Titolo tesi

Marco Tinacci

30 maggio 2012

Indice

1 Introduction

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 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*, 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 *SEAL*, 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 *SEAL* può limitarsi alla descrizione del comportamento del soggetto e di come viene gestita la visione dell'ambiente. La compilazione del codice *SEAL* 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 *SEAL* 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 alcuni casi di studio confrontando i risultati delle simulazioni con quelli ottenuti tramite approcci più "tradizionali". Per la visualizzazione delle simulazioni viene utilizzato *ARGoS* implementando le interfacce dei moduli *C++* che determinano il comportamento dei *marXbot*.

1.1 Scaletta

- agenti adattivi, adattività (ricercare interpretazione tra gli articoli)
- esempi di casi di studio, *swarm robotics*, *e-veicles* e *cloud computing* (ASCENS)
  - concetto di vista
  - concetto di target
  - schema di deploy
  - modello prism
  - codice c++ precompilato
- *prism*: model checking come risolutore di scelte
- linguaggio per agenti adattivi *SEAL*
- *xtext*: implementazione di *SEAL*
  - implementazione automatica del plugin a partire dalla grammatica
  - scrittura di template in *xtend* per la generazione del codice
- *argos*: simulazione di casi di studio

program	::=	**actions**{*actions*} **subject** *module* *targets* *environment* **ranges**{*ranges*}
actions	::=	action-label | *actions*, *actions*
targets	::=	...
environment	::=	*module* | *environment environment*

Tabella 1: Sintassi di un programma *SEAL*

2 Syntax

Per descrivere la sintassi *SEAL* 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 2 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 2) 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 2: Sintassi di un modulo *SEAL*

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

L'insieme dei moduli sarà quindi del tipo

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

2.1 Syntactic sugar

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 Semantic

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* *targets* *environment* **ranges**{*ranges*}
targets	::=	...
environment	::=	*module* | *environment environment*
actions	::=	action-label | *actions*, *actions*
ranges	::=	module-name.id **in**(value, value, value)
module	::=	**module** module-name{*variables rules*}
variables	::=	type id = *expression*; | *variables variables*
expression	::=	...
system	::=	module-name(*params*) | *system* $\|\|_A$ *system*
params	::=	**void** | *expression* | *params*, *params*
rules	::=	*condition*[action-label] $\Rightarrow$ *distribution* | *rules*, *rule*
condition	::=	**E** id : module-name . *condition* | *expression* $\bowtie$ *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: Sintassi *SEAL* completa

- *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 *Markov Decision Processes*. 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 dal *MDP* risultante dal parallelo dei *MDP* che lo compongono.

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

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

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

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

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

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

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

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

4 Examples

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:

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

5 Code translation

5.1 Prism

La traduzione del sorgente *SEAL* 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 4. Con `a`, `b` e `delta` rappresentiamo costanti intere, con `e` un'espressione *SEAL* e con `e'` la rispettiva traduzione in *PRISM*. I dati necessari alla discretizzazione vengono attualmente forniti direttamente nel file *PRISM* nella sezione *ranges*.

SEAL	*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 4: Traduzione da *SEAL* a *PRISM*