Towards dynamic systems

Antonio Natali

Alma Mater Studiorum – University of Bologna
via Sacchi 3,47023 Cesena, Italy,
viale Risorgimento 2,40136 Bologna, Italy
antonio.natali@studio.unibo.it

Table of Contents

7

1 Overcoming static configurations

In the *QActor* metamodel, a software system is made of a collections of *Actors*, each included in a computational node called *Context*. A *Context* can be viewed as a *subsystem* with a unique name and a unique IP address (host,port).

In several applications, the set of *Contexts* and actors that compose the system can be statically fixed, but there are other cases in which a system must dynamically change its configuration. In particular:

1. we should allow an actor to dynamically create other actors (actor instances) within its *Context*;
2. we should allow the dynamic introduction of new *Contexts* within a given system, together with the possibility to support the interaction among all the actors, both statically defined and dynamically created.

2 Dynamic creation of actors within a Context

Let us introduce the model of a very simple actor that receives and prints 2 messages and sends a reply to the sender;

```
System agentprototype -regeneratesrc
/*
 * ==========================================================
 * agentprototype.qa in project it.unibo.actor.dynamic
 * ==========================================================
 */
Dispatch info : info( X )

Context agentprototype ip [ host="localhost" port=8060 ] -g yellow
/*
 * ---------------------------------------------
 * An actor that receives and prints 2 messages
 * acting as a prototype for dynamic creation
 * (see agentcreation,qa)
 * ---------------------------------------------
 */
 QActor agentprototype context agentprototype -g green {
   Plan init normal
       [ !? actorobj(NAME) ] println( waits( NAME ) );
       receiveMsg time(300000) ;
       [ !? actorobj(NAME) ] onMsg info : info(X) -> replyToCaller -m info : info( reply(NAME, thanks(X)) ) ;
       printCurrentMessage ;
       repeatPlan 1
}
```

Listing 1.1. `agentprototype.qa`

This actor has the 'static' name `agentprototype`, but it refers to its own name with the built-in rule `actorobj/1`. The generated code provides (in the directory `src`) the class `Agentprototype` that defines the following constructor:

Constrcutor in class *it.unibo.agentprototype.Agentprototype*

```
public Agentprototype(String actorId, ActorContext myCtx, IOutputEnvView outEnvView )  throws Exception{
...
}
```

This constructor can be used to build new instances in a dynamical way, by using an actor-creation rule already defined in the built-in actor's `WorldTheory`:

2.1 An actor-creation rule

```
createActor(Name, PrototypeClass, NewActor):-
    actorobj(CREATOR),
    CREATOR <- getName returns CurActorName,     %%get the name of the actor prototypee
    CREATOR <- getContext returns Ctx,           %%get the conetxt of the actor prototypee
    CREATOR <- getOutputEnvView returns View,    %%get the output view of the actor prototypee
    %% CREATE A NEW ACTIR INSTANCE
    java_object(PrototypeClass, [Name,Ctx,View], NewActor),
    NewActor <- getName returns NewActorName,
    actorPrintln( createActor(NewActorName, NewActor, Ctx) ).
```

Listing 1.2. `createActor rule (in WorldTheory)`

The rule calls the constructor of the prototype given in the argument `PrototypeClass` by using the same context and the same output view of the creator's agent. Thus each new instance is always creted in the context of the creator agent.

2.2 An actor-instance creator

For example, the following actor `agentcreator` dynamically generates two instances of `agentprototype` and sends to each instance a messages, waiting for a reply:

```
System agentcreation -regeneratesrc
/*
 * ==============================================================
 * agentcreation.qa in project it.unibo.actor.dynamic
 * ==============================================================
 */
Dispatch info   : info( X )

Context ctxAgentCreation ip [ host="192.168.43.229" port=8060 ] -g yellow
/*
 * The ctxPivot declaration is required, since any instance will work in the
 * context of the creator
 */
Context ctxPivot ip [ host="192.168.43.209" port=8088 ] -standalone
/*
 * ------------------------------------------------------------
 * Creates two instances of agentprototype and intercats with
 * them by sending a message and then waiting for a reply
 * ------------------------------------------------------------
 */
QActor agentcreator context ctxAgentCreation {
     Plan init normal
         println(starts(agentcreator)) ;
         switchToPlan createInstance;
         switchToPlan sendHello ;
         switchToPlan getReply ;
         println(ends(agentcreator))
     Plan createInstance resumeLastPlan
          [ !? newName(agentprototypepivot,Name,N) ]
           solve createActor(Name,'it.unibo.agentprototypepivot.Agentprototypepivot',A) time(0) onFailSwitchTo
                prologFailure ;
         [ !? newName(agentprototypepivot,Name,N) ]
           solve createActor(Name,'it.unibo.agentprototypepivot.Agentprototypepivot',_) time(0) onFailSwitchTo
                prologFailure
     Plan sendHello resumeLastPlan
        [!? instance( agentprototypepivot, 1, A ) ] forward A -m info : info( hello(world)) ;
        [!? instance( agentprototypepivot, 2, A ) ] forward A -m info : info( hello(world))
    Plan getReply resumeLastPlan
        println( waits( agentcreator, reply ) );
        receiveMsg time(300000) ;
        printCurrentMessage ;
        repeatPlan 3
```

```
    Plan prologFailure resumeLastPlan
        println("agentcreator has failed to solve a Prolog goal" )
}
QActor pivot context ctxPivot {
     Plan init normal
        println("pivot place holder ")
}
```

Listing 1.3. `agentcreation.qa`

The guard `[!? newName(agentprototype,Name,N)]` in Plan `createInstance` makes reference to the built-in `newname` rule for instance-names creation, that stores in the actor's knowledge base a fact `instance/3`:

```
instance( PROTOTYPE_NAME, INSTANCE_COUNTER, INSTANCE_NAME )
```

Listing 1.4. instance fact

2.3 The name-creation rule

The `newname/3` name creation rule is defined in the *WorldTheory* as follows:

```
newName( Prot, Name,N1 ) :-
    inc(nameCounter,1,N1),
    text_term(N1S,N1),
    text_term(ProtS,Prot),
    text_concat(ProtS,N1S,Name),
    assert( instance( Prot, N1, Name ) ),
    actorPrintln( newname(Name,N1) ) .

value(nameCounter,0).

inc(I,1,N):-
    value( I,V ),
    N is V + 1,
    assign( I,N ).
```

Listing 1.5. `newName rule (in WorldTheory)`

3 Dynamic addition of Contexts

In this section we give an example of a dynamic system that starts from a single actor (and thus from a single node) called `pivot`. Other actors can interact with `pivot` by asking it to perform two basic operations:

- ***register(NickName,Name)***: the actor of name `Name` informs `pivot` of its existence by providing a *NickName* that can be used by other actors to find the actor;
- ***getactor(NickName)***: the actor makes a query to `pivot` in order to obtain the (name of the) actor that corresponds to the given `NickName`. The answer given by `pivot` can be used as destination in message-passing operations.

With these two basic operations, we can extends a systems with new actors working in different nodes and we can allow an actor to interact with another one, just knowing its nickname.

3.1 A first dynamic actor

In the following example:

1. an actor (`agent1`) registers itself to the `pivot` with the nickname `zorro`;
2. `agent1` makes a request to `pivot` for an actor with nickname `batman`;
3. `agent1` forwards a *dispatch* to the actor given in the answer to its request.

```
System agent1 -regeneratesrc
/*
 * ===============================================================
 * agent1.qa in project it.unibo.actor.dynamic
 * ===============================================================
 */
Event alarm : alarm(X)

Request register : actorname( NICKNAME, NAME )
Request getactor : nickname( NAME )
Dispatch answer : answer( X )
Dispatch info    : info( X )
/*
 * -------------------------------------------------
 *
 * -------------------------------------------------
 */
Context ctxPivot ip [ host="192.168.43.209" port=8088 ] -standalone
Context ctxAgent1 ip [ host="192.168.43.229" port=8040 ] -g white
EventHandler evh for alarm -print ;

QActor pivot context ctxPivot {
     Plan init normal
        println("pivot place holder ")
}

QActor agent1 context ctxAgent1 {
     Plan init normal
        println(agent1(zorro, starts)) ;
        switchToPlan register ;
        switchToPlan interact ;
        println(agent1(ends))
     Plan register resumeLastPlan
        println(agent1(zorro, register)) ;
        demand pivot -m register : actorname(zorro,agent1) ;
        receiveMsg time(300000) ;
        onMsg answer : answer(R) -> println( R )
      Plan interact resumeLastPlan
        println(agent1(lookat,batman)) ;
        demand pivot -m getactor : nickname(batman) ;
        receiveMsg time(300000) ;
//      printCurrentMessage ;
        onMsg answer : answer(R) -> println( agent1(found,batman,R) ) ;
//      onMsg answer : answer(R) -> forward R -m info : info(learned(1,R)) ;
        onMsg answer : answer(R) -> forward R -m info : info(learned(2,R))
}
```

Listing 1.6. `agent1.qa`

Of course, this kind of system can work only if all the dynamic agents share a common knowledge about the messages that can exchange among them.

3.2 A second dynamic actor

Each dynamic agent starts from a system composed of two actors (and contexts): itself and `pivot`.

Thus, the structural part of the model of the 'batman' agent is quite similar to that of agent1 above; as regards the behaviour, we suppose here that 'batman' first registers itself to the pivot and then waits for messages:

```
System agent2 -regeneratesrc
/*
 * ===============================================================
 * agent2.qa in project it.unibo.actor.dynamic
 * ===============================================================
 */
Event alarm : alarm(X)

Request register : actorname( NICKNAME, NAME )
Request getactor : nickname( NAME )
Dispatch answer : answer( X )
Dispatch info    : info( X )
/*
 * ---------------------------------------------
 *
 * ---------------------------------------------
 */
Context ctxPivot  ip [ host="192.168.43.209" port=8030 ] -standalone
Context ctxagent2 ip [ host="192.168.43.229" port=8050 ] -g green
EventHandler evh for alarm -print ;

QActor pivot context ctxPivot {
     Plan init normal
        println("pivot place holder ")
}

QActor agent2 context ctxagent2 {
     Plan init normal
        println(agent2(batman, starts)) ;
        switchToPlan register ;
        switchToPlan interact ;
        println(agent2(ends))
     Plan register resumeLastPlan
        println(agent2(batman, register)) ;
        demand pivot -m register : actorname(batman,agent2) ;
        receiveMsg time(300000) ;
        onMsg answer : answer(R) -> println( R )
      Plan interact resumeLastPlan
        println( agent2(batman, waits) ) ;
        receiveMsg time(300000) ;
        printCurrentMessage ;
        repeatPlan 1
}
```

Listing 1.7. agent2.qa

Of course, agent2 must run before agent1 since agent1 must find it already registered in the system.

4 The pivot

Le us introduce now a possible model for the pivot :

```
System pivot -regeneratesrc
/*
 * ===============================================================
 * pivot.qa in project it.unibo.actor.dynamic.pivot
 * ===============================================================
 */
Event alarm : alarm(X)

Request register : actorname( NICKNAME, NAME )
```

```
Request getactor : nickname( NAME )
Dispatch answer : answer( X )
Dispatch info    : info( X )
/*
 * ----------------------------------------------
 *
 * ----------------------------------------------
 */
Context ctxPivot ip [ host="192.168.137.2" port=8088 ]
EventHandler evh for alarm -print ;

QActor pivot context ctxPivot {
     Plan init normal
        solve consult("./pivotTheory.pl") time(0) onFailSwitchTo prologFailure ;
        switchToPlan acceptRequest
     Plan acceptRequest
        println( pivot(waits) ) ;
        receiveMsg time(300000) ; //tout=5 min
        memoCurrentMessage ;
//         printCurrentMessage ;
        onMsg register : actorname(_,_) -> switchToPlan register ;
        onMsg getactor : nickname(_) -> switchToPlan getactor ;
        repeatPlan 0
    Plan register resumeLastPlan
        [ ?? msg( register, MSGTYPE, SENDER, RECEIVER, actorname(NICKNAME, NAME), SEQNUM )]
                       solve register(NICKNAME, NAME, RESULT) time(0) onFailSwitchTo prologFailure ;
        [ !? goalResult(register(NICKNAME,NAME,RESULT))] replyToCaller -m answer:answer(RESULT)
    Plan getactor resumeLastPlan
        [ ?? msg( getactor, MSGTYPE, SENDER, RECEIVER, nickname(NICKNAME), SEQNUM )]
                       solve getactor(NICKNAME, RESULT) time(0) onFailSwitchTo prologFailure ;
        [ !? goalResult(getactor(NICKNAME,NAME))] replyToCaller -m answer:answer(NAME)
    Plan prologFailure resumeLastPlan
        println("pivot has failed to solve a Prolog goal" )
}
```

Listing 1.8. `pivot.qa`

4.1 The pivot application theory

Several operations of the `pivot` actor are implemented as tuProlog rules by means of the user-defined *pivotTheory*:

```
/*
===============================================================
pivotTheory.pl
===============================================================
*/
register( NICKNAME , NAME, alreadyRegistered(NICKNAME) ):-
    retract( registered(NICKNAME,_) ),!.
register( NICKNAME, NAME, doneRegistered(NICKNAME,NAME) ):-
    actorPrintln( registered(NICKNAME,NAME) ),
    assert( registered(NICKNAME,NAME) ).


getactor(NICKNAME, NAME):-
    registered(NICKNAME,NAME),
    actorPrintln( getactor(NICKNAME,NAME) ),
    !.
getactor(NICKNAME,null).

/*
------------------------------------------------------------
initialize
------------------------------------------------------------
*/
initialize :-
    actorPrintln("pivotTheory started ...") .
```

```

:- initialization(initialize).
```

Listing 1.9. `pivotTheory.pl`

4.2 The pivot at start-up

Startup

```
sudo crontab -u root -e

@reboot cd /home/pi/nat/pivot && screen sh -c './pivot.sh ; read'

#!/bin/bash
echo hello pivot
sudo java -jar  MainCtxPivot.jar
exec bash
```