \neg(\neg\phi_1 \wedge \neg\phi_2) \\
\phi_1 \rightarrow \phi_2 &\equiv \neg\phi_1 \vee \phi_2 \\
\mathcal{P}_{\bowtie p}[\Diamond \phi] &\equiv \mathcal{P}_{\bowtie p}[\text{true} \mathcal{U} \phi] \\
\mathcal{P}_{\bowtie p}[\Diamond \leq^k \phi] &\equiv \mathcal{P}_{\bowtie p}[\text{true} \mathcal{U}^{\leq k} \phi] \\
\mathcal{P}_{\bowtie p}[\Box \phi] &\equiv \mathcal{P}_{\bowtie 1-p}[\Diamond \neg \phi] \\
\mathcal{P}_{\bowtie p}[\Box \leq^k \phi] &\equiv \mathcal{P}_{\bowtie 1-p}[\Diamond \leq^k \neg \phi] \\
\exists \Diamond \phi &\equiv \neg \mathcal{P}_{\leq 0}[\Diamond \phi]
\end{aligned}$$

Tabella 1.: Operatori derivati di *PCTL*

sentano aspetti probabilistici. I principali modelli probabilistici supportati sono le *DTMC*, le *MDP*, le *Continuous-Time Markov Chain (CTMC)* ed i *Probabilistic Timed Automata (PTA)*, mentre i linguaggi utilizzabili per descrivere le formule sono *PCTL*, *Continuous Stochastic Logic (CSL)* e *Linear Time Logic (LTL)*. In questa sezione descriveremo l'utilizzo di *PRISM* concentrandoci sugli strumenti di interesse diretto per questo lavoro come le *MDP* e il linguaggio *PCTL* precedentemente descritti.

*PRISM* utilizza un suo specifico linguaggio con il quale è possibile definire tutti i modelli sopra citati. Si tratta infatti di un linguaggio basato sugli stati e su formalismi tipici dei sistemi reattivi. I componenti fondamentali rappresentati dal linguaggio sono i *moduli* e le *variabili*, un *modello* viene rappresentato come un insieme di moduli che possono interagire tra loro. Ogni modulo può inoltre contenere delle variabili locali e il valore di queste variabili ad un certo istante rappresentano il suo stato. Allo stesso modo lo stato del modello globale è definito come lo stato locale di tutti i moduli che lo compongono.

Il comportamento di un modulo viene definito dall'insieme di *comandi* che contiene. Un comando ha la seguente forma:

```
[ ] guardia -> prob_1 : update_1 + ... + prob_n : update_n ;
```

La *guardia* è un predicato che può considerare qualsiasi variabile del modello e se risulta soddisfatta il modello può avanzare eseguendo una delle transizioni *update* con le rispettive probabilità (o eventualmente rate). L'intero modulo viene definito specificandone il nome e il contenuto

```
module name ... endmodule
```

All'interno del modulo possono essere presenti sia comandi che variabili. Al fine di avere uno *spazio finito di stati* *PRISM* permette di gestire solo variabili booleane e variabili intere in un range finito. Di seguito mostriamo la dichiarazione di una variabile *x* che può assumere i valori interi compresi tra 0 e 10 compresi inizializzata a 5 e una variabile booleana *b* inizializzata a *true*:

```
x : [0..10] init 5;
b : bool init true;
```

Le guardie sono classiche espressioni booleane che possono fare riferimento a variabili di qualsiasi modulo in quanto viene richiesta solamente la lettura dei valori. Gli update, invece, sono sequenze di assegnamenti intervallate dal separatore &. La sequenza di update vuota viene indicata con true. Un assegnamento di un valore a una variabile può avvenire solo su una variabile locale al modulo a cui appartiene il comando, inoltre il nome della variabile aggiornata deve terminare col simbolo ' ad indicarne il nuovo stato. All'interno del listato 3.1 mostriamo l'utilizzo degli strumenti appena descritti. L'esempio rappresenta due processi in mutua esclusione.

Listati 3.1: Esempio di definizione dei moduli in *PRISM*

```
1 module M1
2
3 x : [0..2] init 0;
4
5 [] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
6 [] x=1 & y!=2 -> (x'=2);
7 [] x=2 -> 0.5:(x'=2) + 0.5:(x'=0);
8
9 endmodule
10
11 module M2
12
13 y : [0..2] init 0;
14
15 [] y=0 -> 0.8:(y'=0) + 0.2:(y'=1);
16 [] y=1 & x!=2 -> (y'=2);
17 [] y=2 -> 0.5:(y'=2) + 0.5:(y'=0);
18
19 endmodule
```

In questo caso i due moduli sono simmetrici, ed entrambi presentano esclusivamente scelte probabilistiche. Possiamo se necessario inserire anche delle scelte non deterministiche sulle quali è quindi impossibile fare osservazioni di tipo quantitativo. Modificando il modulo M1 come mostrato nel listato 3.2 inseriamo una scelta nondeterministica, infatti le due nuove guardie inserite saranno sempre abilitate insieme.

Listati 3.2: Scelta nondeterministica in *PRISM*

```
1 module M1
2
3 x : [0..2] init 0;
4
5 // scelta nondeterministica
```



```

6      [] x=0 -> (x'=0);
7      [] x=0 -> (x'=1);
8
9      [] x=1 & y!=2 -> (x'=2);
10     [] x=2 -> 0.5:(x'=2) + 0.5:(x'=0);
11
12 endmodule

```

In generale il nondeterminismo può verificarsi anche se due guardie vengono soddisfatte assieme anche solo parzialmente. La stessa struttura può quindi presentare scelte deterministiche a seconda dello stato corrente.

In *PRISM* è anche possibile definire delle costanti globali sui domini di interi, reali e booleani:

```

const int z = 12;
const double pi = 3.141592;
const bool flag = true;

```

Queste costanti possono essere utilizzate, al fine di parametrizzare il modello, all'interno delle espressioni che possono coinvolgere i seguenti operatori:

- – meno unario, \* moltiplicazione, / divisione, + somma e – sottrazione,
- le relazioni d'ordine  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ , e di equivalenza  $=$  e  $!=$
- gli operatori booleani di ! negazione, & congiunzione, | disgiunzione,  $\leq$  se e solo se e  $\Rightarrow$  implicazione logica.

Sono presenti inoltre le funzioni:

- min minimo e max massimo,
- floor di arrotondamento all'intero minore più vicino,
- ceil di arrotondamento all'intero maggiore più vicino,
- pow per l'elevamento a potenza,
- mod modulo,
- log logaritmo.

Le espressioni possono essere utilizzate non solo nelle condizioni delle guardie, nelle inizializzazioni e negli aggiornamenti, ma anche per calcolare le proprietà di una transizione.

Come primo parametro di un comando può essere inserita la label di un'azione per far sincronizzare i moduli. Un azione viene quindi specificata inserendone il nome (ad esempio *step*) all'interno delle parentesi quadre, nel seguente modo

```
[step] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
```

Il modello finale viene ricavato di default dalla *composizione parallela* di tutti i moduli che lo compongono, se si vuole invece descrivere una composizione diversa, è possibile usare il costrutto

```
system ... endsystem
```

Al suo interno possono essere usati i seguenti operatori Communicating Sequential Processes (CSP):

- $M1 \parallel M2$ : i moduli si sincronizzano solo su azioni che appaiono in entrambi (composizione parallela di default),
- $M1 ||| M2$ : interleaving completo, senza sincronizzazione,
- $M1 [[a, b, \dots]] M2$ : la sincronizzazione avviene solo sulle azioni specificate nell'operatore,
- $M/\{a, b, \dots\}$  le azioni specificate vengono nascoste all'esterno,
- $M\{a < -b, c < -d, \dots\}$  le azioni vengono rinominate all'esterno.

Alcuni esempi di espressioni di sistema sono mostrati di seguito:

```
system (station1 ||| station2 ||| station3) |[serve]| server endsystem
system ((P1 |[a]| P2) / {a}) |[b]| Q endsystem
system ((P1 |[a]| P2) {a<-b}) |[b]| Q endsystem
```

## Parte II

### LAPSA: UN LINGUAGGIO PER AGENTI ADATTIVI



## LAPSA: LANGUAGE FOR POPULATION OF SELF-ADAPTIVE AGENTS

(LV) lapsa /'lapsa/ : (IT) volpe

Per definire ed implementare un modello di un sistema adattivo possono essere utilizzati molti strumenti già esistenti. L'approccio che vogliamo impiegare coinvolge l'utilizzo di un model checker come supporto alle scelte che possiamo semplicemente scegliere a seconda delle specifiche e delle esigenze dello scenario.

Prendiamo come punto di riferimento, quindi, l'unico elemento sicuro che abbiamo a disposizione: l'agente principale, cioè quello per il quale deve essere trovata una strategia che lo porti al suo obiettivo. Quello che ci si può immaginare è che questo agente proceda in parallelo con la parte che non conosce: l'ambiente (figura 6). L'ambiente può essere costituito da più agenti o altri tipi di sistemi di cui si può avere una conoscenza solamente parziale o nulla.

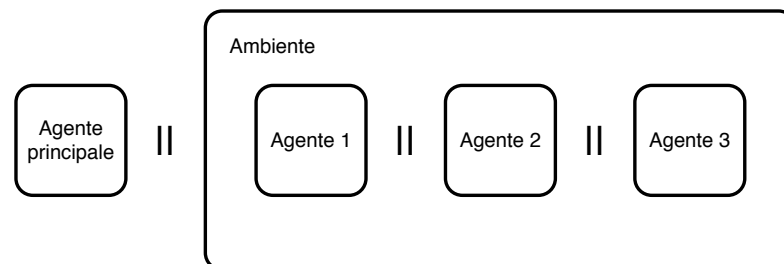


Figura 6.: Schema di come viene modellato uno scenario attorno all'agente che lo analizza

L'approccio che vogliamo utilizzare è quello di *formulare un'ipotesi* sulla composizione ed il comportamento dell'ambiente ed utilizzare il model checker su questo modello verificando la formula che rappresenta l'obiettivo dell'agente principale. Cambiamo quindi la prospettiva classica del model checking fornendo un modello ipotetico per fare *previsioni*. Quello che vogliamo che l'agente finale abbia alla fine di questo procedimento è una valutazione quantitativa di quanto ogni scelta ha la possibilità di portarlo al raggiungimento dell'obiettivo.

In questo capitolo viene introdotta la definizione del linguaggio *LAPSA*, un linguaggio specifico per agenti adattivi. L'obiettivo che si vuole raggiungere con questo linguaggio è definire un'interfaccia che permetta di modellare sistemi

adattivi in modo più efficiente ed efficace possibile. Si parla di interfaccia in quanto, attraverso la definizione di sintassi e semantica, si dichiara *cosa* possono fare i costrutti linguistici di *LAPSA* (front-end) senza entrare in merito del *come* questo verrà implementato (back-end).

Il back-end è stato implementato in *JAVA* tramite *Xtext* [xteb], un meta-tool specifico per la creazione di plugin *ECLIPSE* di linguaggi personalizzati. Definendo la grammatica del proprio linguaggio è possibile implementarne velocemente la traduzione del codice ed i servizi di utilità più diffusi come l'autocompletamento e la colorazione delle parole chiave. Oltre al compilatore è presente anche il model checker *PRISM* in grado di eseguire i controlli di formule *PCTL* su *MDP* definite secondo il suo specifico linguaggio.

L'approccio utilizzato separa il linguaggio dal livello implementativo permettendo di cambiare gli strumenti sottostanti senza che l'utilizzatore debba venirne a conoscenza, a patto che i nuovi strumenti rispettino la semantica dei costrutti linguistici.

#### 4.1 SINTASSI

Per descrivere la sintassi *LAPSA* sarà utilizzato il formalismo *EBNF* indicando con le parole in corsivo i simboli non terminali e con quelle in neretto e quelle in stampatello i terminali. Le parole in neretto sono *keyword* del linguaggio mentre quelle in stampatello descrivono dei valori arbitrari su insiemi come nomi di variabili o costanti numeriche. Procederemo con la descrizione informale di cosa viene identificato con i costrutti sintattici del linguaggio introducendoli gradualmente.

Il non terminale *program* è il simbolo iniziale della grammatica e nella sua struttura racchiude la dichiarazione dell'insieme di azioni considerato, il modulo che descrive il comportamento del soggetto, i moduli che possono essere usati per descrivere l'ambiente e i dati necessari alla discretizzazione delle variabili.

$$\begin{aligned} \textit{program} ::= & \textbf{actions} \{ \textit{actions} \} \\ & \textbf{subject} \textit{module} \\ & \textit{modules} \\ & \textit{environment} \\ & \textbf{ranges} \{ \textit{ranges} \} \end{aligned}$$

Il non terminale *actions* è una semplice lista dove vengono dichiarate i nomi delle azioni che possono essere effettuate. Ovviamente le azioni utilizzate in seguito nelle transizioni dei moduli e nelle sincronizzazioni dovranno essere state dichiarate in questa sezione per considerare il programma *corretto*.

$$\textit{actions} ::= \text{action-id} \mid \textit{actions} \textit{actions}$$

La definizione di un comportamento viene espressa tramite il non terminale *module* che permette di descrivere i suoi dati, le sue transizioni e i suoi obiettivi. I dati sono rappresentati dalla lista *variables* e ogni dato è rappresentato dal tipo di dato, il nome associato e l'espressione che gli attribuisce un valore iniziale. Con il non terminale *rules*, invece, viene descritta una lista di transizioni. Le transizioni vengono definite come la tripla *condizione, azione, distribuzione*: se la condizione è vera allora può essere effettuata l'azione e l'aggiornamento dello stato secondo la distribuzione di probabilità. Gli obiettivi vengono descritti nella lista *targets* dove il primo criterio ha importanza massima e decrementa fino all'ultimo che sarà il meno importante.

$$\text{module} ::= \mathbf{module} \text{ module-id } \{ \text{variables rules targets} \}$$

$$\text{variables} ::= \text{type variable-id} = \text{expression}; \quad | \quad \text{variables variables}$$

$$\text{rules} ::= \text{condition}[\text{action-id}] \Rightarrow \text{distribution}; \quad | \quad \text{rules rules}$$

$$\text{targets} ::= \mathbf{target never} \text{ condition} \quad | \quad \text{targets targets}$$

Le distribuzioni di probabilità sono definite come un insieme di possibili aggiornamenti dello stato associati ad un certo valore di probabilità. Questo valore viene espresso nel costrutto sintattico dall'espressione all'interno delle parentesi angolate e viene normalizzato con gli altri nel caso in cui la somma non sia 1. Questi valori possono anche dipendere dalle variabili di stato e quindi variare con l'avanzamento del modello.

$$\text{distribution} ::= < \text{expression} > \text{update} \quad | \quad \text{distribution} \# \text{distribution}$$

Ogni caso di una distribuzione porta a una lista di aggiornamenti che possono essere descritti con i costrutti sintattici definiti come segue

$$\begin{aligned} \text{update} ::= & \text{variable-id} = \text{expression} \quad | \quad \mathbf{noaction} \\ & | \quad \text{update}, \text{update} \end{aligned}$$

Si può modificare il valore di una variabile tramite assegnamento o specificare che nessun cambiamento verrà eseguito tramite l'operazione nulla *noaction*.

Il non terminale *modules* viene utilizzato per definire un insieme di moduli che poi potranno essere istanziati nell'ambiente con dei riferimenti al loro nome. Questo accade all'interno del non terminale *environment* dove viene descritto l'ambiente come una lista di riferimenti a moduli intervallati dagli insiemi di azioni su cui possono sincronizzarsi, un operatore che riprende sintatticamente e semanticamente dal parallelo di Hoare in CSP [Hoa78].

$$\text{modules} ::= \text{module} \quad | \quad \text{modules modules}$$

$$\text{environment} ::= \text{module-id} \quad | \quad \text{environment } \{ \text{actions} \} \text{ environment}$$

Le condizioni hanno un ruolo importante in quanto parte fondamentale dell'abilitazione delle transizioni. La sintassi di una condizione mette a disposizione, oltre ai costrutti sintattici più classici come gli operatori booleani e il confronto tra espressioni, anche il quantificatore esistenziale. Questo operatore viene considerato soddisfatto se esiste un modulo del tipo specificato che soddisfa la condizione espressa. La condizione del quantificatore esistenziale può fare riferimento al nome temporaneo associato al tipo di modulo interessato in modo da poter indagare sulle sue variabili di stato.

$$\begin{aligned} \text{condition} ::= & \text{exists variable-id : module-id such that condition} \\ & | \text{expression} \bowtie \text{expression} \quad | \quad \text{true} \\ & | \text{condition or condition} \quad | \quad \text{not condition} \\ & | (\text{condition}) \end{aligned}$$

Il simbolo  $\bowtie$  rappresenta l'operatore di confronto e può essere quindi definito come  $\bowtie = \{<, \leq, >, \geq, =, \neq\}$ . Le espressioni sono fondamentalmente variabili e costanti numeriche combinate tra loro tramite i classici operatori binari e gli operatori di confronto seguono l'interpretazione classica. Le variabili referenziate devono essere precedute dall'istanza di appartenenza: se fanno riferimento al modulo nel quale vengono richiamate si utilizza la keyword *this*, altrimenti il riferimento al modulo interessato nel caso in cui si stia esprimendo una condizione interna ad un quantificatore esistenziale.

$$\begin{aligned} \text{expression} ::= & \text{expression bop expression} \quad | \quad \text{reference} \\ & | (\text{expression}) \quad | \quad \text{constant} \\ \text{reference} ::= & \text{variable-id.variable-id} \quad | \quad \text{this.variable-id} \\ \text{bop} ::= & + \quad | \quad - \quad | \quad * \quad | \quad / \end{aligned}$$

Infine il non terminale *ranges* permette di descrivere i range di tutte le variabili dei moduli. Questo costrutto è stato inserito in aggiunta alla normale interfaccia di *LAPSA* per adattarne l'implementazione del backend a *PRISM* che necessita la conoscenza dei possibili valori delle variabili.

$$\text{ranges} ::= \text{module-id.variable-id in[constant, constant]} \quad | \quad \text{ranges, ranges}$$

In tabella 2 viene riportata la sintassi di *LAPSA*.

## 4.2 ZUCCHERO SINTATTICO

La sintassi presentata in tabella 2 è concreta ma contiene solo gli operatori primitivi ed è dunque solo il nucleo del linguaggio che si vuole utilizzare. Al fine di aumentare la semplicità di comprensione e di scrittura del linguaggio



<i>program</i>	::=	<b>actions</b> { <i>actions</i> }
		<b>subject</b> <i>module</i>
		<i>modules</i>
		<i>environment</i>
		<b>ranges</b> { <i>ranges</i> }
<i>actions</i>	::=	action-id   <i>actions actions</i>
<i>module</i>	::=	<b>module</b> module-id { <i>variables rules targets</i> }
<i>variables</i>	::=	type variable-id = <i>expression</i> ;   <i>variables variables</i>
<i>rules</i>	::=	condition[action-id] $\Rightarrow$ <i>distribution</i> ;   <i>rules rules</i>
<i>targets</i>	::=	<b>target never</b> condition   <i>targets targets</i>
<i>distribution</i>	::=	< <i>expression</i> > <i>update</i>   <i>distribution</i> # <i>distribution</i>
<i>update</i>	::=	variable-id = <i>expression</i>   <b>noaction</b>
		<i>update</i> , <i>update</i>
<i>modules</i>	::=	<i>module</i>   <i>modules modules</i>
<i>environment</i>	::=	module-id   <i>environment</i> [{ <i>actions</i> }] <i>environment</i>
<i>condition</i>	::=	<b>exists</b> variable-id : module-id <b>such that</b> <i>condition</i>
		<i>expression</i> $\bowtie$ <i>expression</i>   <b>true</b>
		<i>condition</i> <b>or</b> <i>condition</i>   <b>not</b> <i>condition</i>
		( <i>condition</i> )
<i>expression</i>	::=	<i>expression</i> <i>bop</i> <i>expression</i>   <i>reference</i>
		( <i>expression</i> )   <b>constant</b>
<i>reference</i>	::=	variable-id.variable-id   <b>this</b> .variable-id
<i>bop</i>	::=	+   -   *   /
<i>ranges</i>	::=	module-id.variable-id <b>in</b> [constant, constant]   <i>ranges</i> , <i>ranges</i>

Tabella 2.: Sintassi di LAPSA

*LAPSA* intrudiciamo alcuni costrutti di utilità come una rielaborazione di quelli primitivi già presenti.

Con gli operatori logici possiamo ridefinire i seguenti costrutti all'interno del non terminale *condition*, assumendo  $\alpha$  come un variable-id,  $\gamma$  come un module-id e  $\beta, \beta' \in condition$

**false**  $\equiv$  **not true**

$\beta$  **and**  $\beta' \equiv$  **not (not  $\beta$  or not  $\beta'$ )**

**forall**  $\alpha : \gamma$  **such that**  $\beta \equiv$  **not exists**  $\alpha : \gamma$  **such that not**  $\beta$

Inseriamo un costrutto di *rule* tale da poter inserire una condizione di abilitazione dei singoli casi delle distribuzioni, assumendo  $\alpha$  come un action-id,  $\beta, \beta' \in condition$ ,  $\delta \in distribution$  ed  $\epsilon \in expression$

$\beta[\alpha] \Rightarrow < \epsilon, \beta' > \alpha \# \delta \equiv \beta$  **and**  $\beta'[\alpha] \Rightarrow < \epsilon > \alpha \# \delta, \beta$  **and not**  $\beta'[\alpha] \Rightarrow \delta$

Dato che in *LAPSA* è presente il non determinismo se più guardie sono abilitate allo stesso passo, introduciamo la possibilità di scrivere più distribuzioni con una sola guardia che vale per tutte le regole. Sia  $\alpha$  un action-id,  $\beta \in condition$  e  $\delta, \delta' \in distribution$ , introduciamo il seguente costrutto del non terminale *rule*

$\beta[\alpha] \Rightarrow \delta \Rightarrow \delta'; \equiv \beta[\alpha] \Rightarrow \delta; \beta[\alpha] \Rightarrow \delta';$

Possiamo ottenere con facilità il costrutto che esprime un obiettivo che vogliamo mantenere durante tutta l'esecuzione:

**target always** *condition*  $\equiv$  **target never not** *condition*

Infine, per quanto riguarda i riferimenti a variabile, è possibile assumere in assenza del prefisso di appartenenza che la variabile appartenga al modulo locale di default:

variable-id  $\equiv$  **this**.variable-id

In tabella 3 vengono riportati gli operatori derivati di *LAPSA* appena descritti.

#### 4.3 SEMANTICA

Durante la descrizione della sintassi sono stati presentati informalmente i costrutti di *LAPSA*, passiamo adesso a dare una definizione formale della loro semantica. Rappresenteremo il significato di un programma *LAPSA* tramite una *MDP* perché permette di esprimere le transizioni tra stati come scelte nondeterministiche di distribuzioni di probabilità aventi come supporto l'insieme degli stati stessi.

Ad ogni nonterminale *module* sarà associata una *MDP* della forma

$$M = (\Sigma, Act, \rightarrow_p, \sigma_0)$$

<b>false</b> $\equiv$ <b>not true</b>
$\beta$ <b>and</b> $\beta' \equiv$ <b>not (not <math>\beta</math> or not <math>\beta'</math>)</b>
<b>forall</b> $\alpha : \gamma$ <b>such that</b> $\beta \equiv$ <b>not exists</b> $\alpha : \gamma$ <b>such that not</b> $\beta$
$\beta[\alpha] \Rightarrow \langle \epsilon, \beta' \rangle \alpha \# \delta \equiv \beta$ <b>and</b> $\beta'[\alpha] \Rightarrow \langle \epsilon \rangle \alpha \# \delta, \beta$ <b>and not</b> $\beta'[\alpha] \Rightarrow \delta$
$\beta[\alpha] \Rightarrow \delta \Rightarrow \delta'; \equiv \beta[\alpha] \Rightarrow \delta; \beta[\alpha] \Rightarrow \delta';$
<b>target always condition</b> $\equiv$ <b>target never not condition</b>
variable-id $\equiv$ <b>this</b> .variable-id

Tabella 3.: Operatori derivati di *LAPSA*

Le parti che compongono la *MDP* sono descritte in seguito, tenendo conto che ogni riferimento ai nodi della sintassi viene inteso come appartenente al modulo in questione e, per semplicità, utilizziamo la notazione  $eval(e)$  con  $e \in expression$  per indicare la valutazione di un'espressione nel modo classico.

- $\Sigma = \{\sigma \mid \sigma : \mathbb{VAR} \rightarrow \mathbb{VAL}\}$  è l'insieme degli *stati* rappresentati da funzioni che mappano variabili in valori, dove  $\mathbb{VAR}$  è l'insieme delle variable-id definite nel modulo e  $\mathbb{VAL} \subset \mathbb{N}_0$  di cardinalità finita
- $\sigma_0 \in \Sigma$  è lo *stato iniziale* del modulo ottenuto tramite la valutazione delle espressioni di dichiarazione,

$$\sigma_0(v) = eval(e)$$

dove "type  $v = e$ "  $\in$  *variables*

- $Act$  è l'insieme delle azioni action-id
- $\rho \subseteq condition \times Act \times Dist(\mathbb{U})$  è la *struttura statica* della *MDP* definita come

$$\begin{aligned} \rho &= \{(g, a, d) \mid "g[a] \Rightarrow \langle e_1 \rangle \alpha_1 \# \dots \# \langle e_n \rangle \alpha_n" \in rule, \\ &\quad d = [u_{\alpha_1} : p_1, \dots, u_{\alpha_n} : p_n]\} \end{aligned}$$

dove, per  $i = 1, \dots, n$ , valgono  $\alpha_i \in update$ ,  $e_i \in expression$  e la normalizzazione delle probabilità

$$p_i = \frac{eval(e_i)}{\sum_{j=1}^n eval(e_j)}$$

- $U = \{u \mid u : \text{update} \times \Sigma \rightarrow \Sigma\}$  è l'insieme delle funzioni di aggiornamento di stato definite come

$$u_\alpha(\sigma) = \begin{cases} \sigma[\text{eval}(e)/x] & \text{se } \alpha = "x = e" \\ \sigma & \text{se } \alpha = \text{"noaction"} \\ u_{\alpha_2}(u_{\alpha_1}(\sigma)) & \text{se } \alpha = \alpha_1 \alpha_2 \end{cases}$$

dove  $\alpha, \alpha_1, \alpha_2 \in \text{update}$ ,  $\sigma \in \Sigma$ ,  $x \in \text{VAR}$  ed  $e \in \text{expression}$

- $\rightarrow_\rho \subseteq \Sigma \times \text{Act} \times \text{Dist}(U)$  è la relazione di *avanzamento*: questa relazione descrive come evolve lo stato del modulo col passare del tempo, e viene descritto dalla seguente regola di inferenza

$$\frac{(g, a, d) \in \rho}{\sigma \xrightarrow{a}_\rho d(\sigma)} \sigma \models g \quad (\text{Update 1})$$

Salendo dal livello dei moduli a quello del sistema globale inteso come la composizione parallela del modulo *subject* con l'ambiente, introduciamo  $\Pi \in \text{Dist}(S)$  per indicare distribuzioni di sistemi. Un sistema  $S$  è un insieme che contiene tutti i moduli riferiti in *environment* (eventualmente anche con più istanze dello stesso modulo) e il modulo principale *subject*. Consideriamo quindi per semplicità la semantica definita su  $S$  come nonterminale fittizio

$$S ::= \text{module-id} \quad | \quad S_1 \{ \{ \text{actions} \} \} S_2$$

Possiamo utilizzare  $S$  per descrivere il come *LAPSA* gestisce la composizione del modulo principale con quelli dell'ambiente.

$$\text{subject-id} \{ \{ \text{Act} \} \} \text{environment}$$

Le regole di inferenza riportate di seguito descrivono la semantica di  $S$  in quanto più generale e comprensibile, dalla quale si può facilmente derivare il comportamento specifico. Useremo le notazioni  $\sigma_m$  e  $\rho_m$  per indicare rispettivamente lo stato e la struttura statica del generico modulo  $m \in \text{module}$  e l'insieme  $A \subseteq \text{Act}$ .

Con la prima regola viene descritto l'avanzamento dello stato di un modulo nel tempo a livello del sistema che lo contiene

$$\frac{\sigma_m \xrightarrow{a}_{\rho_m} d(\sigma_m)}{S \xrightarrow{a} \Pi} m \in S \quad (\text{Update 2})$$

Con le seguenti tre regole viene descritta la sincronizzazione tra sistemi che possono effettuare la stessa azione se questa è contenuta nell'insieme  $A$ . Nel

caso in cui l'azione non sia contenuta nell'insieme  $A$  l'avanzamento avviene comunque ma senza sincronizzazione.

$$\frac{S_1 \xrightarrow{a} \Pi_1 \quad S_2 \xrightarrow{a} \Pi_2}{S_1 \parallel \{A\} S_2 \xrightarrow{a} \Pi_1 \parallel \{A\} \Pi_2} \quad a \in A \quad (\text{Sync})$$

$$\frac{S_1 \xrightarrow{a} \Pi_1}{S_1 \parallel \{A\} S_2 \xrightarrow{a} \Pi_1 \parallel \{A\} S_2} \quad a \notin A \quad (\text{Async 1})$$

$$\frac{S_2 \xrightarrow{a} \Pi_2}{S_1 \parallel \{A\} S_2 \xrightarrow{a} S_1 \parallel \{A\} \Pi_2} \quad a \notin A \quad (\text{Async 2})$$

Rimane da definire come si comporta l'operatore di composizione parallela tra un sistema e una distribuzione e tra due distribuzioni.

$$\Pi_1 \parallel \{A\} S_2(S) = \begin{cases} \Pi_1(S'_1) & \text{se } S = S'_1 \parallel \{A\} S_2 \\ 0 & \text{altrimenti} \end{cases}$$

$$S_1 \parallel \{A\} \Pi_2(S) = \begin{cases} \Pi_2(S'_2) & \text{se } S = S_1 \parallel \{A\} S'_2 \\ 0 & \text{altrimenti} \end{cases}$$

$$\Pi_1 \parallel \{A\} \Pi_2(S) = \begin{cases} \Pi_1(S_1) \cdot \Pi_2(S_2) & \text{se } S = S_1 \parallel \{A\} S_2 \\ 0 & \text{altrimenti} \end{cases}$$

Riassumiamo la semantica di *LAPSA* nella tabella 4.

$\frac{(g, a, d) \in \rho}{\sigma \xrightarrow{a}_\rho d(\sigma)} \quad \sigma \models g$	(Update 1)
$\frac{\sigma_m \xrightarrow{a}_{\rho_m} d(\sigma_m)}{S \xrightarrow{a} \Pi} \quad m \in S$	(Update 2)
$\frac{S_1 \xrightarrow{a} \Pi_1 \quad S_2 \xrightarrow{a} \Pi_2}{S_1 \parallel \{A\} S_2 \xrightarrow{a} \Pi_1 \parallel \{A\} \Pi_2} \quad a \in A$	(Sync)
$\frac{S_1 \xrightarrow{a} \Pi_1}{S_1 \parallel \{A\} S_2 \xrightarrow{a} \Pi_1 \parallel \{A\} S_2} \quad a \notin A$	(Async 1)
$\frac{S_2 \xrightarrow{a} \Pi_2}{S_1 \parallel \{A\} S_2 \xrightarrow{a} S_1 \parallel \{A\} \Pi_2} \quad a \notin A$	(Async 2)

Tabella 4.: Semantica di *LAPSA*

La cosa che manca per fornire un'interpretazione semantica completa di un programma *LAPSA* è la sezione *targets*. La semantica finale associa ad un programma *LAPSA* la coppia  $(M, \pi)$  dove  $M$  è la *MDP* che descrive il sistema

globale e  $\pi$  e la formula *PCTL* che descrive l'obiettivo del modulo principale. Pur potendo essere presenti in qualsiasi modulo, solo gli obiettivi del soggetto principale verranno presi in considerazione.

In tabella 5 viene data la semantica denotazionale tramite la definizione della funzione  $\tau : module \times module \times targets$  per l'obiettivo,  $\gamma : module \times module \times condition$  per le condizioni ed  $\epsilon : module \times module \times expression$  per le espressioni.

$\Phi_m^t(\mathbf{target\ never\ } c) = P_{max=?}[G_{\leq k}!''\tau_m^t(c)'']$
$\tau_m^t(\mathbf{exists\ var\ :\ mod\ such\ that\ } c) = \tau_{m_1}^t(c) \mathbf{or\ \dots\ or\ } \tau_{m_n}^t(c)$
$\tau_m^t(\mathbf{true}) = \mathbf{true}$
$\tau_m^t(c_1 \mathbf{or\ } c_2) = \tau_m^t(c_1) \mathbf{or\ } \tau_m^t(c_2)$
$\tau_m^t(\mathbf{not\ } c) = !\tau_m^t(c)$
$\tau_m^t(e_1 \bowtie e_2) = \epsilon_m^t(e_1)\tau_m^t(\bowtie)\epsilon_m^t(e_2)$

Tabella 5.: Semantica del target del modulo principale

Il primo parametro  $t$  indica il modulo di origine da cui parte la valutazione della condizione (nel caso specifico il modulo principale), mentre il secondo  $m$  indica il modulo che si considera per la risoluzione di riferimenti a variabili esterne. Questo diventa necessario quando si valuta il quantificatore esistenziale che ci porta ad indagare sugli stati degli altri moduli.

Sono state assunte due principali semplificazioni:

- è stato considerato un singolo *target*, la valutazione dei successivi si svolge nello stesso modo decrementando la priorità degli obiettivi successivi. Il secondo target sarà quindi interessante solamente nei casi in cui si sarà verificato un pareggio per il primo.
- Anche la valutazione delle espressioni, in particolare dei riferimenti a variabili, è stata omessa come semplificazione: sarà necessaria una struttura di appoggio dove mantenere i nomi delle variabili di riferimento indispensabili in caso di quantificatori esistenziali annidati.

Inoltre sono state usate le abbreviazioni  $c$  ed  $e$  rispettivamente per indicare *condition* ed *expression*.

#### 4.4 ESEMPI

Diamo alcuni esempi di moduli per semplificare la comprensione del linguaggio. Nel listato 4.1 riportiamo il modulo *LAPSA* di un robot che esegue una *random walk* su una griglia escludendo dalla scelta probabilistica le direzioni adiacenti occupate.

Listati 4.1: Esempio di random walk in *LAPSA*

```

1 subject module RandomWalkRobot {
2   // variabili
3   int x = 5;
4   int y = 5;
5
6   // transizioni
7   true [step] =>
8     <1, not exists bot:RandomWalkRobot such that bot.x=x and bot.y=y+1>
9       y=y+1 #
10    <1, not exists bot:RandomWalkRobot such that bot.x=x and bot.y=y-1>
11      y=y-1 #
12    <1, not exists bot:RandomWalkRobot such that bot.x=x+1 and bot.y=y>
13      x=x+1 #
14    <1, not exists bot:RandomWalkRobot such that bot.x=x-1 and bot.y=y>
15      x=x-1 #
16    <1, true> noaction;
17 }

```

Nel listato 4.2 viene riportato l'esempio di un modulo di robot analogo al precedente con la differenza che la scelta della mossa viene fatta in modo nondeterministico, spostando le condizioni dai casi della distribuzione alle guardie delle transizioni.

Listati 4.2: Versione nondeterministica della random walk in *LAPSA*

```

1 subject module NondeterministicRobot {
2   // variabili
3   int x = 5;
4   int y = 5;
5
6   // transizioni
7   not exist bot:RandomWalkBot such that bot.x=x and bot.y=y+1 [step] =>
8     <1> y=y+1;
9   not exist bot:RandomWalkBot such that bot.x=x and bot.y=y-1 [step] =>
10    <1> y=y-1;
11   not exist bot:RandomWalkBot such that bot.x=x+1 and bot.y=y [step] =>
12    <1> x=x+1;
13   not exist bot:RandomWalkBot such that bot.x=x-1 and bot.y=y [step] =>
14    <1> x=x-1;
15   true [step] => <1> noaction;
16 }

```

#### 4.5 DA LAPSA A PRISM

Il compilatore del linguaggio *LAPSA* esegue due fasi: nella prima traduce il modello *LAPSA* producendo un modello *PRISM* e un file di formule *PCTL* che

modellano gli obiettivi ricercati, nella seconda richiama il model checker di *PRISM* per eseguire la verifica della formula sul modello dati in input (figura 7). Quello che viene prodotto in output è una struttura dati contenente i risultati del model checking, in questo caso è una *hashtable* avente come chiave lo stato del modello e come dato la probabilità che si ha di passare da quello stato al raggiungimento dell'obiettivo. In *JAVA* è rappresentata dal seguente tipo:

```
Hashtable<List<Integer>,Double> results;
```

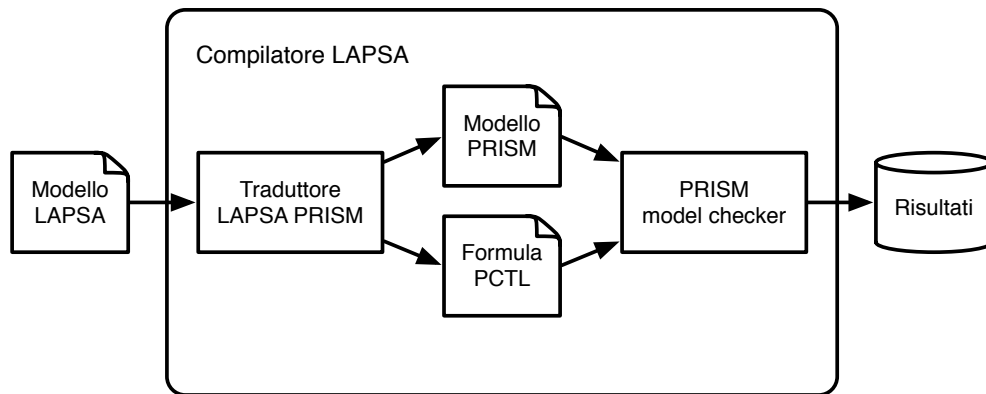


Figura 7.: Schema del compilatore *LAPSA*

Al fine di migliorare le prestazioni è stato pensato di strutturare i risultati in modo leggermente più complesso:

```
Hashtable<List<Integer>,Hashtable<List<Integer>,Double>> results;
```

La struttura è quindi una hashtable di hashtable di probabilità. Il primo livello ha come chiave la lista dei parametri che identifica lo stato del modulo principale mentre la seconda è la lista di parametri che identifica lo stato dei moduli che compongono l'ambiente. In questo modo i tempi di lettura rimangono costanti, ma risulta più semplice applicare algoritmi di ricerca del massimo dei minimi: generalmente si procede cercando il caso peggiore che si può verificare nell'ambiente, in termini di probabilità, e si indaga sulla scelta che porta al risultato migliore con una ricerca del massimo.

Come illustrato in figura 8 il compilatore *LAPSA*, la cui implementazione è riportata in appendice A, è stato pensato per effettuare la decisione delle scelte in una fase precedente alla effettiva esecuzione. Durante la fase di compilazione vengono forniti tramite la hashtable tutti i dati necessari per prendere le decisioni sul campo, quindi il software utilizzato dall'agente non dovrà fare altro che accedere ai dati e prendere la scelta desiderata secondo una politica che ovviamente dovrà concordare con quella espressa nel target del modulo *LAPSA*.



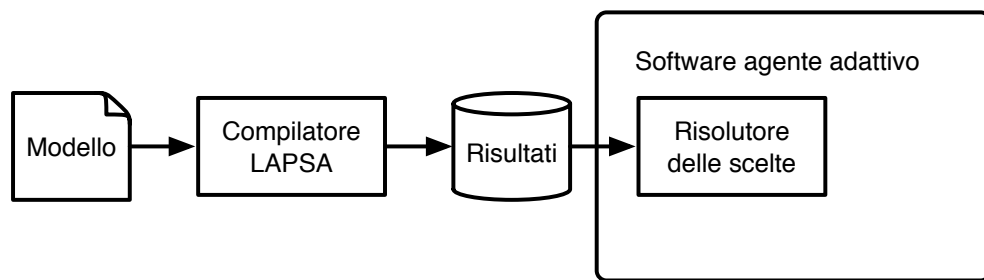


Figura 8.: Utilizzo dei risultati nell'agente adattivo



Parte III

CASO DI STUDIO



## CASO DI STUDIO

Lo scenario preso come caso di studio prevede una popolazione di agenti mobili che si muovono casualmente all'interno di un'area limitata. Gli agenti possono venire a conoscenza, tramite dei *sensori di prossimità*, della presenza di altri agenti entro un raggio limitato. Ogni singolo agente può decidere periodicamente se muoversi verso nord, sud, est o ovest o se stare fermo. L'obiettivo primario è quello di associare uno scheduler all'agente protagonista che minimizzi il numero di collisioni con altri agenti che si verificheranno.

In figura 9 viene riportata la rappresentazione dello scenario in un certo istante di tempo: gli eventi si succedono in un'arena  $5 \times 5$  chiusa tra quattro pareti, tre agenti mobili rossi che si muovono in modo casuale e l'agente protagonista verde. L'agente principale riesce a percepire agenti vicini nell'area rappresentata dal disco grigio, in questo caso riconosce la presenza di un agente in direzione ovest e un altro agente in direzione nord-est. Le posizioni locali che vengono monitorate dai sensori sono infatti nove: nord, sud, ovest, est, nord-est, nord-ovest, sud-est, sud-ovest e la posizione centrale, in caso di collisioni. La griglia tratteggiata rappresenta le strade percorribili dagli agenti, supporremo che ad ogni passo gli agenti si possano spostare tra incroci adiacenti. Questa astrazione a griglia dell'arena rappresenta un vantaggio ai fini della modellazione in quanto lo spazio degli stati è discreto. Considerando anche il tempo discreto e le scelte dell'agente principale nondeterministiche ci ritroviamo in un contesto facilmente riconducibile alle *MDP*.

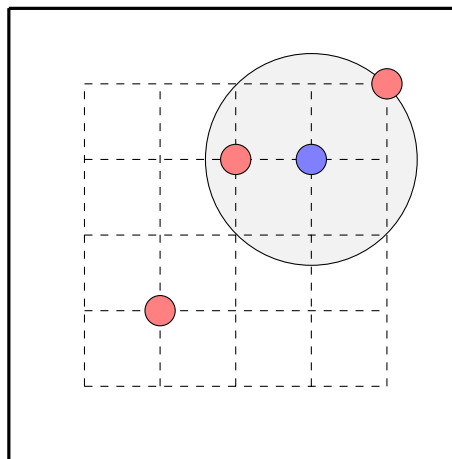


Figura 9.: Rappresentazione grafica di un possibile scenario del caso di studio

## 5.1 ANALISI

Per l'analisi del problema vengono effettuate le seguenti assunzioni:

- ogni agente si può muovere solo nelle quattro direzioni (nord, sud, ovest, est) o può decidere di rimanere fermo,
- tutti gli agenti sono *sincronizzati nello spazio*: esiste una griglia globale che rappresenta le strade percorribili dagli agenti,
- tutti gli agenti sono *sincronizzati nel tempo*: tutti eseguono allo stesso tempo un passo nella direzione scelta.

Possiamo quindi rappresentare la zona con una matrice  $Z \in \mathbb{N}_0^{m \cdot m}$ , dove  $m \in \mathbb{N}$  indica la dimensione della zona e gli elementi della matrice indicano il numero di agenti in una certa posizione. Se  $Z_{i,j} = 0$  la posizione è *libera*, se  $Z_{i,j} = 1$  allora la posizione è *occupata* da un agente, mentre se  $Z_{i,j} > 1$  allora in quella posizione si sta verificando una *collisione*.

Per analizzare la successione temporale dello scenario possiamo parametrizzare la matrice  $Z$  rispetto al tempo definendola come una funzione  $Z : \mathbb{N}_0 \rightarrow \mathbb{N}^{m \cdot m}$  che, dato un certo istante temporale  $n \in \mathbb{N}_0$ , restituisce una matrice  $Z(n)$  che descrive la zona in quell'istante. Lo scenario iniziale è rappresentato da  $Z(0)$ . Assumendo che i sensori di un agente permettano di rilevare se un altro agente si trova entro un passo di distanza, definiamo le *posizioni adiacenti* come le posizioni raggiungibili all'interno della zona che non portino ad uno scontro diretto con un altro agente:

$$N_{i,j} = \{(i', j') : |i - i' + j - j'| = 1 \wedge i', j' \in \{1, \dots, m\}\} \cup \{(i, j)\}.$$

Il criterio seguito dal generico agente sarà quindi l'algoritmo 1. I parametri

---

**Algorithm 1** Algoritmo di scelta generico
 

---

**Require:**  $m, n, n_{rob} \in \mathbb{N} \wedge i, j \in \{1, \dots, m\} \wedge Z \in \mathbb{N}_0^{m \cdot m}$

```

 $p_0 = (i, j)$ 
for  $k = 1$  to  $n$  do
   $p_k = \text{schedule}(\text{local}(Z, p_{k-1}, m), m, n_{rob})$ 
  barriera
   $Z_{p_{k-1}} = Z_{p_{k-1}} - 1$ 
   $Z_{p_k} = Z_{p_k} + 1$ 
  barriera
end for

```

---

dell'algoritmo hanno il seguente significato:

- $m$  è la dimensione dell'area,

- $n$  è il numero di passi che vengono eseguiti da ogni agente (generalmente possiamo immaginarlo come un numero molto grande),
- $nrob$  è il numero di agenti presenti nell'area,
- $i$  e  $j$  sono la posizione iniziale dell'agente,
- $Z$  è lo stato iniziale della matrice globale.

La funzione *local* viene così definita

$$local(Z, i, j, m) = Z[I(i, m), J(j, m)].$$

Il primo sottoinsieme di indici è definito come

$$I(i, m) = \begin{cases} \{i, i + 1\} & \text{se } i = 1 \\ \{i - 1, i\} & \text{se } i = m \\ \{i - 1, i, i + 1\} & \text{altrimenti} \end{cases}$$

e il secondo, in modo analogo  $J(j, m) = I(j, m)$ . Con la funzione *local* si vuole definire formalmente la sottomatrice locale che viene rilevata dal sensore di prossimità dell'agente.

La funzione *schedule* dipende invece dal comportamento che si vuole associare all'agente e quindi che criterio utilizzerà per risolvere le scelte. Lo scheduler avrà quindi pochi dati su cui prendere una decisione e se si esclude anche una memorizzazione dello storico allora il dominio di *schedule* diventa il numero di combinazioni di un massimo di  $nrob$  agenti all'interno delle posizioni locali. La conoscenza dell'agente si limita quindi alle posizioni locali  $local(Z, i, j, m)$ , il numero totale di agenti in gioco  $nrob$  e la dimensione dell'area  $m$ .

Le *barriere* sono il costrutto di programmazione parallela dove ogni agente attende che tutti gli altri abbiano raggiunto il suo stesso punto, dopodichè tutti possono riprendere l'esecuzione. In questo caso vengono utilizzate per un doppio scopo: il primo è di evitare la *race condition* e il secondo è quello di avere una separazione netta tra le fasi globali di decisione del prossimo passo e aggiornamento della mappa.

## 5.2 APPROCCI PROPOSTI

Introduciamo i due principali algoritmi di scheduling utilizzati in questo caso di studio:

- semi-casuale,
- basato sul model checking.

Lo scheduling *semi-causale* non fa altro che scegliere casualmente una delle posizioni libere raggiungibili al prossimo passo.

Lo scheduling basato sul model checking, invece, è incentrato sulla costruzione di un modello globale ottenuto facendo ipotesi sugli aspetti che non si conoscono. Una volta che si ha a disposizione il modello globale “stimato” si procede valutando la probabilità di soddisfare la formula che rappresenta l’obiettivo nel caso in cui si effettua una determinata scelta tramite model checking.

In questo caso di studio si conosce *quanti agenti sono presenti* ma non il loro comportamento e con che criterio prediligono una direzione piuttosto che un’altra. Si ipotizza il movimento degli altri agenti come uno scheduling casuale che, a differenza di quello semi-casuale, può scegliere anche direzioni occupate da altri agenti vietando comunque direzioni che farebbero uscire dal perimetro dell’arena. Il modello globale viene quindi correttamente rappresentato da una MDP in quanto è composizione parallela dei modelli degli agenti contenenti scelte nondeterministiche e probabilistiche.

Assumiamo l’implementazione in LAPSA dello scheduler dell’agente mostrata nel listato 5.1, ipotizzando che esista solo un altro agente nell’area e che questo si muova secondo uno scheduler casuale:

- ogni variabile indica se una posizione è occupata da un altro agente (valore 1) o libera (valore 0), la posizione p1 indica l’area a nord-ovest, p2 quella a nord, fino alla p9 che è quella a sud-est,
- la posizione p5 è l’area interna di collisione con l’agente, se altri agenti vengono rivelati in quella zona lo stato viene interpretato come collisione, per questo motivo l’obiettivo dell’agente è formulato in termini di questa zona
- le transizioni contengono già l’assunzione di come gli agenti esterni effettueranno le loro scelte, nella prima transizione di esempio viene mostrato dove si può muovere un agente che si trova inizialmente in posizione p3: la scelta nondeterministica indica la direzione intrapresa dal modulo mentre la distribuzione descrive come la scelta dell’agente esterno modifica lo stato percepito,
- gli agenti esterni sono già considerati all’interno del modulo principale, quindi possiamo assumere che non ci siano moduli nell’ambiente.

Listati 5.1: Implementazione LAPSA dello scheduler basato su model checking

```

1 actions { a }
2
3 subject module NDRobot {
4   // variabili area locale
5   int p1 = 0; int p2 = 0; int p3 = 0;
6   int p4 = 0; int p5 = 0; int p6 = 0;

```



```

7   int p7 = 0; int p8 = 0; int p9 = 0;
8
9   // transizioni
10  p1+p2+p4+p5+p6+p7+p8+p9 == 0 and p3 = 1 [a]
11  => // resta fermo
12      <0.2> noaction #
13      <0.2> p3 = 0, p6 = 1 #
14      <0.4> p3 = 0 #
15      <0.2> p2 = 1, p3 = 0
16  => // vai a nord
17      <0.2> noaction #
18      <0.2> p3 = 0, p5 = 1 #
19      <0.2> p3 = 0, p6 = 1 #
20      <0.2> p3 = 0 #
21      <0.2> p3 = 0, p9 = 1
22  => // vai a ovest
23      <0.2> noaction #
24      <0.8> p3 = 0
25  => // vai a est
26      <0.2> noaction #
27      <0.2> p3 = 0, p5 = 1 #
28      <0.2> p1 = 1, p3 = 0 #
29      <0.2> p3 = 0 #
30      <0.2> p2 = 1, p3 = 0
31  => // vai a sud
32      <0.2> noaction #
33      <0.8> p3 = 0;
34
35  // ...
36
37  // obiettivo
38  target never p5 > 0
39 }
40
41 environment is empty
42
43 ranges {
44     NDRobot.p1 in [0,1], NDRobot.p2 in [0,1],
45     NDRobot.p3 in [0,1], NDRobot.p4 in [0,1],
46     NDRobot.p5 in [0,1], NDRobot.p6 in [0,1],
47     NDRobot.p7 in [0,1], NDRobot.p8 in [0,1],
48     NDRobot.p9 in [0,1]
49 }

```

Per brevità è stata riportata nel listato solo una transizione.

La compilazione del file *LAPSA* costruirà un modello *PRISM* e una formula *PCTL* ed otterrà, tramite model checking, una lista di probabilità di successo che verranno salvate e serializzate all'interno di una struttura dati utilizzabile da un

programma *JAVA* esterno. Si tratta di una *hashtable* che ha come chiave lo stato del modulo e come dato la probabilità di successo in quello stato. Il programma *JAVA* che utilizza questa hashtable decide la scelta da fare in base all'algoritmo 2. In questo caso la hashtable generata dal compilatore *LAPSA* ha un solo livello di

---

**Algorithm 2** Algoritmo di scheduling basato sul model checking

---

**Require:** Actions: insieme delle azioni disponibili,  
 States: insieme degli stati possibili,  
 current: stato attuale dell'agente,  
 index: hash table delle probabilità calcolate dal model checker.  
 max = -1  
**for all** act  $\in$  Actions **do**  
   **for all** next  $\in$  States **do**  
     *calcola la probabilità  $p_{\text{step}}$  di passare da current a next con l'azione act*  
      $p_{\text{MC}} = \text{index}(\text{next})$   
      $\text{prob} = p_{\text{step}} \cdot p_{\text{MC}}$   
     **if** prob > max **then**  
       max = prob  
       best\_act = act  
     **end if**  
   **end for**  
**end for**  
**return** best\_act

---

profondità in quanto non sono presenti moduli nell'ambiente. Per questo motivo è sufficiente una semplice ricerca del massimo al posto della più generale ricerca del massimo dei valori minimi.

### 5.3 SIMULAZIONI

Gli esperimenti condotti su questo caso di studio consistono nell'osservare il numero di scontri che coinvolgono l'agente principale in simulazioni di 100 passi, variando il numero di agenti a scheduler casuale ipotizzati nel modulo *LAPSA*, il numero di agenti effettivi presenti nello scenario e lo scheduler utilizzato dagli agenti.

Gli scenari saranno quindi i seguenti:

- lo scheduler dell'agente principale è semi-casuale mentre quello dei secondari è casuale, lo scenario reale è composto da
  - un agente principale e un agente secondario
  - un agente principale e due agenti secondari
  - un agente principale e tre agenti secondari

in un'arena  $5 \times 5$ ,

- lo scheduler dell'agente principale è basato sul model checker che assume la presenza di 1, 2 o 3 agenti secondari casuali, gli scheduler degli agenti secondari sono casuali, lo scenario reale è composto da
  - un agente principale e un agente secondario
  - un agente principale e due agenti secondari
  - un agente principale e tre agenti secondari

in un'arena  $5 \times 5$ ,

- lo scheduler dell'agente principale è basato sul model checker che assume la presenza di 1, 2 o 3 agenti secondari casuali, gli scheduler degli agenti secondari sono casuali e lo scenario reale è composto da un agente principale e 40 agenti secondari in un arena  $10 \times 10$ ,
- lo scheduler dell'agente principale è semi-casuale, gli scheduler degli agenti secondari sono casuali e lo scenario reale è composto da un agente principale e 40 agenti secondari in un arena  $10 \times 10$ ,
- tutti gli scheduler sono basati sul model checker assumendo la presenza di tre agenti secondari casuali, lo scenario reale è composto da un agente principale e tre agenti secondari in un'arena  $5 \times 5$ .

In figura 10 viene riportata la rappresentazione grafica di sei passi di una simulazione. Il cerchio blu indica l'agente principale che si muove secondo uno scheduler basato su model checker, lo scenario ipotizzato coincide con quello reale: nell'ambiente sono presenti due altri agenti (cerchi rossi) che si muovono secondo uno scheduler casuale. Nel primo passo l'agente principale è in un angolo e non rileva la presenza di nessuno nelle vicinanze, quindi decide di spostarsi verso est. Al secondo passo i due agenti secondari sono entrati contemporaneamente nell'area di visibilità e lo scheduler prende la decisione di stare fermo. Osserviamo che quella è effettivamente la scelta più saggia in quanto nessuno dei due agenti secondari sarebbe in grado di scontrarsi direttamente col principale mentre le altre tre direzioni sarebbero state tutte a rischio. Inoltre lo scheduler sta ipotizzando che ci siano solamente altri due agenti nell'ambiente, quindi la possibilità di venire intrappolato al turno successivo non sussiste. Nel terzo passo la situazione cambia e percepisce solamente un agente a sud, quindi decide di spostarsi a est tornando nell'angolo. Al quarto passo si allontana nuovamente dall'angolo andando a est. Al quinto passo percepisce nuovamente un altro agente e si allontana tornando sui suoi passi. Nel sesto passo decide di rimanere nell'angolo.

Andando ad osservare i dati forniti dalla hashtable nella simulazione appena descritta possiamo notare che le probabilità presenti sono solo uguali a 1 (se uno stato non è presente nella tabella significa che la sua probabilità è nulla).

Considerando infatti solo due agenti non c'è modo che questi possano mettere in difficoltà lo scheduler costringendolo ad abbassare la sua probabilità massima di successo. Il modello *LAPSA* infatti non considera le pareti e gli angoli ma ogni stato come se fosse sempre spazio aperto. In questo modo il modello è più leggero ma non viene considerato ad esempio il caso in cui l'agente è bloccato in un angolo.

Tutti e soli gli stati di collisione non sono presenti nella hashtable, quindi hanno probabilità 0, e questo garantisce che lo scheduler non sceglierà mai di muoversi verso un altro agente frontalmente. Dato che tutte le altre configurazioni hanno probabilità 1 la preferenza ricadrà sullo scenario che tra questi sarà quello raggiungibile con probabilità più alta al passo successivo.

In tabella 6 vengono riportate le medie dei test effettuati, raffigurate nel grafico 11. Dal grafico si riesce ad osservare un effettivo miglioramento del risultato dello scheduler basato sul model checker rispetto a quello semi-casuale. Si percepisce, seppure in minor misura, un leggero miglioramento all'aumentare della complessità dello scenario ipotizzato: anche nel caso in cui le simulazioni prevedano solamente un agente secondario si riscontra una media di collisioni per test più bassa se la hashtable viene generata su ipotesi più complesse, assumendo due o tre agenti secondari. Nello scenario reale che comprende due agenti esterni si verifica un leggero miglioramento medio passando dallo scheduler generato sull'ipotesi di un solo agente semi-casuale a quello che ne considera due: questo miglioramento è giustificato dal fatto che gli stati previsti sono un numero maggiore e se lo scheduler basato sul model checker non dovesse trovare lo stato che sta cercando all'interno della hashtable allora si comporterà come uno scheduler semi-casuale. Le simulazioni con 40 agenti sono state eseguite con un'arena più grande rispetto agli altri casi ( $10 \times 10$  invece che  $5 \times 5$ ) e potrebbe sembrare che l'agente abbia meno difficoltà, ma in realtà la densità è molto più alta. In questo ultimo caso notiamo un ulteriore miglioramento rispetto allo scheduler semi-casuale.

	1 casuale	2 casuali	3 casuali	40 casuali
<b>semi-casuale</b>	4.22	7.67	11.98	32.86
<b>model checker 1</b>	2.87	6.52	10.02	25.07
<b>model checker 2</b>	2.82	6.36	9.88	24.99
<b>model checker 3</b>	2.44	6.27	9.82	25.10

Tabella 6.: Risultati delle simulazioni

Le simulazioni tra agenti che utilizzano solo scheduler basati su model checker danno origine a fenomeni che rendono i risultati non confrontabili con quelli mostrati finora. Tendono a crearsi situazioni ottime e pessime a seconda dei casi:

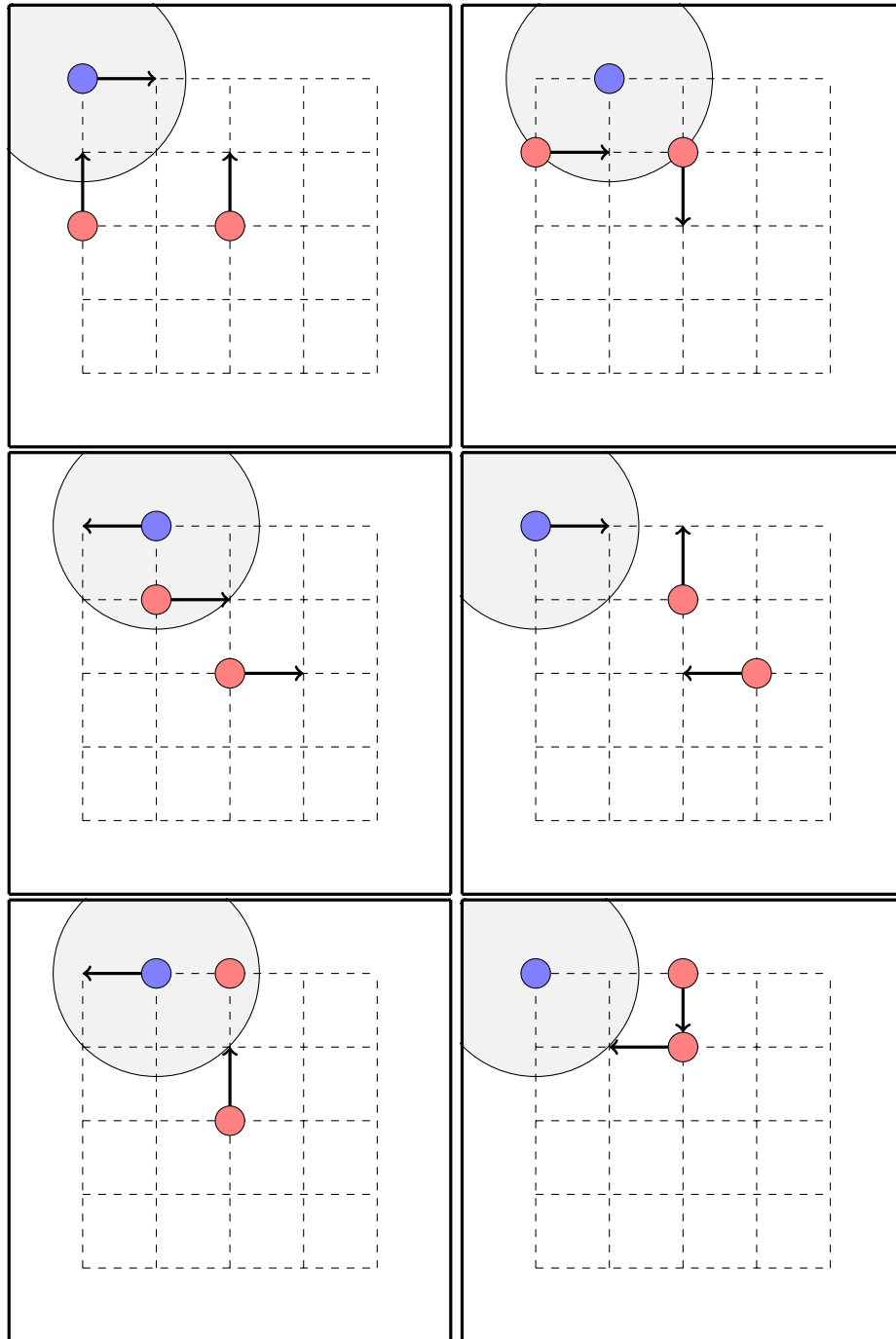


Figura 10.: Alcuni passi di una simulazione. L'agente blu si muove secondo uno scheduler basato sul model checker che considera due altri agenti nell'arena. Gli agenti due agenti sono rossi e si muovono con uno scheduler casuale. Le frecce indicano la direzione verso la quale gli agenti si vogliono spostare al passo successivo.

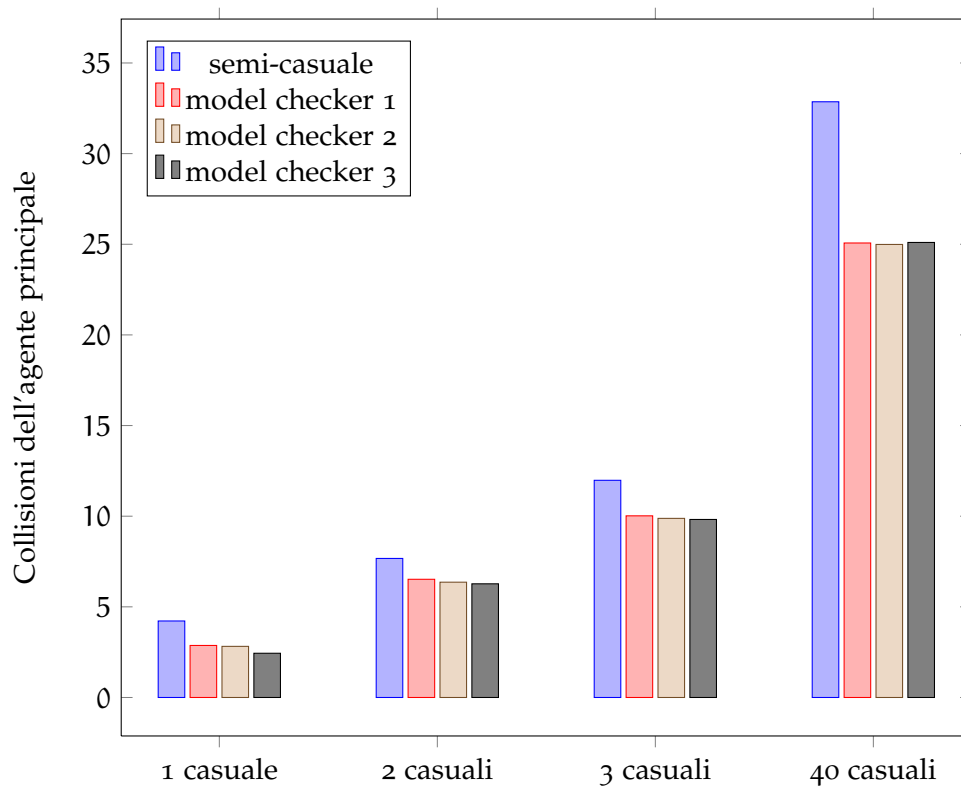


Figura 11.: Grafico dei risultati delle simulazioni: sull'ascissa variano gli agenti che sono realmente nell'area assieme all'agente principale, sull'ordinata viene indicato il numero di collisioni e i diversi colori rappresentano di diversi scheduler utilizzati dall'agente principale.

in alcune occasioni tutti gli agenti si stabilizzano in una posizione di equilibrio dove nessuno ha interesse nello spostarsi, in altre lo scenario si ripete con un periodo più ampio portando a oscillazioni tra più zone ma ripetendo gli stessi errori dovuti all'approccio deterministico dello scheduler. Nel caso positivo si ha un numero di collisioni quasi sempre nullo mentre in quello negativo ci si avvicina al 50% della lunghezza della simulazione per il fenomeno sopra descritto. Questo comportamento è giustificato dal fatto che tutti gli agenti hanno lo stesso comportamento deterministico e quindi ricercano le stesse condizioni. Si osserva quindi una situazione di equilibrio instabile che dipende dallo stato iniziale della situazione: se un agente incorre in un caso di collisioni cicliche sarà impossibile uscirne.





Parte IV

CONCLUSIONI



## CONCLUSIONI

---

In questo lavoro di tesi è stato affrontato il problema di fornire ad un agente una strategia adattiva. L'idea di base utilizzata è stata quella di utilizzare il model checking per fare previsioni anziché verifica come nell'approccio classico. Abbiamo quindi introdotto *LAPSA*: un linguaggio specifico per agenti adattivi.

Nei risultati ottenuti dalle simulazioni si possono osservare degli effettivi miglioramenti rispetto a una strategia casuale, i quali suggeriscono che le scelte prese siano tendenzialmente adatte alla situazione. D'altra parte sarebbe semplice elaborare uno scheduler ad hoc per caso di studio proposto che, rispettando i vincoli dati dalla conoscenza parziale dell'ambiente, ottenga esiti migliori. L'obiettivo principale però era ad un livello più generale, cioè lavorare ad un approccio che permetta di elaborare strategie per sistemi adattivi, e queste simulazioni ci hanno permesso di verificarne la possibilità.

Se da una parte l'approccio di precomputazione delle scelte dello scheduler proposto da *LAPSA* libera l'agente finale da quasi tutto il carico di lavoro, i tempi di compilazione di un modello possono essere molto alti se il modello è molto complesso. Per far fronte a questo è possibile effettuare astrazioni a scapito della precisione della strategia. Nel caso di studio proposto abbiamo infatti modellato il sistema astraendo dalla posizione dell'agente all'interno dell'arena, ignorando quindi la distanza dalle pareti e dagli angoli, sui quali non viene elaborata alcuna strategia dedicata.

Da questo lavoro si aprono molte prospettive. Una prima evoluzione naturale è la modifica della conoscenza dell'ambiente durante l'esecuzione: prendendo ad esempio il nostro caso di studio, se l'ipotesi iniziale considerasse solo un agente esterno e durante l'esecuzione ne venissero avvistati due contemporaneamente le informazioni a disposizione permetterebbero di inserire nel modello una maggiore accuratezza. Per realizzare è possibile estendere *LAPSA* con delle nuove primitive che permettano la gestione delle informazioni conosciute a run-time: questi nuovi operatori permetterebbero all'agente principale di aggiornare la composizione dell'ambiente aggiungendo e rimuovendo moduli.

Nel caso di studio proposto, inoltre, ci si concentra principalmente sull'obiettivo locale: l'agente deve minimizzare il numero delle sue collisioni con altri agenti. Un possibile obiettivo globale potrebbe estendere naturalmente quello locale ricercando una strategia che permetta di minimizzare gli scontri tra tutti gli agenti nell'arena. Quello che succede nella simulazione in cui tutti gli agenti hanno uno scheduler basato sul model checker è che ognuno ha l'obiettivo locale di minimizzare le sue collisioni. Questo però non implica una ricerca di un ottimo globale, infatti nelle simulazioni si verificano situazioni di equilibrio

che si ripetono ciclicamente producendo sia casi positivi che negativi. Questo aspetto potrebbe essere approfondito sfruttando la teoria dei giochi.

Attualmente gli scheduler degli agenti prendono decisioni solamente sullo stato corrente ignorando la storia precedente. Conservare un certo numero di stati precedenti potrebbe aiutare a produrre strategie migliori, potendo ad esempio dedurre dei limiti massimi di zona che un agente appena uscito dalla visuale non può aver superato.

## IMPLEMENTAZIONE DI LAPSA IN XTEXT

Il compilatore del linguaggio *LAPSA* è stato implementato tramite il tool *Xtext* [xteb] e l'implementazione può essere reperita tramite il repository online <https://github.com/marcotinacci/SwarmRobotics>.

Il tool mette a disposizione gli strumenti per definire la grammatica concreta del linguaggio che si vuole tradurre, generando automaticamente tutte le classi *JAVA* che rappresentano la struttura dei nodi terminali e nonterminali. In questo modo sarà poi possibile implementare il backend facendo riferimento a queste classi ed utilizzandone i metodi visitando l'albero sintattico. La sintassi di *LAPSA* è riportata nel listato A.1 e segue i costrutti linguistici dati in tabella 2 ad eccezione di alcune modifiche strettamente legate all'implementazione.

Listati A.1: Sintassi di *LAPSA* in *Xtext*

```

1 grammar unifi.marcotinacci.thesis.seal.Seal with org.eclipse.xtext.common.
  Terminals
2
3 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
4 generate seal "http://www.marcotinacci.unifi/thesis/seal/Seal"
5
6 // === main program syntax ===
7
8 Program:
9   'actions' '{' (actions+=Action)+ '}'
10  'subject' modules+=ModuleDefine
11  (modules+=ModuleDefine)*
12  'environment' ((environment=Environment) | isEmptyEnv?='is empty')
13  ('ranges' '{' ranges+=Range (',' ranges+=Range)* '}')?
14 ;
15
16 Range:
17   module=[ModuleDefine] '.' variable=[VariableDeclaration] 'in' '[' from=
     Value ',' to=Value ']' ('delta' '=' delta=Value)?
18 ;
19
20 Action: name=ID ;
21
22 // === module syntax ===
23
24 ModuleDefine:
25   'module' name=ID '{'
26   (variables+=VariableDeclaration ';')+

```

```

27         (rules+=Rule ';'')+
28         ('target' 'never' never+=Expression)*
29     '}'
30 ;
31
32 VariableDeclaration:
33     type=Type name=ID '=' expr=Expression
34 ;
35
36 Type:
37     name='int' | name='float' | name='bool'
38 ;
39
40 Rule:
41     cond=Expression '[' action=[Action] ']' (ndCases+=NDCase)+
42 ;
43
44 NDCase:
45     '=>' cases+=Case ('#' cases+=Case)*
46 ;
47
48 Case:
49     '<'weight=Expression (hasCondition?=', ' cond=Expression)? '>' update+=
        Update (',' update+=Update)*
50 ;
51
52 Update:
53     { NoAction } 'noaction' |
54     { Assign } variable=[VariableDeclaration] '=' expr=Expression
55 ;
56
57 Environment:
58     modules+=[ModuleDefine] ('|{' (actions+=[Action])+ '}'| modules+=[
        ModuleDefine])*
59 ;
60
61
62 // === expression syntax ===
63
64 Expression:
65     Logical;
66
67 Logical returns Expression:
68     Relation
69     (({And.left=current} 'and'
70      | {Or.left=current} 'or')
71      right=Relation)*;
72

```

```

73 Relation returns Expression:
74     Addition
75         ({Leq.left=current} '<='
76         | {Less.left=current} '<'
77         | {Eq.left=current} '=='
78         | {Neq.left=current} '!='
79         | {Geq.left=current} '>='
80         | {Gtr.left=current} '>')
81         right=Addition)?;
82
83 Addition returns Expression:
84     Multiplication
85         ({Plus.left=current} '+'
86         | {Minus.left=current} '-')
87         right=Multiplication)*;
88
89 Multiplication returns Expression:
90     PrimaryExpression
91         ({Multi.left=current} '*'
92         | {Div.left=current} '/' )
93         right=PrimaryExpression)*;
94
95 PrimaryExpression returns Expression:
96     '(' Expression ')'
97     | { Not } 'not' cond=PrimaryExpression
98     | { Literal } value=Value
99     | { Quantifier } 'exists' name=ID ':' module=[ModuleDefine] 'such that'
100        cond=PrimaryExpression
101     | { ExternalReference } module=[Quantifier] '.' variable=[
102        VariableDeclaration]
103     | { LocalReference } ('this' '.')? variable=[VariableDeclaration]
104 ;
105
106 Value: INT | FLOAT | BOOL;
107
108 terminal INT returns ecore::EInt: ('0'..'9')+;
109
110 terminal FLOAT returns ecore::EDouble:
111     ('-')? (INT)* ('.' (INT)+)? |
112     ('-')? (INT)+ ('.' (INT)+)? |
113     ('-')? (INT)+ ('.' (INT)+)? (('e'|'E') ('-'|'+')? (INT)+);
114
115 terminal BOOL returns ecore::EBoolean: 'true' | 'false';

```

L'implementazione del compilatore avviene attraverso la definizione della funzione `doGenerate` offerta da *Xtext*. L'implementazione è riportata nel listato ?? e consiste nelle fasi principali citate in precedenza: generazione del modello *PRISM* e della formula obiettivo *PCTL*, esecuzione del model checking ed esportazione

dei risultati serializzando la hashtable nel file `hashtable.ser`.

Listati A.2: Implementazione della funzione di generazione in *Xtend*

```

1  override void doGenerate(Resource resource, IFileSystemAccess fsa) {
2      modules = new LinkedList<ModuleDefine>
3      renaming = new Hashtable<String, Integer>
4      moduleCounter = -1
5
6      var Program p = resource.contents.get(0) as Program
7      if(p.isEmptyEnv) env = null
8      else env = p.environment
9
10     // prism model generation
11     fsa.generateFile("model.pm", p.prismCompile)
12
13     // model checking
14     var fh = new FormulaHandler(resource)
15
16     moduleCounter = 0
17     fh.execModelCheck(p.modules.get(0).never.get(0).
18         prismCompileExpression.toString)
19
20     // serialize hashmap
21     var objOut = new ObjectOutputStream(
22         new FileOutputStream(commons::getSrcGenURI(resource)+"
23             hashtable.ser")
24     )
25     objOut.writeObject(fh.index);
26     objOut.close();
27 }

```

La generazione del modello *PRISM* ha inizio a partire dal nodo iniziale *program* e scende fino alle foglie dell'albero sintattico. All'interno del listato A.3 vengono mostrate le funzioni che traducono il nodo *program* e i nodi *module* in codice *PRISM*. Nel nodo *program* viene specificato che si tratta di un modello *MDP* e viene data la traduzione del modulo principale e di quelli dell'ambiente, se ce ne sono. Ogni modulo, a sua volta, viene compilato rimandando la traduzione di ogni nodo *variable* e *rule* che contiene. I nomi delle variabili sono gestiti con un valore numerico incrementale (*moduleCounter*) in modo da non avere ambiguità.

Listati A.3: Generazione dei moduli in *Xtend*

```

1  def CharSequence prismCompile(Program p){
2      var CharSequence tpl = '''
3      mdp
4
5      // === MODULES ===
6

```



```

7      <<p.modules.get(0).prismCompile>>
8      '''
9
10     if(!p.isEmptyEnv){
11         tpl =
12         '''
13         <<tpl>>
14         <<FOR m:p.environment.modules>>
15             <<m.prismCompile>>
16         <<ENDFOR>>
17         '''
18     }
19     tpl
20 }
21
22 def prismCompile(ModuleDefine m){
23     moduleCounter=moduleCounter+1
24     '''
25     module module_<<moduleCounter>>
26     // variables
27     <<FOR v:m.variables>>
28         <<v.prismCompile>>
29     <<ENDFOR>>
30     // rules
31     <<FOR r:m.rules>>
32         <<r.prismCompile>>
33     <<ENDFOR>>
34     endmodule
35     '''
36 }

```

La dichiarazione di variabili viene implementata come descritto nel listato A.4, recuperando il range dal nodo *ranges* e distinguendo il caso sui tre tipi di variabile che si possono verificare. La variabile booleana ha una mappatura quasi naturale in quanto ci si limita ad aggiungergli solamente la valutazione dell'assegnamento iniziale. Per le variabili intere viene anche specificato il range riportando i valori selezionati. Per gestire le variabili *float* si esegue una discretizzazione: all'interno della sezione *ranges* del file *LAPSA* dovranno essere stati specificati i parametri di inizio e fine del dominio e un valore  $\delta$  (di default uguale a uno) che specifichi l'ampiezza dell'intervallo di discretizzazione. Il dominio discretizzato viene quindi mappato nel dominio degli interi a partire dallo zero e anche il valore di inizializzazione verrà riadattato allo stesso modo.

Listati A.4: Traduzione della dichiarazione di variabili da *LAPSA* a *PRISM*

```

1  def prismCompile(VariableDeclaration v){
2      var from = 0
3      var to = 1

```

```

4      var delta = 1
5
6      // look at ranges and get bounds
7      for(r:(v.eContainer.eContainer as Program).ranges){
8          if(r.variable==v){
9              to = Integer.parseInt(r.to)
10             from = Integer.parseInt(r.from)
11             if(r.delta != null){
12                 delta = Integer.parseInt(r.delta)
13             }
14         }
15     }
16     '''
17     <<v.localName>> :
18     <<IF v.type.name.equals('bool')>>
19         bool init <<v.expr.prismCompileExpression>>;
20     <<ENDIF>>
21     <<IF v.type.name.equals('int')>>
22         [<<from>>..<<to>>] init <<v.expr.prismCompileExpression>>;
23     <<ENDIF>>
24     <<IF v.type.name.equals('float')>>
25         [0..floor((<<to>>-<<from>>)/<<delta>>)] init
26         ceil((<<v.expr.prismCompileExpression>>-<<from>>)/<<delta>>);
27     <<ENDIF>>
28     '''
29 }

```

Come ultima procedura descriviamo la traduzione delle transizioni da *LAPSA* a *PRISM* (listato A.5). La condizione di base rappresenta la guardia che abilita la transizione e viene definita una volta inizialmente e riutilizzata se necessario. Può infatti succedere, nel caso in cui utilizziamo il costrutto di scelta nondeterministica, che la condizione debba essere riscritta nel codice *PRISM* che non può prevedere quel tipo di espressione. Il primo ciclo scandisce infatti ogni caso nondeterministico e genera una transizione per ciascuno. Il secondo invece si occupa dei casi delle distribuzioni probabilistiche provvisti di guardie generando tutte le combinazioni di condizioni soddisfatte e non. Viene quindi inserita la condizione di base seguita da tutte le condizioni aggiunte dalle dai casi probabilistici provvisti di guardia. Se non ci sono casi allora viene inserita la transizione *true* a probabilità 1, altrimenti un ulteriore ciclo scandisce i casi presenti. Le probabilità vengono normalizzate in modo che una transizione abbia sempre la distribuzione dei casi a somma 1, mentre le espressioni di aggiornamento vengono ricavate dai nodi *update*.

Listati A.5: Traduzione delle transizioni da *LAPSA* a *PRISM*

```

1  def prismCompile(Rule r){
2      val baseCondition = '''
3          [<<r.action.name>>] <<r.cond.prismCompileExpression>>'''

```

```

4      // add unconditioned cases and true cases
5      '''
6      <<FOR ndcase:r.ndCases>>
7      <<FOR comb:ndcase.cases.reduced.allCombinations
8          SEPARATOR ';' AFTER ';'>>
9      <<baseCondition>>
10     <<FOR c:ndcase.cases.filter(c|c.hasCondition)
11         BEFORE '&' SEPARATOR '&'>>
12         <<IF !comb.contains(c)>>!<<ENDIF>>(<<c.cond.
13             prismCompileExpression>>)
14     <<ENDFOR>>
15     ->
16     <<IF comb.size==0 && ndcase.cases.filter(c | !c.hasCondition).size
17         ==0>>
18         1 : true
19     <<ELSE>>
20         <<FOR c : ndcase.cases.filter(c | !c.hasCondition
21             || comb.contains(c)) SEPARATOR '+'>>
22             (<<c.weight.prismCompileExpression>>) /
23             (<<ndcase.totalWeight(comb)>>) :
24             <<FOR u:c.update SEPARATOR '&'>>
25                 <<u.prismCompileUpdate>>
26             <<ENDFOR>>
27         <<ENDFOR>>
28     <<ENDIF>>
29     <<ENDFOR>>
30     '''
31 }

```



## BIBLIOGRAFIA

---

- [BCG<sup>+</sup>12] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembling strategies with maude. *ASCENS*, 2012.
- [BCL<sup>+</sup>12] Roberto Bruni, Andrea Corradini, Alberto Lluch Lafuente, Fabio Gadducci, and Andrea Vandin. A conceptual framework for adaptation. *LNCs*, 7212:240–254, 2012.
- [Bil12] P. Billingsley. *Probability and Measure*, chapter 1. Anniversary edition edition, 2012.
- [GLPT11] Edmond Gjondrekaj, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A robot foraging scenario in klaim. 2011.
- [Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.
- [KNP11] M. Kwiatkowska, G. Norman, and D. Parker. *PRISM 4.0: Verification of Probabilistic Real-time Systems*, pages 101–135. American Mathematical Society, 2011.
- [PPQ95] Terence J Parr, T. J. Parr, and R W Quong. Antlr: A predicated-ll(k) parser generator. <http://wwwantlr.org/>, 1995.
- [SRA<sup>+</sup>11] Nikola Serbedzija, Stephan Reiter, Maximilian Ahrens, José Velasco, Carlo Pinciroli, Nicklas Hoch, and Bernd Werther. Requirement specification and scenario description of the ascens case studies. *ASCENS*, 2011.
- [xtea] Xtend. <http://www.eclipse.org/xtend/>.
- [xteb] Xtext. <http://www.eclipse.org/Xtext/>.