Piccolo manuale d’utilizzo del simulatore AsmetaS

Il progetto ha come scopo lo sviluppo di un interprete per Abstract State Machines (*ASM*) di base.

L’interprete

- analizza un modello conforme al metamodello *ASMM* [1] prodotto dal parser sulla base di un file di specifiche *AsmetaL* [3]
- valuta ogni costrutto secondo la semantica delle *ASM* [1, 2]
- produce come output la traccia di esecuzione della macchina simulata.

Sommario

1 MODALITA' D'USO

Il simulatore può funzionare in due modalità, in base a come sono letti i valori delle funzioni monitorate (l'ambiente esterno in cui opera la macchina simulata):

- interattiva, in cui il simulatore chiede i valori da standard input. In caso d'errore, stampa su standard output un messaggio diagnostico e invita l'utente a correggersi;
- batch, in cui il simulatore legge i valori da file. In caso d'errore, termina lanciando un'eccezione.

L'ambiente converte le rappresentazioni esterne dei valori appartenenti ai domini supportati dal metamodello *ASMM* negli opportuni oggetti di tipo *Value* manipolati dall'interprete.

1.1 Modalità interattiva

Per eseguire una macchina specificata in un file *.asm*, bisogna creare un'istanza della classe *Simulator* indicando il pathname del file. La classe *InteractiveUI* si occupa dell'interazione con l'utente.

```
>java –jar AsmetaS.jar modello.asm
```

Esecuzione per 100 transizioni

Il comando nel riquadro esegue la macchina contenuta in *modello.asm* per 100 transizioni (il valore di default). Se si vuole modificare questo valore, bisogna aggiungere l'opzione *–n* seguita dal numero di transizioni richieste.

```
>java –jar AsmetaS.jar –n 3 modello.asm
```

Esecuzione per 3 transizioni

In alternativa, la macchina può essere eseguita finché la regola principale non produce un insieme di aggiornamenti vuoto specificando l'opzione *–ne*.

```
>java –jar AsmetaS.jar –ne modello.asm
```

Esecuzione finché l'insieme di aggiornamenti della regola principale non è vuoto

Il simulatore produce come output la traccia d'esecuzione della macchina.
L'output è disponibile all'utente in due forme:

- come testo non formattato inviato sullo standard output
- come documento *xml* memorizzato nel file *log.xml* residente nella directory di lavoro.

Per cambiare stili di presentazione e media di memorizzazione, l'opzione *–log* consente di precisare un file log4j [4] che sarà considerato dal simulatore in sostituzione di quello di default.

```
>java –jar AsmetaS.jar –log filelog4j modello.asm
```

L'ouput è generato secondo le direttive contenute nel file *filelog4j*.

Un esempio di file di configurazione log4j

```
log4j.rootLogger=DEBUG, A0
log4j.logger.org.asmeta.interpreter=DEBUG, A1, A2
log4j.additivity.org.asmeta.interpreter=false

log4j.appender.A0=org.apache.log4j.ConsoleAppender
log4j.appender.A0.layout=org.apache.log4j.SimpleLayout

log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.asmeta.interpreter.util.XmlToTextLayout

log4j.appender.A2=org.apache.log4j.FileAppender
log4j.appender.A2.layout=org.asmeta.interpreter.util.RawLayout
log4j.appender.A2.File=log.xml
log4j.appender.A2.Append=false

log4j.logger.org.asmeta.interpreter.ReflectiveVisitor=WARN
#log4j.logger.org.asmeta.interpreter.TermEvaluator=WARN
#log4j.logger.org.asmeta.interpreter.RuleEvaluator=WARN
log4j.logger.org.asmeta.interpreter.TermSubstitution=WARN
log4j.logger.org.asmeta.interpreter.RuleSubstitution=WARN
```

La sintassi ed il significato delle direttive accettate da log4j sono descritti in [4] e nella documentazione allegata alla libreria [http://logging.apache.org/log4j/docs/index.html].

I messaggi di testo inviati ai logger sono frammenti di codice xml. L'interprete è fornito con due classi di tipo layout per la formattazione dei messaggi: *XmlToTextLayout* e *RawLayout*. La prima classe indenta il codice xml per migliorare la visualizzazione su console, mentre la seconda invia l'input così com'è agli appender (solitamente dei file xml). In quest'ultimo caso, si consiglia di aprire il file prodotto con un viewer xml per facilitare la navigazione.

1.2 Modalità batch

Aggiungendo come ultimo argomento alla riga di comando il pathname del file d'ambiente, l'utente è sollevato dal compito di fornire manualmente i valori poiché essi saranno letti direttamente dal file.

```
>java –jar AsmetaS.jar modello.asm ambiente.env
```

La classe *FileUI* consente la lettura da file di testo. In questo caso, il file è organizzato in linee che contengono una costante letterale, un commento (preceduto dal carattere '#') oppure una stringa vuota.

```
# un numero intero
124
# un valore booleano
true
# una stringa
"abc"
```

Esempio di un file d'ambiente

N.B. La possibilità di leggere le funzioni monitorate da file ha come scopo principale la validazione del programma. Il formato stesso è estremamente semplice in quanto non permette di associare un valore alla funzione monitorata che si suppone debba leggerlo. Per questo motivo, si consiglia di utilizzare l'opzione con attenzione.

1.3 Formato delle costanti letterali

Interi e reali

Es. 1234, 7.6, -8, +3.14

Booleani

true, false

Undef

undef

Stringhe

Sono sequenze di caratteri racchiuse tra doppi apici.
Es. “questa è una stringa.”.

Le costanti di tipo enumerativo, astratto e concreto sono rappresentate per mezzo di identificatori composti da lettere, cifre, il carattere ‘_’ ed il carattere ‘!’. Il primo carattere è una lettera.
In tutti e tre i casi, il simulatore controlla a run-time che le costanti denotino dei valori appartenenti ai domini associati.

Costanti enumerative

Es. RED, BLUE, GREEN.

Costanti astratte

Es. prodotto!1, prodotto!2, ordine, ordine!127

Costanti di domini concreti

La rappresentazione dipende dal dominio associato.

Tuple

Le tuple sono ennuple di costanti racchiuse fra i caratteri ‘(‘ e ‘)’.
Es. (), (1, “abc”, false)

Insiemi

Gli insiemi sono collezioni di costanti racchiuse fra i caratteri ‘{‘ e ‘}’.
Es. {}, {1, 2, 3}, {(1, “a”), (2, “b”)}

Sequenze

Gli insiemi sono liste di costanti racchiuse fra i caratteri ‘[‘ e ‘]’.
Es. [], [true, true, false], [{}, {1, 2}]

1.4 COSTRUTTI SUPPORTATI

Tipi

- *Naturali*: ok
- *Interi*: ok
- *Reali*: ok
- *Complessi*: no
- *Char*: no
- *String*: ok
- *Boolean*: ok
- *Undef*: ok
- *Rule*: no
- *Enumerativi*: ok
- *Astratti*: ok
- *Concreti*: ok
- *Tuple*: ok
- *Insiemi*: ok
- *Sequenze*: ok
- *Bag*: no
- *Map*: no

Termini

- *VariableTerm*: ok
- *TupleTerm*: ok
- *FunctionTerm*: ok
- *LocationTerm*: ok
- *ConditionalTerm*: ok
- *CaseTerm*: ok
- *DomainTerm*: solo se il dominio associato è un *AbstractTd, EnumTd* oppure un *ConcreteDomain*. In quest'ultimo caso, la valutazione del termine che lo definisce deve restituire un *SetValue*.
- *RuleAsTerm*: solo se compare come argomento di una *MacroCallRule* e non contiene variabili libere
- *SetTerm*: ok
- *SequenceTerm*: ok
- *BagTerm*: no
- *MapTerm*: no
- *ExistTerm*: ok
- *ForallTerm*: ok
- *ExistUniqueTerm*: no
- *SetCt*: ok
- *SequenceCt*: ok
- *BagCt*: no
- *MapCt*: no

Regole

- *SkipRule*: ok
- *UpdateRule*: ok
- *BlockRule*: ok
- *SeqRule*: ok
- *ConditionalRule*: ok
- *CaseRule*: ok
- *ExtendRule*: ok

Esempio

```
asm Ex01

import ../STDL/StandardLibrary

signature:

        abstract domain Products

        static p1: Products
        static p2: Products
        static p3: Products

definitions:

        main rule r_main =
                extend Products with $p do skip
```

La regola *extend* serve per estendere il contenuto di un dominio astratto (*Products*). I valori che appartengono al dominio prima che la simulazione inizi sono dichiarati come costanti (*p1*, *p2*, *p3*). La regola dell'esempio, aggiunge un nuovo elemento ad ogni iterazione. Nel riquadro sottostante, possiamo osservare la traccia d'esecuzione. Il simulatore associa ad ogni dominio astratto una funzione controllata unaria che ha lo stesso nome del dominio, che ha come dominio il dominio astratto e come codominio i booleani. I nuovi valori prodotti dalla *extend* si riconoscono poiché sono denotati da costanti con lo stesso nome del dominio seguite dal carattere '!' e da un numero progressivo (*Products!1*, *Products!2*, *Products!3*).
NOTA: i numeri progressivi non sono necessariamente consecutivi.

```
<Run>
      <Transition>
            <SkipRule>
                  <UpdateSet>{}</UpdateSet>
            </SkipRule>

      <State>{Agent(self)=true,Products(Products!1)=true,Products(p1)=true,Pr
oducts(p2)=true,Products(p3)=true}</State>
      </Transition>
      <Transition>
            <SkipRule>
                  <UpdateSet>{}</UpdateSet>
            </SkipRule>

      <State>{Agent(self)=true,Products(Products!1)=true,Products(Products!2)
=true,Products(p1)=true,Products(p2)=true,Products(p3)=true}</State>
      </Transition>
      <Transition>
            <SkipRule>
                  <UpdateSet>{}</UpdateSet>
            </SkipRule>

      <State>{Agent(self)=true,Products(Products!1)=true,Products(Products!2)
=true,Products(Products!3)=true,Products(p1)=true,Products(p2)=true,Products
(p3)=true}</State>
      </Transition>
</Run>
```

- *LetRule*: ok
- *ChooseRule*: ok
- *ForallRule*: ok
- *MacroCallRule*: ok

Esempio

```
asm Ex02

import ../STDL/StandardLibrary

signature:

        abstract domain Products

        static p1: Products
        static p2: Products
        static p3: Products

        controlled var1: Products
        controlled var2: Products

definitions:

        macro rule r_swap($x in Products, $y in Products) =
                par
                        $x := $y
                        $y := $x
                endpar

        main rule r_main =
                r_swap[var1, var2]

default init s0:

        function var1 = p1
        function var2 = p2
```

La regola r_swap[] scambia il contenuto di due variabili controllate. Le regole sono chiamate per nome. Questo significa che in una chiamata i parametri formali sono sostituiti nel corpo della regola dai parametri attuali. I parametri attuali non sono valutati nello stato in cui avviene la chiamata ma in seguito quando sono usati nel corpo (eventualmente in stati differenti a causa della composizione sequenziale). Per facilitare la modellazione è utile generalizzare la semantica della chiamata ammettendo dichiarazioni di regole con variabili-location e variabili-regole. In quest'estensione ogni parametro formale di una dichiarazione è una variabile logica, una variabile-location oppure una variabile-regola. Nel corpo della regola, una variabile-location può essere usata come un termine qualsiasi oppure a sinistra di un'update rule. Una variabile-regola può comparire ovunque è ammissibile una regola. In una chiamata, una variabile logica deve essere sostituita da un termine. Una variabile-location deve essere sostituita da un termine-location, in altre parole un termine che inizia con un nome di funzione dinamica Una variabile-regola deve essere sostituita da una regola.

1.5 Controllo degli invarianti

Il simulatore controlla gli assiomi al termine d'ogni transizione, ossia dopo aver applicato l'update set prodotto dalla main rule allo stato corrente.

Esempio

```
asm Ex01

import ../STDL/StandardLibrary

signature:

	abstract domain Products

	static p1: Products
	static p2: Products
	static p3: Products

	controlled stockQuantity: Products -> Integer
	monitored quantity: Products -> Integer

definitions:

	invariant over stockQuantity:
		(forall $p in Products with stockQuantity($p) >= 0)

	main rule r_main =
		choose $p in Products with true do
			stockQuantity($p) := stockQuantity($p) - quantity($p)

default init s0:

	function stockQuantity($p in Products) = 100
```

Nell'esempio riportato in figura, la funzione *stockQuantity* è decrementa senza accertarsi che non assuma valori negativi. Se lo stato della macchina non soddisfa l'invariante, il simulatore lancia un'eccezione del tipo *InvalidInvariantException*.

1.6 Controllo degli update inconsistenti

Il simulatore controlla che durante l'esecuzione della macchina non si producano update set inconsistenti.

Esempio

```
asm Ex02

import ../STDL/StandardLibrary

signature:

        abstract domain Orders
        enum domain Status = {PENDING | INVOICED | CANCELLED}

        static o1: Orders
        static o2: Orders
        static o3: Orders

        controlled orderStatus: Orders -> Status

definitions:

        main rule r_main =
                par
                        choose $o in Orders with true do
                                orderStatus($o) := INVOICED
                        choose $oo in Orders with true do
                                orderStatus($oo) := CANCELLED
                endpar

default init s0:

        function orderStatus($o in Orders) = PENDING
```

Nell'esempio riportato in figura, alla funzione *orderStatus* sono assegnati in parallelo due valori potenzialmente in conflitto. Se il valore delle variabili logiche *$o* e *$oo* è lo stesso, il simulatore lancia un'eccezione del tipo *UpdateClashException*.

2 COME ESTENDERE IL SIMULATORE

2.1 Implementazione della libreria standard

La corrente implementazione dell'interprete supporta una minima parte delle funzioni appartenenti alla libreria standard *StandardLibrary.asm*.

L'implementazione di tali funzioni è fatta nella classe *org.asmeta.interpreter.StandardLibrary* che è registrata come valutatore delle funzioni della StandardLibrary.asm con l'istruzione
`StaticFunctionEvaluator.registerFuntionEvaluator("StandardLibrary", org.asmeta.interpreter.StandardLibrary.class);`

Supponiamo di voler implementare la funzione *foo(a, b)* appartenente alla libreria standard che riceve i parametri *a* intero e *b* reale e che restituisce un intero.
Per ognuno dei tipi supportati dal metamodello, l'interprete definisce una corrispondente classe con suffisso *Value*. Quindi esiste una classe *IntegerValue* per rappresentare un valore intero e una classe *RealValue* per rappresentare un valore reale.
Modifichiamo la classe org.asmeta.interpreter.StandardLibrary.java nel seguente modo:

```
class StandardLibrary {

        …

        public static IntegerValue foo(IntegerValue a, RealValue b) {
        …
        }

}
```

Implementazione di una funzione della libreria standard

Tutte le volte che l'interprete incontra la funzione *foo* con parametri attuali gli oggetti *a* e *b* rispettivamente di tipo *IntegerValue* e *RealValue* chiama il metodo *foo(a,b)* che è definito nella classe StandardLibrary.

In generale, se l'interprete deve valutare una funzione statica *func*, dichiarata nella ASM con nome *Lib.asm*, ma non definita in termini di altre funzioni, la classe `StaticFunctionEvaluator` controlla quale classe java è registrata con Lib.asm e cerca in quella classe un metodo statico con nome *func* e parametri opportuni. In questo modo si possono introdurre funzioni statiche definite mediante codice Java.

3 Bibliografia

[1] A. Gargantini, E. Riccobene, and P. Scandurra. *Metamodelling a Formal Method: Applying MDE to Abstract State Machines*. Technical Report 97, DTI Dept., University of Milan, November 2006.

[2] E. Boerger and R. Staerk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[3] http://asmeta.sourceforge.net

[4] G. Gulcu. *Short introduction to log4j*. http://logging.apache.org/log4j/docs/index.html