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

Ohana means family.
Family means nobody gets left behind, or forgotten.

— Lilo & Stitch

Dedicated to the loving memory of Rudolf Miede.

1939 – 2005

ABSTRACT

Short summary of the contents in English...

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

Put your acknowledgments here.

Many thanks to everybody who already sent me a postcard!

Regarding the typography and other help, many thanks go to Marco Kuhlmann, Philipp Lehman, Lothar Schlesier, Jim Young, Lorenzo Pantieri and Enrico Gregorio[1], Jörg Sommer, Joachim Köstler, Daniel Gottschlag, Denis Aydin, Paride Legovini, Steffen Prochnow, Nicolas Repp, Hinrich Harms, Roland Winkler, Jörg Weber, and the whole LaTeX-community for support, ideas and some great software.

Regarding: The port was intially done by *Nicholas Mariette* in March 2009 and continued by *Ivo Pletikosić* in 2011. Thank you very much for your work and the contributions to the original style.

1 Members of GuIT (Gruppo Italiano Utilizzatori di TeX e LaTeX)

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

LISTATI

ACRONIMI

CTL *Computation Tree Logic*

CSL *Continuous Stochastic Logic*

CSP Communicating Sequential Processes

CTMC *Continuous-Time Markov Chain*

DTMC *Discrete-Time Markov Chain*

LAPSA *Language for Population of Self-adaptive Agents*

LTL *Linear Time Logic*

MDP *Markov Decision Process*

PCTL *Probabilistic Computation Tree Logic*

PS *Processo Stocastico*

PTA *Probabilistic Timed Automata*

1 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 circostante 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 di agenti: robot, risorse e e-vehicles [7]. 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?*"[2]. 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 [1] assemblandosi in gruppi richiedono una collaborazione esplicita mentre per la raccolta di risorse [3] 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* [5], 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 *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* comporenderà un modello *PRISM* e un file di formule *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* [9], 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* [8] vengono costruiti automaticamente lexer e parser basati su *ANTLR* [6]. 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

2

PROBABILITÀ

In questo capitolo introduciamo gli elementi fondamentali per la comprensione delle sezioni successive. Cominciando dalle basi di probabilità si arriverà a formulare le definizioni dei principali modelli utilizzati: *Discrete-Time Markov Chain* (*DTMC*) e *MDP*.

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 Ω) *Lo* spazio fondamentale Ω *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 Ω*

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

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

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

Gli elementi di una σ-algebra sono chiamati *insiemi misurabili*. Chiamiamo *spazio misurabile* uno spazio fondamentale su cui è definita una σ-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*

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

.

Definizione 6 (Insieme di Borel) *Un* insieme di Borel $\mathcal{B}(\mathbb{R})$ *è la più piccola* σ-*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* $\mathbb{P} : \mathcal{F} \to \mathbb{R}_{\geqslant 0}$ *tale che*

$$\mathbb{P}(\emptyset) = 0 \tag{2.1}$$

$$\mathbb{P}(\Omega) = 1 \tag{2.2}$$

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:*

$$\mathbb{P}\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \mathbb{P}\left(A_i\right) \tag{2.3}$$

Chiamiamo *spazio di probabilità* dell'esperimento casuale $\mathcal{C}$ la tripla $(\Omega, \mathcal{F}, \mathbb{P})$, dove Ω 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. $\forall A \in \mathcal{F} : \mathbb{P}A + \mathbb{P}\overline{A} = 1$,
2. $\forall A, B \in \mathcal{F} : A \subseteq B \Rightarrow \mathbb{P}A \leqslant \mathbb{P}B$,
3. $\forall A \in \mathcal{F} : \mathbb{P}A \leqslant 1$,
4. $\forall A, B \in \mathcal{F} : \mathbb{P}(A \cup B) \geqslant \max\{\mathbb{P}A, \mathbb{P}B\}$,
5. $\forall A, B \in \mathcal{F} : \mathbb{P}A \cap B \leqslant \min\{\mathbb{P}A, \mathbb{P}B\}$,
6. $\forall A, B \in \mathcal{F} : \mathbb{P}(A \cup B) = \mathbb{P}A + \mathbb{P}B - \mathbb{P}(A \cap B)$,
7. $\forall A, B \in \mathcal{F} : A \subseteq B \Rightarrow \mathbb{P}(B \backslash A) = \mathbb{P}B - \mathbb{P}A$,
8. $\forall A_i \in \mathcal{F} : \mathbb{P}\left(\bigcup_{i=0}^{\infty} A_i\right) \leqslant \sum_{i=0}^{\infty} \mathbb{P}A_i$.

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

2.1 VARIABILI CASUALI

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

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

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

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

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

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* $f : \Omega_1 \to \Omega_2$ *funzione misurabile, allora:*

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

è uno spazio di probabilità.

2.2 PROCESSI STOCASTICI

Definizione 13 (*Processo Stocastico* (PS)) *Un* PS *è una famiglia* T*-indicizzata di variabili casuali*

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

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

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

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

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

2.3 DISCRETE-TIME MARKOV CHAINS

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*;*
- $\mathbb{P} : S \times S \to [0, 1]$ *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* path *di una* DTMC $\mathcal{D}$ *che comincia da* s_0 *in* $\mathcal{D}$ *è una sequenza non vuota* $s_0, s_1, s_2, \ldots$ *di stati con* s_0 *come stato iniziale e con* $\mathbb{P}(s_i, s_{i+1}) > 0$*.*

Osservazione 2 *Un* path *di una* DTMC *può essere finito o infinito.*

Definizione 18 (*DTMC* path up to n) *Sia* σ *un* path*, il* path up to n*, o* $\sigma \uparrow n$*, è il prefisso* $s_0, \ldots, s_n$ *di* σ*.*

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

Utilizziamo inoltre le seguenti notazioni riguardanti i path su *DTMC*:

- $\sigma(i)$ indica l'i-esimo stato del path σ,
- $|\sigma|$ indica la lunghezza del path σ,
- $\mathtt{last}(\sigma)$ indica l'ultimo stato del path finito σ e
- $\sigma_{\mathtt{fin}}$ indica l'insieme di tutti i path finiti.

2.4 MARKOV DECISION PROCESSES

Definizione 20 *Un* MDP *è una tupla* $\mathcal{M} = (S, \bar{s}, \mathtt{Act}, \mathtt{Steps})$ *dove:*

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

Definizione 21 (*MDP* path) *Un* path *di una* MDP $\mathcal{M}$ *che comincia da* s_0*, indicato con* $\mathtt{path}^{\mathcal{M}}_{s_0}$*, è una sequenza infinita* $s_0, a_1, \mu_1, s_1, a_2, \mu_2, s_3, \ldots$ *dove* $s_i \in S$*,* $(a_{i+1}, \mu_{i+1}) \in \mathtt{Steps}(s_i)$ *e* $\mu_{i+1}(s_{i+1}) > 0$ *per ogni* $i \geqslant 0$*.*

Osservazione 3 *Un* path *di una* MDP *può essere finito o infinito.*

Definizione 22 (*MDP* path up to n) *Sia* σ *un* path*, il* path up to n*,* o $\sigma \uparrow n$*, è il prefisso* $s_0, \ldots, s_n$ *di* σ*.*

Definizione 23 (*MDP* paths) $\mathtt{paths}^{\mathcal{M}}_{s_0}$ *o* $\mathtt{paths}(\mathcal{M}, s_0)$*, è l'insieme di tutti gli* MDP *path della* MDP $\mathcal{M}$ *che cominciano in* s_0*.*

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

- $\sigma(i)$ indica l'i-esimo stato del path σ,
- $|\sigma|$ indica la lunghezza del path σ,
- $\mathtt{last}(\sigma)$ indica l'ultimo stato del path finito σ e
- $\sigma_{\mathtt{fin}}$ indica l'insieme di tutti i path finiti.

Definizione 24 (Scheduler) *Uno* scheduler Σ *di una* MDP $\mathcal{M}$ *è una funzione che mappa tutti gli elementi di* $\sigma_{\mathtt{fin}}$ *di* $\mathcal{M}$ *in un elemento dell'insieme* $\mathtt{Steps}(\mathtt{last}(\sigma_{\mathtt{fin}}))$ *indicato con* $S(\sigma_{\mathtt{fin}})$*. Con* $\mathtt{Scheduler}_{\mathcal{M}}$ *denotiamo l'insieme di tutti i possibili scheduler di* $\mathcal{M}$ *e, per ogni scheduler* Σ*, indichiamo con* $\mathtt{path}^{\Sigma}_{s}$ *il sotto insieme di* $\mathtt{path}_s$ *che corrisponde a* Σ*.*

3 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, che, in caso di sistemi sequenziali, può portare al rintracciamento del $30 - 93\%$ degli errori. 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 questo problema. 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 1) 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 2).

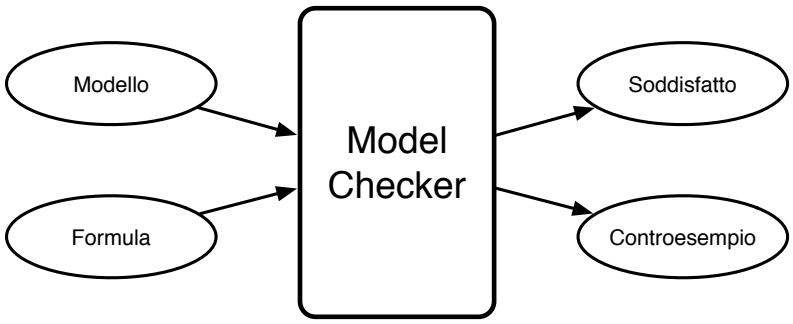

Figura 1: Schema del funzionamento di un model checker

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

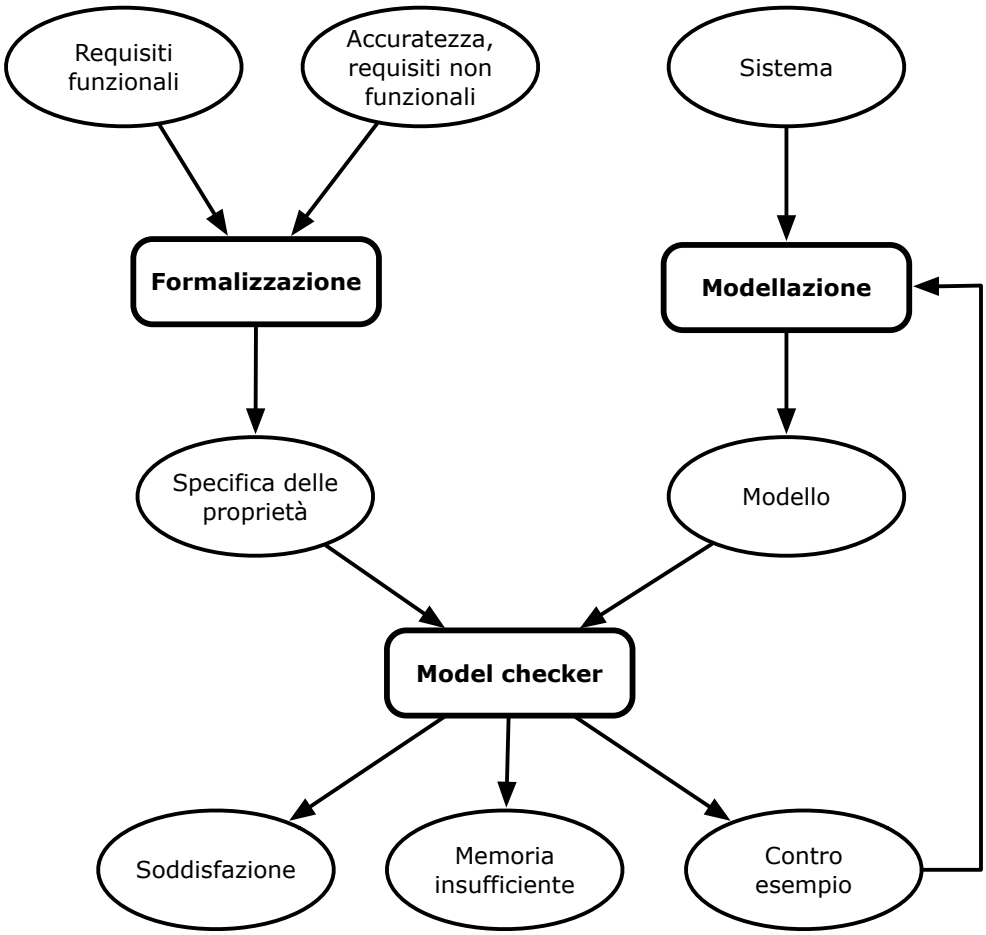

Figura 2: Schema del funzionamento di un model checker probabilistico

due tipi di probabilità elementari delle *DTMC*:

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

Definizione 25 (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 $s \in S$

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

. 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 26 (Probabilità steady state π_j) *La probabilità* steady state $\pi_j = \lim_{n\to\infty} \mathbb{P}\{X_n = j\}$ *è la probabilità che la* DTMC *sia nello stato* j *al passo a lungo andare.*

L'esistenza di questo limite è garantita solo sotto determinate condizioni di *ergodicità* della catena. L'esistenza del limite permette quindi di ricavare una distribuzione di probabilità steady state che è 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 $0 \leqslant \pi_i \leqslant 1$ e $1 \leqslant i \leqslant |S|$.

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 condiderando. 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. Il linguaggio *PCTL* viene infatti utilizzato per descrivere le formule in *PRISM*.

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

Definizione 27 (Sintassi *PCTL*) *La sintassi* PCTL *viene definita come segue:*

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

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

Dalla sintassi distinguiamo due tipi di formula: le formule di stato ϕ e le formule di percorso ψ. 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 ψ 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 percorso, l'operatore *next* $\mathcal{X}\phi$ richiede che ϕ venga soddisfatta dallo stato successivo, l'operatore *bounded until* $\phi_1 \mathcal{U}^{\leqslant k} \phi_2$ richiede che ϕ_2 venga soddisfatto entro k passi e che ϕ_1 resti sempre soddisfatto fino a quel punto, mentre l'operatore *unbonded until* $\phi_1 \mathcal{U} \phi_2$ richiede che prima o poi ϕ_2 venga soddisfatto e che ϕ_1 sia sempre soddisfatto fino a quel punto.

Per definire formalmente la semantica che abbiamo appena descritto è necessario specificare come gli elementi dell'insieme AP delle proposizioni atomiche sono gestiti in una *MDP*.

Definizione 28 (*MDP* etichettata) *Una* MDP *etichettata è una tupla* $(\mathcal{M}, L)$ *dove:*

- $\mathcal{M}$ *è una* MDP*;*
- $L : S \to 2^{AP}$ *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 29 (Relazione di soddisfacibilità) *Sia* $\mathcal{M} = (S, \overline{s}, \mathtt{Act}, \mathtt{Steps}, L)$ *una* MDP *etichettata. Per ogni stato* $s \in S$*, la relazione di soddisfacibilità di stato* $\models$ *è definita per induzione come segue:*

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

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

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

e per ogni percorso $\omega \in \mathtt{Path}$*:*

$$\begin{array}{lllll}
\mathcal{M}, \omega & \models_{pt} & \mathcal{X}\phi & \Leftrightarrow & \mathcal{M}, \omega(1) \models_{pt} \phi, \\
\mathcal{M}, \omega & \models_{pt} & \phi_1 \mathcal{U}^{\leqslant k} \phi_2 & \Leftrightarrow & \exists i \leqslant k \,.\, (\mathcal{M}, \omega(i) \models_{pt} \phi_2 \ e \ \mathcal{M}, \omega(j) \models_{pt} \phi_1 \forall j < i), \\
\mathcal{M}, \omega & \models_{pt} & \phi_1 \mathcal{U} \phi_2 & \Leftrightarrow & \exists k \geqslant 0 \,.\, \mathcal{M}, \omega \models_{pt} \phi_1 \mathcal{U}^{\leqslant k} \phi_2.
\end{array}$$

Possiamo rielaborare i costrutti principali definiti finora per estendere il linguaggio. Mentre gli operatori logici $\mathtt{false}$, $\vee$ e $\rightarrow$ possono essere derivati facilmente, vi sono alcuni operatori meno banali. Gli operatori $\Diamond$ e $\Box$ 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* $\Diamond^{\leqslant k}$ e $\Box^{\leqslant k}$ stabiliscono, tramite il numero di passi $k \in \mathbb{N}$, il tempo entro il quale la proprietà ϕ deve rimanere soddisfatta a partire dall'istante iniziale. La proprietà di cammino $\Diamond^{\leqslant k}\phi$ sarà quindi soddisfatta se entro k passi ϕ si verificherà almeno una volta, mentre la proprietà di cammino $\Box^{\leqslant k}$ sarà soddisfatta se ϕ rimarrà sempre soddisfatta per tutti e k i passi. 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. La definizione dei costrutti derivati appena descritti è riportata in tabella 1 dove $\overline{\leqslant} \equiv \geqslant$, $\overline{<} \equiv >$, $\overline{\geqslant} \equiv \leqslant$ e $\overline{>} \equiv <$.

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

Tabella 1: Operatori derivati di *PCTL*

3.3 PRISM MODEL CHECKER

Il model checker probabilistico che utilizzeremo è *PRISM* [5]. Si tratta di un tool con il quale è possibile fare modellazione e analisi di sistemi che presentano 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 evenutalmente 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*

```
module M1

x : [0..2] init 0;

    [] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
    [] x=1 & y!=2 -> (x'=2);
    [] x=2 -> 0.5:(x'=2) + 0.5:(x'=0);

endmodule

module M2

y : [0..2] init 0;

    [] y=0 -> 0.8:(y'=0) + 0.2:(y'=1);
    [] y=1 & x!=2 -> (y'=2);
    [] y=2 -> 0.5:(y'=2) + 0.5:(y'=0);

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 insierite saranno sempre abilitate insieme.

Listati 3.2: Scelta nondeterministica in *PRISM*

```
module M1

x : [0..2] init 0;

    // scelta nondeterministica
    [] x=0 -> (x'=0);
    [] x=0 -> (x'=1);

    [] x=1 & y!=2 -> (x'=2);
    [] x=2 -> 0.5:(x'=2) + 0.5:(x'=0);

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 $<$, $\leqslant$, $>$, $\geqslant$, e di equivalenza $=$ e $!=$
- gli operatori booleani di ! negazione, & congiunzione, | disgiunzione, $<=>$ se e solo se e $=>$ implicazione logica.

Sono presenti inoltre le funzioni:

- $\mathtt{min}$ minimo e $\mathtt{max}$ massimo,
- $\mathtt{floor}$ di arrotondamento all'intero minore più vicino,
- $\mathtt{ceil}$ di arrotondamento all'intero maggiore più vicino,
- $\mathtt{pow}$ per l'elevamento a potenza,
- $\mathtt{mod}$ modulo,
- $\mathtt{log}$ 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 || 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, ...]|| M2: la sincronizzazione avviene solo sulle azioni specificate nell'operatore,
- M/{a, b, ...} le azioni specificate vengono nascoste all'esterno,
- M{a < −b, c < −d, ...} le azioni vengono rinominate all'esterno.

Parte II

LAPSA: UN LINGUAGGIO PER AGENTI ADATTIVI

4

LAPSA

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 3). L'ambiente può essere costituito da più agenti o altri tipi di sistemi di cui si può avere una conoscenza solamente parziale o nulla.

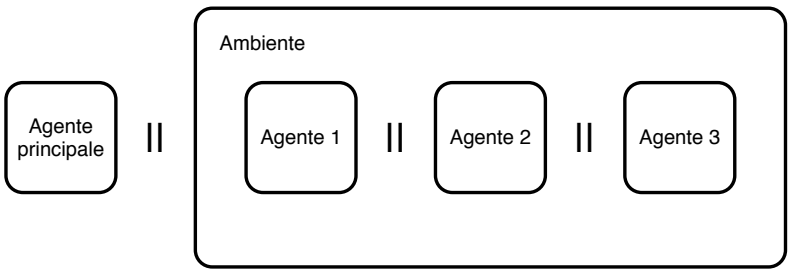

Figura 3: Schema di come viene modellato uno scenario attorno all'agente che lo analizza

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* [9], 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 non 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 (tabella 2).

program	::=	**actions** {*actions*}
		subject *module*
		modules
		environment
		ranges {*ranges*}

Tabella 2: Sintassi *LAPSA* di *program*

Il non terminale *actions* è una semplice lista dove vengono dichiarate i nomi delle azioni che possono essere effettuate (tabella 3). 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*.

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

Tabella 3: Sintassi *LAPSA* di *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 (tabella 4). 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.

$$\begin{aligned}
\textit{module} &::= \textbf{module}\ \text{module-id}\ \{\textit{variables rules targets}\} \\
\textit{variables} &::= \text{type variable-id} = \textit{expression}; \quad | \quad \textit{variables variables} \\
\textit{rules} &::= \textit{condition}[\text{action-id}] \Rightarrow \textit{distribution}; \quad | \quad \textit{rules rules} \\
\textit{targets} &::= \textbf{target never}\ \textit{condition} \quad | \quad \textit{targets targets}
\end{aligned}$$

Tabella 4: Sintassi *LAPSA* di *module* e delle sezioni che lo compongono

Le distribuzioni di probabilità (tabella 5) 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.

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

Tabella 5: Sintassi *LAPSA* di *distribution*

Ogni caso di una distribuzione porta a una lista di aggiornamenti che possono essere descritti con i costrutti sintattici definiti in tabella 6. Si può modificare il valore di una variabile tramite assegnamento o specificare che nessun cambiamento verrà eseguito tramite l'operazione nulla `noaction`. Per mezzo degli operatori `env.add` e `env.remove` è possibile agire sull'ambiente aggiungendo e rimuovendo elementi rispettivamente.

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

Tabella 6: Sintassi *LAPSA* di *update*

Il non terminale *modules* descritto, assieme ad *environment*, in tabella 7 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* [4].

$$\begin{aligned} \textit{modules} \quad & ::= \quad \textit{module} \quad | \quad \textit{modules modules} \\ \textit{environment} \quad & ::= \quad \text{module-id} \quad | \quad \textit{environment} \ |\{\textit{actions}\}| \ \textit{environment} \end{aligned}$$

Tabella 7: Sintassi *LAPSA* di *modules* e di *environment*

Le condizioni hanno un ruolo importante in quanto parte fondamentale dell'abilitazione delle transizioni. La sintassi di una condizione è illustrata in tabella 8 e 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.

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

Tabella 8: Sintassi *LAPSA* di *condition*

Il simbolo $\bowtie$ rappresenta l'operatore di confronto e può essere quindi definito come $\bowtie = \{<, \leqslant, >, \geqslant, =, \neq\}$. Le espressioni (tabella 9) 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.

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

Tabella 9: Sintassi *LAPSA* di *expression*

Infine il non terminale *ranges* (tabella 10) 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.

In tabella 11 viene riportata la sintassi completa di *LAPSA* i cui costrutti possono essere estesi

$$\textit{ranges} ::= \textit{reference } \textbf{in}[\text{constant}, \text{constant}] \quad | \quad \textit{ranges}, \textit{ranges}$$

Tabella 10: Sintassi *LAPSA* di *ranges*

4.2 ZUCCHERO SINTATTICO

La sintassi presentata in tabella 11 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 *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 α come un variable-id, γ come un module-id e $\beta, \beta' \in$ *condition*

$$\textbf{false} \equiv \textbf{not true}$$

$$\beta \textbf{ and } \beta' \equiv \textbf{not } (\textbf{not } \beta \textbf{ or not } \beta')$$

$$\textbf{forall } \alpha : \gamma \textbf{ such that } \beta \equiv \textbf{not exists } \alpha : \gamma \textbf{ such that not } \beta$$

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

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

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

Tabella 11: Sintassi di *LAPSA*

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 α 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 facilmente 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

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 distrubuzioni di probabilità aventi come supporto l'insieme degli stati stessi.

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

$$M = (\Sigma, Act, \rightarrow_\rho, \sigma_0)$$

Le parti che compongono la *MDP* sono descritte in seguto, 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 é l'insieme delle azioni action-id

- $\rho \subseteq \textit{condition} \times \mathtt{Act} \times \mathtt{Dist}(\mathcal{U})$ è la *struttura statica* della *MDP* definita come

$$\begin{aligned}\rho \quad = \quad & \{(g, a, d) \mid \text{“}g[a] \Rightarrow < e_1 > \alpha_1 \# \ldots \# < e_n > \alpha_n \text{ ”} \in \textit{rule}, \\ & d = [u_{\alpha_1} : p_1, \ldots, u_{\alpha_n} : p_n]\}\end{aligned}$$

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

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

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

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

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

- $\rightarrow_\rho \subseteq \Sigma \times \mathtt{Act} \times \mathtt{Dist}(\mathcal{U})$ 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 $\Pi \in \mathtt{Dist}(S)$ 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

$$S ::= \textsf{module-id} \quad | \quad S_1 \; |\{\textit{actions}\}| \; S_2$$

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

$$\textsf{subject-id} \; |\{\textit{Act}\}| \; \textit{environment}$$

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 σ_m e ρ_m per indicare rispettivamente lo stato e la struttura statica del generico modulo $m \in \textit{module}$ e l'insieme $A \subseteq \mathtt{Act}$.

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.

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

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

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

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

$$\Pi_1 \; |\{A\}| \; S_2(S) = \begin{cases} \Pi_1(S_1') & \text{se } S = S_1' \; |\{A\}| \; S_2 \\ 0 & \text{altrimenti} \end{cases}$$

$$S_1 \; |\{A\}| \; \Pi_2(S) = \begin{cases} \Pi_2(S_2') & \text{se } S = S_1 \; |\{A\}| \; S_2' \\ 0 & \text{altrimenti} \end{cases}$$

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

L'ultima 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, π) dove M è la *MDP* che descrive il sistema globale e π 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 12 viene data la semantica denotazionale tramite la definizione della funzione $\tau : \textit{module} \times \textit{module} \times \textit{targets}$ per l'obiettivo, $\gamma : \textit{module} \times \textit{module} \times \textit{condition}$ per le condizioni ed $\epsilon : \textit{module} \times \textit{module} \times \textit{expression}$ per le espressioni.

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:

$$
\begin{array}{l}
\Phi^{t}_{m}(\textbf{target never } c) = P_{max=?}[G_{\leqslant k}\,!''\tau^{t}_{m}(c)''] \\
\tau^{t}_{m}(\textbf{exists var} : \text{mod } \textbf{such that } c) = \tau^{t}_{m_1}(c) \textbf{ or } \ldots \textbf{ or } \tau^{t}_{m_n}(c) \\
\tau^{t}_{m}(\textbf{true}) = \textbf{true} \\
\tau^{t}_{m}(c_1 \textbf{ or } c_2) = \tau^{t}_{m}(c_1) \textbf{ or } \tau^{t}_{m}(c_2) \\
\tau^{t}_{m}(\textbf{not } c) = !\,\tau^{t}_{m}(c) \\
\tau^{t}_{m}(e_1 \bowtie e_2) = \epsilon^{t}_{m}(e_1)\tau^{t}_{m}(\bowtie)\epsilon^{t}_{m}(e_2)
\end{array}
$$

Tabella 12: Semantica denotazionale del target del modulo principale

- 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>
               y=y+1 #
9          <1, not exists bot:RandomWalkRobot such that bot.x=x and bot.y=y-1>
               y=y-1 #
10         <1, not exists bot:RandomWalkRobot such that bot.x=x+1 and bot.y=y>
               x=x+1 #
```

```
11          <1, not exists bot:RandomWalkRobot such that bot.x=x-1 and bot.y=y>
                x=x-1 #
12          <1, true> noaction;
13      }
```

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] =>
           <1> y=y+1;
8      not exist bot:RandomWalkBot such that bot.x=x and bot.y=y-1 [step] =>
           <1> y=y-1;
9      not exist bot:RandomWalkBot such that bot.x=x+1 and bot.y=y [step] =>
           <1> x=x+1;
10     not exist bot:RandomWalkBot such that bot.x=x-1 and bot.y=y [step] =>
           <1> x=x-1;
11     true [step] => <1> noaction;
12 }
```

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 4). 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;
```

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;
```

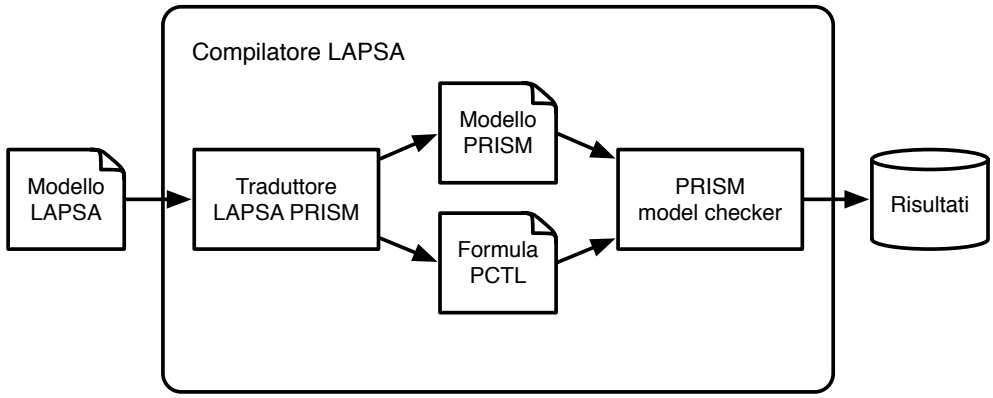

Figura 4: Schema del compilatore *LAPSA*

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 **??** il compilatore *LAPSA* 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*.

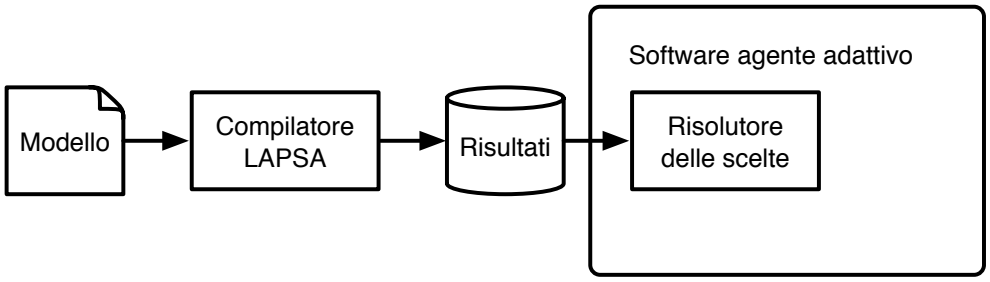

Figura 5: Utilizzo dei risultati nell'agente adattivo

Il compilatore del linguaggio *LAPSA* è stato implementato tramite il tool *Xtext* [9]. 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 4.3 e segue i costrutti linguistici dati in tabella 11 ad eccezione di alcune modifiche strettamente legate all'implementazione.

Listati 4.3: 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
```

```
Type:
    name='int' | name='float' | name='bool'
;

Rule:
    cond=Expression '[' action=[Action] ']' (ndCases+=NDCase)+
;

NDCase:
    '=>' cases+=Case ('#' cases+=Case)*
;

Case:
    '<'weight=Expression (hasCondition?=',' cond=Expression)? '>' update+=
        Update (',' update+=Update)*
;

Update:
    { NoAction } 'noaction' |
    { Assign } variable=[VariableDeclaration] '=' expr=Expression
;

Environment:
    modules+=[ModuleDefine] ('|{' (actions+=[Action])+ '}|' modules+=[
        ModuleDefine])*
;


// === expression syntax ===

Expression:
    Logical;

Logical returns Expression:
    Relation
        (({And.left=current} 'and'
        | {Or.left=current} 'or')
        right=Relation)*;

Relation returns Expression:
    Addition
        (({Leq.left=current} '<='
        | {Less.left=current} '<'
        | {Eq.left=current} '=='
        | {Neq.left=current} '!='
        | {Geq.left=current} '>='
        | {Gtr.left=current} '>')
        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'
            cond=PrimaryExpression
100     | { ExternalReference } module=[Quantifier] '.' variable=[
            VariableDeclaration]
101     | { LocalReference } ('this' '.')? variable=[VariableDeclaration]
102 ;
103
104 Value: INT | FLOAT | BOOL;
105
106 terminal INT returns ecore::EInt: ('0'..'9')+;
107
108 terminal FLOAT returns ecore::EDouble:
109   ('-')? (INT)* ('.' (INT)+)? |
110   ('-')? (INT)+ ('.') |
111   ('-')? (INT)+ ('.' (INT)*)? (('e'|'E')('-'|'+')? (INT)+);
112
113 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 4.4: 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).
              prismCompileExpression.toString)
18
19        // serialize hashmap
20        var objOut = new ObjectOutputStream(
21            new FileOutputStream(Commons::getSrcGenURI(resource)+"
                  hashmtable.ser")
22        )
23        objOut.writeObject(fh.index);
24        objOut.close();
25    }
```

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 4.5 vengono mostrate le funzioni che traducono il nodo *program* e i nodi *module* in condice *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 4.5: Implementazione della funzione di generazione 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>>
```

```
            '''
        }
        tpl
    }

    def prismCompile(ModuleDefine m){
        moduleCounter=moduleCounter+1
        '''
        module module_<<moduleCounter>>
        // variables
        <<FOR v:m.variables>>
            <<v.prismCompile>>
        <<ENDFOR>>
        // rules
        <<FOR r:m.rules>>
            <<r.prismCompile>>
        <<ENDFOR>>
        endmodule
        '''
    }
```

La dichiarazione di variabili viene implementata come descritto nel listato 4.6, 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 δ (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 4.6: Traduzione della dichiarazione di variabili da *LAPSA* a *PRISM*

```
    def prismCompile(VariableDeclaration v){
        var from = 0
        var to = 1
        var delta = 1

        // look at ranges and get bounds
        for(r:(v.eContainer.eContainer as Program).ranges){
            if(r.variable==v){
                to = Integer::parseInt(r.to)
                from = Integer::parseInt(r.from)
                if(r.delta != null){
                    delta = Integer::parseInt(r.delta)
                }
```

```
            }
        }
        '''
        <<v.localName>> :
        <<IF v.type.name.equals('bool')>>
            bool init <<v.expr.prismCompileExpression>>;
        <<ENDIF>>
        <<IF v.type.name.equals('int')>>
            [<<from>>..<<to>>] init <<v.expr.prismCompileExpression>>;
        <<ENDIF>>
        <<IF v.type.name.equals('float')>>
            [0..floor((<<to>>-<<from>>)/<<delta>>)] init
            ceil((<<v.expr.prismCompileExpression>>-<<from>>)/<<delta>>);
        <<ENDIF>>
        '''
    }
```

Come ultima procedura descriviamo la traduzione delle transizioni da *LAPSA* a *PRISM* (listato 4.7). 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 4.7: Traduzione delle transizioni da *LAPSA* a *PRISM*

```
    def prismCompile(Rule r){
        val baseCondition = '''
            [<<r.action.name>>] <<r.cond.prismCompileExpression>>'''
        // add unconditioned cases and true cases
        '''
        <<FOR ndcase:r.ndCases>>
        <<FOR comb:ndcase.cases.reduced.allCombinations
            SEPARATOR ';' AFTER ';'>>
        <<baseCondition>>
        <<FOR c:ndcase.cases.filter(c|c.hasCondition)
            BEFORE '&' SEPARATOR '&'>>
            <<IF !comb.contains(c)>>!<<ENDIF>>(<<c.cond.
                prismCompileExpression>>)
```

```
        <<ENDFOR>>
        ->
        <<IF comb.size==0 && ndcase.cases.filter(c | !c.hasCondition).size
            ==0>>
            1 : true
        <<ELSE>>
            <<FOR c : ndcase.cases.filter(c | !c.hasCondition
                || comb.contains(c)) SEPARATOR '+'>>
                (<<c.weight.prismCompileExpression>>) /
                (<<ndcase.totalWeight(comb)>>) :
                <<FOR u:c.update SEPARATOR '&'>>
                    <<u.prismCompileUpdate>>
                <<ENDFOR>>
            <<ENDFOR>>
        <<ENDIF>>
        <<ENDFOR>>
        <<ENDFOR>>
        '''
    }
```

Parte III

CASO DI STUDIO

5 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.

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 globlale 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 $Z : \mathbb{N}_0 \to \mathbb{N}^{m \cdot m}$ 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:

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

.

Il criterio seguito dal generico agente sarà quindi l'algoritmo 1. I parametri dell'algoritmo hanno il seguente significato:

Algorithm 1 Algoritmo di scelta generico

Require: $m, n, nrob \in \mathbb{N} \wedge i, j \in \{1, \ldots, m\} \wedge Z \in \mathbb{N}_0^{m \cdot m}$
  $p_0 = (i, j)$
  for $k = 1$ **to** n **do**
    $p_k = schedule(local(Z, p_{k-1}, m), m, nrob)$
    barriera
    $Z_{p_{k-1}} = Z_{p_{k-1}} - 1$
    $Z_{p_k} = Z_{p_k} + 1$
    barriera
  end for

- 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 e 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

```
actions { a }

subject module NDRobot {
    // variabili area locale
    int p1 = 0; int p2 = 0; int p3 = 0;
    int p4 = 0; int p5 = 0; int p6 = 0;
    int p7 = 0; int p8 = 0; int p9 = 0;

    // transizioni
    p1+p2+p4+p5+p6+p7+p8+p9 == 0 and p3 = 1 [a]
    => // resta fermo
        <0.2> noaction #
        <0.2> p3 = 0, p6 = 1 #
        <0.4> p3 = 0 #
        <0.2> p2 = 1, p3 = 0
    => // vai a nord
        <0.2> noaction #
        <0.2> p3 = 0, p5 = 1 #
        <0.2> p3 = 0, p6 = 1 #
        <0.2> p3 = 0 #
        <0.2> p3 = 0, p9 = 1
    => // vai a ovest
        <0.2> noaction #
        <0.8> p3 = 0
    => // vai a est
        <0.2> noaction #
        <0.2> p3 = 0, p5 = 1 #
        <0.2> p1 = 1, p3 = 0 #
        <0.2> p3 = 0 #
        <0.2> p2 = 1, p3 = 0
    => // vai a sud
        <0.2> noaction #
        <0.8> p3 = 0;

    // ...

    // obiettivo
    target never p5 > 0
}

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

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_{step} *di passare da* $current$ *a* $next$ *con l'azione* act
    $p_{MC} = index(next)$
    $prob = p_{step} \cdot p_{MC}$
    if $prob > max$ **then**
      $max = prob$
      $best\_act = act$
    end if
  end for
end for
return $best\_act$

un solo livello di 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.

	1 semi-random	2 semi-random	3 semi-random
semi-random	4.22	7.67	11.98
model-checker 1	2.87	6.52	10.02
model-checker 2	2.82	6.36	9.88
model-checker 3	2.44	6.27	9.82

Tabella 13: Risultati delle simulazioni

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:

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

In tabella 13 vengono riportate le medie dei test effettuati, raffigurate nel grafico 6. 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

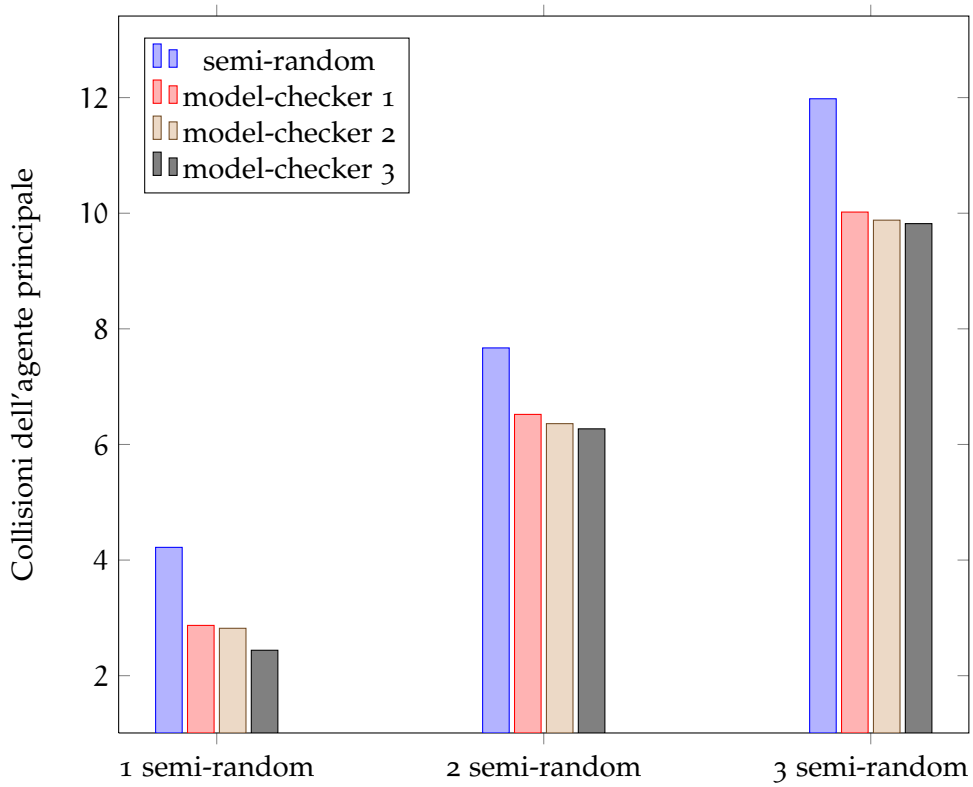

Figura 6: Grafico dei risultati delle simulazioni: sull'ascissa variano gli agenti che sono realmente nell'area assieme all'agente principale, sull'ordinata viene indicato il numero di collisioni e i diversi colori rappresentano di diversi scheduler utilizzati dall'agente principale.

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 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: 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

6 CONCLUSIONI

conclusioni...

BIBLIOGRAFIA

[1] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembling strategies with maude. *ASCENS*, 2012.

[2] Roberto Bruni, Andrea Corradini, Alberto Lluch Lafuente, Fabio Gadducci, and Andrea Vandin. A conceptual framework for adaptation. *LNCS*, 7212:240–254, 2012.

[3] Edmond Gjondrekaj, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A robot foraging scenario in klaim. 2011.

[4] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[5] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. `http://www.prismmodelchecker.org/`, 2011.

[6] Terence J Parr, T. J. Parr, and R W Quong. Antlr: A predicated-ll(k) parser generator. `http://www.antlr.org/`, 1995.

[7] 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.

[8] Xtend. `http://www.eclipse.org/xtend/`.

[9] Xtext. `http://www.eclipse.org/Xtext/`.