An introduction to the usage of QActors and QRobots

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

1 Introduction to QActors

QActor is the name given to the basic concept of a custom programming meta-model inspired to the actor model (as can be found in the Akka library). The qa language is a custom language that can allow us to express in a concise way the structure, the interaction and also the behaviour of (distributed) software systems composed of a set of *QActor*.

The leading Q/q means 'quasi' since the *QActor* meta-model and the qa language do introduce (with respect to Akka) their own peculiarities, including reactive actions and even-driven programming concepts.

This work is an introduction to the main concepts of the *QActor* meta-model and to a 'core' set of constructs of the qa language. Let us start with some example.

1.1 Example: The 'hello world'

The first example of a **qa** specification is obviously the classical 'hello world':

```
1  /*
2   * hello.qa
3   */
4  System helloSystem
5  Context ctxHello ip [ host="localhost" port=8079 ]
6
7  QActor qahello context ctxHello {
8     Plan init normal
9        println("Hello world" )
10 }
```

Listing 1.1. `hello.qa`

This example shows that each qactor works within a *Context* that models a computational node associated with a network `IP` (`host`) and a communication `port` (see Subsection 3.5). The behaviour of a is modelled as a set of macro-actions called *plans* (see Subsection 2.2).

1.2 QActor specification

A *QActor* specification can be viewed as:

- an executable specification of the (logic) architecture of a distributed (*heterogeneous*) software system, according to three main dimensions: *structure*, *interaction* and *behaviour*;
- a prototype useful to fix software requirements ;
- a (operational) scheme useful to define the *product-backlog* in a `SCRUM` process;
- a model written using a custom, extendible meta-model/language tailored to the needs of a specific application domain.

Thus, a *QActor* specification aims at capturing main *architectural* aspects of the system by providing a support for *rapid software prototyping*. A *QActor* specification can express the intention of an actor to:

- execute actions (see Subsection 2.1
- send/receive messages (see Subsection 3.6)
- emit/perceive events (see Subsection 3.11)

A *QActor* support is implemented in Java and in tuProlog in the *it.unibo.qactors* project and is deployed in the file `qa18Akka.jar`.

1.3 The generated code

The *QActor* language/metamodel is associated to a *software factory* that automatically generates the proper system configuration code, so to allow Application designers to focus on application logic. In fact, aach actor requires some files, each storing a description written in tuProlog syntax:

- A file that describes the configuration of the system. In the case of the example, this file is named `hellosystem.pl`; it stored in the directory `srcMore/it/unibo/ctxHello`.
- A file named `sysRules.pl` that describes a set of rules used at system configuration time. In the case of the example, this file it stored in the directory `srcMore/it/unibo/ctxHello`.
- A (optional) file that describes a set of (tuProlog) rules and facts that give a symbolic representation of the "world" in which a qactor is working. In the case of the example, this file is `srcMore/it/unibo/qahello/WorldTheory.pl`.

For each actor and for each context, the `qa` software factory generates (Java) code in the directories `scr-gen` and in the `src`. Moreover, a gradle build file is also generated; for the example, it is named `build_ctxHello.gradle`.

1.4 The work of the application designer

In order to produce executable code in an Eclipse workspace, the application designer must:

1. Copy in the current workspace the project `it.unibo.iss.libs`.
2. Execute the command `gradle -b build_ctxHello.gradle eclipse` in order to set the required libraries.
3. Modify the code of the actors by introducing in the generated actor class (in the `src` directory) the required application code. In the case of the example, we have no need to modify the generated class `src/it/unibo/qahello/Qahello.java`. In any case, this class is generated only once, so that code changes made by the application designer are not lost if the model is modified.
4. define a set of `JUnit` testing operations within a source folder named `test`. Each test unit should start with the string `"Test"` (see the generated build file). For an example see Subsection 8.4
5. Run the generated main program (`src-gen/it/unibo/ctxHello/MainCtxHello.java`).

1.5 Example: Actor as a finite state machine

The next example defines the behaviour of a *QActor* able to execute two plans: an `init` plan (qualified as '`normal`' to state that it represents the starting work of the actor) that calls another plan named `playMusic` that plays a sound and, once terminated, returns the control to the previous one:

```
1  /*
2   * basic.qa
3   * A system composed of a qactor (player)
4   * working in a context named 'ctxBasic' associated with a GUI
5   */
6  System basic    //the testing flag avoids automatic termination
7  Context ctxBasic ip [ host="localhost" port=8079 ] -g cyan
8
9  QActor player context ctxBasic{
10     Plan init normal
11         println("Hello world" ) ;
12         switchToPlan playMusic ;
13         println("Bye bye" )
14     Plan playMusic resumeLastPlan
15         sound time(2000) file('./audio/tada2.wav') ;
```

```
16        [ ?? tout(X,Y) ] println( tou(X,Y) )
17        repeatPlan 1
18
19  }
```

Listing 1.2. `basic.qa`

A plan can be viewed as the specification of a **state** of a *finite state machine* (FSM) (see Subsection 2.5). State transition can be performed with no-input moves (e.g. `switchToPlan` action) or when a *message* is received or an *event* is sensed,

1.6 Example: Message-based interaction

A *Qactor* is an element of a (distributed) software system (*qactor-system* from now-on) that an work in cooperation/competition with other actors and other components, each modelled as a *QActor*. *QActors* do not share memory (data): they can interact only by exchanging messages or by emitting/sensing events.

As an example of a message-based interaction, let us introduce a very simple producer-consumer system:

```
1  /*
2   * basicProdCons.qa
3   * A system composed of a producer a consunmer
4   */
5  System basic -testing
6  Dispatch info : info(X)
7
8  Context ctxBasicProd ip [ host="localhost" port=8079 ] -g cyan
9  Context ctxBasicCons ip [ host="localhost" port=8089 ] -g yellow
10
11 QActor producer context ctxBasicProd{
12     Plan init normal
13         println( producer(starts) ) ;
14         //The producer sends a message to itself
15         forward producer -m info : info("self message");
16         switchToPlan produce ;
17         switchToPlan getSelfMessage ;
18         println( producer(ends) )
19     Plan produce
20         //delay time(1500);   //to force a timeout in the consumer
21         println( producer(sends) ) ;
22         forward consumer -m info : info(1);
23         resumeLastPlan
24     Plan getSelfMessage
25         receiveMsg time(1000);
26         printCurrentMessage;
27         resumeLastPlan
28  }
29
30  QActor consumer context ctxBasicCons{
31     Plan init normal
32         println( consumer(starts) ) ;
33         switchToPlan consume ;
34         println( consumer(ends) )
35     Plan consume
36       receiveMsg time(2000) ;
37       [ ?? tout(R,W)] endPlan "consumer timeout";
38       printCurrentMessage -memo; //-memo: the current message is stored in the actor KB
39       resumeLastPlan
40  }
```

Listing 1.3. `basicProdCons.qa`

Here is an example of testing (written by the application designer):

```
1  package it.unibo.ctxBasicCons;
2  import static org.junit.Assert.*;
3  import java.util.concurrent.ScheduledThreadPoolExecutor;
4  import org.junit.After;
5  import org.junit.Before;
6  import org.junit.Test;
7  import alice.tuprolog.SolveInfo;
8  import it.unibo.ctxBasicProd.MainCtxBasicProd;
9  import it.unibo.qactors.QActorUtils;
10 import it.unibo.qactors.akka.QActor;
11
12 public class TestProdCons {
13       private QActor prod;
14       private QActor cons;
15       private  ScheduledThreadPoolExecutor sched =
16              new ScheduledThreadPoolExecutor( Runtime.getRuntime().availableProcessors() );
17
18    @Before
19    public void setUp() throws Exception {
20       activateCtxs();
21       //Get a reference to the two actors:
22       //getQActor waits until the actor is active
23       prod  = QActorUtils.getQActor("producer_ctrl");
24       cons  = QActorUtils.getQActor("consumer_ctrl");
25    }
26     @After
27     public void terminate(){
28        System.out.println("====== terminate " );
29     }
30     @Test
31     public void execTest() {
32         System.out.println("====== execTest ===============" );
33         try {
34               assertTrue("execTest prod", prod != null );
35               assertTrue("actorTest cons", cons != null );
36               System.out.println("====== execTest waits for work completion ... " );
37               Thread.sleep(2000); //give the time to work
38               //Acquire the message stored at the consumer site
39               SolveInfo sol = cons.solveGoal("msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )");
40               System.out.println("execTest CONTENT="+sol.getVarValue("CONTENT"));
41               assertTrue("actorTest info", sol.getVarValue("CONTENT").toString().equals("info(1)") );
42         } catch (Exception e) {
43               fail("actorTest " + e.getMessage() );
44         }
45     }
46     private void activateCtxs( ){
47         sched.submit( new Runnable(){
48            @Override
49            public void run() {
50                try {
51                    //QActorContext ctxCons =
52                        MainCtxBasicCons.initTheContext();
53                } catch (Exception e) { e.printStackTrace(); }
54            }
55         });
56         sched.submit( new Runnable(){
57            @Override
58            public void run() {
59                try {
60                    //QActorContext ctxProd =
61                        MainCtxBasicProd.initTheContext();
62                } catch (Exception e) { e.printStackTrace(); }
63            }
64         });
65     }
66 }
```

Listing 1.4. `TestProdCons.qa`

2 QActor concept overview

2.1 Actions

A *QActor* can execute a set of (predefined or user-defined) *actions* that must always terminate. A *timed action* (see Subsection 2.1.5) always terminates within a prefixed time interval.

The effects of actions can be perceived in one of the following ways:

1. as changes in the state of the "*actor's 'mind'*";
2. as changes in the actor's working environment.

The first kind of actions are referred here as *logical actions* since they do not affect the physical world. The ***actor-mind*** is represented in the following by a Prolog theory named `WorldTheory` associated with the actor (see Subsection 2.1.1 and Subsection 3.3)).

Actions that change the actor's physical state or the actor's working environment are called *physical actions*.

2.1.1 The actor's `WorldTheory`. The `WorldTheory` includes computational rules written in tuProlog and facts about the state of the actor and of the world. For example:

- the rule `actorPrintln/1` prints a given tuProlog `Term` (see Subsection 3.6.1) in the standard output of the actor;
- the fact `actorobj/1` memorizes a reference to the Java/Akka object that implements the actor (see Subsection 6.3).
- the rule `actorOp/1` puts in execution a Java method written by the application designer (see Subsection 6.4).
- the fact `actorOpDone/2` memorizes the result of the last `actorOp` executed (see Subsection 6.4)
- the fact `goalResult/1` memorizes the result of the last Prolog goal given to a `solve` operation (see Subsection 6.1)
- the fact `result/1` memorizes the result of the last plan action (see Subsection 2.1.5) performed by the actor

Facts like `actorOpDone/1`, `goalResult/1`, etc. are 'singleton facts'. i.e. there is always one tuple for each of them, related to the last action executed.They can be used to express *guards* (see Subsection 2.1.5) related to action evaluation.

2.1.2 Logical actions. *Logical actions* usually are 'pure' computational actions defined in some general programming language actually we use Java, Prolog and JavaScript.

For example, any *QActor* is 'natively' able to compute the `n-th` Fibonacci's number in two ways: in a fast way (`fib/2` Prolog rule) and in a slow way (`fibo/2` Prolog rule).

2.1.3 Physical actions. *Physical actions* can be implemented by using low-cost devices such as `RaspberryPi` and `Arduino`.

2.1.4 Application actions. Besides the predefined actions, a *QActor* can execute actions defined by an application designer according to the constraints imposed by its logical architecture. More on this in Section 6.

2.1.5 PlanActions. A *PlanAction* is a logical or physical action defined by the system or by the application designer. A *PlanAction* can assume different logical forms according to different attributes that can be associated to it:

Actions attribute sets

```
 ACTION
[ GUARD ] , ACTION
[ GUARD ] , ACTION , DURATION
[ GUARD ] , ACTION , DURATION , ENDEVENT
[ GUARD ] , ACTION , DURATION , [EVENTLIST], [PLANLIST]
```

We will use the following terminology:

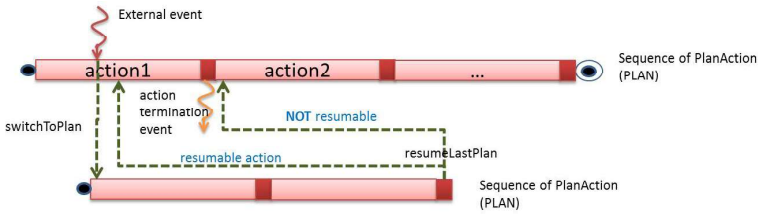
- an action that does not specify any `DURATION` is called ***basic action*** (see Subsection 5.1);
- an action that does specify a `[GUARD]` is called ***guarded action*** (see Subsection 5.6);
- an action that specifies a `DURATION` is called ***timed action*** (see Subsection 5.2);
- an action that specifies a `ENDEVENT` is called (timed) ***asynchronous action*** (see Subsection 5.4);
- an action that specifies `[EVENTLIST],[PLANLIST]` is called (timed) ***(synchronous) reactive action*** (see Subsection 5.5).

A *timed action*:

- emits (when it terminates) a built-in *termination event*;
- can be interrupted by events; it is qualified as ***resumable*** if it can continue its execution after the interruption.

2.2 Plans

A ***Plan*** is a sequence of *PlanActions*.



2.3 Messages and (reactive) message

A ***message*** is defined here as information sent in ***asynchronous*** way by some source to some specific destination. For *asynchronous* transmission we intend that the messages can be 'buffered' by the infrastructure, while the 'unbuffered' transmission is said to be ***synchronous.***

Messages can be ***sent*** and/or ***received*** by the *QActors* that compose the *qactor-system*. A message does not force the execution of code: it can be managed only after the execution of an explicit *receive* action performed by a *QActor*. Thus we talk of *massage-based* behaviour only, by excluding *massage-driven* behaviour (the default behaviour in Akka).

Messages are represented as follows:

```
1 msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
```

where :

MSGID	Message identifier
MSGTYPE	Message type (e.g.:dispatch,request,invitation,event,token)[1]
SENDER	Identifier of the sender
RECEIVER	Identifier of the receiver
CONTENT	Message payload
SEQNUM	Unique natural number associated to the message

The `msg/6` pattern can be used to express *guards* (see Subsection 5.6) to allow conditional evaluation of *PlanActions*.

2.4 Events and event-driven/event-based behaviour

An ***event*** is defined here as information emitted by some source without any explicit destination. Events can be ***emitted*** by the *QActors* that compose the *actor-system* or by sources external to the system.

The occurrence of an event can put in execution some code devoted to the management of that event. We qualify this kind of behaviour as ***event-driven*** behaviour, since the event 'forces' the execution of code.

An event can also trigger state transitions in components, usually working as finite state machines that call operations to explicitly ***perceive*** events. We qualify this kind of behaviour as ***event-based*** behaviour, since the event is 'lost' if no machine is in a state waiting for it.

Events are represented as messages (see Subsection 2.3) with no destination (`RECEIVER=none`):

```
1 msg( MSGID, event, EMITTER, none, CONTENT, SEQNUM )
```

2.5 Actor programs as plans

A ***qactor-program*** consists in a set of *Plans*; the (unique, mandatory) *Plan* qualified as `normal` is executed as the actor main activity. Other plans can be put in execution by the main plan according to action-based, event-based or message-based behaviour.

Each plan has a name and can be put into execution by a proper *PlanAction* (e.g. `switchToPlan`, see Subsection 5.1). Plans can be stored in files and dynamically loaded by the user into the *actor-mind*.

A plan is represented in files as a sequence of 'facts', each expressed in the following way:

Internal representation of plans

```
plan(ACTIONCOUNTER,PLANNAME,sentence(GUARD,MOVE,EVENTLIST,PLANLIST))
```

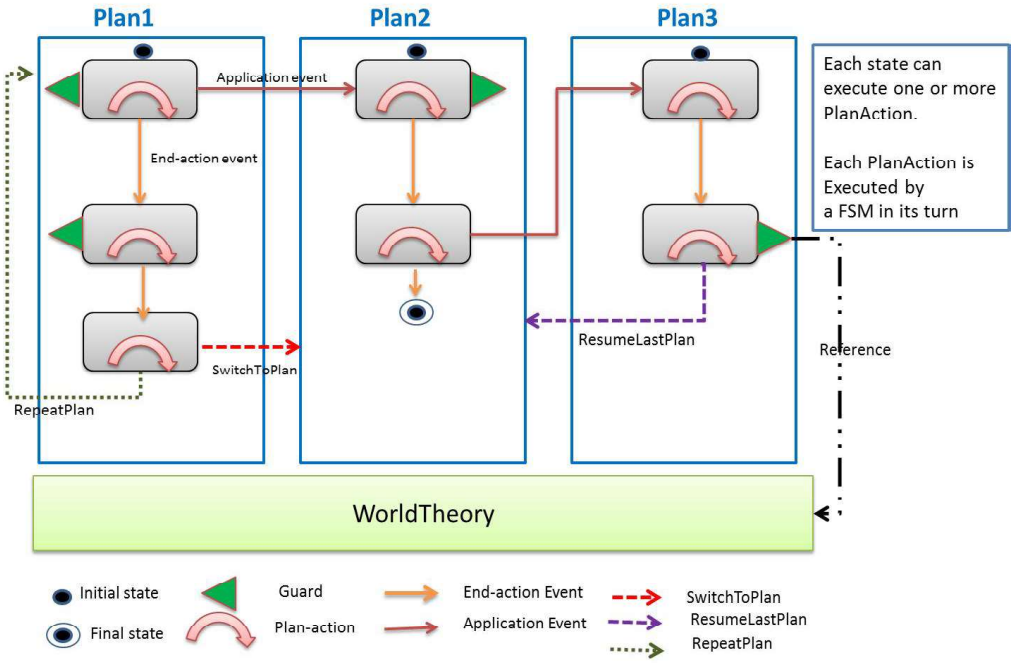
For example:

Internal representation of a plan

```
plan(0,p0,sentence(true,move(playsound,'./audio/tada2.wav',1500),'',''))
plan(1,p0,sentence(true,move(playsound,'./audio/music_interlude20.wav',20000),'usercmd,alarm','handleUsercmd,handleAlarm'))
```

This internal representation of Plans can be the input of a run-time *PlanInterpreter* that can execute plans dynamically created by the actor. This can be a support for experiments in the field of *automated planning*.

2.5.1 Actor programs as finite state machines (FSM) A *Plan* can be viewed as the specification of a **state** of a *finite state machine* (`FSM`) of the Moore's type and should **not** be interpreted as a procedure. In fact a plan can be put in execution by events (see Subsection 2.4) and returns the control to the 'calling' plan only when it declares to '`resumeLastPlan`' (otherwise the computation ends).



State transitions are usually caused by messages or events but that can be caused also by explicit built-in actions such as `switchToPlan`.

A timed *PlanAction* is in its turn implemented as `FSM` that can generate different kinds of termination events:

- a (built-in) normal termination event;
- a user-defined termination event (see Subsection 5.4);
- a time-out termination event;
- a abnormal termination event;