Introduction to QActors (2018)

Antonio Natali

Alma Mater Studiorum – University of Bologna
via dell'Università 50,47521 Cesena, Italy,
Viale Risorgimento 2,40136 Bologna, Italy
antonio.natali@studio.unibo.it

Table of Contents

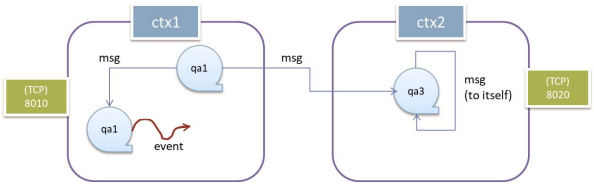
1 Introduction to QActors

QActor is the name given to ta custom meta-model inspired to the actor model (as supported by 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 the behaviour of (distributed) software systems whose components interacts by using messages and events.

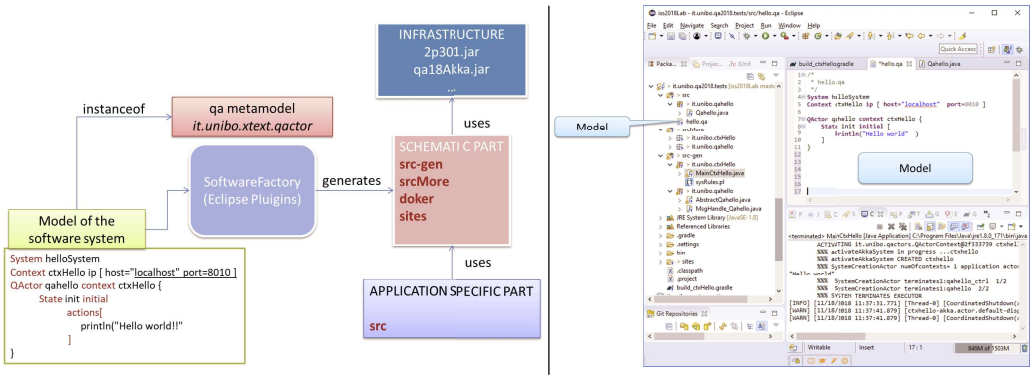
1.1 Overview

The leading Q/q in the *QActor* word, 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. Let us summarize the main features of a qa system:

- A QA-System is a collection of active entities (QActors) each working in a computational node (Context). A *QActor* can interact with other QActors using (see Subsection ??) Messages of different types (Dispatch, Request, Invitation, ...) and Events.

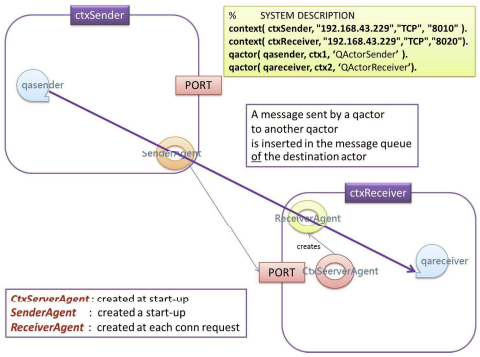


- A QA-System can be specified in textual form by means of the *QActor* language/metamodel. The *QActor* language is associated to a software factory (Subsection 2.2) that automatically generates the proper system *run-time support* so to allow Application designers to focus on application logic. The qa software factory is implemented as a set of Eclipse plug-ins, built by using the XText framework.



- The *configuration* of a QA-System is explicitly represented by a set of 'facts' written in tuProlog syntax (Subsection 5.6) replicated in each Context (Context Knowledge Base or simply ContextKB). A QA-System con be configured in a static or in a dynamic way. In case of dynamic configuration, the knowledge about the configuration is dynamically updated in each Context of the system (see Subsection 9).
- The start-up of a distributed *QActor* system (i.e. a system made of two or more Contexts) is handled by the run-time support. In particular, the Application code (i.e. the code written into the actors) begins to run only when all the Contexts are activated.

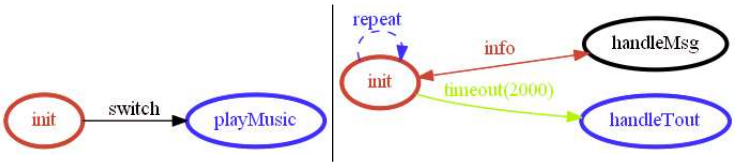
– The exchange of information among the `QActors` is implemented by the `QA-Infrastructure` (Subsection 5.6), based on the `Akka-Java` library. The `QA-Infrastructure` supports interaction among `QActors` working in the same Context and/or in different Contexts. In the latter case, the `QA-Infrastructure` exploits the `ContextKB` in order to deliver a message from the Context of the sender to the Context of the destination. An event raised in some Context, is delivered to **all** the other Contexts of the system.



– To deliver information among the Contexts, the `QA-Infrastructure` can use pairwise `TCP` connections between the Contexts or a `MQTT` broker. The choice is up to the Application designer.

– Each *QActor* behaves as a (Moore's) *Finite State Automaton* (`FSM`). While in a state, a *QActor* can execute both `synchronous` and asynchronous actions. An `asynchronous` action terminates immediately and emits an event (see Subsection 4.1) when it terminates.

– A `state-transition` from a state `S1` to another state `S2` can be done (triggered) only when all the actions activated in `S1` are terminated. A state-transition can be triggered by a message or by an event or as a choice of the *QActor* (no-input transition).



– A state-transition can be associated to a boolean condition (`guard`) and can be triggered only when the guard is true (see Subsection 3.0.3). When two or more transitions are possible, the systems checks the possibility to trigger one of them by following the order in which they are written. If no transition can be triggered, the *QActor* remains in the state until the occurrence of some message or event that allows a transition.

– A *QActor* is able to execute a set of `pre-defined actions` (see Subsection 11.3, Subsection 3.0.5). The a application designer can define `Application specific actions` by using Java or other languages (in particular `JavaScript/Node` and Prolog).

– A *QActor* is associated to a private knowledge-base (`QaKB`, Section ??) written in Prolog that can be dynamically extended by the Application designer.

Let us start with some example.

The code of the examples shown in this work can be found in the project `it.unibo.qa2018.tests`.

1.2 Example: the 'hello world'

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

```
/*
 * hello.qa
 */
System helloSystem
Context ctxHello ip [ host="localhost" port=8010 ]

QActor qahello context ctxHello {
   State init initial [
       println("Hello world" )
   ]
}
/*
 * Another style: Plan vs of State, normal vs initial
 */
QActor qahelloold context ctxHello {
   Plan init normal
       actions[javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveForward', 'arg': 60000 }");
          println("Hello world old style" )
       ]
}
```

Listing 1.1. `hello.qa`

The 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 5.6).

A *QActor* must define one (and just one!) `State`, qualified as '`initial`' to state that it represents the starting work of the actor. Each `State` of a *QActor* represents the state of a Moore `Finite State Machine`, whose `state-actions` are defined in a proper section, enclosed within the 'brackets' `[ ... ]`.

The example shows also an older version in which:

- The key-word `Plan` is used at place of the key-word `State`;
- The key-word `normal` is used at place of the key-word `initial`;
- The *state-action* section can be optionally prefixed by the word `actions`.

In fact, it is extremely easy (by using the `Xtext` framework) to change the key-words of our language, since `Xtext` automatically generates (starting from the grammar of the language) a compiler and a syntax-driven editor.

The introduction of an appropriate set of keywords is crucial to better transmit the intended semantics of a language construct. With the keyword `State`, we call to mind the idea of a machine, while with the keyword `Plan` we could evoke the idea of a sequence of high-level actions the our machine must execute to fulfil some goal.

In the following we will use both the notations, since the language semantics is identical.

A *QActor* is intended to be a software component that can perform application `actions` by interacting with other *QActor*s by means of `messages` (see Subsection 5.1) or `events` (see Subsection 5.4).

`State transitions` can be performed when:

- all the actions in the state body are terminated, and
- a *message* is or an *event* is available, or
- a no-input transition (`switchTo`) is specified.

1.3 Example: no-input transitions

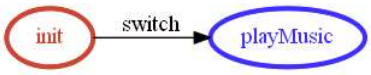
The next example defines the behaviour of a *QActor* that performs a no-input transition:

```
System noinputTansition
Context ctxNoinputTansition ip [ host="localhost" port=8079 ]

QActor qahellonoinputtrans context ctxNoinputTansition{
    State init initial [
        println( playSomeMusic );
        delay 1000 //wait for a second
    ]
    switchTo playMusic

    State playMusic [
        sound time(2000) file('./audio/tada2.wav')
    ]
```

Listing 1.2. noinputTansition.qa

switchTo is a predefined operation of the *QActor* language (see Subsection 2.1).
sound is a QActor built-in action (see Subsection 11.3, Subsection 3.0.5) that ends after 2 seconds.
delay is a built-in action that waits for the specified number of msecs.

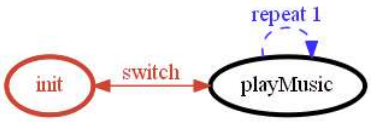
The *QActor* performs a state-switch from its init state to the playMusic state with a no-input transition switchTo. The behaviour of the actor can be represented by a state diagram like that shown in the following picture:



This state diagram is automatically generated [1] by the *QActor* software factory (see Subsection 2.2).

1.4 Example: repeat/resume plans

Let us define the behaviour of a *QActor* that implements the state machine shown in the following state diagram:




```
/*
 * repeatResume.qa
 */
System repeatResume
Context ctxRepeatResume ip [ host="localhost" port=8079 ]

QActor player context ctxRepeatResume{
    State init initial [
          println("player STARTS" ) ;
           println("player ENDS" )
       ]
       switchTo playMusic
//     finally repeatPlan 1  //(1)
```

[1] See the file srcMore/it/unibo/qahellonoinputtrans/qahellonoinputtrans.gv

```
14
15      Plan playMusic resumeLastPlan [
16            println( playSomeMusic );
17             sound time(2000) file('./audio/tada2.wav') ;
18             delay 500
19          ]
20          finally repeatPlan 1
21  }
22  /*
23  OUTPUT
24  ----------------------------
25  "player STARTS"
26  "player ENDS"
27  playSomeMusic
28  playSomeMusic
29  */
```

Listing 1.3. `repeatResume.qa`

The *QActor* performs a state-switch from its initial state to the `playMusic` state with a no-input transition `switchTo`. The `playMusic` state repeats its actions two times (because of `repeatPlan 1`) and then makes a no-input transition to its previous ('calling') state (`player`).

From the output, we note that all the actions of a plan are execute before the transition. This can be better explained by introducing the main rules that define the behaviour of a *State/Plan* (i.e. its operational semantics).

1.5 How a Plan/State works

When a *QActor* enters in a Plan, it works as follows:

1. the *QActor* executes, in sequential way, all the actions specified in the *Plan*. Each action must be defined as an `algorithm`, i.e. it must terminate;
2. if the *State/Plan* specification ends with the sentence `finally repeatPlan N` (`N` natural number `N>=1`), the state actions are repeated `N` times. If `N` is omitted, the *Plan* is repeated forever. The key word `finally` highlights the fact that the repetition a sentence must always written at the end of a Plan specification;
3. when the state actions are terminated (and before any repetition), the *QActor* can enter in a `transition phase` in order to perform a switch to another state (let us call it '*nextState*' and '*oldState*' the original one). For the details, see Subsection 5.2;
4. when a state has terminated its work (i.e. its actions, transition and repetition), it can resume the execution of its *oldState*. This happens if the `resumeLastPlan` keyword is inserted after the state name. Otherwise, the state is considered a `termination state` and the *QActor* does not perform any other work.

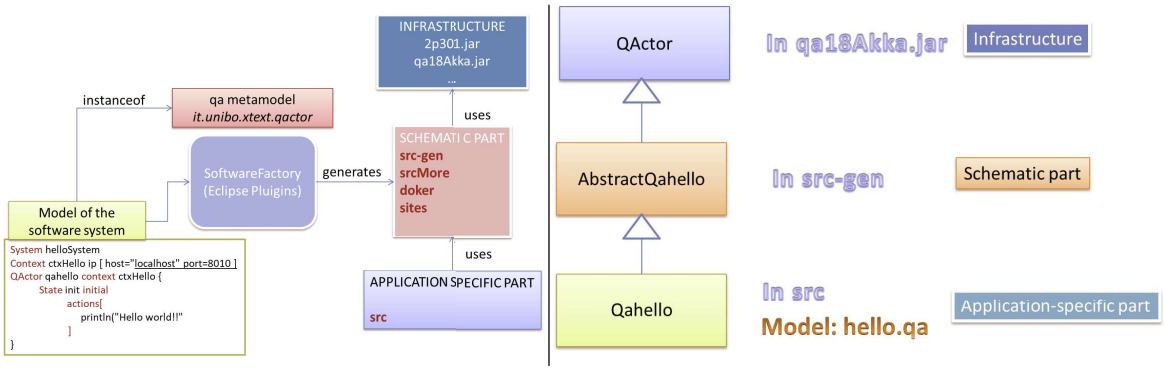
Thus, if we remove the comment `(1)` from the example of Subsection 1.3, the output will be:

```
1  "player STARTS"
2  "player ENDS"
3  playSomeMusic
4  playSomeMusic
5  "player STARTS"
6  "player ENDS"
7  playSomeMusic
8  playSomeMusic
```

If we remove the `resumeLastPlan` specification from `playMusic`, the control does not return to `player` and the output is the same as Subsection 1.3.

2 From specifications to execution

A *QActor* model aims at capturing main architectural aspects of a software system, by allowing a graceful transition from object-oriented programming to message-based and event-based computations. Moreover, a *QActor* model provides a support for rapid software prototyping, since the *QActor* software factory creates the software layer that 'adapts' the application layer to an infrastructure layer ('schematic part' in the picture) that provides a run-time support for the *QActor* modelling language.



2.1 The custom language qa

The qa language is a custom language[2] built by exploiting the XText technology; thus, it is also a metamodel. Technically we can say that qa is a 'brother' of UML, since it is based on EMOF.

The qa language aims at overcoming the abstraction gap between the needs of distributed proactive-reactive systems and the conventional (object-based) programming language (mainly Java, C#, C, etc) normally used for the implementation of software systems.

> In fact, a qa specification aims at capturing main architectural aspects of the system by providing a support for rapid software prototyping and a graceful transition from object-oriented programming to message-based and event-based computations.

The syntax of the qa language is expressed by a EBNF notation:

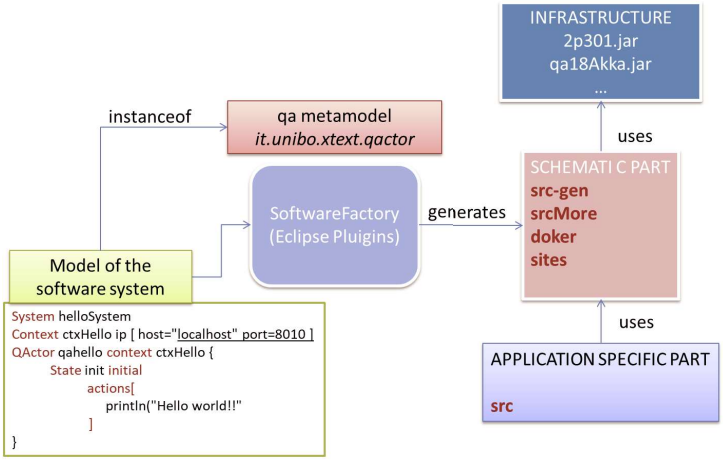
```
QActorSystem:
   "System" spec=QActorSystemSpec
;
QActorSystemSpec:
   name=ID
   ( message  += Message )*
   ( context  += Context )*
   ( actor    += QActor  )*
;
```

> From a grammar specification, the XText framework is able to generate in automatic way Java code for a parser and a syntax-driven editor, that can extend the Eclipse ecosystem by means of a set of plug-ins.

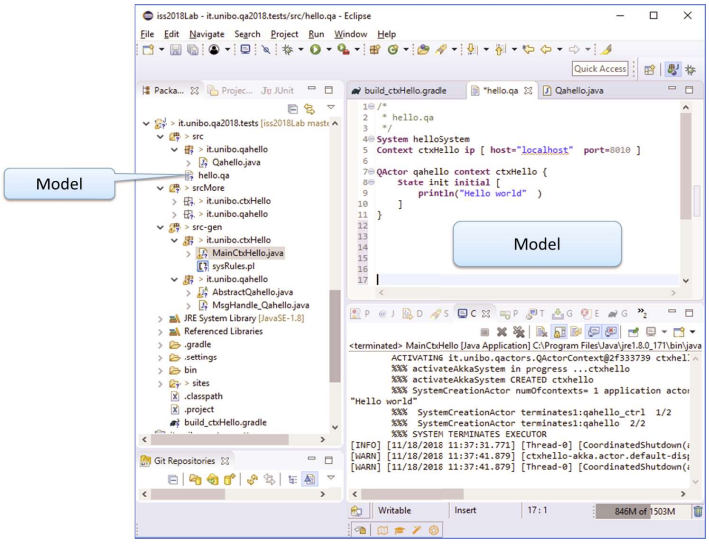
[2] The qa language/metamodel is defined in the project *it.unibo.xtext.qactor*.

2.2 The qa software factory

The qa language/metamodel is associated to a software factory that automatically generates the proper system run-time support (including system configuration code) so to allow Application designers to focus on the Application logic. The qa software factory is implemented as a set of Eclipse plug-ins, built by the XText framework.



For each *QActor* and for each `Context`, the *QActor* software factory generates (`Java`) code in the directories `scr-gen` and `src`. Further information (for example about system configuration) is included in the directory `srcMore`, to allow explicit visibility after deployment[3]. Moreover, a `gradle` build file is also generated for each `Context`.



[3] Usually, Java software is deployed by creating an executable `jar` file with the bytecode, and this file is normally not accessible by a system manager.

2.2.1 Example of generated files .

In the case of the example of Subsection 1.2:

- The file (named build_ctxHello.gradle) that describes the build rules of the system, is generated in the directory /.
- The file (named hellosystem.pl) that describes the configuration of the system, is generated in the directory srcMore/it/unibo/ctxHello.
- The file (named sysRules.pl) that includes the tuProlog rules used (mailny for the system start-up) by the qa-infrastructure (see Subsection 5.6) is generated in the directory srcMore/it/unibo/ctxHello.
- The file (named WorldTheory.pl) that includes the actor's world-theory (see Section ??) is generated in the directory srcMore/it/unibo/qahello.
- The file (named qahell.gv) that gives a graphical representation of the actor's state diagram (see the example of Subsection 1.3), is generated in the directory srcMore/it/unibo/qahello.

For a more complete view of the generated files, see Subsection 5.6.

3 The actor's WorldTheory

For a *QActor* named myactor, the qa software factory generates a file (named WorldTheory.pl) in the directory srcMore/it.unibo.myactor. Thus file includes a set of rules and facts written in tuProlog that give a symbolic representation of the "world" in which a *QActor* is working.

The file WorldTheory.pl includes also several computational rules written in tuProlog that extend the action-set of a *QActor*. For example, it includes rules 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).

The facts and rules stored in the WorldTheory.pl file of a *QActor* can be used to specify conditional execution of actions, by prefixing an action with a guard of the form [GUARD] where GUARD (see Subsection 3.0.3) is written as a Phead (see Subsection ??) tuProlog Term.

More details about the WorldTheory and its role are given in Section ??.

3.0.2 Facts about the state .

Examples of facts related to the current computational state of a *QActor* are:

actorobj/1	memorizes a reference to the Akka object that implements the actor (see Subsection ??)
goalResult/1	memorizes the result of the last goal given to a demo operation (see Subsection 3.0.4)

These facts are 'singleton facts'. i.e. there is always one clause for each of them, related to the last action executed.

3.0.3 Guarded actions .

Actions (see Subsection ??) prefixed by a [GUARD] are executed only when the GUARD is evaluated true. The GUARD can include *unbound variables*[4], possibly bound during the guard evaluation phase. Moreover:

- the prefix !? before the guard condition means that the knowledge (a fact or a rule) that makes the guard true is not removed form the actor's *WorldTheory*;
- the prefix ?? means that the fact/rule that makes the guard true is removed from the actor's *WorldTheory*.

[4] We recall that a Prolog variable is syntactically expressed as an identifier starting with an upcase letter.

Let us consider the following example;

```
System naiveGuardedActions
Context ctxNaiveGuardedAction ip [ host="localhost" port=8037 ]
QActor qanaiveguarded context ctxNaiveGuardedAction {
    State init initial [
        [ !? true    ] println( alwaysHere ) ;
        [ !? false   ] println( neverHere ) ;
        [ !? fib(6,X) ] println( fib(6,X) ) ;
        [ ?? fibo(4,X) ] println( fibo(4,X) )
    ]
    finally repeatPlan 1
}

/*
OUTPUT
----------------------------
alwaysHere
fib(6,8)
fibo(4,3)
alwaysHere
fib(6,8)
*/
```

Listing 1.4. `naiveGuardedActions.qa`

The output shows that the second iteration does not compute any more `fibo/2`, since that rule has been removed in the first iteration. The example show also that `variables` unified by a guard execution can be referred in the action (the `scope` id the action itself).

3.0.4 The `demo` operator .

A *QActor* can use the built-in `demo` operator to execute actions implemented in tuProlog within the actor's `WorldTheory`. The result of the `demo` operator is memorized in the singleton fact `goalResult/1` that can be inspected by using a guard.

```
System demoExample
Context ctxDemoExample ip [ host="localhost" port=8079 ]
QActor qademoexample context ctxDemoExample{
    Plan init normal
        [   println("qademoexample STARTS" ) ;
            [ !? actorobj(X) ] println( actorobj(X) ) ;
            demo actorobj(X) ;
            [ ?? goalResult(R) ] println( qademoexample( R ) ) ;
            demo fibo(6,X);
            [ ?? goalResult(R) ] println(R) ;
            demo fibo(6,8);
            [ ?? goalResult(R) ] println(R) ;
            demo fibo(X,8);   //fails since fibo/2 is not invertible
            [ ?? goalResult(R) ] println(R) ;
            println("qademoexample ENDS" )
        ]
 }
/*
OUTPUT
----------------------------
"qademoexample STARTS"
actorobj(qatuqademoexample_ctrl)
qademoexample(actorobj(qatuqademoexample_ctrl))
fibo(6,8)
fibo(6,8)
failure
"qademoexample ENDS"
*/
```

Listing 1.5. `demoExample.qa`

3.0.5 Built-in Prolog rules .

Besides the 'facts' introduced in Subsection 3.0.2, the *WorldTheory* (QaKb) of an actor includes several other computational rules written in tuProlog. Example of operations written as tuProlog rules in the QaKb are:

actorPrintln(T)	prints the given term T (see Subsection 5) in the actor standard output;
assign(K,V)	associates the given key K the the given value V, by removing nay previous association (if any)
getVal(K,V)	unifies the term V with the given key K
inc(I,K,N)	inc(I,K,N) :- value(I,V), N is V + K, assign(I,N)
addRule(R)	adds the given rule R in the *WorldTheory*
removeRule(R)	removes the given rule R from the *WorldTheory*
replaceRule(R,R1)	replace the given rule R with the other rule R1 of the same 'signature'
eval(plus,V1,V2,R)	unifies R with the result of V1+V2. Also available: minus, times, div
eval(lt,X,Y)	true if X<Y. Also available: gt
divisible(V1,V2)	true if V1 is divisible for V2

Moreover, the Rules option within a *QActor* specification allows us to define operations in a declarative style within facts and rules by using a subset of the tuProlog language.

3.0.6 Rules at model level .

Sometimes can be useful to express tuProlog facts and rules directly in the model specification, especially for configuration or action-selection purposes. The Rules option within a *QActor* allows us to define facts and rules by using a subset of the tuProlog syntax[5]

For example, let us define the model of a system that plays some music file by consulting its knowledge-base about the sound files, defined in the Rules section:

```
System rulesInModel
Context ctxRulesInModel ip [ host="localhost" port=8059 ]
QActor rulebasedactor context ctxRulesInModel {
    Rules{
        music(1, './audio/tada2.wav',2000).
        music(2,'./audio/any_commander3.wav',3000).
//      music(3,'./audio/computer_complex3.wav',3000).
//      music(4,'./audio/illogical_most2.wav',2000).
//      music(5,'./audio/computer_process_info4.wav',4000).
//      music(6,'./audio/music_interlude20.wav',3000).
//      music(7,'./audio/music_dramatic20.wav',3000).
    }
    State init initial[
        [ !? actorobj(X) ] println( myName(X) )
    ]
    switchTo work
    State work [
        [ ?? music(N,F,T) ] sound time(T) file(F) else endPlan "bye"
    ]
    finally repeatPlan
}
```

Listing 1.6. `rulesInModel.qa`

More on the actorKb can be found in Subsection **??**.

[5] The extension of this option with full Prolog syntax is a work to do.

3.0.7 Example: using built-in tuProlog rules .

The following example shows different ways of using built-in and user-defined rules.

```
/*
 * builtinPrologExample.qa
 */
System builtinPrologExample
Context ctxBuiltInExample ip [ host="localhost" port=8079 ]

QActor qabuiltinexample context ctxBuiltInExample{
    Rules{
        r1 :- assign(x,10),getVal(x,V1),eval(plus,V1,3,RV),actorPrintln(r1(RV)).
        r2 :- distance(X),actorPrintln(r2(X) ).
        r3 :- assert( distance(200) ).
    }
    Plan init normal
        [   println("execute built-in rules" ) ;
            demo assign(x,30) ;
            demo getVal(x,V) ;
            [ ?? goalResult(getVal(x,V))] println( valueOfx(V) ) ;
            println("execute a user-defined rule r1 (in Rules)" ) ;
            demo r1 ;
            println("add a distance/1 fact" ) ;
            addRule distance(100);
            println("execute a user-defined rule r2 that refers to distance/1" ) ;
            demo r2 ;
            println("execute a user-defined rule r3 that adds another distance/1" ) ;
            demo r3 ;
            println("conditional execution using distance/1 facts as guards" ) ;
            [ ?? distance(D)] println( distance(D) ) else println( nodistance );
            [ ?? distance(D)] println( distance(D) ) else println( nodistance );
            [ ?? distance(D)] println( distance(D) ) else println( nodistance );
            println("remove the rule r1" ) ;
            removeRule r1 ;
            println("attempt to run the removed rule r1" ) ;
            demo r1 ;
            println("END" )
        ]
}
```

Listing 1.7. `builtinPrologExample.qa`

The code should be self-explaining. The output is:

```
"execute built-in rules"
valueOfx(30)
"execute a user-defined rule r1 (in Rules)"
r1(13)
"add a distance/1 fact"
"execute a user-defined rule r2 that refers to distance/1"
r2(100)
"execute a user-defined rule r3 that adds another distance/1"
"conditional execution using distance/1 facts as guards"
distance(100)
distance(200)
nodistance
"remove the rule r1"
"attempt to run the removed rule r1"
"END"
*/
```

Listing 1.8. `builtinPrologExample.qa`

3.1 Loading and using a user-defined theory

The *WorldTheory* of an actor can be extended by the application designer by using the directive[6] `consult`.

For example, the following system loads a user-defined theory and then works with sensor data, for two times in the same way (plan `accessdata`) :

```
/*
 * atheoryUsage.qa
 */
System atheoryUsage
Context ctxTheoryUsage ip [ host="localhost" port=8049 ]

QActor qatheoryuser context ctxTheoryUsage{
    Plan init normal
    [   println( "qatheoryuser STARTS" ) ;
 /*0*/  demo consult("./src/it/unibo/qatheoryuser/aTheory.pl")
    ]
    switchTo accessdata

    Plan accessdata resumeLastPlan
    [  println( "-----------------------------------" ) ;
/*1*/  [ !? data(S,N,V) ] println( data(S,N,V) ) ;
/*2*/  [ !? validDistance(N,V) ] println( validDistance(N,V) ) ;
/*3*/  demo nearDistance(N,V) ;
/*4*/  [ !? goalResult(nearDistance(N,V)) ] println( warning(N,V) ) ;
/*5*/  demo nears(D)  ;
/*6*/  [ !? goalResult(G) ] println( list(G) )
    ]
    finally repeatPlan 1
}
```

Listing 1.9. `aTheoryUsage.qa`

The theory stored in `aTheory.pl` includes data (facts) and rules to compute relevant data:

```
/* ============================================================
aTheory.pl
============================================================ */
 data(sonar, 1, 10).
 data(sonar, 2, 20).
 data(sonar, 3, 30).
 data(sonar, 4, 40).

 validDistance( N,V ) :- data(sonar, N, V), V>10, V<50.
 nearDistance( N,X ) :- validDistance( N,X ), X < 40.
 nears( D ) :- findall( d( N,V ), nearDistance(N,V), D).

aTheoryInit :- output("initializing the aTheory ...").
:- initialization(aTheoryInit).
```

Listing 1.10. `aTheory.pl`

3.1.1 The initialization directive .

The following directive:

```
:- initialization(goal).
```

sets a starting goal to be executed just after the theory has been consulted.

Thus, the output of the `theoryusage` actor is:

[6] A tuProlog directive is a query immediately executed at the theory load time.

```
/*
OUTPUT
---------------------------
"qatheoryuser STARTS"
initializing the aTheory ...
"-------------------------------------"
data(sonar,1,10)
validDistance(2,20)
warning(2,20)
list(nears([d(2,20),d(3,30)]))
"-------------------------------------"
data(sonar,1,10)
validDistance(2,20)
warning(2,20)
list(nears([d(2,20),d(3,30)]))
*/
```

Listing 1.11. `aTheoryUsage.qa`

3.1.2 On backtracking .

The output shows that the rules `validDistance and nearDistance` exploit *backtracking* in order to return the first valid instance (2), while the repetition of the plan *accessdata* returns always the same data[7]. In fact, `backtracking` is a peculiarity of Prolog and is not included in the computational model of *QActor*. However, an actor could access to different data at each plan iteration, by performing a proper query in which the second argument of `data/3` is used as an index.

[7] Remember from Subsection 3.0.2 that the fact `goalResult/1` is a 'singleton'.

4 Actions (built-in and user-defined)

A *QActor State/Plan* specifies a sequence of predefined or user-defined `actions` that must always terminate. Actions can be classified into a set of types, as reported in the following table:

`Logical action`	usually is a 'pure' computation defined in some general programming language. Actually we use Java, Prolog and JavaScript. Example: `fibo(14,X)`.
`Physical action`	an actions that changes the actor's physical state or the actor's working environment. Example: `addRule`.
`Timed action`	always terminates within a prefixed time interval. Example: `sound`.
`Application action`	defined by the application designer according to the constraints imposed by its logical architecture.

As a general statement, we can say that[8]:

An Application action executed by a *QActor* in a state `S`, should terminate as soon as possible in order to make the actor in `S` as much *reactive* as possible to events and messages. Long-lasting activities should be implemented as `asynchronous actions` (see Subsection 4.1).

4.0.3 Action results .

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

1. as changes in the state of the actor stored in the actor's `WorldTheory` (see Subsection **??**)).
2. as changes of the state stored in some computational object associated to the actor;
3. as changes in the actor's working environment.

4.0.4 Built-in and User-defined actions .

Each *QActor* is able to execute:

- a set of built-in operations, defined by the `qa` language as key-words. Examples are: `println`, `sound`, `forward`, `emit`, `addRule`, `javaRun`, `javaOp`, `nodeOp`, etc.
- a set of actions implemented in Java by the *QActor* System designer in the class `QActor` of the library `qaAkka18.jar`. Examples are: `debugStep`, `publish`, `subscribe`, `sendPost`, `doCoapPost`, etc.. This kind of actions can be executed by means of the built-in operation `javaOp` (see Subsection 4.0.6).
- a set of actions implemented in some Java class written by the *Application designer*. These actions can be executed by means of the built-in action `javaRun` (see Subsection 4.0.5).
- a set of actions written by the *Application designer* in tuProlog (see Subsection 3.0.5, Subsection 3.0.7).

4.0.5 The operation `javaRun` .

As happens for any language, the expressive power of the *QActor* modelling language is limited. The `javaRun` key-word provides an 'escape mechanism' to run Java code provided by the Application designer. Let us consider the following sentence:

```
1 javaRun it.unibo.utils.clientRobotTcp.sendMsg("{ 'type': 'moveForward', 'arg': 60000 }")
```

This operation puts in execution the `sendMsg` (static) method of the (user-defined) class `it.unibo.utils.clientRobotTcp`[9] that sends a command to a remote robot.

[8] However, 'true' `real-time systems` are out of the scope of this work.

[9] Nte that the name of the class starts with a lower-case letter. This is a constraint imposed by the current *QActor* implementation.

4.0.6 The operation javaOp .

Another 'escape mechanism' is the operation javaOp that puts in execution a Java operation denoted by a given String. Let us consider the following sentence:

```
1 javaOp "debugStep()"
```

This operation puts in execution the method debugStep() that the actor inherits from the system class QActor. This method facilitates the debug of the actor, since it blocks the activity of the actor until the software designer hits the 'g' keyboard key.

Actions executed by the javaOp operation extend the computational features of a *QActor* without any change in the *QActor* language; however, the language could be extended at a later time, so to 'promote' the action (with some proper new key-word) as a 'primitive' operation.

The set of action currently defined in the in the QActor class are reported in Subsection 11.3.

4.0.7 The operation nodeOp .

In order to promote heterogeneity, the *QActor* System designer has introduced the operation nodeOp operation, that puts in execution the Node.js program denoted by a given String.

Let us consider the following sentence:

```
1 nodeOp "C:/robot/server/src/main.js 8999 " -o
```

This operation puts in execution a Node.js server that provides some useful feature (for example, a virtual robot working in a virtual environment). With the optional -o set, the system will show the output of the Node.js program (if any) on the standard output of the actor.

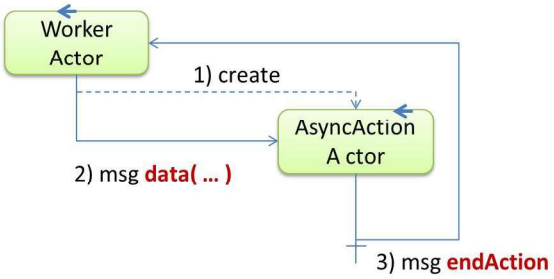
4.1 Asynchronous actions

In Section 4 we said that an ***action*** is an activity that must always terminate. Let us consider here an action that computes the n-th number of Fibonacci (slow, recursive version):

```
1   protected long fibonacci( int n ){
2       if( n<0 || n==0 || n == 1 ) return 1;
3       else return fibonacci(n-1) + fibonacci(n-2);
4   }
```

Usually an action expressed in this way is executed as a procedure that keeps the control until the action is terminated. Since this is a 'pure computational action', its effects can be perceived (as a result of type long) when the action returns the control to the caller.

The growing demand for asynchronous, event-driven, parallel and scalable systems, lead us to introduce the idea of an action that can be activated in **asynchronous way** and that, when it terminates, sends a **termination message** to its activator actor.



Let us introduce here a possible implementation of asynchronous actions.

4.1.1 A base-actor for asynchronous action implementation .

The following abstract class defines the behaviour of an Akka actor generic with respect to the result type T, that delegates to the application designer the task to define the operations execTheAction and endOfAction.

```
public abstract class ActionObservableGenericActor<T> extends UntypedAbstractActor{
    protected String name = "noname";
    protected IOutputEnvView outEnvView = null;
    protected String terminationEvId = "endAction";
    protected long tStart = 0;
    protected long durationMillis = -1;
    protected QActorContext ctx = null;
    protected T result;
    protected QActor myactor = null;

    public ActionObservableGenericActor(String name, QActor qa, IOutputEnvView outEnvView) {
        this.name          = name.trim();
        this.myactor       = qa;
        this.ctx           = (qa != null) ? qa.getQActorContext() : null;
        this.outEnvView    = outEnvView;
    }
    /*
     * TO BE DEFINED BY THE APPLICATION DESIGNER
     */
    protected abstract void execTheAction(Struct actionInput) throws Exception;
    protected abstract T endOfAction() throws Exception;
```

Listing 1.12. ActionObservableGenericActor<T>

The execTheAction operation is called when the actor receives a string of the form data(X), while endOfAction is executed after the termination of the action, according to the following pattern:

4.1.2 onReceive .

```
    public void onReceive(Object msg) {
//      System.out.println(" %%% ActionObservableGenericActor onReceive: " + msg + " from " + getSender().path() );
        try{
            Struct msgt = (Struct) Term.createTerm(msg.toString()); //check syntax data( ... )
            tStart = Calendar.getInstance().getTimeInMillis();
            execTheAction( msgt );
            result = endActionInternal();
        }catch(Exception e){
            System.out.println(" %%% ActionObservableGenericActor onReceive ERROR " + e.getMessage());
            //stop the actor. we could propagate to the parent (SystemCreationActor)
            getContext().stop( getSelf() ); //LOCAL RECOVERY POLICY
        }
    }
```

Listing 1.13. onReceive<T>

The operation endActionInternal evaluates the duration of the action and prepares (by calling the endOfAction) the result to be sent as a *dispatch* to the actor that activated the action:

4.1.3 endActionInternal .

```
    protected T endActionInternal() throws Exception{
        evalDuration();
        T res = endOfAction();
        String payload = terminationEvId+"(ANAME,RES)".replace("ANAME", name).replace("RES", res.toString() );
        //System.out.println(" %%% ActionObservableGenericActor SEND msg:" + terminationEvId + " to " +
                myactor.getName().replace("_ctrl", ""));
        myactor.sendMsg(terminationEvId, myactor.getName().replace("_ctrl", ""), "dispatch", payload );
        return res;
```

```
    }
```

Listing 1.14. endActionInternal

The application designer must introduce a specialization of the class ActionObservableGenericActor<T>. As an example, let us define an asynchronous computation of a Fibonacci number:

```
System asynchFibo
Dispatch endAction  : endAction(A,R)
Dispatch data       : data(X)

Context ctxAsynchFibo ip[ host="localhost" port=8018 ] //-g cyan

QActor asynchcallerfibo context ctxAsynchFibo{
    Plan init normal[
        javaOp "doAsynAction(\"asynchfibo\",
                  \"it.unibo.actionAsynch.ActionActorFibonacci\", \"30\")"
    ]
    switchTo handleActionEnd

    State handleActionEnd [ println("WAIT FOR ASYNCH ACTION TERMINATION") ]
    transition stopAfter 30000
        whenMsg endAction -> callActionResult

    Plan callActionResult resumeLastPlan[
        //onMsg endAction : endAction(actionFiboAsych,V) -> println( V ) ;
        onMsg endAction : endAction(A,V) -> println( endAction(A,V) )
    ]
 }
/*
 * OUTPUT
"WAIT FOR ASYNCH ACTION TERMINATION"
endAction(asynchfibo,fibo(30,832040,timemsec(58)))
 */
```

Listing 1.15. asynchFibo.qa

4.1.4 ActionActorFibonacci .

The code written by the application designer as an extension of the model is:

```
package it.unibo.actionAsynch;
import alice.tuprolog.Struct;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.action.ActionObservableGenericActor;
import it.unibo.qactors.akka.QActor;

public class ActionActorFibonacci extends ActionObservableGenericActor<String> {
private long myresult = 0;
private int n       = 0;
    public ActionActorFibonacci(String name, QActor actor, IOutputEnvView outEnvView) {
        super(name, actor, outEnvView);
    }
    @Override
    public void execTheAction(Struct actionInput) throws Exception {
      //actionInput : data( n )
      n = Integer.parseInt( actionInput.getArg(0).toString() );
      myresult = fibonacci( n );
//    throw new Exception("simulateFault");  //(1)
    }
    protected long fibonacci( int n ){
        if(  n == 1 || n == 2 ) return 1;
        else return fibonacci(n-1) + fibonacci(n-2);
    }
    @Override
    protected String endOfAction() throws Exception {
        return "fibo("+n+","+this.myresult+",timemsec("+durationMillis+")"+")";
    }
```

```
28 }
```

Listing 1.16. ActionActorFibonacci

5 Messages and events

The syntax for *message* and *event* declarations is:

```
1  Message :         OutOnlyMessage | OutInMessage ;
2  OutOnlyMessage :  Dispatch | Event | Signal | Token ;
3  OutInMessage:     Request | Invitation ;
4
5  Event:     "Event"     name=ID ":" msg = PHead ;
6  Signal:    "Signal"    name=ID ":" msg = PHead ;
7  Token:     "Token"     name=ID ":" msg = PHead ;
8  Dispatch:  "Dispatch"  name=ID ":" msg = PHead ;
9  Request:   "Request"   name=ID ":" msg = PHead ;
10 Invitation: "Invitation" name=ID ":" msg = PHead ;
```

PHead: The PHead rule defines a subset of Prolog syntax:

```
1     PHead : PAtom | PStruct ;
2     PAtom : PAtomString | Variable | PAtomNum | PAtomic ;
3     PStruct : name = ID "(" (msgArg += PTerm)? ("," msgArg += PTerm)* ")";
4     PTerm  : PAtom | PStruct ...
```

The grammar states that the declaration of messages and events (if any) must immediately follow the System sentence, since they represent system-wide information. For example:

```
1  System basicProdCons
2  Dispatch info : info(X)
3  Context ctxBasicProdCons ip [ host="localhost" port=8019 ]
```

Listing 1.17. An example of message declaration

At the moment, only dispatch, request and event are implemented.

5.1 Messages

In the *QActor* metamodel, a message is intended 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.

A message does not force the execution of code: a message m sent from an actor sender to an actor *receiver* can trigger a state transition (see Subsection 5.2) in the receiver. If the receiver is not 'waiting' for a transition including m, the message is enqueued in the receiver queue.

At application-level, we say that a *QActor* works according to a massage-based behaviour, while at the lower level (in the qa-infrastructure, see Subsection 5.6) it works according to the massage-driven (Akka) behaviour.

At application level, a message is denoted by specifying a message type, a name and a payload. For example[10]:

```
Dispatch msgName : msgPayload( PHead )
```

At low level (in the `qa-infrastructure`, see Subsection 5.6) messages are syntactically represented as follows:

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

where:

MSGID	Message identifier
MSGTYPE	Message type (e.g.: dispatch,request,event)
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 at application level to express `guards` (see Subsection 3.0.3) to allow conditional evaluation of *Actions*.

5.1.1 Sending/Receiving messages .

The *QActor* language defines built-in action that allow software designer to send messages and that facilitate the handling of received messages.

For example, the `SendDispatch` grammar rule defines how to write the `forward` of a *dispatch*:

```
SendDispatch: name="forward" dest=VarOrQactor "-m" msgref=[Message] ":" val = PHead ;
VarOrQactor : var=Variable | dest=[QActor] ;
```

Once a message has triggered a state transition, the `onMsg` action (in the *newState*) allows us to select a message and execute actions according to the specific structure of that message.

```
MsgSwitch:  "onMsg" message=[Message] ":" msg = PHead "->" move = StateMoveNormal ;
```

5.1.2 Example: a producer-consumer system .

As an example of a message-based transition, let us introduce a very simple producer-consumer system[11], in which the producer sends two times a *dispatch* to the consumer:

```
System basicProdCons
Dispatch info : info(X)
Context ctxBasicProdCons ip [ host="localhost" port=8019 ]
QActor producer context ctxBasicProdCons{
   Rules{
          item(1).
          item(2).
   }
   State init initial [
          [ !? item(X)] println( producerSending( item(X)) ) ;
          [ !? item(X)] addRule produced( item(X) );
          [ ?? item(X)] forward consumer -m info : info( item(X) ) ;
          delay 500
   ]
   finally repeatPlan 1
 }
 QActor consumer context ctxBasicProdCons{
```

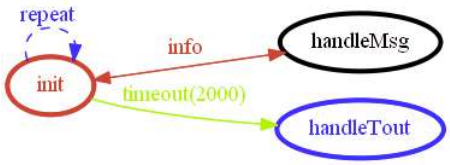
[10] For the syntax of `PHead`, see Subsection ??.

[11] This example is in the project `it.unibo.qa2018.prodcons`.

```
18      State init normal [ println( consumer(waiting) ) ]
19      transition whenTime 2000 -> handleTout
20                 whenMsg info -> handleMsg
21      finally repeatPlan
22      State handleMsg resumeLastPlan [
23          printCurrentMessage ;
24          onMsg info : info(X) -> addRule consumed( X );
25          onMsg info : info(X) -> println( consumerReceiving(X) )
26      ]
27      State handleTout [ println( consumerTout ) ]
28  }
```

Listing 1.18. `basicProdCons.qa`

The state diagram generated by the *QActor* software factory for the consumer is:



Note that the producer stores in its `actorKb` the fact `produced( item(X) )` for each produced element `X`. The consumer instead stores in its `actorKb` the fact `consumed( item(X) )` for each produced element `X` received by the producer. This information is useful for testing purposed, as shown in Subsection 6.2.

The output is:

```
1   /*
2   OUTPUT
3   ---------------------------------------
4   producerSending(item(1))
5   consumer(waiting)
6   -------------------------------------------------------------------------------------------
7   consumer_ctrl currentMessage=msg(info,dispatch,producer_ctrl,consumer,info(item(1)),1)
8   -------------------------------------------------------------------------------------------
9   consumerReceiving(item(1))
10  consumer(waiting)
11  producerSending(item(2))
12  -------------------------------------------------------------------------------------------
13  consumer_ctrl currentMessage=msg(info,dispatch,producer_ctrl,consumer,info(item(2)),2)
14  -------------------------------------------------------------------------------------------
15  consumerReceiving(item(2))
16  consumer(waiting)
17  consumerTout
18  */
```

Listing 1.19. `basicProdCons.qa`

The meaning of a `transition` is discussed in Subsection 5.2.

5.2 State transitions

A transition from a state (*oldState*) to another state (*nextState*) can be specified in different ways, according to the following syntax:

PlanTransition

```
PlanTransition : SwitchTransition | MsgTransition   ;
```

Thus, a state transition sentence can start:

1. with the keyword switchTo: in this case the automaton performs a SwitchTransition (no-input switch, see Subsection 1.3).

```
1 SwitchTransition: "switchTo" nextplantrans = NextPlanTransition ;
2 NextPlanTransition: (guard = Guard)? nextplan=[Plan] ;
```

Note that a transition specification can be prefixed by a guard.

2. with the keyword transition: in this case the automaton performs a MsgTransition that can be triggered by a message or by an event.

```
1 MsgTransition: "transition" duration=Duration (msgswitch+=StateTransSwitch )?("," msgswitch+=StateTransSwitch)* ;
2 Duration : (guard = Guard)? "whenTime" (msec=INT | var=Variable) "->" move=[Plan] ;
3 StateTransSwitch : MsgTransSwitch | EventTransSwitch ;
4
5 MsgTransSwitch: "whenMsg" (guard = Guard)? message=[Message] next=TransSwitch ;
6 EventTransSwitch: "whenEvent" (guard = Guard)? message=[Event] next=TransSwitch ;
```

Thus, a typical state transition involving messages and/or events takes the following form:

A transition specification

```
transition
   whenTime  <timeOut>  ->  <nextState1>
   whenEvent <eventId>  ->  <nextState2>,
   whenMsg   <msgId>    ->  <nextState3>
```

The meaning is:

> the automaton must switch to <nextState1> after <timeOut> milliseconds. In the mean time, it shall switch to <nextState2> if the event named <eventId> occurs or to <nextState3> if the message named <msgId> is sent to the *QActor*.

5.2.1 Switch part .

A TransSwitch grammar rule allows us to specify

- either an explicit new *State/Plan* to reach;
- or an action to be executed as part of an implicit Plan that 'returns' to its caller at the end of its work.

TransSwitch

```
TransSwitch: PlanSwitch | ActionSwitch ;
PlanSwitch:     "->" move = [Plan]  ;
ActionSwitch:  ":" msg = PHead "do" action = StateMoveNormal ;
StateMoveNormal: StateActionMove | OutMessageMove |  ActionDelay | ExtensionMove |
                    BasicMove | StatePlanMove | GuardMove | BasicRobotMove ;
```

An example of PlanSwitch has been given in Subsection 5.1.2. A simple example of ActionSwitch is:

```
System demoActionSwitch
Dispatch info : payload( CONTENT )
Context ctxDemoActionSwitch ip [ host="localhost" port=8019 ]

 QActor sendertoself context ctxMsgToSelf{
     Plan init initial [
          selfMsg info : payload( "helloWorld" )
     ]
     transition stopAfter 100
           whenMsg info : payload(V) do println(V)
 }
```

The meaning of the *QActor* operation selfMsg is that the actor forwards the dispatch to itself.

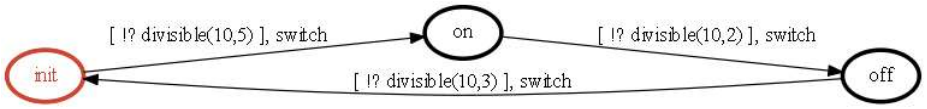
5.3 Guarded transitions

Also transitions can be prefixed by a [GUARD]; in this case the transition 'fires' only if the GUARD is evaluated true. Let us show a very simple example:

```
System guardedTransitions
Context ctxGuardedTransitions ip [ host="localhost" port=8037 ]
QActor qaguardedtrans context ctxGuardedTransitions {
    Plan init normal
        [   println("qaguardedtrans STARTS" ) ]
        switchTo [ !? divisible(10,5) ] on
    Plan on
        [   println( on )  ]
        switchTo [ !? divisible(10,2) ] off
    Plan off
        [   println( off )    ]
        switchTo [ !? divisible(10,3) ] init //possible loop!
}
/*
OUTPUT
----------------------------
"qaguardedtrans STARTS"
on
off
*/
```

Listing 1.20. guardedTransitions.qa

The generated state diagram is:



More interesting examples will be given in the following.

5.3.1 Example: action after a self-message .

A more interesting example is given in the next section

```
System msgToSelf
Dispatch execute : code( CODE )
Dispatch eval    : code( CODE )

Context ctxMsgToSelf ip [ host="localhost" port=8019 ]
```

```
6
7  QActor sendertoself context ctxMsgToSelf{
8      Plan init initial [
9           selfMsg eval : code( fibo(10,V) )
10     ]
11     transition whenTime 100 -> handleTout
12           whenMsg execute -> execProg ,
13           whenMsg eval    : code(P) do selfMsg execute : code(P)
14        finally repeatPlan 1
15
16     State execProg resumeLastPlan [
17        onMsg execute : code(P) -> demo P;
18        [ ?? goalResult(V) ] println(V)
19     ]
20     State handleTout
21        [ println( consumerTout ) ]
22 }
23 /*
24 OUTPUT
25 ----------------------------------------
26 fibo(10,55)
27 ----------------------------------------
```

Listing 1.21. `msgToSelf.qa`

To understand the output, let us trace the behavior of the actor:

1. The actor sends an `eval` messages to itself.
2. The `eval` message is handled by the actor that sends the message `execute` to itself.
3. The Plan `init` is repeated: another `eval` messages is sent by the actor to itself.
4. Now there are two messages 'pending'. The first considered for a transition is `execute`.
5. The actor performs a transition to the state `execProg` and then terminates.

5.4 Events

In the *QActor* metamodel, an `event` is intended as information emitted by some source without any explicit destination. Events can be *emitted* by the *QActors* that compose a *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 (see Subsection 5.5).

An event can also trigger state transitions in components, usually working as finite state machines. We qualify this kind of behaviour as `event-based` behaviour, since the event is 'lost' if no actor is in a state waiting for it.

At application level, an event is denoted by specifying the key-word `Event`, a name and a payload. For example[12]:

```
1 Event evName : evPayload( PHEAD )
```

At low level (in the `qa-infrastructure`, see Subsection 5.6) events are syntactically represented as messages with no destination (`RECEIVER=none`):

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

The *QActor* language defines built-in actions that allow software designer to emit events and actions that facilitate the handling of perceived events.

The `RaiseEvent` grammar rule specifies an operation that allow us to `emit` an event:

[12] For the syntax of `PHead`, see Subsection **??**.

```
RaiseEvent : name="emit" ev=[Event] ":" content=PHead ;
```

Once an event has triggered a state transition, the onEvent action allows us to execute actions according to the specific structure of that event.

```
EventSwitch: "onEvent" event=[Event] ":" msg = PHead "->" move = StateMoveNormal ;
```

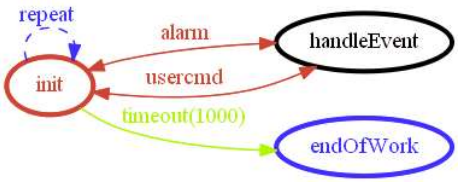
5.4.1 Example: event-based behavior .

As an example of a event-based transition, let us introduce a very simple system, in which an actor works as event-emitter and another actor that handles the emitted events:

```
/*
 * basicEvents.qa
 */
System basicEvents
Event usercmd : usercmd(X)
Event alarm  : alarm(X)

Context ctxBasicEvents ip [ host="localhost" port=8037 ]

QActor qaeventemitter context ctxBasicEvents {
    State init initial
    [  println("qaeventemitter STARTS ") ;
        delay 500 ;   //(1)
        println("qaeventemitter emits alarm(fire)") ;
        emit alarm : alarm(fire) ;
        delay 500 ;   //(2)
        println("qaeventemitter emits usercmd(hello)") ;
        emit usercmd : usercmd(hello) ;
        println( "qaeventemitter ENDS" )
    ]
}
QActor qaeventperceptor context ctxBasicEvents {
    State init initial
        [  println("qaeventperceptor STARTS ") ]
        transition whenTime 1000 -> endOfWork
            whenEvent alarm -> handleEvent,
            whenEvent usercmd -> handleEvent
        finally repeatPlan

    State handleEvent resumeLastPlan
        [  println("ex4_perceptor handleEvent " ) ;
            printCurrentEvent ;
            onEvent alarm : alarm(X)  -> println( handling(alarm(X)) ) ;
            onEvent usercmd : usercmd(X) -> println( handling(usercmd(X)) )
        ]
    State endOfWork
        [  println("qaeventperceptor ENDS (tout) ") ]
}
```

Listing 1.22. `basicEvents.qa`

The state diagram generated by the *QActor* software factory for the consumer is:



The output is

```
/*
OUTPUT
----------------------------
"qaeventemitter STARTS "
"qaeventperceptor STARTS "
"qaeventemitter emits alarm(fire)"
"ex4_perceptor handleEvent "
-----------------------------------------------------------------------------------------
qaeventperceptor_ctrl currentEvent=msg(alarm,event,qaeventemitter_ctrl,none,alarm(fire),3)
-----------------------------------------------------------------------------------------
handling(alarm(fire))
"qaeventperceptor STARTS "
"ex4_alarmemitter emits usercmd(hello)"
"ex4_perceptor handleEvent "
-----------------------------------------------------------------------------------------
qaeventperceptor_ctrl currentEvent=msg(usercmd,event,qaeventemitter_ctrl,none,usercmd(hello),7)
-----------------------------------------------------------------------------------------
handling(usercmd(hello))
"qaeventperceptor STARTS "
"qaeventemitter ENDS"
"qaeventperceptor ENDS (tout)
*/
```

Listing 1.23. `basicEvents.qa`

Note that if we comment the `delay (1)` and `(2)`, the emitted events are `not perceived` by the `qaeventperceptor`, since it has no time to enter its `transition` phase before the event emission.

5.5 Event handlers and event-driven behaviour

The occurrence of an event activates, in `event-driven` way, all the `EventHandlers` declared in actor *Context* for that event, with the following syntax:

```
EventHandler :
    "EventHandler" name=ID ( "for" events += [Event] ( "," events += [Event] )* )?
    ( print ?= "-print") ?
    ( "{" body = EventHandlerBody "}" )?
    ";"
EventHandlerBody: op += EventHandlerOperation (";" op += EventHandlerOperation)* ;
```

The syntax shows that, in a `qa` model, we can express only a limited set of actions within an `EventHandler`[13]:

```
EventHandlerOperation: MemoOperation | SolveOperation | RaiseOtherEvent | SendEventAsDispatch ;

MemoOperation: doMemo=MemoCurrentEvent "for" actor=[QActor] ;
MemoCurrentEvent : "memoCurrentEvent" (lastonly?="-lastonly")? ;

SolveOperation: "demo" goal=PTerm "for" actor=[QActor] ;

RaiseOtherEvent: "emit" ev=[Event] ("fromContent" content = PHead "to" newcontent=PHead )? ;

SendEventAsDispatch: "forwardEvent" actor=[QActor] "-m" msgref=[Message] ;
```

- `MemoOperation`: memorize and event into the *WorldThery* of a specific *QActor*
- `SolveOperation`: 'tell' to a specific *QActor* to solve a goal
- `RaiseOtherEvent`: emit another event with the content of the event
- `SendEventAsDispatch`: forward a dispatch with the content of the event

The `SolveOperation` rule sends an 'internal system message' to the specific *QActor* and does not force any immediate execution within that *QActor*.

In the example that follows, the system reacts to all the events by storing them in the knowledge base (*WorldTheory*) related to a event tracer actor, that periodically shows the events available.

[13] Of course, other actions can be defined directly in Java by the Application designer.

```
/*
 * eventTracer.qa
 */
System eventTracer
Event usercmd   : usercmd(X)
Event alarm     : alarm(X)

Context ctxEventTracer ip [ host="localhost" port=8027 ]
EventHandler evh for usercmd,alarm -print {
    memoCurrentEvent for qaevtracer
};
//WARNING: any change in the model modifies the EventHandlers
QActor qaevtracer context ctxEventTracer {
    Plan init normal
    [  println("qaevtracer starts") ;
       [ ?? msg(E,'event',S,none,M,N) ]
          println(qaevtracer(E,S,M)) else println("noevent") ;
       delay 300
    ]
    finally repeatPlan 5
}
QActor qatraceremitter context ctxEventTracer {
    Plan init normal
    [  println("qatraceremitter STARTS ") ;
       delay 500 ;   //(1)
       println("qatraceremitter emits alarm(fire)") ;
       emit alarm : alarm(fire) ;
       delay 500 ;   //(2)
       println("qatraceremitter emits usercmd(hello)") ;
       emit usercmd : usercmd(hello) ;
       println( "qaeventemitter ENDS" )
    ]
}
```

Listing 1.24. `eventTracer.qa`

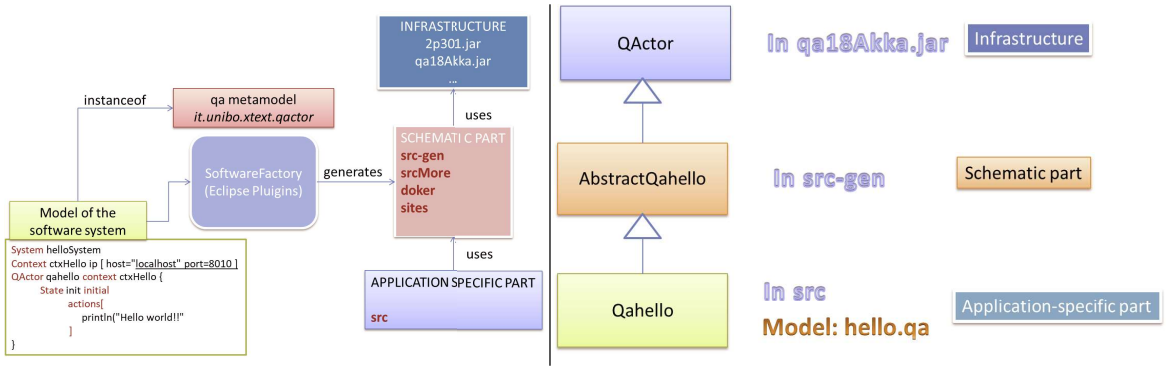
The output is:

```
/*
OUTPUT
----------------------------
"qatraceremitter STARTS "
"qaevtracer starts"
"noevent"
"qaevtracer starts"
"noevent"
"qatraceremitter emits alarm(fire)"
>>> evh          (defaultState, TG=01:00:00)          || msg(alarm,event,qatraceremitter_ctrl,none,alarm(fire),5)
"qaevtracer starts"
qaevtracer(alarm,qatraceremitter_ctrl,alarm(fire))
"qaevtracer starts"
"noevent"
"qatraceremitter emits usercmd(hello)"
>>> evh          (defaultState, TG=01:00:00)          || msg(usercmd,event,qatraceremitter_ctrl,none,usercmd(hello),7)
"qaeventemitter ENDS"
"qaevtracer starts"
```

Listing 1.25. `eventTracer.qa`

5.6 The qa-infrastructure

A *QActor* model aims at capturing main architectural aspects of a software system, by allowing a graceful transition from object-oriented programming to message-based and event-based computations. Moreover, a *QActor* model provides a support for rapid software prototyping, since the *QActor* software factory creates the software layer that 'adapts' the application layer to the infrastructure layer ('schematic part' in the picture):



The *QActor* infrastructure is mainly provided by the library qa18Akka.jar (project it.unibo.qactors) that is in its turn based on other custom libraries, including the following ones:

uniboInterfaces.jar	includes a set of interfaces (project *it.unibo.interfaces*).
uniboEnvBaseAwt.jar	a framework for (graphical) user interfaces (project *it.unibo.envBaseAwt*).
unibonoawtsupports.jar	a support for two-way protocols (TCP,UDP,etc). (project *it.unibo.noawtsupports*)

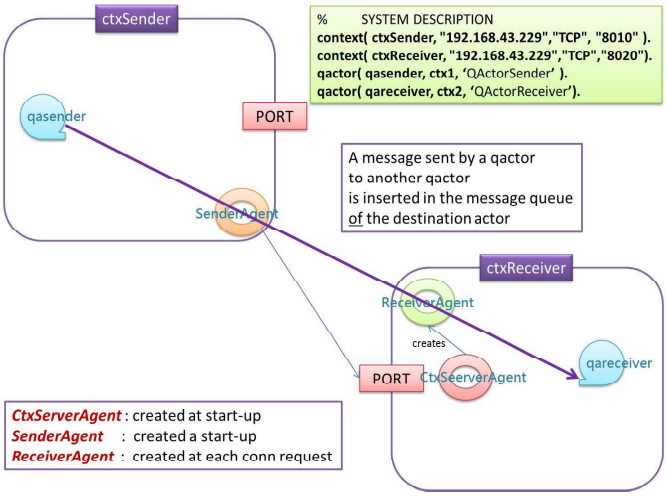
For each *Context*, the *QActor* software factory generates:

- in the directory src-gen:
  - a class for each Context, in a package related to that Context, that contains a main program that performs the initialization of the Context;
  - an abstract class for each *QActor*, in a package related to that *QActor*;
- in the directory src:
  - in the package related to a Context, a class that extends the abstract class generated for that Context in the src-gen directory;
  - in the package related to a *QActor*, a class that extends the abstract class generated for that actor in the src-gen directory;
- in the directory srcMore:
  - for each Context, a high-level description of the Context and of the actors that work in each of them. This `Context Knowledge Base` or simply ContextKB is identical for each context of the system and automatically consulted at the system start-up.
    Example: for a Context named aCtx, the ContextKB is written in a file named aCtx.pl within the directory srcMore/it.unibo.aCtx. In the same directory, it is generated also the file sysRules.pl, that includes the `tuProlog` rules used by the `qa run-time system` (see also Subsection 2.2.1).

The main program that initializes a Context, creates an Akka actor-system and a `Akka` SystemCreationActor that work as the 'father' of all the other actors in that *Context*. In its turn, the SystemCreationActor creates:

- a Akka actor of class EventLoopActor. This actor manages the events raise in the system;
- a Akka actor of class CtxServerAgent. This actor implements a server that manages connections requests coming from other Contexts;

– a Akka actor for each *QActor* defined in the context.



For each connection, the `CtxServerAgent` creates an Akka actor of class `ReceiverAgent`. This actor handles the message sent on that connection, by dispatching a `message` to the (local) destination actor and an `event` to the `EventLoopActor`.

5.7 Example: a distributed producer-consumer system

Once a system is creates and tested in a local environment (i.e. within a single Context/JVM), it can evolve into a distributed system by allocating one or more actors on different, new Contexts. As an example, let us transform the single-Context producer-consumer system of Subsection 5.1.2 into a distributed one.

```
System prodConsDistributed
Dispatch info : info(X)
Context ctxBasicProdDistr ip [ host="localhost" port=8079 ]
Context ctxBasicConsDistr ip [ host="localhost" port=8089 ] -g green
QActor producerdistr context ctxBasicConsDistr{
       Rules{
           item(1).
           item(2).
       }
   State init initial
       [   [ !? item(X) ] println( producer(sends(X)) ) ;
           [ ?? item(X)] forward consumerdistr -m info : info(X)
                                 else endPlan "producer(ends)"
       ]
       finally repeatPlan
}
QActor consumerdistr context ctxBasicConsDistr{
   State init initial [ println( consumer(waiting) ) ]
   transition whenTime 2000 -> handleTout
       whenMsg info : info(X) do println( consumerHandles(X) )
       finally repeatPlan
   State handleTout [
        println( consumer(tout) ) ;
         delay 10000  //to avoid immediate GUI disappearance
```

Listing 1.26. `prodConsDistributed.qa`

To run the system, the Application designer must activate (in any order and within a prefixed amount of time) the main programs generated from the two contexts `ctxBasicProd` and `ctxBasicCons`. The Application code (i.e. the code written into the actors) begins to run only when **all the Contexts** are activated.

6 Automated Testing

Testing is a process intimately related to the development process. In particular, the *iterative* development process methodology highlights testing, by proposing some popular motto:

Analyse a little. Design a little. Code a little. Test what you can.
Test early. Test often. test enough.

Among the main testing strategies, we can recall the following ones:

- *Black box testing*: testing on the target public `API` without knowledge of the target source code.
- *White box testing*: testing with knowledge of the target source code.

.

6.1 Types of testing

The main types of testing can be summarized as follows:

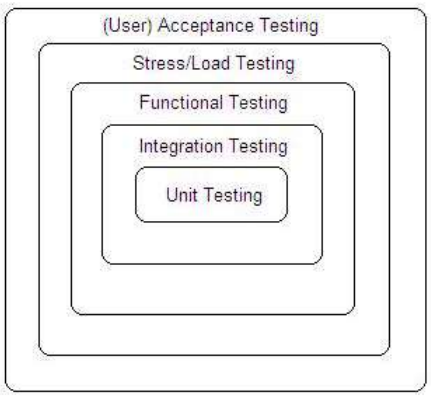


Fig. 1. Test types

- *Unit Testing*: testing single units of work.
- *Integration Testing*: testing how different units of work interact.
- *Functional Testing*: testing subsystems (usually on a boundary `API`).
- *Stress/Load Testing*: testing the system performance.
- *(User) Acceptance Testing*: testing the system as a user.

With reference to our motto:

There is no code without project, no project without problem analysis and no problem without requirements.

---

the goal here is to automate the testing phase as soon as possible, by starting from the models built during the requirement analysis or the problem analysis.

6.2 Testing of the producer-consumer system

Here is an example of Testing written by the application designer with reference to `Junit4`. From the logical point of view, the test is an acceptance test that can be planned before writing any code:

> The system works correctly if each element produced by the producer is consumed (in the same order) by the consumer.

Thus, during the test plan activity, we assume to have the possibility to know the sequence of elements produced and the sequence of elements consumed. This kind of knowledge can be stored in a `log file` or on a database (e.g. `Mongo`) or published on a `MQTT` broker, and so on. In this example we will base the test on the facts stored by the `producer` and the `consumer` actors in their `actorKb`. More precisely, the testing activity will work as follows:

1. The test unit creates the 'real' system by exploiting the static method `initTheContext()` defined in the generated class `MainCtxBasicProdCons`.
2. The test unit waits for a while, until the system has completed its job.
3. The test unit looks at the `actorKb` of the `producer` and of the `consumer` and checks that they include the same sequence of produced/consume elements.

```
1  public class TestProdCons {
2  private Producer producer;
3  private Consumer consumer;
4
5      @Before
6      public void setUp(){
7              System.out.println("================== setUp ==================");
8                  try {
9                      System.out.println("setUp STARTS producer="+producer );
10                     it.unibo.ctxBasicProdCons.MainCtxBasicProdCons.initTheContext();
11                     Thread.sleep( 1000 ); //give the time to start and execute
12                     producer = (Producer) QActorUtils.getQActorForTesting("producer");
13                     consumer = (Consumer) QActorUtils.getQActorForTesting("consumer");
14                     System.out.println("setUp ENDS producer="+producer );
15                 } catch (Exception e) {
16                       fail( e.getMessage() ) ;
17                 }
18     }
19
20     private void checkSolution(SolveInfo solProd, SolveInfo solCons) {
21         if( ! solProd.isSuccess() || ! solCons.isSuccess() ) {
22             System.out.println("test1 NO SOLUTION FOUND");
23             fail("NO SOLUTION FOUND");
24         }
25     }
26     @Test
27     public void test(){
28         System.out.println("&&&&&&&&&&&&&&&& test &&&&&&&&&&&&&&&&&&&&&&&&");
29         try {
30             while( producer == null || consumer == null ) { System.out.print(".");Thread.sleep(400);}
31             Prolog enginProd = producer.getPrologEngine();
32             Prolog engineCons = consumer.getPrologEngine();
33             SolveInfo solProd = enginProd.solve("produced(ITEM).");
34             SolveInfo solCons = engineCons.solve("consumed(ITEM).");
35             checkSolution(solProd, solCons);
36             String itemProd = solProd.getVarValue("ITEM").toString();
37             String itemCons = solCons.getVarValue("ITEM").toString();
38             System.out.println("test-1 itemProd="+itemProd + " itemCons="+ itemCons );
39             assertTrue( "test", itemProd.equals(itemCons) );
40             while( enginProd.hasOpenAlternatives() && enginProd.hasOpenAlternatives() ) {
41                 solProd   = enginProd.solveNext();
```

```
42              solCons   = engineCons.solveNext();
43              checkSolution(solProd, solCons);
44              itemProd  = solProd.getVarValue("ITEM").toString();
45              itemCons  = solCons.getVarValue("ITEM").toString();
46              System.out.println("test-2 itemProd="+itemProd + " itemCons="+ itemCons );
47              assertTrue( "test", itemProd.equals(itemCons) );
48          }
49          boolean b1 = engineProd.hasOpenAlternatives();
50          boolean b2 = engineProd.hasOpenAlternatives();
51          if( (b1 && ! b2) || (! b1 && b2 ) ) fail("test: DIFFERENT NUMBER OF ITEMS ");
52      } catch (Exception e) {
53          System.out.println("test ERROR: "+ e.getMessage() );
54          fail("test ERROR: "+ e.getMessage());
55      }
56  }
57
58
59  }
```

Listing 1.27. `TestProdCons.java`

Now, let us run the `build` task of the generated `gradle` file associated to the Context,

```
1  gradle -b build_ctxBasicProdCons.gradle build
```

This activity will create the `build` directory in the project workspace with the test reports generated by `gradle` and by `jacoco`.

6.3 Introspection

The knowledge about a context configuration can be acquired at application level by rules like the following ones:

```
1  System introspection
2
3  Context ctxInspect ip [ host="localhost" port=8040 ]
4
5  QActor inspector context ctxInspect{
6      State init initial [
7          demo consult("curConfigTheory.pl")
8      ]
9      switchTo work
10
11     Plan work [
12         demo showSystemConfiguration;
13         println( "inspector READY" )
14     ]
15
16 }
```

Listing 1.28. `introspection.qa`

```
1  getTheContexts(Ctx,CTXS) :-
2      Ctx <- solvegoal("getTheContexts(X)","X") returns CTXS .
3  getTheActors(Ctx,ACTORS) :-
4      Ctx <- solvegoal("getTheActors(X)","X") returns ACTORS.
5
6   /*
7   ---------------------------------
8  Show system configuration
9   ---------------------------------
10  */
11 showSystemConfiguration :-
12     actorPrintln('-------------------------------------------------'),
13     actorobj(A),
```

```
14      A <- getQActorContext returns Ctx,
15      %% Ctx <- getName returns CtxName, actorPrintln( CtxName ),
16      getTheContexts(Ctx,CTXS),
17      actorPrintln('CONTEXTS IN THE SYSTEM:'),
18      showElements(CTXS),
19      actorPrintln('ACTORS IN THE SYSTEM:'),
20      getTheActors(Ctx,ACTORS),
21      showElements(ACTORS),
22      actorPrintln('-------------------------------------------------').
23
24  showElements(ElementListString):-
25      text_term( ElementListString, ElementList ),
26      %% actorPrintln( list(ElementList) ),
27      showListOfElements(ElementList).
28  showListOfElements([]).
29  showListOfElements([C|R]):-
30      actorPrintln( C ),
31      showElements(R).
```

Listing 1.29. curConfigTheory.pl

7 Built-in GUI interfaces

Any modern (IOT) application should provide the possibility to be used by humans or by other machines by means of a web interface.

In order to reduce the time required to build human web-based interfaces during the system prototyping phase, the *QActor* software factory does introduce a built-in support, by creating a HTTP web-socket server on port 8080 when the -httpserver flag for a Context is set. The factory generates also (just once) a set of HTML pages in the in the package of the srcMore directory associated with that *Context*.

7.1 Built-in web server

The page that provide the web interface is shown in the following picture:

All the buttons are implemented as devices that emit some predefined *QActor* event. In particular:

- The RUN button at the top of the GUI allows an human user to emit string commands
- The buttons at the bottom are introduced to emit prefixed commands and send alarms

This interface emits the following events:

`inputcmd : usercmd(executeInput(CMD))`	(RUN button)
`usercmd : usercmd(robotgui(w(low)))`	(FORWARD button)
`alarm : alarm(fire)`	(FIRE button)
`obstacle : obstacle(X)`	(OBSTACLE button)
`cmd : cmd(start)`	(START button)
`cmd : cmd(stop)`	(STOP button)

This built-in behaviour can be modified by the Application designer by changing the given HTML page (QActorWebUI.html). However, these events usually provide a sort of '**internal standard**' that increases understanding and promotes fast prototyping.

When the -httpserver flag is set, a simple HTTPserver[14] is created and started during the Context configuration phase. This server is based on the WebSocket technology [15] and answers to HTTP requests on

[14] The class of the server is it.unibo.qactors.web.QActorHttpServer) (in the infrastructure library qa18Akka.jar).

[15] The *WebSocket* specification defines an API establishing "socket" connections between a web browser and a server, i.e. a persistent client-server connection so that both parties can start sending data at any time.

port 8080 by returning the web page named QActorWebUI.html stored in the context package generated in the srcMore directory.

The QActorHttpServer server performs different actions according to the prefix of the string sent by the page QActorWebUI.html.

7.1.1 Input string prefix m- .

In this case the rest of the input string is assumed to be a qa message of the form:

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

The QActorHttpServer sends the message to the RECEIVER actor.

For the default page, there no case of this type, that is instead present in the Led web page.

7.1.2 Input string prefix i- .

In this case the rest of the input string is assumed to be a user command (CMD) that is translated into a string of the form:

```
usercmd( executeInput( MOVE ) )
```

where MOVE can take one of the following forms:

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

For the default page, this is the case of the user-command interface; the QActorHttpServer emits the following event:

usercmd : usercmd(executeInput(CMD)) (RUN button)

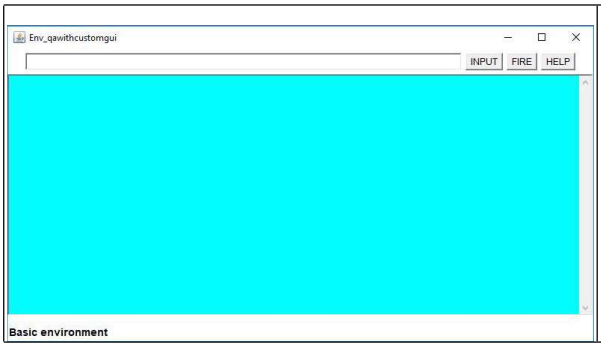
7.1.3 Input string without special prefix .

In this case, the string is assumed to be written in Prolog syntax. For the default page, this is the case of the robot-console buttons, that generate input strings of the form:

```
e(alarm(fire))
e(alarm(obstacle))
w( high )
...
```

7.2 The local actor-GUI interface

When the -g flag for an Actor is set, the *QActor* software factory generates a `GUI` interface for that actor, whose form shown in the following picture:



The `INPUT` button generates the event :

```
local_inputcmd : usercmd(executeInput(CMD))
```

where `CMD` is the content of the input field on the left.

The Application designer can modify this Actor-specific `GUI` with two actions:

1. exclude the built-in panels by redefining with an empty body the following methods:

   ```
   protected void addInputPanel(int size){  }
   protected void addCmdPanels(){  }
   ```

2. create his/her own Panels and inject (with the built-in method `addPanel`) them in the built-in `GUI` environment.

Of course, the Application designer can avoid to introduce the built-in interface support (the -g flag) and define application-specific methods for provide the actor with a custom `GUI` interface.

8 Components written in other languages

A modern (IOT) software system is composed of several computational nodes (it is a distributed system), each providing one or more (micro)services implemented in their proper language/infrastructure (it is an heterogeneous system).

A *QActor* model aims at describing the architecture of a distributed software system, by focussing the attention on the system components and their interaction. Since a *QActor* model is also executable, it can be used in the early stages of software development to provide working prototypes that help in the interaction with the customer and with the final users. However, when we execute a *QActor* model, we exploit a runtime support implemented in Java based on TCP node interactions, while the application could demand for the usage of different languages and infrastructures.

In this section we explore the possibility to use *QActor* models as a sort of system integrator, that overcomes the previous constraints.

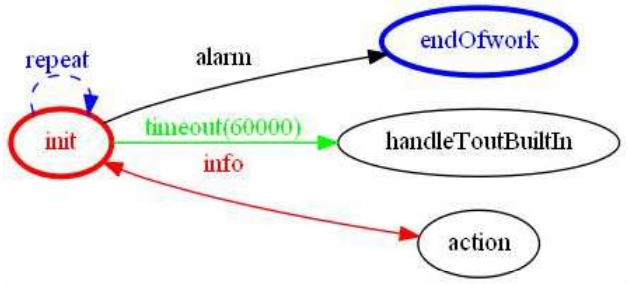
8.1 Using Node.js

A first example of the possibility to use a *QActor* model as an integrator of heterogeneous distributed activities is given hereunder:

```
System jsToQa
Event alarm  : alarm(X)
Dispatch info : info(X)

Context ctxJsToQa ip[ host="localhost" port=8031]
EventHandler evh for alarm -print;
/*
 * This actor activates a NodeJs process that sends messages to the qareceiver
 */
QActor qajaactoivator context ctxJsToQa {
    Plan init normal [
        println("qajaactoivator START") ;
        nodeOp "./nodejsCode/TcpClientToQaNode.js localhost 8031" -o ;
//      actorOp runNodeJs("./nodejsCode/TcpClientToQaNode.js localhost 8031", "true") ;
        //[?? actorOpDone(OP,R)] println( rrr(R) );
        println("qajaactoivator END")
    ]
}

/*
 * This actor handles message sent by the NodeJs process by distinguishing between
 * messages (dispatch info : info(X)) and events (alarm : alarm(X)).
 */
QActor qareceiver context ctxJsToQa {
    Plan init normal [
        println("qareceiver WAITS")
    ]
    transition stopAfter 60000
            whenEvent alarm -> endOfwork ,
            whenMsg info : info(X) do printCurrentMessage
    finally repeatPlan

    Plan endOfwork [
        onEvent alarm : alarm(X) -> println( endOfwork(X) )
    ]

 }
```

Listing 1.30. jsToQa.qa

The system is composed of a NodeJs client that sends messages (and events) to the qareceiver that implements the finite state machine shown in the picture:



The NodeJs client is activated by the *QActor* system itself by means of the `qajaactoivator`. Of course, such a client could be activated as an independent activity through a command like:

```
node TcpClientToQaNode.js localhost 8031
```

8.2 The operation `runNodeJs`

The operation that activates a `NodeJs` computation should be written by the application designer. For example:

```
protected String runNodeJs(String prog, String showOutput){
    try {
        String cmd = "node "+ prog;
        java.lang.Process nodeExecutor = runtimeEnvironment.exec("node "+prog);
        if( showOutput.equals("false") ) return "";
        InputStream nodeInputStream = nodeExecutor.getInputStream();
        return getOutput(prog,nodeInputStream);
    } catch (Exception e) {
        println("      " + prog+ " WARNING >"+ e.getMessage() ) ;
        return "";
    }
}
```

The method `getOutput` is called when we want to see what the `NodeJs` computation writes on the `console.log`.

However, a method with this signature has been defined in the `QActor` class, to provide a built-in operation to activate a `NodeJs` computation; the string `*** nodjs>` will denote its output. Moreover, the buil-in operation `nodeOp` of the *QActor* language is translated in a call to the `runNodeJs` method.

8.2.1 The NodeJs client .

The first activity of the NodeJs client is to establish a connection with a server host whose IP is given (together with a port) as argument at start-up (e.g. `node TcpClientToQaNode.js localhost 8031`):

```
/*
* ======================================
* TcpClientToQaNode.js
* ======================================
*/
var net = require('net');
var host = process.argv[2];
var port = Number(process.argv[3]);  //23 for telnet

console.log('connect to ' + host + ":" + port);
var socket = net.connect({ port: port, host: host });
console.log('connect socket allowHalfOpen= ' + socket.allowHalfOpen );
socket.setEncoding('utf8');

// when receive data back, print to console
socket.on('data',function(data) {
    console.log(data);
});
// when server closed
```

```
socket.on('close',function() {
    console.log('connection is closed');
});
socket.on('end',function() {
    console.log('connection is ended');
});
/*
 * TERMINATION
 */
process.on('exit', function(code){
    console.log("Exiting code= " + code );
});
//See https://coderwall.com/p/4yis4w/node-js-uncaught-exceptions
process.on('uncaughtException', function (err) {
cursor.reset().fg.yellow();
    console.error('got uncaught exception:', err.message);
cursor.reset();
```

Listing 1.31. `TcpClientToQaNode.js: connect`

The NodeJs client provides also some utility function to send messages:

```
================================================================
*/
function sendMsg( msg ){
    try{
        socket.write(msg+"\n");
    }catch(e){
        console.log(" ------------------ EVENT " + e );
    }
}
function sendMsgAfterTime( msg, time ){
    setTimeout(function(){
        //println("SENDING..." + msg );
        sendMsg( msg ); },
```

Listing 1.32. `TcpClientToQaNode.js: utilities`

Finally, the NodeJs client send some application-level message:

```
================================================================
*/
var msgNum=1;
sendMsgAfterTime("msg(info,dispatch,jsSource,qareceiver,info(ok1)," + msgNum++ +")", 200);
sendMsgAfterTime("msg(info,dispatch,jsSource,qareceiver,info(ok2)," + msgNum++ +")", 500);
sendMsgAfterTime("msg(alarm,event,jsSource,none,alarm(obstacle)," + msgNum++ +")", 700);
sendMsgAfterTime("msg(info,dispatch,jsSource,qareceiver,info(ok3)," + msgNum++ +")", 1000);
```

Listing 1.33. `TcpClientToQaNode.js: application`

8.2.2 The result .

The NodeJs client sends a sequence of 3 messages, but before the last one, it 'emit' an alarm event. Thus, the output of the system is:

```
"qajaactoivator START"
"qareceiver WAITS"
              *** nodjs> connect to localhost:8031
              *** nodjs> connect socket allowHalfOpen= false
   %%% CtxServerAgent ctxjstoqa_Server WORKING on port 8031
-------------------------------------------------------------------------------------------
qareceiver_ctrl currentMessage=msg(info,dispatch,jsSource,qareceiver,info(ok1),1)
-------------------------------------------------------------------------------------------
"qareceiver WAITS"
-------------------------------------------------------------------------------------------
qareceiver_ctrl currentMessage=msg(info,dispatch,jsSource,qareceiver,info(ok2),2)
-------------------------------------------------------------------------------------------
```

```
"qareceiver WAITS"
endOfwork(obstacle)
    %%% SystemCreationActor terminates1:qareceiver_ctrl 1/4
    %%% SystemCreationActor terminates1:qareceiver 2/4
>>> evh          (defaultState, TG=01:00:00)          || msg(alarm,event,jsSource,none,alarm(obstacle),3)
...
              *** nodjs> SOCKET END
"qajaactoivator END"
...
```