SEAL: a language for self adaptive agents

Marco Tinacci

28 maggio 2012

Indice

1 Syntax

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*

Sono state omesse le descrizioni di alcuni simboli per alleggerire la lettura della sintassi:

- m: riferimento alla definizione di un modulo,
- A: insieme di azioni di sincronizzazione,
- e: espressione,
- $\bowtie$: operatore di confronto,
- x: dichiarazione di un identificatore,
- a: label azione.

L'insieme dei moduli sarà quindi del tipo

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

1.1 Syntactic sugar

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

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

2 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 $A \subseteq Act$;
- *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.

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

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

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

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

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

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

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

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

3 Examples

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

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

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

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

4 Code translation

4.1 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 1. 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 1: Traduzione da *SEAL* a *PRISM*