Introduction to QActors and QRobots (2017)

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 ta custom 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':

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

QActor qahello context ctxHello {
    Plan init normal
        actions[
            println("Hello world" )
        ]
}
```

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 **??**).

A *QActor* must define at least one `Plan`, qualified as '`normal`' to state that it represents the starting work of the actor. Each *Plan* of a *QActor* represents the state of a Moore finite state machine (see also Subsection **??**), whose `actions` are defined in a proper section of a *Plan*, enclosed within the 'brackets' `actions[ ... ]`[1].

A *QActor* is intended to be a software component that can perform application *Tasks* by interacting with other *QActor*s by means of `messages` (see Subsection **??**) or `events` (see Subsection **??**). Thus, state transitions can be performed with no-input moves or when a *message* is or an *event* is available. Some simple example will be given in the following sections.

1.2 Example: no-input transitions

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

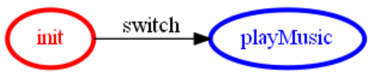
```
/*
 * noinputTansition.qa
 */
System noinputTansition
Context ctxNoinputTansition ip [ host="localhost" port=8079 ]

QActor qahellonoinputtrans context ctxNoinputTansition{
    Plan init normal
        [  println("Hello world" ) ]
        switchTo playMusic
```

[1] The key word `actions` can be omitted

```
    Plan playMusic
        [   println( playSomeMusic );
            sound time(1500) file('./audio/tada2.wav')
        ]
 }
/*
OUTPUT
----------------------------
"Hello world"
playSomeMusic
*/
```

Listing 1.2. `noinputTansition.qa`

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, among many other things, by the *QActor* software factory.

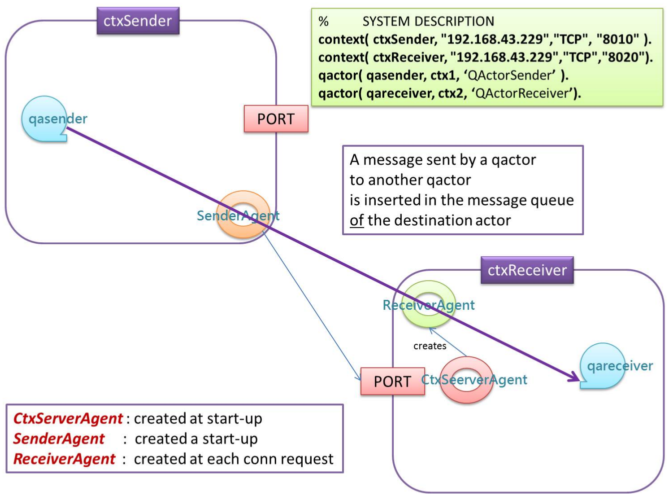
1.3 The *QActor* software factory

The *QActor* 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 application logic. The `qa`software factory is implemented as a set of Eclipse plug-ins, built by using the `XText` framework.

For each *Context*, the *QActor* software factory generates an Akka actor-system and a `SystemCreationActor` that work as the 'father' of all the other actors in that *Context*. More specifically, this actor creates:

– a Akka actor of class `EventLoopActor`
– a Akka actor of class `CtxServerAgent`
– a Akka actor for each *QActor* defined in the context

The `CtxServerAgent` allows messages exchanged among *QActor* working on different contexts to flow throw the context ports (using the `TCP/IP` protocol) and to deliver the message in the message-queue of the destination *QActor*.

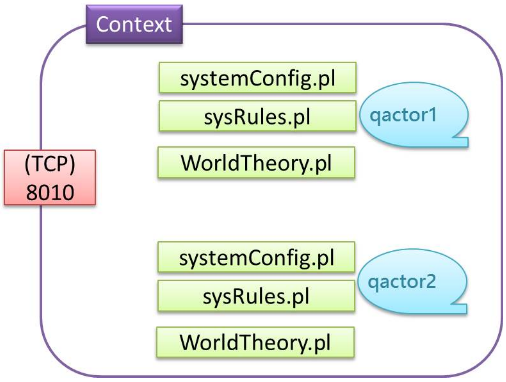


For each connection, an Akka actor of class ReceiverAgent is created. This actor handles the message sent on that connection, by sending a message to the destination actor and an event to the EventLoopActor.

1.4 The *QActor* knowledge-base

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

Each actor requires a set of configuration files, each storing a description written in tuProlog syntax. The picture hereunder shows the set of tuProlog theories associated with each actor:



– The theory systemConfig.pl (the name can be different from system to system) describes the configuration of the system.
– The theory sysRules.pl describes a set of rules used at system configuration time.
– The theory WorldTheory.pl describes a set of rules and facts that give a symbolic representation of the "world" in which a *QActor* is working.

In the case of the example of Subsection 1.1:

– The file that describes the system configuration (named hellosystem.pl) is generated in the directory srcMore/it/unibo/ctxHello.
– The file sysRules.pl it generated in the directory srcMore/it/unibo/ctxHello.
– The WorldTheory.pl is generated in the directory srcMore/it/unibo/qahello.
– The state diagram (named qahell.gv) is generated in the directory srcMore/it/unibo/qahello.

1.5 Facts about the state

The WorldTheory includes facts about the state of the actor and of the world. For example:

actorobj/1	memorizes a reference to the Java/Akka object that implements the actor (see Subsection ??)
actorOpDone/2	memorizes the result of the last actorOp executed (see Subsection 2.10)
goalResult/1	memorizes the result of the last goal given to a demo operation (see Subsection 1.6)

Facts like actorOpDone/1, goalResult/1, etc. are 'singleton facts'. i.e. there is always one clause for each of them, related to the last action executed.

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 is written as a tuProlog Term (see Subsection 2.5).

1.6 Example: the demo operator

A *QActor* can use the built-in `demo` operator to execute actions implemented in tuProlog within the actor's *WorldTheory*. 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).

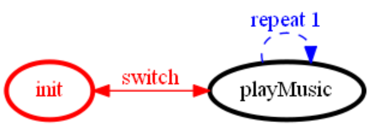
The result of the `demo` operator is memorized in the singleton fact `goalResult/1` that can be inspected by using a guard (see Subsection 2.5).

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

Listing 1.3. `demoExample.qa`

1.7 Example: repeat/resume plans

The next example defines the behaviour of a *QActor* that implements the state machine shown in the following state diagram:




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

QActor player context ctxBasic{
   Plan init normal
       [   println("player STARTS" ) ;
           println("player ENDS" )
       ]
       switchTo playMusic
//     finally repeatPlan 1  //(1)

   Plan playMusic resumeLastPlan
```

```
        [   println( playSomeMusic );
            sound time(1500) file('./audio/tada2.wav') ;
            delay 500
        ]
        finally repeatPlan 1
 }
/*
OUTPUT
----------------------------
"player STARTS"
"player ENDS"
playSomeMusic
playSomeMusic
*/
```

Listing 1.4. basic.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 *Plan* (i.e. its operational semantics).

1.8 How a Plan 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 *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 2.3;
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 1.2, the output will be:

```
"player STARTS"
"player ENDS"
playSomeMusic
playSomeMusic
"player STARTS"
"player ENDS"
playSomeMusic
playSomeMusic
```

If we remove the `resumeLastPlan` specification from `playMusic`, the control does not return to `player` and the output is the same as Subsection 1.2.

2 The qa language/metamodel.

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 has the following form :

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

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

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

Listing 1.5. Message declaration

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

```
Message :         OutOnlyMessage | OutInMessage ;
OutOnlyMessage :  Dispatch | Event | Signal | Token ;
OutInMessage:     Request | Invitation ;

Event:     "Event"     name=ID ":" msg = PHead ;
Signal:    "Signal"    name=ID ":" msg = PHead ;
Token:     "Token"     name=ID ":" msg = PHead ;
Dispatch:  "Dispatch"  name=ID ":" msg = PHead ;
Request:   "Request"   name=ID ":" msg = PHead ;
Invitation: "Invitation" name=ID ":" msg = PHead ;
```

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

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

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

2.1 Messages

In the *QActor* metamodel, a message is defined 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 2.3) in the receiver. If the receiver is not 'waiting' fro

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

a transition including m, the message is enqueued in the receiver queue. Thus we talk of massage-based behaviour, by excluding massage-driven behaviour (the default behaviour in Akka).

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 to express guards (see Subsection 2.5) to allow conditional evaluation of *PlanActions*.

2.2 Events

In the *QActor* metamodel, an event is defined 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 4.2).

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.

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

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

2.3 State transitions

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

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

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 1.2).

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

2. with the keyword reactive: in this case the automaton performs a ReactiveAction that can lead to another state. This case is discussed in Subsection ??.

```
ReactiveAction: "reactive" (guard = Guard)? action=RAction ...
```

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

```
MsgTransition: "transition" duration=Duration (msgswitch+=StateTransSwitch )?("," msgswitch+=StateTransSwitch)* ;
Duration : (guard = Guard)? "whenTime" (msec=INT | var=Variable) "->" move=[Plan] ;
StateTransSwitch : MsgTransSwitch | EventTransSwitch ;

MsgTransSwitch: "whenMsg" (guard = Guard)? message=[Message] next=TransSwitch ;
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*.

2.3.1 Switch part.

A `TransSwitch` can specify either an explicit new *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 ;
```

For an example, see Section 4

2.4 Plan Actions

A *QActor Plan* specifies a sequence of predefined or user-defined `actions` that must always terminate. The `effects` of actions can be perceived in one of the following ways:

1. as changes in the state of the actor (the actor's `'mind'` , see Subsection ??);
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 by the *WorldTheory* associated with the actor (see Subsection 1.4)). Actions that change the actor's physical state or the actor's working environment are called `physical actions`.

`Logical action`	usually is a 'pure' computation defined in some general programming language. Actually we use Java, Prolog and JavaScript.
`Physical action`	can be implemented by using low-cost devices such as `RaspberryPi` and `Arduino`
`Timed action`	always terminates within a prefixed time interval. An example is the built-in `sound` action introduced in Subsection 1.2
`Application action`	defined by the application designer according to the constraints imposed by its logical architecture. More on this in Section ??.

Each *QActor* is able to execute a set of built-in actions, defined by the `qa` language, implemented in Java. An example is the `println` action.

The application designer can define new actions either in Java or in tuProlog.

- an action can be defined in tuProlog by introducing a rule in the `Rules` specification (see Subsection 2.7 and Subsection 2.9) or by loading a `user-defined theory` (see Subsection 2.11). This kind of action can be invoked within an actor model by means of the `demo` operator (see Subsection 1.6).
- an action can be defined in Java by writing a `public` method in the actor class generated within the `src` directory. This kind of action can be invoked within an actor model by means of the `actorOp` operator (see Subsection 2.10).

2.4.1 The syntax of a Plan.

From the syntax rules, we can see that each action of a *Plan* can be prefixed by a *guard* (see Subsection 2.5) and that there are three types of actions: `StateMoveNormal`, `EventSwitch` and `MsgSwitch`.

```
Plan : "Plan" name=ID ( normal ?= "normal" )? ( resume ?= "resumeLastPlan" )?
      ("actions")? "["
       action += PlanTerminatingAction (";" action += PlanTerminatingAction)*
       "]"
      ( transition      = PlanTransition )?
      ( "finally" repeat = AgainPlan )?
;
PlanTerminatingAction: (guard = Guard)? move = StateMove ( "else" elsemove=StateMove) ? ;
StateMove: StateMoveNormal | EventSwitch | MsgSwitch ;
```

`StateMoveNormal` actions have been introduced in Subsection 2.3.1 and are the conventional actions that one expects in a computational system. The action `EventSwitch` and `MsgSwitch` are introduced to facilitate the handling of received messages and will be introduced in Section 3.

2.5 Guarded actions

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` is written as a tuProlog `Phead` (see Section 2).

Actions prefixed by a [GUARD] are executed only when the GUARD is evaluated true. The GUARD can include *unbound variables*[3], 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 or rule that makes the guard true is removed from the actor's *WorldTheory*.

Let us consider the following example;

```
System guardedActions
Context ctxGuardedAction ip [ host="localhost" port=8037 ]
QActor qaguarded context ctxGuardedAction {
   Plan init normal
   [
      [ !? divisible( 10, 2 )] println( isdivisible(10,2) ) ;
      [ !? divisible( 10, 3 )] println( isdivisible(10,3) )
                        else println( notdivisible(10,3) ) ;
      demo fibo(10,R) ;
      [ !? goalResult( fibo(10,R) )] println( result( fibo(10,R) ) ) ;
      [ ?? goalResult( fibo(10,55) )]
        println( fibo(10,55,correct) ) else println( fibo(10,55,wrong) ) ;
      [ ?? goalResult( fibo(10,58) )]
        println( fibo(10,58,correct) ) else println( fibo(10,58,wrong) )
   ]}
```

Listing 1.6. guardedActions.qa

The term divisible/2 is the head of a built-in rule defined in the actor *WorldTheory* (see Subsection 2.8). The output is:

```
/*
OUTPUT
---------------------------
isdivisible(10,2)
notdivisible(10,3)
result(fibo(10,55))
fibo(10,55,correct)
fibo(10,58,wrong)
*/
```

Listing 1.7. guardedActions.qa

2.6 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:

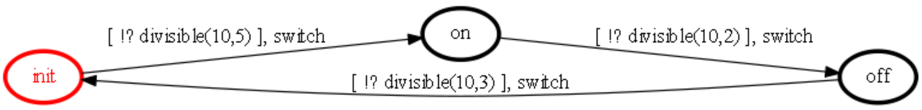
```
System guardedTransitions
Context ctxGuardedTransitions ip [ host="localhost" port=8037 ]
QActor qaguardedtrans context ctxGuardedTransitions {
   Plan init normal
      [  println("qaguardedtrans STARTS" ) ]
      switchTo [ !? divisible(10,5) ] on
   Plan on
      [  println( on )  ]
      switchTo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

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

```
/*
OUTPUT
----------------------------
"qaguardedtrans STARTS"
on
off
*/
```

Listing 1.8. `guardedTransitions.qa`

The generated state diagram is:



More interesting examples will be given in the following.

2.7 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[4]

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

```
/*
 * rulesInModel.qa
 */
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).
    }
    Plan init normal
    [
        [ ?? music(N,F,T) ] sound time(T) file(F) else endPlan "bye"
    ]
    finally repeatPlan
}
```

Listing 1.9. `rulesInModel.qa`

2.8 Built-in rules

The *WorldTheory* can also define computational rules written in tuProlog. For example:

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

actorPrintln(T)	prints the given term T (see Subsection 2) in the actor standard output;
actorOp(M	puts in execution the given Java method M written by the application designer (see Subsection 2.10)
assign(K,V)	associates the given 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

2.9 Example: using built-in tuProlog rules

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

```
/*
 * builtinExample.qa
 */
System basic
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 1.10. builtinExample.qa

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

```
/*
OUTPUT
----------------------------
"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 1.11. `builtinExample.qa`

2.10 The operator `actorOp`

The qa operator `actorOp` allows us to put in execution a Java method written by the application designer as an application-specific part.

Here is an example (project *it.unibo.qactors2017.tests*) that shows how to execute methods that return primitive data and methods that return objects:

```
System actorOpExample
Context ctxActorOpExample ip [ host="localhost" port=8037 ]
QActor qaactoropexample context ctxActorOpExample {
   Plan init normal
      [ println("qaactoropexample STARTS " ) ]
      switchTo testReturnPrimitiveData

   Plan testReturnPrimitiveData
      [
         println("------------- testReturnPrimitiveData START " ) ;
         actorOp getHello ;
         [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
         actorOp intToVoid(5) ;
         [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
         actorOp intToString(5) ;
         [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
         actorOp intToInt(5) ;
         [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
         actorOp intToFloat(5) ; //qa floats not yet implemented
         [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
         println("------------- testReturnPrimitiveData END " )
      ]
      switchTo testReturnPojo

   Plan testReturnPojo
   [
      println("------------- testReturnPojo START" ) ;
      actorOp getDate ;
      [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
      println("------------- testReturnPojo END " )
   ]
}
/*
OUTPUT
----------------------------
```

```
"qaactoropexample STARTS "
"------------- testReturnPrimitiveData "
       Java getHello
done(getHello,'hello world')
       Java intToVoid 5
done(intToVoid(5),null)
       Java intTostring 5
done(intToString(5),'51')
       Java intToInt 5
done(intToInt(5),6)
       Java iniToFloat 5
done(intToFloat(5),2.0)
"------------- testReturnPojo "
       Java getDate
done(getDate,'Tue Sep 26 09:04:19 CEST 2017') */
```

Listing 1.12. `actorOpExample.qa`

The code written by the application designer is:

```
/* Generated by AN DISI Unibo */
/*
This code is generated only ONCE
*/
package it.unibo.qaactoropexample;
import java.util.Calendar;
import java.util.Date;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.QActorContext;

public class Qaactoropexample extends AbstractQaactoropexample {
   public Qaactoropexample(String actorId, QActorContext myCtx, IOutputEnvView outEnvView ) throws Exception{
      super(actorId, myCtx, outEnvView);
   }
/*
 * Introduced by the Application Designer
 */
   public String getHello(){
      println( "    Java getHello " );
      return "hello world";
   }
   public void intToVoid( int n ){
      println( "    Java intToVoid " + n );
   }
   public int intToInt( int n ) {
      println( "    Java intToInt " + n );
      return n+1;
   }
   public String intToString( int n ) {
      println( "    Java intTostring " + n );
      return ""+n+1;
   }
   public float intToFloat( int n ) {
      println( "    Java iniToFloat " + n );
      return n/2;
   }

   public Date getDate( ) {
      println( "    Java getDate " );
      Calendar rightNow = Calendar.getInstance();
      Date d = rightNow.getTime();
      return d;
   }
}
```

Listing 1.13. `Qaactorop.java`

2.11 Loading and using a user-defined theory

The *WorldTheory* of an actor can be extended by the application designer by using the directive[5] `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 1.14. `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 1.15. `aTheory.pl`

2.11.1 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:

[5] 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 1.16. `aTheoryUsage.qa`

2.11.2 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[6]. 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 (for an example, see Subsection ??).

[6] Remember from Subsection 1.5 that the fact goalResult/1 is a 'singleton'.

3 Message-based behaviour

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` 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 been received, the **onMsg** action 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 ;
```

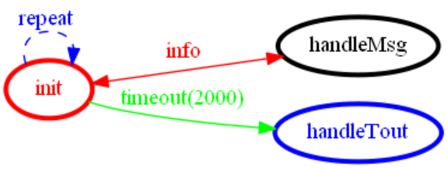
3.1 Example: a producer-consumer system

As an example of a message-based transition, let us introduce a very simple producer-consumer system, in which the producer sends two times a *dispatch* (see Subsection 2.1) to the consumer:

```
/*
 * basicProdCons.qa
 */
System basicProdCons
Dispatch info : info(X)
Context ctxBasicProdCons ip [ host="localhost" port=8019 ]
QActor producer context ctxBasicProdCons{
    Plan init normal
        [   println( "producer sends info(1)" ) ;
            forward consumer -m info : info(1) ;
            delay 500;
            println( "producer sends info(2)" ) ;
            forward consumer -m info : info(2)
        ]
 }
 QActor consumer context ctxBasicProdCons{
    Plan init normal
        [ println( consumer(waiting) ) ]
        transition whenTime 2000 -> handleTout
                   whenMsg info -> handleMsg
        finally repeatPlan
    Plan handleMsg resumeLastPlan
        [ printCurrentMessage ;
          onMsg info : info(X) -> println( msgcontent(X) )
        ]
    Plan handleTout
        [ println( consumerTout ) ]
```

Listing 1.17. `basicProdCons.qa`

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



The output is:

```
}
/*
OUTPUT
----------------------------------------
consumer(waiting)
"producer sends info(1)"
--------------------------------------------------------------------------------------------
consumer_ctrl currentMessage=msg(info,dispatch,producer_ctrl,consumer,info(1),1)
--------------------------------------------------------------------------------------------
msgcontent(1)
consumer(waiting)
"producer sends info(2)"
--------------------------------------------------------------------------------------------
consumer_ctrl currentMessage=msg(info,dispatch,producer_ctrl,consumer,info(2),2)
--------------------------------------------------------------------------------------------
msgcontent(2)
consumer(waiting)
consumerTout
*/
```

Listing 1.18. `basicProdCons.qa`

4 Event-based and event-driven behaviour

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

The `RaiseEvent` rule defines how to `emit` and event:

```
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 ;
```

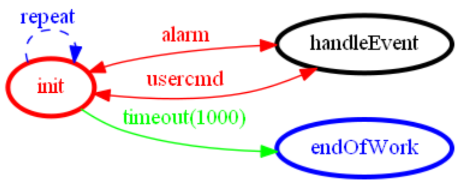
4.1 Example: event-based behavior

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 {
    Plan init normal
    [   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 {
    Plan init normal
        [   println("qaeventperceptor STARTS ") ]
        transition whenTime 1000 -> endOfWork
            whenEvent alarm -> handleEvent,
            whenEvent usercmd -> handleEvent
        finally repeatPlan

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

Listing 1.19. `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 1.20. `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.

4.2 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[7]:

```
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] ;
```

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

- 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
- 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.

```
/*
 * 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 1.21. eventTracer.qa

The output is:

```
"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"
qaevtracer(usercmd,qatraceremitter_ctrl,usercmd(hello))
"qaevtracer starts"
"noevent"
```

Listing 1.22. eventTracer.qa

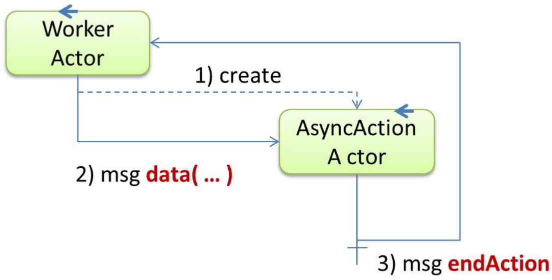
5 Asynchronous actions

In Subsection 2.4 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):

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

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.

5.0.1 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 1.23. `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:

5.0.2 onReceive .

```
    public void onReceive(Object msg) { // msg=startAction
//     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 1.24. `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:

5.0.3 endActionInternal .

```
    protected T endActionInternal() throws Exception{
        evalDuration();
        T res = endOfAction();
        String payload = terminationEvId+"(ANAME,RES)".replace("ANAME", name).replace("RES", res.toString() );
        myactor.sendMsg(terminationEvId, myactor.getName().replace("_ctrl", ""), "dispatch", payload );
        return res;
    }
}
```

Listing 1.25. `endActionInternal`

The application designer must introduce a specialization of the class `ActionObservableGenericActor<T>`. Let us show different possible ways to execute a computation in asynchronous way.

5.1 Asynchronous actions: an example

.

In the following model, we activate the asynchronous computation of a Fibonacci number in three ways:

- by calling an application-specific operation `fibonacciAsynch` implementd by the application designer (see Subsection 2.10) as an asynchronous operation;
- by calling a conventional application-specific operation `fibonacciNormal` by specifying an asynchronous execution (flag `-asynch`) of the operator `actorOp` (see Subsection 2.10);
- by solving a tuProlog rule `fibo/2` vy calling the operator `demo` (see Subsection 1.6) in asynchronous way (flag `-asynch`).

```
System asyncActionsFibo
Dispatch endAction  : endAction(A,R)
Dispatch endActorOp : endActorOp(A,R)

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

QActor asynchworkerfibo context ctxAsynchFibo{
    Plan init normal[
        actorOp fibonacciAsynch( "actionFiboAsych", 37 ) ;
        actorOp fibonacciNormal( 23 ) -asynch;
```

```
        demo fibo(23,V) -asynch; //fibo(N,V) is defined ain the WorldTheory
        println("asynchworkerfibo END")
    ]
    switchTo handleActionEnd

    Plan handleActionEnd [ println("WAIT FOR ASYNCH ACTION TERMINATION") ]
    transition stopAfter 3000
        whenMsg endAction -> useActionResult,
        whenMsg endActorOp : endActorOp(A,R) do println( endActorOpMs(A,R) )
        finally repeatPlan

    Plan useActionResult resumeLastPlan[
        onMsg endAction : endAction(actionFiboAsych,V) -> println( V ) ;
        onMsg endAction : endAction(A,V) -> println( endAction(A,V) )
    ]
}
```

Listing 1.26. `asyncActionsFibo`

5.1.1 ActionActorFibonacci .

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

```
public class Asynchworkerfibo extends AbstractAsynchworkerfibo {
    public Asynchworkerfibo(String actorId, QActorContext myCtx, IOutputEnvView outEnvView ) throws Exception{
        super(actorId, myCtx, outEnvView);
    }
/*
 * ADDED BY THE APPLICATION DESIGNER
 */
    public void fibonacciAsynch(String actionName, int n){
        ActorRef actionActorRef =
              myCtx.getCreatorAkkaContext().actorOf( //son of the SystemCreationActor instance
                Props.create(ActionActorFibonacci.class, //action implementation class
                             actionName, this, outEnvView), //constructor arguments
                             actionName );
        actionActorRef.tell("data("+n+")", getSelf() );
    }
```

Listing 1.27. `fibonacciAsynch`

The class `ActionActorFibonacci` is a specialized version of `ActionObservableGenericActor<String>`, also written by the application designer:

```
package it.unibo.asynchworkerfibo;
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+")"+")";
    }
}
```

Listing 1.28. `ActionActorFibonacci`

5.1.2 fibonacciNormal .

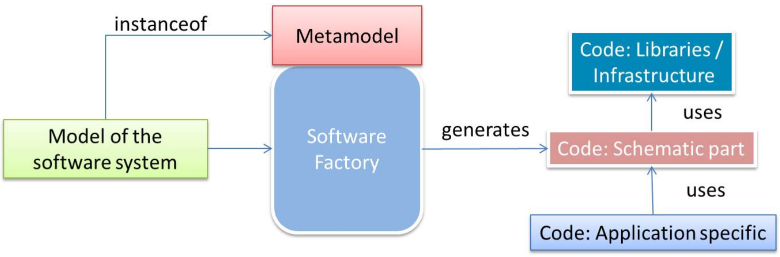
The application designer writes in the actor class the usual code:

```
    public long fibonacciNormal( int n ){
        if(  n == 1 || n == 2 ) return 1;
        else return fibonacci(n-1) + fibonacci(n-2);
    }
```

Listing 1.29. `fibonacciNormal`

6 Application-specific actions

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 (Subsection 1.3) creates in automatic way 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. Defined by the project *it.unibo.interfaces*.
`uniboEnvBaseAwt.jar`	provides a framework for building basic (graphical) user interfaces. Defined by the project *it.unibo.envBaseAwt*.
`unibonoawtsupports.jar`	provides a support for communications base on connection-based protocols such as TCP, UDP, ... Defined by the project *it.unibo.noawtsupports*.

The Subsection 2.10 has shown how application-specific Java code can be put in execution from a *QActor* model by means of the `actorOp` operator. The Java code must be written by the application designer as a `public` method in the proper actor class generated within the `src` directory.

In this section we give some other example of how Java libraries can be exploited to enrich the model with non-trivial application-specific parts. The code is available in the project `it.unibo.lss17`.

6.1 A Custom GUI

In this example, we want to create an application-specific GUI for a *QActor*. Thus, we delegate the task to an application-specific operation (`buildCustomGui`):

```
System customGui

Context ctxCustomGui ip[ host="localhost" port=8038 ] //-g cyan

 QActor qawithcustomgui context ctxCustomGui //-g cyan
 {
    Plan init normal[
        actorOp buildCustomGui("customGUI") ;
        delay 30000 ;  //to avoid immediate termintion
        println("qawithcustomgui END")
    ]
 }
```

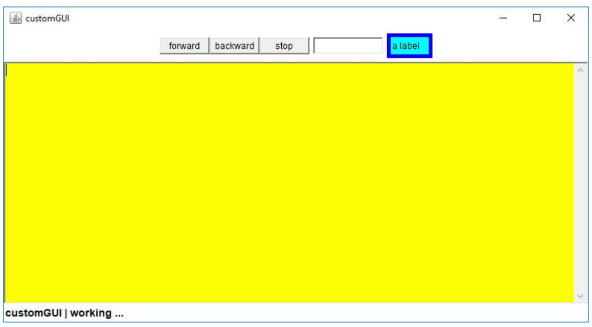
Listing 1.30. `customGui.qa`

The application-specific Java code can be written as follows (by exploiting the `uniboEnvBaseAwt.jar` library):

```java
/* Generated by AN DISI Unibo */
/*
This code is generated only ONCE
*/
package it.unibo.qawithcustomgui;
import java.awt.Color;

import alice.tuprolog.SolveInfo;
import it.unibo.baseEnv.basicFrame.EnvFrame;
import it.unibo.is.interfaces.IBasicEnvAwt;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.QActorContext;

public class Qawithcustomgui extends AbstractQawithcustomgui {
    public Qawithcustomgui(String actorId, QActorContext myCtx, IOutputEnvView outEnvView ) throws Exception{
        super(actorId, myCtx, outEnvView);
    }
/*
 * ==========================================================
 * ADDED BY THE APPLICATION DESIGNER
 * ==========================================================
 */
    /*
     * OVERRIDE in order to NOT insert built-in panels if the actor has a local gui
     */
    protected void addInputPanel(int size){ }
    protected void addCmdPanels(){ }
    /*
     * If the actor ha no local GUI, create a new Frame,
     * otherwise, just set the given logo
     */
    public void buildCustomGui(String logo){
        IBasicEnvAwt env = outEnvView.getEnv();
        if( env == null){
            env = new EnvFrame( logo, Color.yellow, Color.black );
            env.init();
            ((EnvFrame)env).setSize(800,430);
        }
        env.writeOnStatusBar(logo + " | working ... ",14);
        new it.unibo.customGui.CustomGuiSupport( env );
    }
}
```

Listing 1.31. `Qawithcustomgui.java`

Most of the work is further delegated to an utility object of class `it.unibo.customGui.CustomGuiSupport` that defines the desired aspect of the `GUI`. The result is:



6.1.1 Using the `uniboEnvBaseAwt` framework .

To create the application-specific `GUI`, the application designer defines an utility class `it.unibo.customGui.CustomGuiSupport` that, in its turn, exploits the `uniboEnvBaseAwt` framework:

```java
public class CustomGuiSupport extends SituatedPlainObject{
private IActivityBase cmdHandler ;
private IBasicEnvAwt envAwt;
     public CustomGuiSupport(IBasicEnvAwt env) {
        super(env);
        //env is declared of type IBasicUniboEnv in SituatedPlainObject
        //that does not provide any addPanel method. Thus, we memorize it
        envAwt = env;
        init();
    }
    protected void init(){
        cmdHandler = new CmdHandler(envAwt);
        setCommandUI();
        setInputUI();
        addCustomPanel();
    }
```

Listing 1.32. `CustomGuiSupport.java`

The `init` operation first creates some input panel and then adds a new custom panel to the `GUI`. All these operations are supported by the `uniboEnvBaseAwt` framework that makes these actions quite simple (see Subsection 6.1.5,Subsection 6.1.6).

The class `SituatedPlainObject` has been defined in the project *it.unibo.noawtsupports* as an observable entity that extends `java.util.Observable` and implements the interface `it.unibo.is.interfaces.IObservable`.

Let us recall other basic 'contracts' defined by the framework.

6.1.2 Observable POJO objects .

```java
package it.unibo.is.interfaces;

public interface IObservable {
    public void addObserver(IObserver arg0); //modifier
}
```

Listing 1.33. `IObservable`

```java
package it.unibo.is.interfaces;
import java.util.Observer;

public interface IObserver extends Observer {
//  public void update(Observable source, Object state); //inherited modifier
}
```

Listing 1.34. `IObserver`

6.1.3 Environment interfaces .

Any `uniboEnvBaseAwt` environment must always provide the following operations, that do not require any `GUI`:

```java
package it.unibo.is.interfaces;

public interface IBasicUniboEnv {
    public void init();
    public String readln( );
    public IOutputView getOutputView();
    public void println( String msg );
    public void close( );
 }
```

Listing 1.35. `IBasicUniboEnv`

An environment based on a GUI should provide also:

```java
package it.unibo.is.interfaces;
import java.awt.Component;
import java.awt.Panel;

public interface IBasicEnvAwt extends IBasicUniboEnv{
    public void initNoFrame();
    public IOutputEnvView getOutputEnvView();
    /**
     *  Write on the status bar
     */
    public void writeOnStatusBar( String s, int size);
    /**
     * @return true in case of a standalone application
     */
    public boolean isStandAlone();
    /**
    *   Add a panel in the environment.
    */
    public void addInputPanel( int size );
    public void addInputPanel( String msg );
    public void addPanel( Panel p );
    public void addPanel( Component p );
```

Listing 1.36. `IBasicEnvAwt (partial)`

Our custom GUI works in an environment that implements `IBasicEnvAwt`. Note that on other platforms (e.g. *Android*, *Raspberry*) the reference environment should support a `IBasicUniboEnv` only, in order to avoid any dependency on graphical libraries.

6.1.4 Command interfaces .

Since a user interface is usually a means to send commands to an application, the application software must define entities able to execute actions given a command `String`. These entities must support the following interface:

```java
package it.unibo.is.interfaces;
/*
 * Interface of any entity that can execute an action
 */
public interface IActivityBase {
    public void execAction( String cmd );
}
```

Listing 1.37. `IActivityBase`

For example, the `CmdHandler` introduced in Subsection 6.1.1 can be defined as follows:

```java
package it.unibo.customGui;
import it.unibo.is.interfaces.IActivityBase;
import it.unibo.is.interfaces.IBasicEnvAwt;
import it.unibo.system.SituatedPlainObject;

public class CmdHandler extends SituatedPlainObject implements IActivityBase{
    public CmdHandler(IBasicEnvAwt env) {
        super(env);
    }
    @Override
    public void execAction(String cmd) {
        String input = env.readln();
        println("CmdHandler -> " + cmd + " input= " + input);
    }
}
```

Listing 1.38. `CmdHandler.java`

6.1.5 Adding a command panel .

The operation that adds a command user interface can be performed by simply calling the addCmdPanel operation with proper arguments:

```
    protected void setCommandUI(){
        envAwt.addCmdPanel("commandPanel",
                new String[]{"forward" ,"backward" ,"stop" },
                cmdHandler );
    }
```

Listing 1.39. CustomGuiSupport.java

In this case, we handle each command button with the same handler, introduced in Subsection 6.1.4 that simply prints the command on the standard output.

6.1.6 Adding an input panel .

Quite simple too:

```
    protected void setInputUI(){
        envAwt.addInputPanel( 10 );
    }
```

Listing 1.40. CustomGuiSupport.java

6.1.7 Adding a new panel .

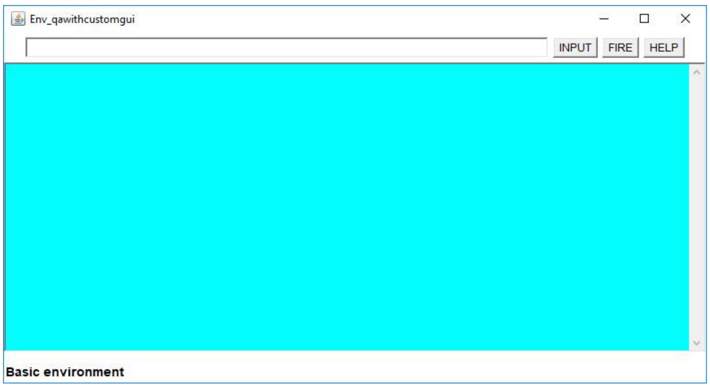
Let us add a new panel we a label inside:

```
    protected void addCustomPanel(){
        Panel p = new Panel( );
        p.setBackground(Color.blue);
        Label l = new Label("a label");
        l.setBackground(Color.cyan);
        p.add( l );
        envAwt.addPanel( p );
    }
```

Listing 1.41. CustomGuiSupport.java

6.1.8 Built-in GUI .

The qa meta-model allows the insertion of a -g flag (e.g. -g cyan) after the declaration of an actor. In this case the factory generates a GUI interface like that shown in the picture:



The application designer can decide to 'specialize' in some way such a built-in interface as done in Subsection 6.1. The output is the same as before, with the background color specified in the `-g` declaration (in fact, the application code does not change it).

6.1.9 Built-in commands .

With reference to the built-in `GUI` of an actor:

- the `FIRE` button generates the event `local_alarm : alarm(fire)`;
- The `INPUT` button generates the event `local_inputcmd : usercmd(executeInput(CMD))` where `CMD` is the content of the input field on the left.

The name of events generated by the `GUI` buttons is prefixed by the string `local_` in order to state that these events are local to the current computational node and must not be propagated to other nodes (more on this in Section **??**).

By clicking the button `HELP` we can visualize the syntax of a possible `CMD`. For example:

```
 GOAL
[ GUARD ], ACTION
[ GUARD ], ACTION, DURATION
[ GUARD ], ACTION, DURATION, ENDEVENT
[ GUARD ], ACTION, DURATION, EVENTS, PLANS
```

With reference to the current implementation of the built-in `GUI` of an actor, let us see what happens when we insert in the `Input` field one of the previous command strings:

COMMAND	event `local_inputcmd : usercmd(executeInput(CMD))`
`eval(gt,5,2)`	usercmd(`executeInput(eval(gt,5,2))`)
`[eval(gt,5,2)],nears(D)`	usercmd(`executeInput(do([eval(gt,5,2)],nears(D))`)
`[eval(gt,5,2)],nears(D),2000`	usercmd(`executeInput(do([eval(gt,5,2)],nears(D),2000)`)

6.2 A Command interpreter

Let us define here a simple interpreter of user commands expressed as strings of the form:

```
 GOAL
[ GUARD ], ACTION
```

The logic architecture of the interpreter can be defined as an actor with a built-in `GUI`:

```
/*
 * cmdExecutor.qa
 */
System cmdExecutor
Event local_inputcmd : usercmd(X) //generated by cmd actor gui-interface
Event alarm         : alarm(X)    //generated by HTTP cmd user-interface

Context ctxCmdExecutor ip [ host="localhost" port=8039 ]

QActor qacmdexecutor context ctxCmdExecutor -g cyan {
```

Listing 1.42. `cmdExecutor.qa`

The actor does introduce also a set of rules, to be used in a demo:

```
Rules{
data(sonar, 1, 10).
data(sonar, 2, 20).
data(sonar, 3, 30).
data(sonar, 4, 40).
validDistance( N,V ) :- data(sonar, N, V), eval(gt,V,10), eval(lt,V,50) .
nearDistance( N,X ) :- validDistance( N,X ), eval(lt,X,40) .
nears( D ) :- findall( d( N,V ), nearDistance(N,V), D ).
}
```

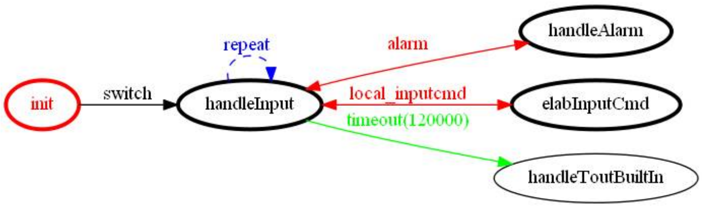
Listing 1.43. cmdExecutor.qa

At its start-up, the actor loads the application tuProlog theory ./src/cmdInterpreterSimple.pl that defines the interpretation rules of the user commands:

```
    Plan init normal[
       println("===================================================" ) ;
       println("An actor that executes user commands " ) ;
       println("===================================================" ) ;
       demo consult("./src/cmdInterpreterSimple.pl")
    ]
    switchTo handleInput
```

Listing 1.44. cmdExecutor.qa

Afterwards, the actor waits for input events (local_inputcmd and alarm).




```
    Plan handleInput [ println("WAIT ...") ]
       transition stopAfter 120000
       whenEvent local_inputcmd -> elabInputCmd,
       whenEvent alarm -> handleAlarm
       finally repeatPlan
    Plan elabInputCmd resumeLastPlan[
      printCurrentEvent;
      onEvent local_inputcmd : usercmd(executeInput(CMD)) -> demo executeInput(CMD) ; //INTERPRET
      [ ?? goalResult(R)] println( cmdResult(R))
    ]
    Plan handleAlarm resumeLastPlan [
      sound time(1500) file("./audio/tada2.wav") -asynch;
      printCurrentEvent
    ]
 }
```

Listing 1.45. cmdExecutor.qa

The actor handles the alarm event by playing a short sound, while the local_inputcmd event is handled by calling a tuProlog rule executeInput(CMD) with CMD set as reported in Subsection 6.1.9.

The command interpretation theory ./src/cmdInterpreterSimple.pl can be defined as follows:

```
/*
=================================================================
cmdInterpreterSimple.pl
=================================================================
*/
```

```
6  executeInput( do( [ true ], MOVE ) ):- !,
7      %% output( executeInputGuardedTrue(MOVE)),
8      execMove( MOVE ).
9  executeInput( do( [ GUARD ] , MOVE ) ):-
10     %% output( executeInputGuarded(GUARD,MOVE) ),
11     GUARD,
12     %% output( executeInputGuarded(MOVE) ),
13     execMove( MOVE ).
14
15 executeInput( MOVE ) :-
16     %% output( executeInput( MOVE ) ),
17     execMove( MOVE ).
18 execMove( MOVE ) :-
19     %% output( execMove( MOVE ) ),
20     MOVE, !,
21     %% output( done( MOVE) ),
22     setPrologResult(MOVE). %% defined in the WorldTheory
23 %% MOVE is delegated to an operation (if any) written in Java
24 execMove( MOVE ):-
25     actorobj(Actor),
26     Actor <- MOVE.
27
28 initSimple :-
29    actorPrintln(" *** cmdInterpreterSimple loaded *** ").
30 :- initialization(initSimple).
```

Listing 1.46. `cmdInterpreterSimple`

Note that the interpreter first attempts to execute the user command by using the *WorldTheory* (lines 18-22) . If this attempts fails, the interpreter delegates the work to the Java code (lines 23-26) by exploiting the tuProlog features (more on this in Subsection **??**).

7 Android

In this section, we want to *write a software system on a conventional PC that receives the data of the accelerometer sensor embedded in an Android device connected via Tethering*[8].

The system model (project *it.unibo.lss17*) simply defines an actor that makes a request of data to the `Android` device and then waits for the answer message.

```
System acceleromFromAndroid

Context ctxAcceleromFromAndroid ip [ host="localhost" port=8143 ]

QActor qaandroidpartner context ctxAcceleromFromAndroid {
Rules{
    addr( usbtethering, otium, "192.168.42.129").
    addr( natspot, otium, "192.168.43.71").
}
    Plan init normal [
        println("qaandroidpartner STARTS" ) ;
        [ !? addr( usbtethering, otium, IP)]
                actorOp initConnWithAndroid("qaandroidpartner",IP, 8123);
        println("qaandroidpartner ENDS" )
    ]
    switchTo getSensorData

    Plan getSensorData[
        delay 500;
        actorOp sendMsgToAndroid( "getData" );
        actorOp receiveMsgFromAndroid;
        [ ?? actorOpDone( OP,R ) ] println( answer(OP,R) )
    ]
    finally repeatPlan 3
}
```

Listing 1.47. `acceleromFromAndroid`

Since `Android` devices does not enter (at the moment) in the implementation scope of the *QActor* meta-model, the application designer fulfils the goal by introducing application-specific operations:

- `initConnWithAndroid(String actorName,String hostName,int port)`: sets a connection via TCP
- `sendMsgToAndroid(String msg)` : sends a message on the connection
- `String receiveMsgFromAndroid()` : receives a message from the connection

The application-specific code is:

```
public class Qaandroidpartner extends AbstractQaandroidpartner {
    public Qaandroidpartner(String actorId, QActorContext myCtx, IOutputEnvView outEnvView ) throws Exception{
        super(actorId, myCtx, outEnvView);
    }
/*
 * ADDED by the APPLICATION DESIGNER
 */
private IConnInteraction conn = null;
    public void initConnWithAndroid(String actorName, String hostName, int port){
        conn = CommsWithOutside.initConnection(actorName,hostName, port);
    }//initConnWithAndroid
    public void sendMsgToAndroid( String msg) {
        try {
            CommsWithOutside.sendMsg(conn, msg);
        } catch (Exception e) {
            e.printStackTrace();
        }
```

[8] Tethering, or phone-as-modem (`PAM`), is the sharing of a mobile device's internet connection with other wirelessly connected computers. Connection of a mobile device with other devices can be done over wireless LAN (`Wi-Fi`), over `Bluetooth` or by physical connection using a cable, for example through `USB`.

```
    }//sendMsgToAndroid
    public String receiveMsgFromAndroid( ) {
        try {
            String answ = CommsWithOutside.receiveMsg(conn);
            return answ;
        } catch (Exception e) {
            return "msg(sensor,event,android,none,null,0)";
        }
    }//receiveMsgFromAndroid

//  public void requestDataToAndroid(){
```

Listing 1.48. `Qaandroidpartner`

The operations are implemented by means of an utility class `CommsWithOutside`.

7.1 The utility class *CommsWithOutside*

```
public class CommsWithOutside {
    public static final String protocol = FactoryProtocol.TCP;
    public static final int port      = 8123;
    public static Hashtable<String, IConnInteraction> androConns = new Hashtable<String, IConnInteraction>();

    public static IConnInteraction initConnection(String actorName, String hostName, int port) {
        System.out.println("initConnection actorName=" + actorName + " hostName=" + hostName);
        FactoryProtocol factoryP =
                new FactoryProtocol(SituatedSysKb.standardOutEnvView, protocol, "androclient");
        IConnInteraction conn = null;
        try {
            conn = factoryP.createClientProtocolSupport(hostName, port);
            androConns.put(actorName, conn); //memo the connection for the actor
        } catch (Exception e) {
            System.out.println("WARNING : no connection possible to Android for " + actorName);
            //e.printStackTrace();
        }
        return conn;
    }
    public static void sendMsg( IConnInteraction conn, String msg) throws Exception {
        if( conn != null ) conn.sendALine( msg );
        else System.out.println("WARNING : no connection to Android " );
    }
    public static void sendMsg( String actorName, String msg) throws Exception {
        sendMsg( androConns.get(actorName), msg );
    }
    public static String receiveMsg( IConnInteraction conn ) throws Exception {
        if( conn != null ){
            String msg = conn.receiveALine();
            return msg;
        }
        else{
            System.out.println("WARNING : no connection to Android for " );
            return "noconnection_";
        }
    }
    public static String receiveMsg( String actorName ) throws Exception {
        return receiveMsg( androConns.get(actorName) );
    }
}
```

Listing 1.49. `CommsWithOutside`

This class makes us of the `unibonoawtsupports` for communications introduced in Section 6. In particular, we note that the `IConnInteraction` interface allows us to write code independent of any specific (connection-based) protocol.

7.2 The exchanged messages

The message sent from the application to `Android` is kept as simpler as possible. It has the form:

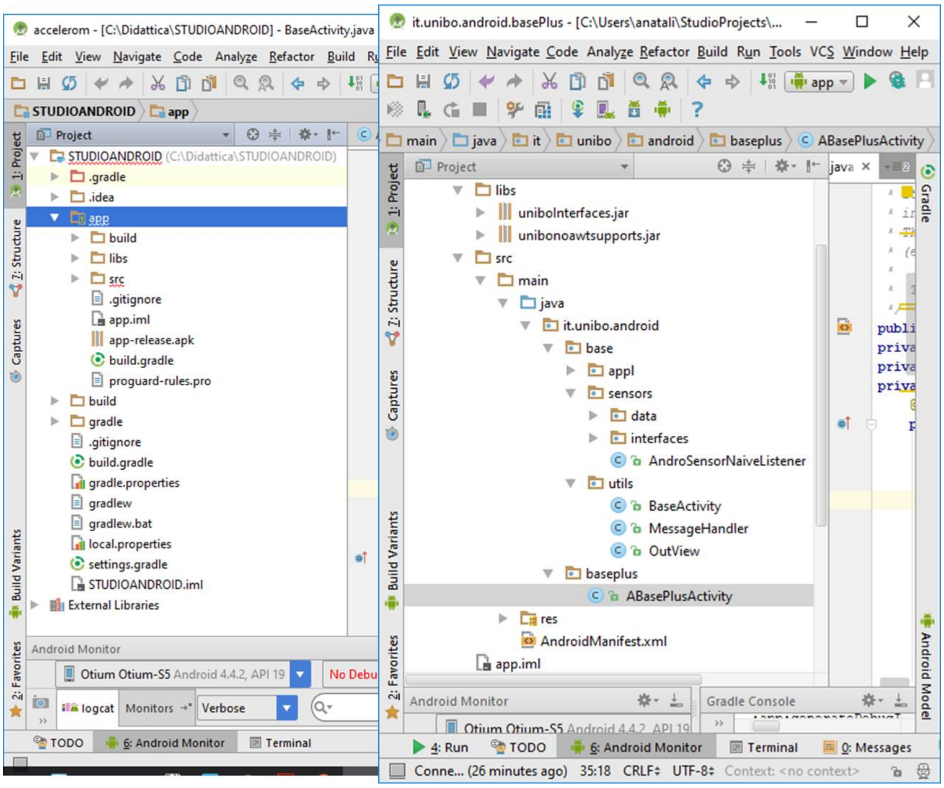
`getData`

The message sent from `Android` to the application has the standard *QActor* form introduced in Subsection 2.1:

`msg(sensor,event,android,none,sensor(androidaccelerometerdata,V),MSGNUM))`

In this way the `Android` device can be viewed as an external source of data-events, able to produce information expressed in the *QActor* internal format.

7.3 The code on the Android device



7.3.1 AndroidManifest :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.unibo.android.baseplus"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="26.0.1" />

    <uses-permission android:name="android.permission.INTERNET"/>
```

```

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".ABasePlusActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Listing 1.50. AndroidManifest.xml

7.3.2 BaseActivity :

```
public class BaseActivity extends Activity{
 public MessageHandler myHandler;
 protected TextView output;
 protected IOutputView outView = null;
 protected Bundle myBundle;
 protected boolean verbose = false;

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        myHandler = new MessageHandler(this);
        myBundle = new Bundle();
        output   = (TextView) findViewById(R.id.output);
        outView  = new OutView(this);
    }
/*
*  ------------------------------------
*   Print Utilities
*  ------------------------------------
*/
    public void println(String msg) {
        if (output == null){
            output = (TextView) findViewById(R.id.output);
        }
        output.append(msg+"\n");
    }
    public void printMsg(String msg) {
        if (output == null){
            output = (TextView) findViewById(R.id.output);
        }
        output.setText( msg );
    }
    public void showMsg(String msg) {
        if (outView != null)
        outView.addOutput(msg);
    }
    /*
     *  ------------------------------------
     *   Android usage support
     *  ------------------------------------
     */
    protected int notifNum = 0;
    public void sendNotification( String logo, String msg ) {
        sendNotification(notifNum++,R.drawable.ic_launcher,logo, msg, null );
    }
    public void sendNotification(int id, int iconId, String notifType, String msg,
```

```
                              Class notifyClass) {
        NotificationManager notificationManager =
                (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
        //Create Notification
        Notification notification =
                new Notification(iconId,notifType, System.currentTimeMillis());
        notification.flags |= Notification.FLAG_AUTO_CANCEL;
        Intent intent;
        if( notifyClass != null )
            intent = new Intent(this, notifyClass);
        else intent = new Intent( );
        intent.putExtra(notifType, msg);
        PendingIntent pIntent =
                PendingIntent.getActivity(this, 0, intent,PendingIntent.FLAG_CANCEL_CURRENT);
        notification.setLatestEventInfo(this, notifType, msg, pIntent);
        //Use the Notification Manager
        notificationManager.notify(id, notification);
    }
    protected PendingIntent buildPendingIntent(String actionName, int requestCode){
        Intent myIntent = new Intent(actionName);
        PendingIntent resIntent=
                PendingIntent.getBroadcast(this, requestCode, myIntent, PendingIntent.FLAG_ONE_SHOT);
        return resIntent;
    }
```

Listing 1.51. `BaseActivity`

7.3.3 Layout :

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
<ScrollView
    android:id="@+id/a2Scroll"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
    <TextView
        android:id="@+id/output"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="@drawable/white"
        android:text=""
        android:textColor="@drawable/black"
        android:textSize="6pt"
        android:textStyle="italic" />

</ScrollView>
</LinearLayout>
```

Listing 1.52. `activity_unibo.xml`

7.3.4 An Android implmentation of `IOutputView` :

```
/*
* ======================================================
* By AN DISI University of Bologna
* ======================================================-
*/
package it.unibo.android.base.utils;
import android.os.Bundle;
import android.os.Message;
```

```java
import it.unibo.is.interfaces.IOutputView;

public class OutView implements IOutputView {
protected BaseActivity myActivity;
protected int nm = 1;
protected String curVal = "";

    public OutView(BaseActivity myActivity) {
        this.myActivity = myActivity;
    }
    public String getCurVal() {
        return curVal;
    }
    public synchronized void addOutput(String msg) {
        curVal = msg ;
        Message m = myActivity.myHandler.obtainMessage();
        Bundle data = new Bundle();
        data.putString("addOutputMsg", msg);
        m.setData(data);
        myActivity.myHandler.sendMessage(m);
    }
    public synchronized void setOutput(String msg) {
        curVal = msg ;
        Message m = myActivity.myHandler.obtainMessage();
        Bundle data = new Bundle();
        data.putString("setOutputMsg", msg);
        m.setData(data);
        myActivity.myHandler.sendMessage(m);
    }
}
```

Listing 1.53. `OutView`

7.3.5 Main activity :

```java
import it.unibo.android.base.appl.SysKb;
import it.unibo.android.base.sensors.AndroSensorNaiveListener;
import it.unibo.android.base.utils.BaseActivity;
import it.unibo.android.baseplus.R;
import it.unibo.system.SituatedSysKb;
import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.widget.Toast;
/*
 * This activity creates a TCP server that can work over a USB connection
 * It creates also a local client that sends to the server a pir of messages
 * including some sensor data (proximity)
 * The server waits for 10 minutes for some other remote client message.
 * (e.g from ClientAndroidBase of project it.unibo.andro.partner)
 *
 *  The application can work also without a USB cable by activating the WIFI
 */
public class ABasePlusActivity extends BaseActivity {
private ServerNoEnv server;
private ClientNoEnv localClient;
private SensorManager sensorManager;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_unibo);
        println("-------------------------------------------------------------------");
        println("ABasePlusActivity(extends BaseActivity)");
        println("using unibo Comm, IOutputView, JSON, sensors ");
```

```
        println("ncores=" + SituatedSysKb.numberOfCores + " port="+ SysKb.port + " mem=" +
             Runtime.getRuntime().maxMemory());
        sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        findSensors();
        println("-----------------------------------------------------------------");
        configureSensor();
        startServer();
        startClient();
        Toast.makeText(this, "ABasePlusActivity ends creation" , Toast.LENGTH_SHORT).show();
    }
    protected void findSensors(){
        List<Sensor> availableSensors = sensorManager.getSensorList(Sensor.TYPE_ALL);
        Iterator<Sensor> itsens = availableSensors.iterator();
        while( itsens.hasNext() ){
            Sensor sens = itsens.next();
            println("SENSOR " + sens.getType() + " " + sens.getName());
        }
```

Listing 1.54. `ABasePlusActivity`

7.4 From `Android` messages to ***QActor*** events

A message sent from the **Android** device can be easily mapped into an event that can be properly handled by actors:

```
System acceleromFromAndroidEvents
Event sensor : sensor( SENSORID, data(DATA) )
//Dispatch info : info(X)

Context ctxAcceleromFromAndroidEvents ip [ host="localhost" port=8143 ]

QActor qaandroidpartnerevents context ctxAcceleromFromAndroidEvents {
Rules{
    addr( usbtethering, otium, "192.168.42.129").
    addr( natspot, otium, "192.168.43.71").
}
    Plan init normal [
        println("qaandroidpartnerevents STARTS" ) ;
        [ !? addr( usbtethering, otium, IP)]
            actorOp initConnWithAndroid("qaandroidpartnerevents",IP, 8123);
        println("qaandroidpartnerevents ENDS" )
    ]
    switchTo requestSensorData

    Plan requestSensorData[
        delay 500;
        actorOp requestDataToAndroid
    ]
    finally repeatPlan 5
}

QActor qasensordatahandler context ctxAcceleromFromAndroidEvents {
    Plan init normal [
        actorOp noOp
    ]
    transition stopAfter 5000
        whenEvent sensor : sensor(ID,DATA) do println( qasensorhandler(ID,DATA) )
        finally repeatPlan
}
```

Listing 1.55. `acceleromFromAndroidEvents`

The application-specific code now is:

```
    public void requestDataToAndroid(){
        try{
```

```
            CommsWithOutside.sendMsg(conn, "getData");
            String answer = CommsWithOutside.receiveMsg(conn);
            Struct ta = (Struct) Term.createTerm(answer); //check
            this.emit("sensor", ta.getArg(4).toString() );
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Listing 1.56. `Qaandroidpartnerevents`

8 (Qa)Models as system integrators

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.

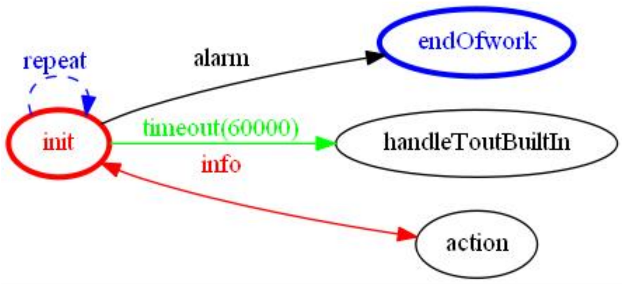
8.1 Using Node.js

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") ;
        actorOp runNodeJs("./nodejsCode/TcpClientToQaNode.js localhost 8031", "true") ;
        //[?? actorOpDone(OP,R)] println( rrr(R) );
        println("qajaactoivator END")
    ]
}

/*
 * This actor handles message ent 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 1.57. `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 Implementation details

The system is available on the GIT https://github.com/anatali/lss0/. A working copy of system is inserted in the directory testOutEclipse. The system is built as a conventional *QActor* project in which the build_ctxJsToQa.gradle has been changed by the application designer:

- by modifying the dependencies on the unibo libraries, since they are now local to the project;
- by adding in the distribution a copy of nodejsCode.

The directory nodejsCode includes the NodeJs code. Recall that:

- the command npm init generates the file package.json that describes the project and declares the dependencies
- the command npm install xxx -save loads a module xxx and automatically adds this dependency to the file package.json.

8.3 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> denotes its output. In some future release of the *QActor* metamodel, it could could be provided as a *QActor* primitive.

8.4 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();
    process.exit(1);      //MANDATORY!!!
});
```

Listing 1.58. `TcpClientToQaNode.js: connect`

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

```
/*
=================================================================
Interaction
=================================================================
*/
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 ); },
        time);
}
```

Listing 1.59. `TcpClientToQaNode.js: utilities`

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

```
/*
=================================================================
Application
=================================================================
*/
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);

setTimeout(function(){ console.log("SOCKET END");socket.close(); }, 2500);
```

Listing 1.60. `TcpClientToQaNode.js: application`

8.5 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"
...
```

8.6 File watcher

Let us introduce a NodeJs computation that allows us to write a sentence in file and read its content:

```
/*
* ====================================
* fileWrite.js
* ====================================
*/
//works with reference to the file sharedFiles/cmd.txt
var fs  = require('fs');
var path = require('path');
var args = process.argv.splice(2);
var command       = args.shift();
var inputData     = args.join(' ');
var file          = path.join(process.cwd(), '../sharedFiles/cmd.txt');

switch(command) {
  case 'read':
    readData(file);
```

```
    break;
  case 'write':
      storeData(file, inputData+"\n");
    break;
  default:
    console.log('Usage: ' + process.argv[0] + ' read|write [inputData]');
}

function loadOrInitializeTaskArray(file, cb) {
  fs.exists(file, function(exists) {
    var oldData = [];
    if (exists) {
        fs.readFile(file, 'utf8', function(err, data) {
        if (err) throw err;
        var newdata = data.toString();
        cb(newdata);
      });
    } else {
      cb([]);
    }
  });
}
function readData(file) {
  loadOrInitializeTaskArray(file, function(data) {
      console.log( data );
  });
}
function storeData(file, newData) {
  fs.writeFile(file, newData, 'utf8', function(err) {
    if (err) throw err;
    console.log('Saved: ' + newData);
  });
}
```

Listing 1.61. `fileWrite.js`

Our intent is to generate an event each time the file is changed. To this end, we can define the following Java operation, that exploits the:

```
    public void watchFileInDir(String dirpath) {
        Path myDir = Paths.get(dirpath);
        try {
           WatchService watcher = myDir.getFileSystem().newWatchService();
           myDir.register(watcher, StandardWatchEventKinds.ENTRY_CREATE,
           StandardWatchEventKinds.ENTRY_DELETE, StandardWatchEventKinds.ENTRY_MODIFY);
           System.out.println("WATCHING " + dirpath);
           WatchKey watckKey = watcher.take();
           List<WatchEvent<?>> events = watckKey.pollEvents();
           for (WatchEvent event : events) {
                if (event.kind() == StandardWatchEventKinds.ENTRY_CREATE) {
                    System.out.println("Created: " + event.context().toString());
                }
                if (event.kind() == StandardWatchEventKinds.ENTRY_DELETE) {
                    System.out.println("Delete: " + event.context().toString());
                }
                if (event.kind() == StandardWatchEventKinds.ENTRY_MODIFY) {
                    System.out.println("Modify: " + event.context().toString());
                   emitFileContent( dirpath+"/"+event.context().toString() );
                }
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.toString());
        }
    }
```

This operation is inserted in the `QActor` class, to provide a new built-in operation. In some future release of the *QActor* metamodel, it could could be provided as a *QActor* primitive. The operation `emitFileContent` reads the file and generates an event of the form:

```
fileChanged  : fileChanged(FNAME,FCONTENT)
```

Our final step is the definition of a *QActor* system that activates the file watcher and handles the `fileChanged` events:

```
System fileChange
Event fileChanged : fileChanged(F,X) //F filename X current content
Context ctxFileChange ip[ host="localhost" port=8061]
/*
 * This system acts as an OBSERVER for change of files in the directory
 * C:/repoGitHub/it.unibo.qa.integrator/sharedFiles
 * Any change in some file (e.g. cmd.txt) generates the event: fileChanged : fileChanged(F,X)
 */
QActor qafilechange context ctxFileChange {
    Plan init normal[ println("START WATCHING");
        actorOp watchFileInDir("C:/repoGitHub/it.unibo.qa.integrator/sharedFiles")
    ]
    finally repeatPlan
}
QActor qafilechangehandler context ctxFileChange {
    Plan init normal[ actorOp noOP ]
    transition stopAfter 30000
    whenEvent fileChanged : fileChanged(F,X) do printCurrentEvent,
    whenEvent fileChanged : fileChanged(F,X) do println(X)
    finally repeatPlan
}
```

Listing 1.62. `fileChange.qa`

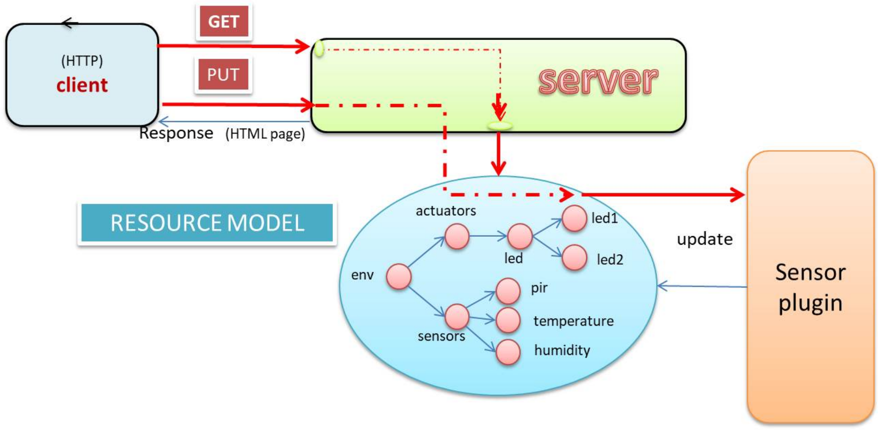
9 Towards the Web of Things

The goal of this section is to build the prototype of a software system whose logic architecture is inspired to the HTTP/REST model adopted in the field of the *Web Of Things* (WoT).

The Internet of Things (IoT) can be defined as a system of physical objects that can be discovered, monitored, controlled, or interacted with by electronic devices that communicate over various networking interfaces and eventually can be connected to the wider internet. Since actually there is no unique and universal application protocol for the IoT that can work across the many networking interfaces, the IoT is essentially a growing collection of isolated *Intranets of Things* that can't be connected to each other. We need a single universal application layer protocol (language) for devices and applications to talk to each other, regardless of how they're physically connected.

The WoT relies exclusively on Application-level protocols and tools. Working with such a high level of abstraction, makes it possible to connect data and services from many devices regardless of the actual transport protocols they use. By hiding the complexity and differences between various transport protocols used in the IoT, the Web of Things allows developers to focus on the logic of their applications without having to bother about how this or that protocol or device actually works.

In the WoT, devices and their services are fully integrated in the web because they use the same standards and techniques as traditional websites. We can write applications that interact with embedded devices in exactly the same way as we would interact with any other web service that uses web APIs, in particular, RESTful architectures. Recent embedded web servers with advanced features can be implemented with only 8 KB of memory. Thanks to efficient cross-layer TCP/HTTP optimizations, they can run on tiny embedded systems or even smart cards.



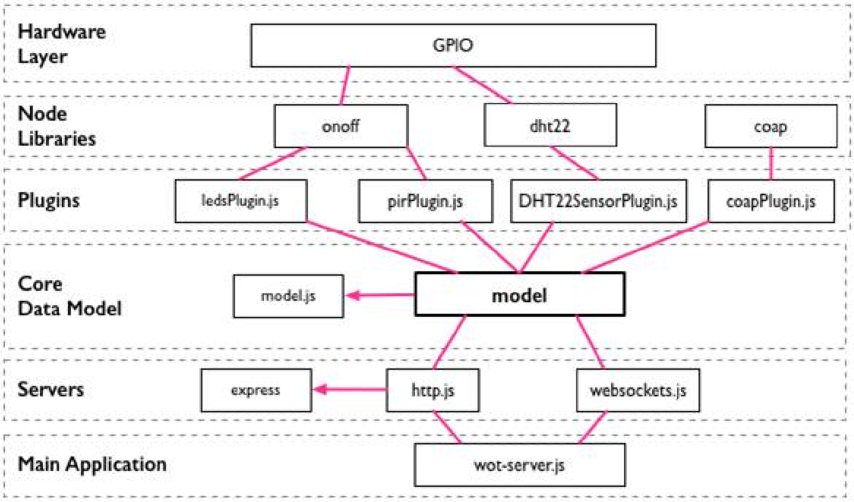
The main aspects that qualify such an architecture are:

- Integration patterns. Things must be integrated to the Internet and the WEB in several ways: using REST (*Representational State Transfer*) on device, by means of Applications Gateways (via specific IoT protocols, like the UDP-based CoAP (*Constrained Application Protocol*)), or by means of remote servers using the Cloud (via publish-subscribe protocols like MQTT).
- Resource (model) design. Each Thing should provide some functionality or service that must be modelled and organized into an hierarchy. Usually, physical resources are currently mapped into REST resources by means of description files written in JSON.
- Representation design. Each resource must be associated with some representation, e.g. JSON, HTML, MessagePack, ect.
- Interface design. Each service can be used by means of a set of commands that must be properly designed. In the REST model, commands are expressed by means of HTTP verbs (GET, PUT, POST, etc.) and often associated with *publish-subscribe* interaction via WebSockets.

– Resource linking design. The different resources must be discovered over the network are often logically linked to each other, for example according to the HATEHOAS (*Hyepermedia as Enigine of Application State*) principle, based on the Web-linking mechanisms: the HTTP header of a response contains the links to related resources.

The world of *Things* is massively dominated by devices running low-level C programs. However, although JavaScript and Node.js are not the optimal solution for every IoT project, their usage is growing in modern IoT and WoT development, since a single language can be used to build the client application, the cloud engine or gateways, and even the code running on the embedded device. In fact, a number of embedded device platforms (most Linux-based platforms indeed) today directly support JavaScript and Node.js to write embedded code.

The following picture (taken from the WoT book[9]) shows the component architecture of a WoT server completely built by using JavaScript and Node.js.

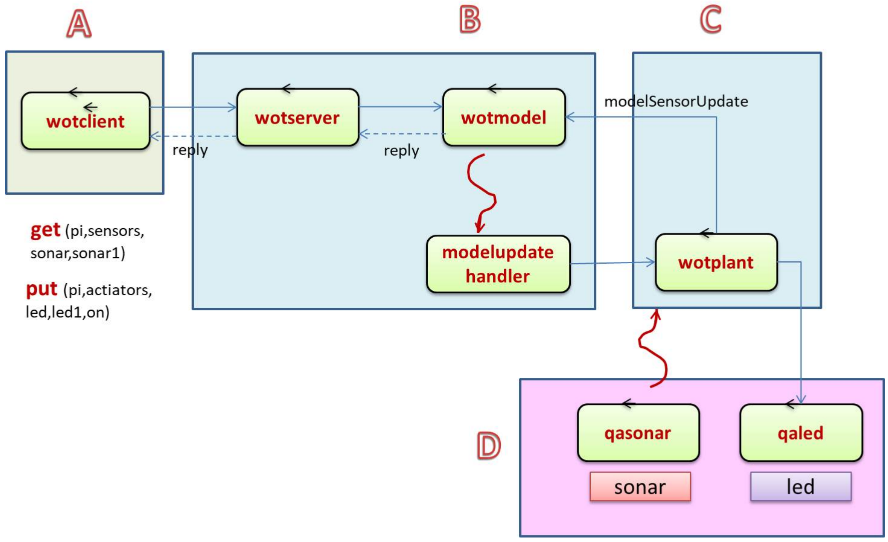


But IoT and WoT application require to build software for distributed systems whose nodes are quite always programmed with a set of different languages and frameworks (i.e they are heterogeneous systems). In this scenario, we can exploit a *QActor* model as an integrator of heterogeneous distributed activities (see Section 8).

[9] Building the Web of Things: With Examples in Node.js and Raspberry Pi. Dominique Guinard, Vlad M. Trifa, Manning 2016, ISBN-10: 1617292680

9.1 A (WoT) logical model

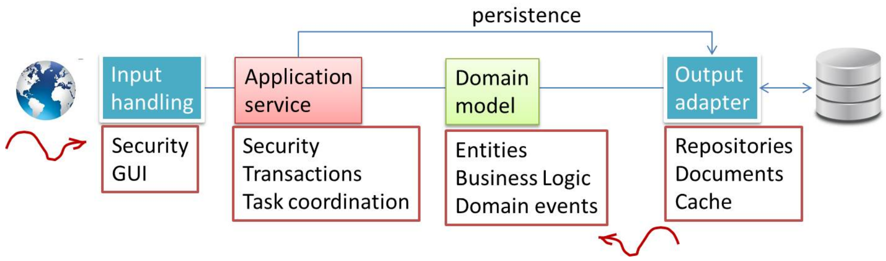
The model shown in the following picture aims at capturing the main components of a `WoT` application, by highlight the key-role of the `wotmodel` (*Resource Model*) in the logical architecture of the system.



From the structural point of view, our model does introduce 4 contexts:

- The `user` context, represented by `A`. This part of the software system could be implemented by a browser.
- The `server` context, represented by `B`. This part is usually implemented on a set of `N>1` computers.
- The `gateway` context, represented by `C`. This part is usually implemented on a conventional PC.
- The `device` context, represented by `D`. This part is usually implemented on low-cost devices like Arduino, Raspberry, etc.

The model can be viewed as an instance of the following schema (isnpired by the concept of *hexagonal architecture*):



The relevant points are:

- all the components are modelled as active entities (e.g. `micro-services`) that interact via message passing or via events;
- the application layer and the physical devices are not directly connected: they interact by means of a `resource model` that is put at the centre of the system;
- the software that implements the domain layer does not depend on the other layers.

9.1.1 A resource model :

```
System wotLogic
Dispatch request        : request( CMD )   //sent by a client (via a browser)
Dispatch reply          : reply( V )       //reply to the client request
Dispatch resModelAction : cmd(CMD )        //request to read/change the resource model
Dispatch resModelAnswer : resanswer( A )   //answer from the model
Dispatch modelSensorUpdate : update(TYPE,ID,V) //request from a plant to change the model of a sensor

Event modelUpdate : modelUpdate( V )       //event raised when the resource model is changed
Event sonar     : p( Distance, SONARID )   //event raised by a sonar

Context ctxWotLogic ip [ host="localhost" port=8059 ]

/*
 * Resource model
 */
QActor wotmodel context ctxWotLogic -g yellow {  //
Rules{
root( pi, "WotPi", "Sensor on Raspberry", 8484 ).

sensor( pi, temperature, t1 ).
sensor( pi, pir, pir1 ).
sensor( pi, humidity, hum1 ).
sensor( pi, sonar, sonar1 ).
sensor( pi, sonar, sonar2 ).

actuator( pi, led, led1).
actuator( pi, led, led2).

descr( temperature, t1, data( "Temperature sensor 1", unit(celsius), value(0), gpio(12) ) ).
descr( humidity,  hum1, data( "Humidity sensor 1", unit(percent), value(0), gpio(12) ) ).
descr( pir,       pir1, data( "Passive Infrared 1", value(true), gpio(17) ) ).
descr( sonar,     sonar1, data( p(0,sonar1), gpio(11) ) ).
descr( sonar,     sonar2, data( p(0,sonar2), gpio(13) ) ).

descr( led,       led1, data( "Led 1", value(false), gpio(4) ) ).
descr( led,       led2, data( "Led 2", value(false), gpio(9) ) ).
}
    Plan init normal [
        println("wotmodel STARTS");
        demo consult("./src/it/unibo/wotmodel/routeRules.pl")
    ]
    switchTo handleRequest

    Plan handleRequest[ actorOp noOp ]
        transition stopAfter 30000
        whenMsg resModelAction -> doResourceModelAction ,
        whenMsg modelSensorUpdate -> doResourceModelAction
        finally repeatPlan
    Plan doResourceModelAction resumeLastPlan[
      printCurrentMessage ;
      onMsg modelSensorUpdate : update(TYPE,ID,V) -> demo setsensor( model(pi,sensor,TYPE,ID,V), ANSWER );
      onMsg resModelAction  :   cmd(CMD) -> demo route(CMD, ANSWER);
      [ !? goalResult(R) ] println( wotmodel( R ) );
      [ !? goalResult(route(_, ANSWER)) ] println( wotmodel( ANSWER ) );
      [ ?? goalResult(route(_, ANSWER)) ] replyToCaller -m resModelAnswer : resanswer(ANSWER)
    ]
}
```

Listing 1.63. `wotLogic.qa`: resource model

9.1.2 Routing rules :

```
/*
```

```
----------------------------------
it/unibo/qawotlserver/routeRules.pl
----------------------------------
*/
route( get(ROOT,sensors,TYPE,ID), ANSWER ) :-
     %%output( get(ROOT,sensors,TYPE,ID) ),
     getSensors( model(ROOT,sensor,TYPE,ID), ANSWER).

route( put(ROOT,sensors,TYPE,ID, V), ANSWER ) :-
     %%output( put(ROOT,sensors,TYPE,ID,V) ),
     setSensor( model(ROOT,sensor,TYPE,ID,V), ANSWER).

getSensors( model(ROOT,sensor,temperature,none) , todo(allTemperatore)):- !.
    getSensors( model(ROOT,sensor,none,none) , todo(allSensor)):- !.
getSensors(model(ROOT,sensor,TYPE,ID) , ANSWER ) :-
    sensor( ROOT, TYPE, ID ),
    %%output( sensor( ROOT, TYPE, ID ) ),
    descr(TYPE, ID, ANSWER ).

setSensor( model(ROOT,sensor,temperature,ID,V) , done(temperature,ID,V)) :-
    %%output( setSensor_temperature ),
    replaceRule(
        descr( temperature, ID, data( D, U, value(X), GPIO ) ),
        descr( temperature, ID, data( D, U, value(V), GPIO ) )
    ) ,
    actorobj(Actor),
    Actor <- emit( "modelUpdate", modelUpdate( ID, GPIO ) ).
setSensor( model(ROOT,sensor,sonar,ID,V) , done(sensor,ID,V) ) :-
    %%output( setSensor_sonar ),
    replaceRule(
        descr( sonar, ID, X, GPIO ),
        descr( sonar, ID, V, GPIO )
    ) ,
    actorobj(Actor),
    Actor <- emit( "modelUpdate", modelUpdate( ID, GPIO ) ).

initRules :-  output( initRules ).
:- initialization(initRules).
```

Listing 1.64. `routeRules`

9.1.3 An actor working as a server :

```
QActor wotserver context ctxWotLogic {  //-g cyan
    Plan init normal [ println("wotserver STARTS") ]
    switchTo handleRequest

    Plan handleRequest[ actorOp noOp ]
        transition stopAfter 30000
            whenMsg request -> elabRequest
            finally repeatPlan
    Plan elabRequest resumeLastPlan[
      printCurrentMessage ;
      onMsg request : request( CMD ) -> forward wotmodel -m resModelAction : cmd(CMD) ;
      actorOp memoCurrentCaller  //otherwise replyToCaller does not work properly
    ]
    transition stopAfter 30000
            whenMsg resModelAnswer -> handleModelAnswer
    Plan handleModelAnswer resumeLastPlan[
        onMsg resModelAnswer : resanswer(A) -> println( rrr(A)) ;
        onMsg resModelAnswer : resanswer(A) -> replyToCaller -m reply : reply(A)
   ]
}
```

Listing 1.65. `wotLogic.qa`: server

9.1.4 A client (simulator) :

```
QActor wotclient context ctxWotLogic -g cyan {
Rules{
   test( get(pi,sensors,temperature,t1) ).
   testChange( put(pi,sensors,temperature,t1, 10) ).

   test( get(pi,sensors,temperature,t1) ).
   testChange( put(pi,sensors,temperature,t1, 20) ).

   test( get(pi,sensors,sonar,sonar1) ).
}
   Plan init normal[
      println("wotclient STARTS " )
   ]
   switchTo doRequest

   Plan doRequest [
      //println("qawotlclient GET " ) ;
      [ ?? test(REQ)] forward wotserver -m request : request( REQ ) else endPlan "no more test"
    ]
   transition stopAfter 30000
      whenMsg reply -> getAnswer
   finally repeatPlan

   Plan getAnswer resumeLastPlan [
      onMsg reply : reply(V) -> println( client( V ) )
   ]
   switchTo changevalue

   Plan changevalue resumeLastPlan[
      //println("qawotlclient PUT " ) ;
      [ ?? testChange(REQ)] forward wotserver -m request : request( REQ ) else endPlan "no more testChange"
   ]
   transition stopAfter 30000
      whenMsg reply : reply(V) do println( client( V ) )
}
```

Listing 1.66. `wotLogic.qa`: client (simulator)

9.1.5 An actor that handles model-update events :

```
QActor modelupdatehandler context ctxWotLogic -g green {
   Plan init normal[
      actorOp noOp
   ]
   transition stopAfter 30000
      whenEvent modelUpdate : modelUpdate(V) do printCurrentEvent //println( modelUpdate( V ) )
   finally repeatPlan
}
```

Listing 1.67. `wotLogic.qa`: model update handler

9.1.6 An actor that simulates a sonar :

```
QActor qasonar context ctxWotLogic {
Rules{
   p(80,sonar1).
   p(70,sonar1).
   p(60,sonar1).
   p(40,sonar1).
```

```
}
    Plan init normal[
     delay time(1000) ;
     [ ?? p(DIST, SID) ] emit sonar : p(DIST,SID)
    ]
    finally repeatPlan 9
}
```

Listing 1.68. `wotLogic.qa`: sonar simulator

9.1.7 An actor that simulates a plant (that modifies the model) :

```
QActor qawotplant context ctxWotLogic -g white {
    Plan init normal[
        actorOp noOp
    ]
    transition stopAfter 120000
        whenEvent sonar -> updateTheModel
        finally repeatPlan
    Plan updateTheModel resumeLastPlan[
        printCurrentEvent;
        onEvent sonar : p(DIST,SID) ->
            forward wotmodel -m modelSensorUpdate : update( sonar,SID,p(DIST,SID) )
    ]
}
```

Listing 1.69. `wotLogic.qa`: plant simulator

10 Playing with real devices

In this section, we introduce, as an example of physical input device, a sonar HC-SR04. The device is connected to a RaspberryPi as follows:

- vcc pin connected to the physical GPIO 4 (5V)
- gnd pin connected to the physical GPIO 6 (GND)
- trig pin connected to the physical GPIO 16 (4 in WiringPi numeration)
- echo pin connected to the physical GPIO 18 (5 in WiringPi numeration)

The physical sonar is made working by the following program written in C:

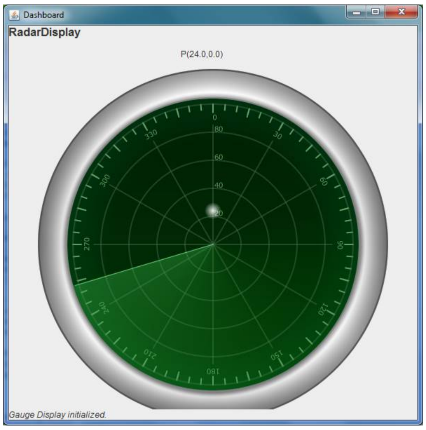
```
#include <iostream>
#include <wiringPi.h>
#include <fstream>
#include <cmath>

//#define TRUE 1
//Wiring Pi numbers for radar with stepper
#define TRIG 4
#define ECHO 5

using namespace std;
/*
 * in the directory: of SonarAlone.c:
1)  [ sudo ../../pi-blaster/pi-blaster ] if servo
2)  g++  SonarAlone.c -l wiringPi -o SonarAlone
sudo ./SonarAlone
In nat/servosonar:
sudo java -jar SonarAloneP2PMain.jar
sudo python radar.py
 */
void setup() {
    wiringPiSetup();
    pinMode(TRIG, OUTPUT);
    pinMode(ECHO, INPUT);

    //TRIG pin must start LOW
    digitalWrite(TRIG, LOW);
    delay(30);
}

int getCM() {
    //Send trig pulse
    digitalWrite(TRIG, HIGH);
    delayMicroseconds(20);
    digitalWrite(TRIG, LOW);

    //Wait for echo start
    while(digitalRead(ECHO) == LOW);

    //Wait for echo end
    long startTime = micros();
    while(digitalRead(ECHO) == HIGH);
    long travelTime = micros() - startTime;

    //Get distance in cm
    int distance = travelTime / 58;

    return distance;
}

int main(void) {
    int cm ;
    setup();
    while(1) {
        cm = getCM();
```

```
        cout << cm <<  endl;
        delay(300);
    }
    return 0;
}
```

Listing 1.70. `sonarAlone.c`

This program writes on the standard output strings of the form `p( Distance )`.

A proper *QActor* model can specify how to transform the data given by the sonar a *QActor* event of the form `polar : p(Distance, Angle)`. The `Angle` value is a integer (`0<=Angle<=90`) that can be used to distinguish among different sonar values. In the project *it.unibo.qactor.radar* this value is used to display the sonar values on an output device that simulates the screen of a radar.



For example, in the *it.unibo.qactor.radar* project, we define an actor to be executed on the RaspberryPi connected to the sonar:

```
/*
 * ==============================================================
 * sonarSensorEmitter.qa
 * ==============================================================
 */
System sonarSensorEmitter
Event polar   : p( Distance, Angle )

Context ctxRadarBase     ip [ host="192.168.43.229" port=8033 ] -standalone
Context ctxSensorEmitter ip [ host="192.168.43.66" port=8073 ]

QActor sensorsonar context ctxSensorEmitter {
Rules{
    /* SIMULATION DATA */
        p(80,0). p(85,20). p(90,40). p(85,60). p(85,80). p(80,100). p(75,120). p(70,140).p(65,160). p(70,180).
    }
    Plan init normal [
        //println("sensorsonar STARTS") ;
        actorOp startSonarC   //activate the program sonarAlone.c
    ]
    switchTo workSimulate
    //Just to show if it works ...
    Plan workSimulate resumeLastPlan [
        [ !? p(D, A) ] println( simulate( p(D, A) ) ) ;
        [ ?? p(D, A) ] emit polar : p(D,A) else endPlan "endSimulate" ;
        delay  500
    ]
    switchTo workReal
    //Handle real data
```

```
    Plan workReal [
       println("sensorsonar workReal") ;
       actorOp getDistanceFromSonar ; //get data from the output of sonarAlone.c
       [ !? actorOpDone( OP,d(D) )] println( p(D,90) ) ;
       [ ?? actorOpDone( OP,d(D) )] emit polar : p(D,90)
    ]
    finally repeatPlan
}
```

Listing 1.71. `sonarSensorEmitter.qa`

The flag `-standalone` in the declaration of a *Context* (`ctxRadarBase` in the example) indicates that our system is using actors working in a computational node already in existence. The actors declared (if any) in a `-standalone` *Context* are 'place holders' for already defined (and working) actors. Such a declaration is required for reference purposes (e.g. message passing, that requires a reference to an actor name). In this case we do not need any declaration, since we do not use messages; the events are automatically propagated to all the nodes by the *QActor* runtime.

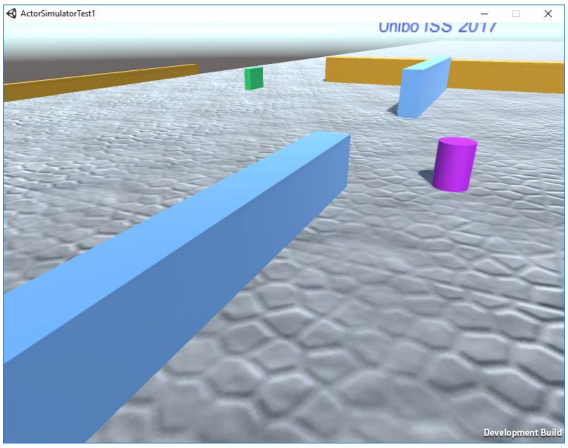
The code written by the application designer is related to the operations `startSonarC` and `getDistanceFromSonar`:

```
public class Sensorsonar extends AbstractSensorsonar {
    public Sensorsonar(String actorId, QActorContext myCtx, IOutputEnvView outEnvView ) throws Exception{
        super(actorId, myCtx, outEnvView);
    }
/*
 * ADDED BY THE APPLICATION DESIGNER
 */
    protected BufferedReader readerC;

    public void startSonarC(){
        try {
            println("startSonarC" );
            Process p = Runtime.getRuntime().exec("sudo ./SonarAlone");
            readerC  = new BufferedReader(new java.io.InputStreamReader(p.getInputStream()));
            println("Process in C STARTED " + readerC);
            println("Process in C reads " + getDistanceFromSonar() );
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public String getDistanceFromSonar(){
        try {
            String inpS = readerC.readLine();
            println("getDistanceFromSonar " + inpS );
            return "d("+inpS+")";
        } catch (Exception e) {
            e.printStackTrace();
            return "d(0)";
        }
    }

}
```

Listing 1.72. `Sensorsonar.java`

11 Playing with virtual devices (in Unity)

During software analysis and testing, it could be preferable to use virtual devices rather real, physical devices. Unity can be used as a simulation environment able to provide a virtual working environment and sensor data. For example, the following Unity scene shows an environment made of a set of walls and fixed obstacles, a mobile obstacle (the cylinder) and a sonar (the small box in green).



Our Unity environment has been modified to interact with *QActor* systems.

The operation `createSimulatedActor` can be used in a *QActor* model to 'inject' into the virtual environment a qactor (to be called `rover`, at the moment) that can be moved in a Unity scene by using proper commands (**`onward, backwards, left, right, stop`**).

When the `rover` is intercepted by the sonar, the (modified) Unity system emits the event `sonar : sonar(SONAR, TARGET, DISTANCE)`.

Moreover, the virtual game object that represents the `rover` is equipped (in its front) with a sonar, that emits the event `sonarDetect : sonarDetect(X)` when detects an obstacle.

Both these events that cam be handled by our applications. For example:

```
System testRover
Event sonarDetect   : sonarDetect(X) //From (virtual robot) sonar
Event sonar         : sonar(SONAR, TARGET, DISTANCE) //From (virtual) sonar

Context ctxRover ip [ host="localhost" port=8070 ]
EventHandler evh for sonarDetect , sonar -print ;

QActor rover context ctxRover {
   Plan init normal [
         println("rover START");
         actorOp workWithUnity("localhost") ;
         actorOp createSimulatedActor("rover","Prefabs/CustomActor") ;
         right 50 time ( 1000 ) //position
      ]
      switchTo moveVitualRobot

   Plan moveVitualRobot [
      println("moveVitualRobot")
   ]
   reactive onward 40 time ( 5000 )
         whenEnd -> endOfMove
         whenTout 30000 -> handleTout
         whenEvent sonarDetect -> handleObstacle
         or whenEvent sonar  -> handleSonar
   finally repeatPlan

```

```
  Plan handleSonar resumeLastPlan [
      onward 50 time ( 300 ) ; //out of sonar range
      stop 50 time ( 1000 )    //stop for a while ...
  ]
  Plan handleObstacle resumeLastPlan [ backwards 50 time ( 3500 ) ]
  Plan endOfMove resumeLastPlan [ println("endOfMove") ]
  Plan handleTout [ println("handleTout") ]

}
```

Listing 1.73. `testRover.qa`

11.1 The MqttUtils

The MQ *Telemetry Transport* (MQTT) is an ISO standard (ISO/IEC PRF 20922) for a "light weight" messaging protocol (on top of TCP/IP) based on the publish-subscribe pattern.

The Eclipse Paho project provides open-source client implementations of MQTT and MQTT-SN for Sensor Networks[10]) to be used in emerging applications for *Machine-to-Machine* (M2M) and *Internet of Things* (IoT).

There are already MQTT C and Java libraries with Lua, Python, C++ and JavaScript at various stages of development.

The *QActor* runtime software provides a Java utility class to be used as a support for MQTT interaction. It is defined as follows:

```java
package it.unibo.qactors.mqtt;
import it.unibo.qactors.akka.QActor;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import alice.tuprolog.Struct;
import alice.tuprolog.Term;

public class MqttUtils implements MqttCallback{
private static MqttUtils myself = null;
    protected String clientid = null;
    protected String eventId  = "mqtt";
    protected String eventMsg = "";
    protected QActor actor    = null;
    protected MqttClient client = null;

    public static MqttUtils getMqttSupport( QActor qa ){
//      System.out.println("  %%% MqttUtils getMqttSupport qa="+ qa );
        if( myself == null ) myself = new MqttUtils();
        return myself ;
    }
    public MqttUtils(){
        try {
            myself  = this;
        } catch (Exception e) {
            println("  %%% MqttUtils WARNING: "+ e.getMessage() );
        }
    }
    public void connect(QActor actor, String clientid, String brokerAddr, String topic ) {
        try{
            this.actor = actor;
            client = new MqttClient(brokerAddr, clientid);
//          println("  %%% MqttUtils connect/4 "+ clientid + " " + brokerAddr + " " + topic + " " + client);
            MqttConnectOptions options = new MqttConnectOptions();
            options.setKeepAliveInterval(480);
            options.setWill("unibo/clienterrors", "crashed".getBytes(), 2, true);
            client.connect(options);
        }catch(Exception e){
            actor.println("MqttUtils connect ERROR " + e.getMessage());
        }
    }
    public void disconnect( ) {
        try{
            println("  %%% MqttUtils disconnect "+ client );
            if( client != null ) client.disconnect();
        }catch(Exception e){
            actor.println("MqttUtils disconnect ERROR " + e.getMessage());
        }
    }

```

[10] MQTT-SN is a protocol derived from MQTT, designed for connectionless underlying network transports such as UDP

```
    public void subscribe(QActor actor, String clientid, String topic) throws Exception {
        try{
            this.actor = actor;
            client.setCallback(this);
            client.subscribe(topic);
        }catch(Exception e){
                println("  %%% MqttUtils subscribe error "+ e.getMessage() );
                eventMsg = "mqtt(" + eventId +", failure)";
                println("  %%% MqttUtils subscribe error "+ eventMsg );
                if( actor != null ) actor.sendMsg("mqttmsg", actor.getName(), "dispatch", "error");
                throw e;
        }
    }
    @Override
    public  void connectionLost(Throwable cause) {
        println("  %%% MqttUtils connectionLost = "+ cause.getMessage() );
    }
    @Override
    public  void deliveryComplete(IMqttDeliveryToken token) {
//      println("         %%% MqttUtils deliveryComplete token= "+ token );
    }
    /*
     * sends to a tpoic a content of the form
     *      msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
     */
    public void publish(QActor actor, String clientid, String topic, String msg,
            int qos, boolean retain) throws MqttException{
        MqttMessage message = new MqttMessage();
        message.setRetained(retain);
        if( qos == 0 || qos == 1 || qos == 2){//qos=0 fire and forget; qos=1 at least once(default);qos=2 exactly once
            message.setQos(0);
        }
        message.setPayload(msg.getBytes());
        try{
            client.publish(topic, message);
//          println("         %%% MqttUtils published "+ message + " on topic=" + topic);
        }catch(MqttException e){
            println("  %%% MqttUtils publish ERROR "+ e );
        }
    }
    /*
     * receives a message of the form
     *          msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
     * and sends it to the RECEIVER
     */
//  private String msgID     = null;
//  private String msgType   = null;
    private String msgSender = null;
//  private String dest      = null;
//  private String msgcontent = null;

    @Override //MqttCallback
    public void messageArrived(String topic, MqttMessage msg) {
         String msgID      = null;
         String msgType    = null;
         String dest       = null;
         String msgcontent = null;
        try {
//              println("  %%% MqttUtils messageArrived on "+ topic + ": "+msg.toString());
            Struct msgt     = (Struct) Term.createTerm(msg.toString());
            println("messageArrived msgt "+ msgt + " actor=" + actor.getName() );
             msgID      = msgt.getArg(0).toString();
             msgType    = msgt.getArg(1).toString();
             msgSender = msgt.getArg(2).toString();
             dest       = msgt.getArg(3).toString();
             msgcontent = msgt.getArg(4).toString();
             if( actor != null ) //send a msg to itself (named without _ctrl)
                actor.sendMsg( msgSender, msgID, actor.getName().replace("_ctrl", ""), msgType, msgcontent);
        } catch (Exception e) {
//          println("messageArrived ERROR "+e.getMessage() );
```

```
        }
    }

    //Used by QActor
    public String getSender(){ return (msgSender!=null) ? msgSender.replace("_ctrl", "") : "notyet"; }
```

Listing 1.74. The utility class `MqttUtils.java`

An object of class `MqttUtils` is used as a *singleton* and works as the support for the actor that calls the operation `connect` (that creates a `MqttClient`). Thus, in this version a *QActor* can connect to a single `MQTTT` server only.

The `subscribe` operation sets this singleton support as the object that provides the callback (`messageArrived`) to be called when the `MqttClient` is a subscriber. The message must have the form of a *QActor* message:

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

The callback `messageArrived` sends the received message to the actor that is using the singleton support.