QakActors25Linguaggio

Linguaggio qak

Il linguaggio si pone nel solco dei *Domain Specific Languages* e permette di esprimere un insieme dei concetti che forma *Il metamodello Qak*.

Il ruolo 'strategico' dei linguaggi in informatica si comprende subito considerando che **ogni computazione** (ogni sistema software) può essere espressa usando un insieme molto limitato di 'mosse' (istruzioni) studiate dalla teoria come *macchine astratte elementari*.

In sintesi, possiamo dire che l'uso du un linguaggio comporta descrizioni di un sistema software:

più compatte, più esplicie e semanticamente più ricche

Il linguaggio qak intende promuovere la definizione in tempi brevi di *prototipi di sistemi distribuiti*, utilizzabili nelle fasi preliminari di un progetto di sviluppo software, al fine di **interagire con il committente**, per chiarire e stabilizzare i requisiti.

In molti casi, la formalizzazione dei requsiti e della analisi del problema in ternini di modelli eseguibili qak costituisce anche un passo pragmaticamente utile per la costruzione effettiva del prodotto finale.

Qak syntax

La sintassi del linguaggio è riportata in *Qak syntax*.

Scopo della grammatica

Lo 'scopo' della grammatica è la produzione relativa alla specifica del sistema.

```
QActorSystemSpec:
  name=ID                                 //(1)
  ( mqttBroker = BrokerSpec)?             //(2)
  ( libs       = UserLibs  )?             //(3)
  ( message   += Message   )*             //(4)
  ( context   += Context   )*             //(5)
  ( actor     += QActorDeclaration )* //(6)
  (display    = DisplayDecl)?             //(7)
  (facade     = FacadeDecl)?              //(8)
```

Specifica del sistema

1. nome del sistema
2. indirizzo di un broker MQTT
3. dichiarazione di una o più librerie applicative
4. *Dichiarazione dei messaggi*
5. *Dichiarazione dei contesti*
6. *Dichiarazione degli attori*
7. Dichiarazione di un Display di sistema
8. *Dichiarazione di una Facade* di sistema

(...)? signitifca **opzionale**

(...)* signitifca **zero o più volte**

Le regole sintattche del linguaggio impongono che un modello Qak venga definito organizzando la sua descrizione in una **sequenza di dichiarazioni**.

Dichiarazione dei messaggi

I diversi *Tipi di messaggi* sono dichiarati usando una *sintassi* Prolog-like (si veda *tuProlog* >):

```
Message      :  BasicMessage | Event | OtherMsg;
BasicMessage :  Dispatch | Request;
OtherMsg     :  Reply;

Event:    "Event"     name=ID  ":" msg = PHead  (cmt=STRING)?;
Dispatch: "Dispatch"  name=ID  ":" msg = PHead  (cmt=STRING)?;
Request:  "Request"   name=ID  ":" msg = PHead  (cmt=STRING)?;
Reply:    "Reply"     name=ID  ":" msg = PHead  ( "for" reqqq = [Request] )? (cmt=STRING)?;

PHead :       PAtom | PStruct | PStructRef ;  //sintassi Prolog
...
```

Dichiarazione dei contesti

```
Context : "Context"  name=ID "ip"  ip = ComponentIP   ( "commonObj" commonObj = STRING)?  ;
```

```
ComponentIP : {ComponentIP} "[" "host=" host=STRING "port=" port=INT "]" ;
```

Un contesto può introdurre un oggetto accessibile a tutti gli attori.

Dichiarazione degli attori

```
QActorDeclaration: QActorInternal |
                   QActorExternal;

QActorInternal:  QActor |
                 QActorCoded;
```

La sintassi indica che vi sono tre tipi di attori.

1. *Attori normali*
2. *Attori coded*
3. *Attori external*

Gli attori possono inoltre comportarsi come generatori di stream

Attori normali

Gli attori 'normali' sono descritti come *Automi a stati finiti*.

1. Nome dell'attore
2. Riferimento al Contesto
3. Oggetto locale usato dall'attore (per un esempio si veda *conway25qak0*)
4. Attore creato solo dinamicamente
5. Libreirie importate
6. Azioni iniziali dell'attore
7. Stati dell'attore
8. Dichiarazione dell'oggetto locale
9. Dichiarazione delle librerie importate

```
QActor: "QActor"
/*1*/ name=ID
/*2*/"context" context = [ Context ]
/*3*/("withobj" withobj = WithObject)?
/*4*/( dynamic ?= "dynamicOnly")?
  "{"
/*5*/( imports += UserImport )*
/*6*/( start   = AnyAction )?
/*7*/( states += State )*
  "}";

/*8*/WithObject: name=ID
       "using" method=STRING;

/*9*/UserImport:
    "import" file=STRING ;
```

Si veda *AnyAction*

Stati di un attore normale

1. Nome dello stato
2. Stato iniziale. Il tag **initial** deve essere presente in un **unico stato**.
3. Azioni locali allo stato
4. Transizioni verso lo stato futuro

```
/*1*/State: "State" name=ID
/*2*/( normal?="initial")?
  "{"
/*3*/( actions += StateAction )*
  "}"
/*4*/( transition = Transition )?
```

Attori coded

Un CodedQActor è un attore scritto direttamente in codice (kotlin, Java o altro) che si comporta come fli attori qak.

Es> *democodedqactor.qak*

```
/*1*/QActorCoded : "CodedQActor"    name=ID
    "context" context = [ Context ]
    "className" className = STRING
    ( dynamic ?= "dynamicOnly")?;
```

Attori external

Un attore dichiearato external è un attore cha fa parte del sistema ma senza essere definito nel modello corrente, in quanto parte di un altro contesto.

Es> *demoaddtocore.qak*

```
/*1*/QActorExternal: "ExternalQActor" name=ID
    "context" context = [ Context ] ;
```

Attori streamer

La *Reactive programming* è una combinazione di idee riconducibili al modello Observer, al modello Iterator e al modello di programmazione funzionale.

In questo stile di programmazione, un servizio-consumatore reagisce ai dati non appena arrivano, con la capacità anche di propagare le modifiche come eventi agli osservatori registrati.

Un sistema Qak può essere impostato seguendo questo stile programmazione emettendo eventi usando la primitiva *subscribeTo*. Per un esempio si veda: Es> *demostreams.qak*

Dichiarazione di una Facade

Una Facade:

- rappresenta un componente che offre accessi tramite Web ad un sistema qak.
- vuole essere un modo per dotare un sistema qak di interfacce Web in accordo ai principi della Clean architecture
- deve essere realizzata da una classe applicativa definita dall'application designer
- lascia alla *Qak infrastructure* il compito di interagire con un attore di riferimento del sistema qak mediante invio/ricezione di messaggi.

```
FacadeDecl : "Facade"
/*1*/ name=ID
/*2*/ "port" port=INT
/*3*/ "usingactor" actor=[QActorInternal]
                   "inctx" context= [Context]
/*4*/ "appl" appl=STRING
/*4*/ libs=UserLibs ;
```

1. identificatore della Facade
2. porta di accesso alla Facade
3. attore di riferimento per la Facade
4. classe applicativa che realizza la Facade
5. librerie per la Facade

Es> ServiceMath24Facade.

Transizioni di stato

```
Transition        :  EmptyTransition | NonEmptyTransition  ;
```

La transizone da uno stato a uno stato successivo può avvenire senza attesa di alcun messaggio (*EmptyTransition*) oppure (*NonEmptyTransition*) in relazione alla disponibilità di un messaggio tra quelli definiti in *Dichiarazione dei messaggi*.

EmptyTransition

```
EmptyTransition:
/*1*/    "Goto" targetState=[State]
/*2*/    ("if" eguard=AnyAction
/*3*/    "else" othertargetState=[State] )?  ;
```

1. Riferimento allo stato futuro (nel caso di guardia **true**)
2. Specifica (opzionale) di una *Guardia*
3. Stato futuro nel caso di *Guardia* **false**

Es> *demoguards.qak*.

Guardia

- Una guardia è una espressione scritta in Kotlin, che può essere valutata come **true** o **false**.
- Una transizione associata a una guardia, viene attivata solo se la valutazione della condizione espressa dalla guardia produce il valore **true**.

NonEmptyTransition

```
NonEmptyTransition:
/*1*/  "Transition" name=ID
/*2*/  (duration=Timeout)?
/*3*/  (trans+=InputTransition)*
```

1. Nome della transizione
2. Specifica di un tempo massimo di attesa: si veda *Timeout per transizioni*
3. Transizione relativa a un messaggio

Es> *demoguards.qak*.

Una *NonEmptyTransition* associata alla disponibilità di un messaggio distingue tra i diversi *tipi di messaggio*:

```
InputTransition    : MsgTransSwitch | RequestTransSwitch |  ReplyTransSwitch |
                     EventTransSwitch | InterruptTranSwitch ;

MsgTransSwitch     : "whenMsg"     message=[Dispatch]
                      ("and"  guard=AnyAction  )?  ">"  targetState=[State]  ;
RequestTransSwitch : "whenRequest" message=[Request]
                      ("and"  guard=AnyAction  )?  ">"  targetState=[State]  ;
ReplyTransSwitch   : "whenReply"   message=[Reply]
                      ("and"  guard=AnyAction  )?  ">"  targetState=[State]  ;
EventTransSwitch   : "whenEvent"   message=[Event]
                      ("and"  guard=AnyAction  )?  ">"  targetState=[State]  ;
InterruptTranSwitch: "whenInterrupt"   message=[Dispatch]
                      ("and"  guard=AnyAction  )?  ">"  targetState=[State]  ;
```

whenInterrupt

Esegue la transzione da uno stato **SA** a uno stato **SB**, con ritorno allo stato **SA**, quando **SB** esegue l'istruzione qak *returnFromInterrupt*.

Es> *demointerrupt.qak*.

Timeout per transizioni

Per evitare una attesa indefinita di messaggi in uno stato, è poosibile associare alla transizione un *time-out* (come un numero naturale in *msec*) scaduto il quale l'automa transita nello stato specificato.

```
Timeout           : TimeoutInt | TimeoutVar | TimeoutSol | TimeoutVarRef;
TimeoutInt        : "whenTime"    msec=INT  ">" targetState = [State]  ;
TimeoutVar        : "whenTimeVar"    variable   = Variable    ">" targetState = [State]  ;
TimeoutVarRef     : "whenTimeVarRef" refvar     = VarRef      ">" targetState = [State]  ;
TimeoutSol        : "whenTimeSol"    refsoltime = VarSolRef   ">" targetState = [State]  ;
```

Lo scadere del tempo indicato in *whenTime* (regola **TimeoutInt**) provoca l'emissione di un **evento**, con identificatore **local_tout_actorname_state** ove **actorname** è il nome dell'attore e **state** è il nome dello stato corrente. Es> *demo0 perceiver*

Le forme che si aggiungono a **TimeoutInt** sono utili in situazioni in cui il tempo non sia noto a priori, ma derivi da elaborazioni. Es> *demo0 sender*

Indice: *QakActors25Index*