pseudo keywords=procedure,end,return,integer,real,foreach,for,each, do,if,then,else,while,until,tru
comment=[l]// lapsa morekeywords=subject, module, noaction, int, float,
bool, and, or, not, exists, target, never, always, actions, environment, is,
empty, ranges, in, morecomment=[l]//, morecomment=[s]/**/, morestring=[s]'',
morestring=[s]""

oaw morekeywords=import, this, create, let, then, Void, extension, JAVA,
IMPORT, DEFINE, ENDDEFINE, LET, ENDLET, FOR, FILE, END-
FILE, ITERATOR, FOREACH, AS, IF, ENDFOREACH, ENDIF, EX-
PAND, INSTANCEOF, USING, SEPARATOR, CSTART, CEND, PRO-
TECT, ENDPROTECT, ID, EXTENSION, context, ERROR, WARNING,
INFO, enum, morecomment=[l]//, morecomment=[s]/**/, morecomment=[s]≪REM≫≪EN-
DREM≫, morecomment=[s]≪REM≫, morestring=[s]'', morestring=[s]"", esca-
pechar=@, literate=≪≪1 ≫≫1

shell alsodigit=¿, morekeywords=linux¿, backgroundcolor=

csimul alsodigit=¿, alsodigit=%, morekeywords=CONDITIONER¿, COUN-
TER¿, %, backgroundcolor=, morecomment=[l]#

Esterel morekeywords= abort, and, assert, await, bool, call, case, clock,
combine, constant, data, do, doup, dodown, dopar, each, emit, else, elseif,
end, every, exec, exit, extends, for, function, generic, goto, halt, handle, if,
immediate, in, input, inputoutput, integer, interface, loop, map, module,
multiclock, mux, not, nothing, or, output, pause, positive, pre, present,
procedure, relation, repeat, return, run, sensor, seq, signal, signed, suspend,
sustain, task, then, times, trap, type, unsigned, var, weak, when, with,
morecomment=[l]%

Lustre keywords= and, assert, bool, current, else, float, if, int, let,
node, not, or, pre, returns, tel, then, var, when , literate=-–2 -¿→ 2 ,
morecomment=[l]–,

KEP morekeywords= ABORT, ABORTI, ADD, ADDC, AWAIT, AWAITI,
AWAITN, CALL, CAWAIT, CAWAITE, CMP, CMPS, EMIT, EMITV,
EMITD, EMITR, EXIT, GOTO, HALT, INPUT, JOIN, JW, LABORT,
LABORTI, LWABORT, LWABORTI, LOAD, MUL, NOTHING, OUTPUT,
PAUSE, PAR, PARE, PRE, PRIO, PRESENT, SETV, SIGNAL, SIGNALV,
SUSPEND, SUSPENDI, SUSTAIN, SUSTAINV, SUSTAIND, SUSTAINR,
TABORT, TABORTI, TWABORT, TWABORTI, WABORT, WABORTI,
, morecomment=[l]//, morecomment=[s]/**/, morecomment=[l]%, more-
string=[b]"

KLP morekeywords= ADD, ADDI, AND, DIV, DIVI, DONE, EQ, GOTO,
IADD, IAND, IDIV, IEQ, IGE, IGT, ILE, ILT, INITC, INITV, INPUT,
ICMOV, IRMOV, ISUB, IVMOV, IOR, IXOR, IMUL, JF, JMP, JNZ, JT,
JZ, LOCAL, MUL, MULI, NEQ, OUTPUT, PRIO, RRMOV, SETCLK,
SETPC, SUB, SUBI, VVMOV, XOR , morecomment=[l]//, morecomment=[l]#,
morecomment=[s]/**/, morecomment=[l]%, morestring=[b]",

Kit morekeywords=statechart, model, version, input, output, var, la-
bel, type, history, hi, deephistory, dh, initial, in, fork, fk, join, jn, junction,
jc, sync, sy, choice, ch, dynamicchoice, dc, suspend, sd, final, doActivity,
do, entryActivity, entry, exitActivity, exit, bindActivity, bind, localEvent,
localVariable, priority, weakAbortion, wa, strongAbortion, sa, normalTer-
mination, nt, suspension, sp, conditional, co, internalTransition, it, integer,
float, double, boolean, combine, with, pos, pp, lp, width, height, collapsed,
true, false, morecomment=[l]#,

Repic morekeywords= abort, await, chkabort, emit, fork, goto, jmp, join,
ldaaddr, present, sync, morecomment=[l]%

tinybasicstyle= scriptsizebasicstyle= footnotesizebasicstyle= smallbasic-

Model checking come supporto per le scelte di sistemi adattivi

Marco Tinacci

14 settembre 2012

Indice

Capitolo 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 [**?**]. 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?*"[**?**]. 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 [**?**] assemblandosi in gruppi richiedono una collaborazione esplicita mentre per la raccolta di risorse [**?**] 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 vei-

coli 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* [**?**], 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!** (**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 *probabilistic computation tree lo-*

gic (*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* [**?**], 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* [**?**] vengono costruiti automaticamente lexer e parser basati su *ANTLR* [**?**]. 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.

Capitolo 2

Probabilità

In questo capitolo saranno forniti gli strumenti necessari alla comprensione del lavoro.

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* $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$*.*

2.1 Probabilità

Definizione 8 (Assiomi di Kolmogoroff)**.** *Dato lo spazio misurabile* $(\Omega, \mathcal{F})$*, una* misura di probabilità *su di esso è una funzione* $\mathbb{P} : \mathcal{F} \to \mathbb{R}_{\geq 0}$ *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. $\forall A, B \in \mathcal{F} : A \subseteq B \Rightarrow \mathbb{P}A \leq \mathbb{P}B$,
3. $\forall A \in \mathcal{F} : \mathbb{P}A \leq 1$,
4. $\forall A, B \in \mathcal{F} : \mathbb{P}(A \cup B) \geq \max\{\mathbb{P}A, \mathbb{P}B\}$,
5. $\forall A, B \in \mathcal{F} : \mathbb{P}A \cap B \leq \min\{\mathbb{P}A, \mathbb{P}B\}$,
6. $\forall A, B \in \mathcal{F} : \mathbb{P}(A \cup B) = \mathbb{P}A + \mathbb{P}B - \mathbb{P}(A \cap B)$,
7. $\forall A, B \in \mathcal{F} : A \subseteq B \Rightarrow \mathbb{P}(B \backslash A) = \mathbb{P}B - \mathbb{P}A$,
8. $\forall A_i \in \mathcal{F} : \mathbb{P}\left(\bigcup_{i=0}^{\infty} A_i\right) \leq \sum_{i=0}^{\infty} \mathbb{P}A_i$.

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

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

o alternativamente

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

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

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

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

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

2.2 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.

2.3 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* $T = \mathbb{R}$

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

2.4 Discrete-Time Markov Chains

Definizione 16 (*Discrete Time Markov Chain* (*DTMC*))**.** *Una* DTMC *è una tupla* $\mathcal{D} = (S, \overline{s}, \mathbb{P})$ *dove:*

- S *è un insieme finito di* stati*;*
- $\overline{s} \in S$ *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)**.** $paths^{\mathcal{D}}_{s_0}$ *o* $paths(\mathcal{D}, s_0)$*, 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,
- $last(\sigma)$ indica l'ultimo stato del path finito σ e
- σ_{fin} indica l'insieme di tutti i path finiti.

2.5 Markov Decision Processes

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

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

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

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)**.** $paths^{\mathcal{M}}_{s_0}$ *o* $paths(\mathcal{M}, s_0)$*, 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,
- $last(\sigma)$ indica l'ultimo stato del path finito σ e
- σ_{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* σ_{fin} *di* $\mathcal{M}$ *in un elemento dell'insieme* $Steps(last(\sigma_{fin}))$ *indicato con* $S(\sigma_{fin})$*. Con* $Scheduler_{\mathcal{M}}$ *denotiamo l'insieme di tutti i possibili scheduler di* $\mathcal{M}$ *e, per ogni scheduler* Σ*, indichiamo con* $path^{\Sigma}_{s}$ *il sotto insieme di* $path_s$ *che corrisponde a* Σ*.*

TODO risolvere uno scheduler permette di trasformare una mdp in una dtmc

2.6 Model Checking

TODO introduzione e schemi su model checking probabilistico e non

2.6.1 Probabilità elementari

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

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

2.6.2 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*).

2.6.3 Sintassi di PCTL

2.6.4 Semantica di PCTL

2.7 PRISM

Capitolo 3

LAPSA

3.1 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.

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, gli obiettivi del soggetto, i moduli che possono essere usati per descrivere l'ambiente e i dati necessari alla discretizzazione delle variabili.

In tabella 3.1 viene descritta anche la sintassi dell'insieme delle azioni, degli obiettivi e dell'ambiente. Le azioni e l'ambiente sono una semplice lista, rispettivamente, di *label* e di *module*. La lista di moduli di ambiente non esprime direttamente quali moduli sono considerati presenti nell'ambiente ma quelli che possono esserci inseriti.

La definizione di un modulo (tabella 3.1) consiste nell'attribuzione di un nome che lo identifica, un insieme di variabili che ne descrive lo stato, e un insieme di regole che ne descrive l'avanzamento. Le variabili vengono dichiarate all'inizio del modulo specificandone il tipo, il nome e il valore iniziale attribuito. Una regola, invece, è composta da una condizione di guarda che la abilita, un'azione da eseguire, e una distribuzione di possibli funzioni di *update*, ognuna delle quali ha associata una probabilità di essere eseguita. Le probabilità della distribuzione vengono determinate dalla normalizzazione dei pesi associati alle funzioni. La funzione *update* è una sequenza di azioni di aggiornamento di stato o di visione dell'ambiente.

- A
- $\bowtie$

module	::=	**module** module-name{*variables rules*}
variables	::=	type id = *expression*; | *variables variables*
rules	::=	*condition*[action-label] ⇒ *distribution* | *rules*, *rule*
condition	::=	**E** id : module-name . *condition* | *expression* ⋈ *expression*
	|	**true** | *condition* **or** *condition* | ! *condition*
distribution	::=	< *expression* > *update*; | *distribution* # *distribution*
update	::=	id = *expression* | **noaction** | *update* ; *update*
	|	**env.add** *system* | **env.remove** id : module-name . *condition*

Tabella 3.1: Sintassi di un modulo **lapsa!**

- id
- module-name
- action-label
- type
- value

L'insieme dei moduli sarà quindi del tipo

$$M \subseteq \gamma \times \mathbb{VAR} \times \cdots \times \mathbb{VAR}$$

3.2 Zucchero sintattico

Inseriamo un costrutto tale da poter inserire una condizione di abilitazione sugli elementi del supporto della distribuzione:

$$\beta[a] \Rightarrow < e, \beta' > \alpha \oplus \delta \equiv \beta \wedge \beta'[a] \Rightarrow < e > \alpha \oplus \delta, \beta \wedge \neg\beta'[a] \Rightarrow \delta$$

3.3 Semantica

Andiamo a dare un'introduzione informale alla semantica del linguaggio descritto:

- *System*: un sistema può essere definito tramite un singolo modulo (m è un riferimento alla definizione di un modulo) o attraverso la composizione parallela di più sistemi su di un insieme di azioni di sincronizzazione $A \subseteq Act$;

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

Tabella 3.2: Sintassi completa di **lapsa!**

- *Rule*: un insieme di regole definisce il comportamento di un modulo, se vale la condizione β si può passare alla valutazione della distribuzione δ;
- *Condition*: descrive una condizione fornendo anche un operatore di quantificatore esistenziale sui moduli;
- *Distribution*: descrive una distribuzione probabilistica di azioni α, dove ogni azione è accompagnata da un'espressione e, che ne descrive il peso, e un'azione a;
- *Action*: un azione descrive un aggiornamento dello stato, che può consistere nell'assegnamento di una, nessuna, o più variabili.

Definiamo la semantica del linguaggio in termini di *MDP*. Alla definizione di ogni modulo sarà assegnata una *MDP* della forma:

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

dove

- $\Sigma = \{\sigma | \sigma : \mathbb{VAR} \rightarrow \mathbb{VAL}\}$ è l'insieme degli *stati* rappresentati da funzioni che mappano variabili in valori,
- *Act* l'insieme delle azioni,
- $\rightarrow_\rho \subseteq \Sigma \times Act \times Dist(U)$ è la relazione di *avanzamento* di stato,
- $\sigma_0 \in \Sigma$ è lo *stato iniziale*,
- $\rho \subseteq \beta \times Act \times Dist(U)$ è la *struttura statica* del *MDP*,
- $U = \{u | u : \Sigma \rightarrow \Sigma\}$ è l'insieme delle funzioni *update* di aggiornamento di stato.

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

$$\rho_m = \{(g, a, d) \mid \gamma_m = g[a] \Rightarrow < e_1 > \alpha_1 \oplus \cdots \oplus < e_n > \alpha_n, d = [u_{\alpha_{i1}} : p_1, \ldots, u_{\alpha_{in}} : p_n]\}$$

dove $u_\alpha \in U$ è una funzione *update* definita nel seguente modo

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

Le probabilità sono invece calcolate nel seguente modo

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

Salendo dal livello dei moduli a quello dei sistemi, introduciamo $\Pi \in Dist(S)$ per indicare distribuzioni di sistemi. Il sistema sarà rappresentato dalla *MDP* risultante dal parallelo di quelle che lo compongono.

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

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

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

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

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

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

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

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

3.4 Esempi

Esempio di un modulo di robot che esegue una *random walk* su una griglia escludedo dalla scelta probabilistica le direzioni adiacenti occupate:

$$\begin{aligned} m_1() \triangleq\ & \mathbf{true}[step] \Rightarrow \\ & < 1, \neg\, \exists\ m_1\ v : v.x = x\ \mathbf{and}\ v.y = y + 1 > y = y + 1; \# \\ & < 1, \neg\, \exists\ m_1\ v : v.x = x\ \mathbf{and}\ v.y = y - 1 > y = y - 1; \# \\ & < 1, \neg\, \exists\ m_1\ v : v.x = x + 1\ \mathbf{and}\ v.y = y > x = x + 1; \# \\ & < 1, \neg\, \exists\ m_1\ v : v.x = x - 1\ \mathbf{and}\ v.y = y > x = x - 1; \# \\ & < 1, \mathbf{true} > \mathbf{noaction}; \end{aligned}$$

Esempio di un modulo di robot analogo al precedente con la differenza che la scelta della mossa viene fatta in modo nondeterministico:

$$
\begin{array}{lcllcl}
m_2() & \triangleq & \neg\, E\ m_1 : v.(v.x = x \wedge v.y = y+1) & [north] & \Rightarrow & < 1 > y = y+1; \\
 & & \neg\, E\ m_1 : v.(v.x = x \wedge v.y = y-1) & [south] & \Rightarrow & < 1 > y = y-1; \\
 & & \neg\, E\ m_1 : v.(v.x = x+1 \wedge v.y = y) & [east] & \Rightarrow & < 1 > x = x+1; \\
 & & \neg\, E\ m_1 : v.(v.x = x-1 \wedge v.y = y) & [west] & \Rightarrow & < 1 > x = x-1; \\
 & & \textbf{true} & [stay] & \Rightarrow & < 1 > \textbf{noaction};
\end{array}
$$

3.5 Traduzione del codice

3.6 PRISM

La traduzione del sorgente **lapsa!** in codice *PRISM* necessita di alcuni dati aggiuntivi riguardo le variabili dei moduli. Dato che *PRISM* lavora con uno spazio degli stati finito è necessario aggiungere informazioni che permettano di trattare le variabili intere e reali. Per le variabili intere sarà sufficiente specificare il range, mentre per le variabili reali dovrà essere specificata anche l'ampiezza dell'intervallo di discretizzazione.

La traduzione delle variabili viene descritta in tabella 3.3. Con `a`, `b` e `delta` rappresentiamo costanti intere, con `e` un'espressione **lapsa!** e con `e'` la rispettiva traduzione in *PRISM*. I dati necessari alla discretizzazione vengono attualmente forniti direttamente nel file *PRISM* nella sezione *ranges*.

lapsa!	*Input file*	*PRISM*

`module m{`
`...`
`bool x = e;`
`...`
`}`
	`-`	`x:bool init e';`

`module m {`
`...`
`int y = e;`
`...`
`}`
	`m.y in (a,b);`	`y:[a..b] init e';`

`module m {`
`...`
`float z = e;`
`...`
`}`
	`m.z in (a,b,delta);`	`z:[0..floor((a-b)/delta)]`
		`init ceil((e'-a)/delta);`

`z = e`	`m.z in (a,b,delta);`	`z' = ceil((e'-a)/delta)`

Tabella 3.3: Traduzione da **lapsa!** a *PRISM*

Capitolo 4

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.

4.1 Analisi

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

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

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

Per analizzare la successione temporale dello scenario possiamo parametrizzare la matrice Z rispetto al tempo definendola come una funzione $Z : \mathbb{N}_0 \to \mathbb{N}^{m \cdot m}$ 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 para-

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

metri dell'algoritmo hanno il seguente significato:

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

La funzione *local* viene così definita

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

Il primo sottoinsieme di indici è definito come

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

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

La funzione *schedule* dipende invece dal comportamento che si vuole associare all'agente e quindi che criterio utilizzerà per risolvere le scelte. Lo scheduler avrà quindi pochi dati su cui prendere una decisione e se si

esclude anche una memorizzazione dello storico allora il dominio di *schedule* diventa il numero di combinazioni di un massimo di *nrob* agenti all'interno delle posizioni locali. La conoscenza dell'agente si limita quindi alle posizioni locali $local(Z, i, j, m)$, il numero totale di agenti in gioco *nrob* e la dimensione dell'area m.

Le *barriere* sono il costrutto di programmazione parallela dove ogni agente attende che tutti gli altri abbiano raggiunto il suo stesso punto, dopodichè tutti possono riprendere l'esecusione. 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.

4.2 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 **??**, 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.

[language=lapsa,style=eclipse,caption=Implementazione **lapsa!** dello scheduler basato su model-checking,label=code:lapsa$_a$*gent*]*actionsa*

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

ranges NDRobot.p1 in [0,1], NDRobot.p2 in [0,1], NDRobot.p3 in [0,1], NDRobot.p4 in [0,1], NDRobot.p5 in [0,1], NDRobot.p6 in [0,1], NDRobot.p7 in [0,1], NDRobot.p8 in [0,1], NDRobot.p9 in [0,1] 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 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 che utilizza questa `Hashtable` decide la scelta da fare in base all'algoritmo 2. In questo caso la `Hashtable` generata dal compilatore **lapsa!** ha un solo livello di 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.

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 ∈ Actions do
  for all next ∈ States do
    calcola la probabilità p_step di passare da current a next con l'azione act
    p_MC = index(next)
    prob = p_step · p_MC
    if prob > max then
      max = prob
      best_act = act
    end if
  end for
end for
return best_act
```

4.3 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,

	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 4.1: Risultati delle simulazioni

- 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.

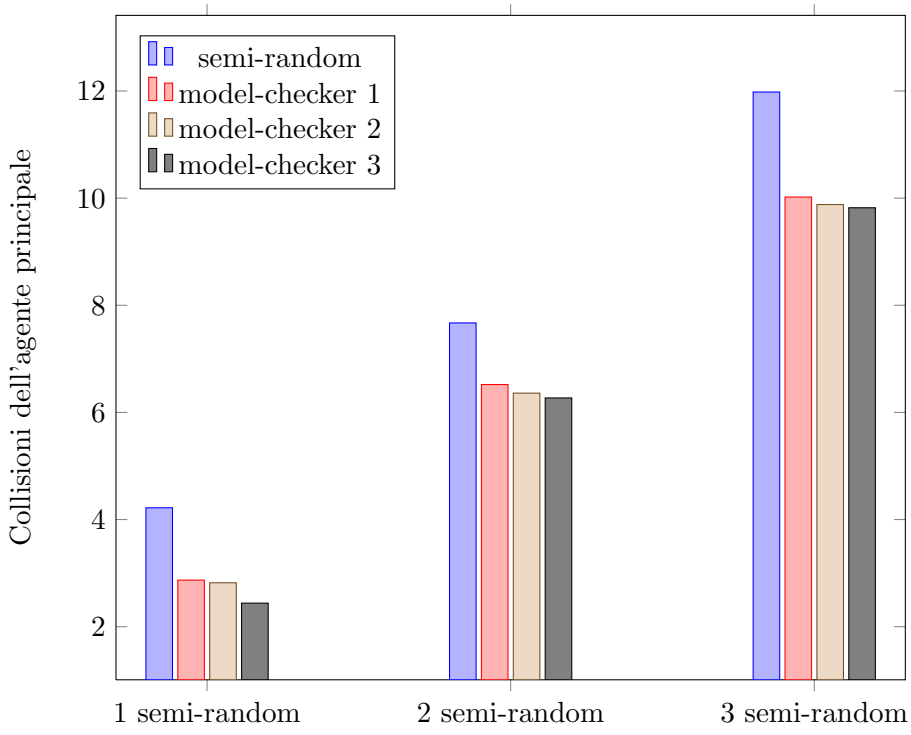

Figura 4.1: Grafico dei risultati delle simulazioni: sull'ascissa variano gli agenti che sono realmente nell'area assieme all'agente principale, sull'ordinata viene indicato il numero di collisioni e i diversi colori rappresentano di diversi scheduler utilizzati dall'agente principale.

In tabella 4.1 vengono riportate le medie dei test effettuati, raffigurate nel grafico 4.1. Dal grafico si riesce ad osservare un effettivo miglioramento del risultato dello scheduler basato sul model-checker rispetto a quello semi-casuale. Si percepisce, seppure in minor misura, un leggero miglioramento all'aumentare della complessità dello scenario ipotizzato: anche nel caso in

cui le simulazioni prevedano solamente un agente secondario si riscontra una media di collisioni per test più bassa se la `Hashtable` viene generata su ipotesi più complesse, assumendo due o tre agenti secondari. Nello scenario reale che comprende due agenti esterni si verifica un leggero miglioramento medio passando dallo scheduler generato sull'ipotesi di un solo agente semi-casuale a quello che ne considera due: questo miglioramento è giustificato dal fatto che gli stati previsti sono un numero maggiore e se lo scheduler basato sul model-checker non dovesse trovare lo stato che sta cercando all'interno della `Hashtable` allora si comporterà come uno scheduler semi-casuale.

Le simulazioni 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.