

Introduction to QActors

(2018)

Antonio Natali

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

Table of Contents

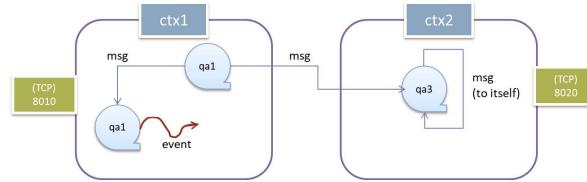
1 Introduction to QActors

QActor is the name given to a custom meta-model inspired to the [actor model](#) (as supported by the Akka library). The [qa](#) language is a custom language that can allow us to express in a concise way the [structure](#), the [interaction](#) and the [behaviour](#) of (distributed) software systems whose components interacts by using messages and events.

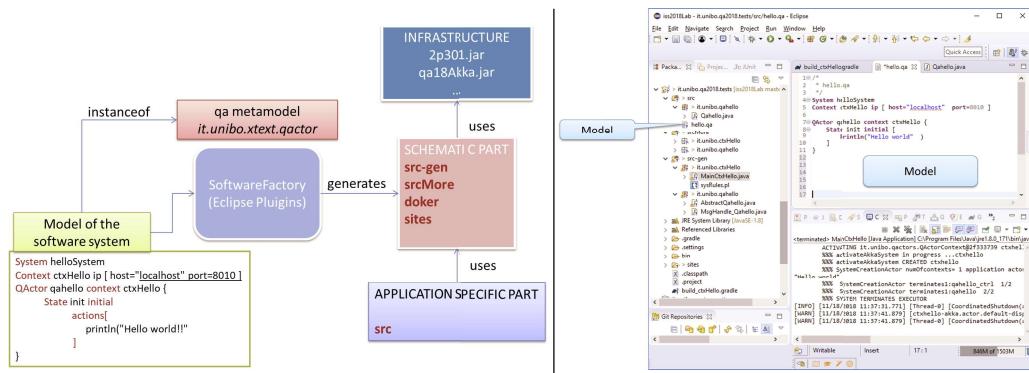
1.1 Overview

The leading [Q/q](#) in the *QActor* word, means 'quasi' since the *QActor* meta-model and the [qa](#) language do introduce (with respect to [Akka](#)) their own peculiarities, including reactive actions and even-driven programming concepts. Let us summarize the main features of a [qa](#) system:

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

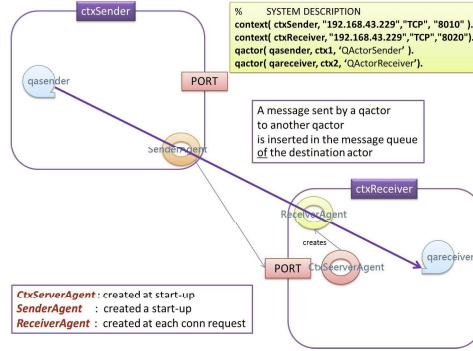


- A [QA-System](#) can be specified in textual form by means of the *QActor* language/metamodel. The *QActor* language is associated to a [software factory](#) (Subsection ??) that automatically generates the proper system *run-time support* so to allow Application designers to focus on application logic.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

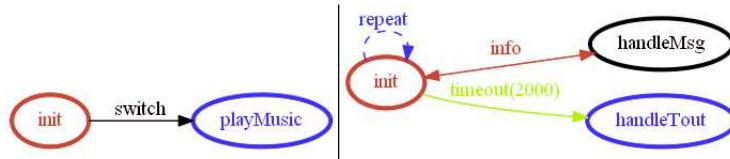


- The [configuration](#) of a [QA-System](#) is explicitly represented by a set of 'facts' written in [tuProlog](#) syntax (Subsection ??) replicated in each Context (Context Knowledge Base or simply [ContextKB](#)). A [QA-System](#) can be configured in a static or in a dynamic way. In case of [dynamic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

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



- To deliver information among the Contexts, the **QA-Infrastructure** can use pairwise TCP connections between the Contexts or a **MQTT** broker. The choice is up to the Application designer.
- Each **QActor** behaves as a (Moore's) *Finite State Automaton* (**FSM**). While in a state, a **QActor** can execute both **synchronous** and asynchronous actions. An **asynchronous** action terminates immediately and emits an event (see Subsection ??) when it terminates.
- A **state-transition** from a state **S1** to another state **S2** can be done (triggered) only when all the actions activated in **S1** are terminated. A state-transition can be triggered by a message or by an event or as a choice of the **QActor** (no-input transition).



- A state-transition can be associated to a boolean condition (**guard**) and can be triggered only when the guard is true (see Subsection ??). When two or more transitions are possible, the systems checks the possibility to trigger one of them by following the order in which they are written. If no transition can be triggered, the **QActor** remains in the state until the occurrence of some message or event that allows a transition.
- A **QActor** is able to execute a set of **pre-defined actions** (see Subsection ??, Subsection ??). The application designer can define **Application specific actions** by using Java or other languages (in particular **JavaScript/Node** and **Prolog**).
- A **QActor** is associated to a private knowledge-base (**QaKB**, Section ??) written in **Prolog** that can be dynamically extended by the Application designer.

Let us start with some example.

The code of the examples shown in this work can be found in the project [it.unibo.qa2018.tests](#).

1.2 Example: the 'hello world'

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

```
1  /*
2   * hello.qa
3   */
4 System helloSystem
5 Context ctxHello ip [ host="localhost" port=8010 ]
6
7 QActor qahello context ctxHello {
8     State init initial [
9         println("Hello world" )
10    ]
11 }
12 /*
13 * Another style: Plan vs of State, normal vs initial
14 */
15 QActor qahelloold context ctxHello {
16     Plan init normal
17     actions[
18         println("Hello world old style" )
19     ]
20 }
```

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

The example shows also an older version in which:

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

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

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

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

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

State transitions can be performed when:

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

1.3 Example: no-input transitions

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

```
1 System noinputTansition
2 Context ctxNoinputTansition ip [ host="localhost" port=8079 ]
3
4 QActor qahellonoinputtrans context ctxNoinputTansition{
5     State init initial [
6         println( playSomeMusic );
7         delay 1000 //wait for a second
8     ]
9     switchTo playMusic
10
11     State playMusic [
12         sound time(2000) file('../audio/tada2.wav')
13 ]
```

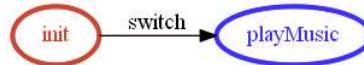
Listing 1.2. noinputTansition.qa

switchTo is a predefined operation of the *QActor* language (see Subsection ??).

sound is a *QActor* built-in action (see Subsection ??, Subsection ??) that ends after 2 seconds.

delay is a built-in action that waits for the specified number of **msecs**.

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



This state diagram is automatically generated¹ by the *QActor* software factory (see Subsection ??).

1.4 The custom language qa

The **qa** language is a custom language² built by exploiting the **XText** technology; thus, it is also a **metamodel**. Technically we can say that **qa** is a 'brother' of UML, since it is based on **EMOF**.

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

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

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

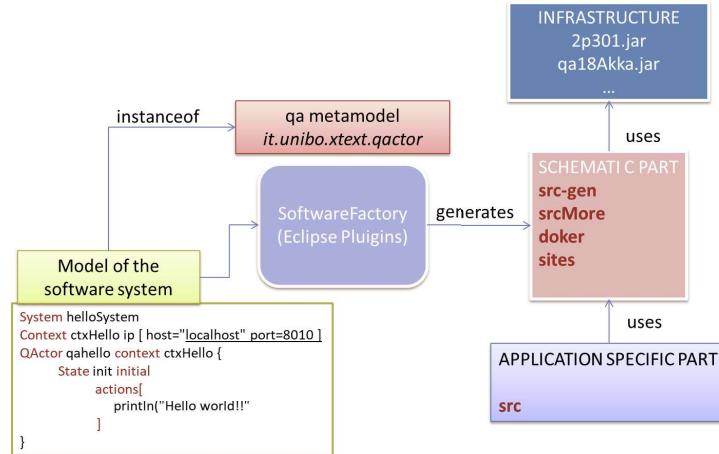
```
1 QActorSystem:
2     "System" spec=QActorSystemSpec
3 ;
4 QActorSystemSpec:
5     name=ID
6     ( message  += Message )*
7     ( context  += Context )*
8     ( actor    += QActor )*
9 ;
```

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

¹ See the file `srcMore/it/unibo/qahellonoinputtrans/qahellonoinputtrans.gv`

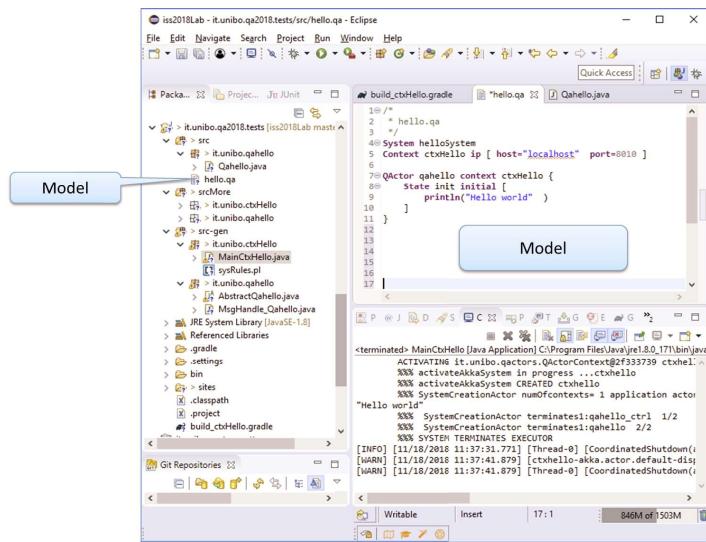
² The **qa** language/metamodel is defined in the project `it.unibo.xtext.qactor`.

1.5 The qa software factory



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

For each *QActor* and for each *Context*, the *QActor* software factory generates (Java) code in the directories `scr-gen` and `src`. Further information (for example about system configuration) is included in the directory `srcMore`, to allow explicit visibility after deployment³. Moreover, a `gradle` build file is also generated for each *Context*.



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

1.5.1 Example of generated files .

In the case of the example of Subsection ??:

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

1.6 The actor's WorldTheory

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

The file `WorldTheory.pl` includes also several computational rules written in tuProlog that extend the action-set of a *QActor*. For example, it includes rules to compute the n-th Fibonacci's number in two ways: in a *fast* way (`fib/2` Prolog rule) and in a *slow* way (`fibo/2` Prolog rule).

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

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

1.6.1 Facts about the state .

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

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

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

1.6.2 Guarded actions .

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

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

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

Let us consider the following example;

```

1 System naiveGuardedActions
2 Context ctxNaiveGuardedAction ip [ host="localhost" port=8037 ]
3 QActor qanaiveguarded context ctxNaiveGuardedAction {
4     State init initial [
5         [ !? true ] println( alwaysHere ) ;
6         [ !? false ] println( neverHere ) ;
7         [ !? fib(6,X) ] println( fib(6,X) ) ;
8         [ ?? fibo(4,X) ] println( fibo(4,X) )
9     ]
10    finally repeatPlan 1
11 }
12 /*
13 OUTPUT
14 -----
15 alwaysHere
16 fib(6,8)
17 fibo(4,3)
18 alwaysHere
19 fib(6,8)
20 */
21 */

```

Listing 1.3. `naiveGuardedActions.qa`

The output shows that the second iteration does not compute any more `fibo/2`, since that rule has been removed in the first iteration.

The example show also that `variables` unified by a guard execution can be referred in the action (the `scope` id the action itself).

1.6.3 The demo operator .

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

```

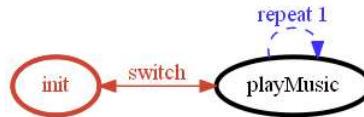
1 System demoExample
2 Context ctxDemoExample ip [ host="localhost" port=8079 ]
3 QActor qademoexample context ctxDemoExample{
4     Plan init normal
5         [ println("qademoexample STARTS" ) ;
6             [ !? actorobj(X) ] println( actorobj(X) ) ;
7             demo actorobj(X) ;
8             [ ?? goalResult(R) ] println( qademoexample( R ) ) ;
9             demo fibo(6,X);
10            [ ?? goalResult(R) ] println(R) ;
11            demo fibo(6,8);
12            [ ?? goalResult(R) ] println(R) ;
13            demo fibo(X,8); //fails since fibo/2 is not invertible
14            [ ?? goalResult(R) ] println(R) ;
15            println("qademoexample ENDS" )
16        ]
17    }
18 /*
19 OUTPUT
20 -----
21 "qademoexample STARTS"
22 actorobj(qatuoqademoexample_ctrl)
23 qademoexample(actorobj(qatuoqademoexample_ctrl))
24 fibo(6,8)
25 fibo(6,8)
26 failure
27 "qademoexample ENDS"
28 */

```

Listing 1.4. `demoExample.qa`

1.7 Example: repeat/resume plans

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



```
1  /*
2   * repeatResume.qa
3   */
4  System repeatResume
5  Context ctxRepeatResume ip [ host="localhost" port=8079 ]
6
7  QActor player context ctxRepeatResume{
8      State init initial [
9          println("player STARTS" );
10         println("player ENDS" )
11     ]
12     switchTo playMusic
13 //     finally repeatPlan 1 //(1)
14
15     Plan playMusic resumeLastPlan [
16         println( playSomeMusic );
17         sound time(2000) file('../audio/tada2.wav') ;
18         delay 500
19     ]
20     finally repeatPlan 1
21 }
22 /*
23 OUTPUT
24 -----
25 "player STARTS"
26 "player ENDS"
27 playSomeMusic
28 playSomeMusic
29 */
```

Listing 1.5. repeatResume.qa

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

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

1.8 How a Plan/State works

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

1. the *QActor* executes, in sequential way, all the actions specified in the *Plan*. Each action must be defined as an *algorithm*, i.e. it must terminate;
2. if the *State/Plan* specification ends with the sentence *finally repeatPlan N* (*N* natural number $N \geq 1$), 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 ??;
 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 ??, the output will be:

```

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

```

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

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

More specifically, we introduce the action types reported in the following table:

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

1.10 Messages and events

A *QActor* can interact with other *QActors* using (see Subsection ??) **Messages** of different types (**Dispatch**, **Request**, **Invitation**, ...) and **Events**.

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

```

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

```

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

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

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

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

Listing 1.6. An example of message declaration

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

1.11 Messages

In the *QActor* metamodel, a **message** is intended as information sent in **asynchronous** way by some source to **some specific destination**.

For *asynchronous* transmission we intend that the messages can be '**buffered**' by the infrastructure, while the '**unbuffered**' transmission is said to be **synchronous**.

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

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

At application level, a message is denoted by specifying a message type, a name and a payload. For example⁵:

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

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

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

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

The **msg/6** pattern can be used at application level to express **guards** (see Subsection ??) to allow conditional evaluation of *Actions*.

⁵ For the syntax of **PHead**, see Subsection ??.

1.11.1 Sending/Receiving messages .

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

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

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

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

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

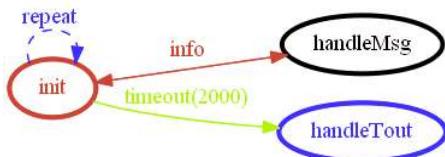
1.11.2 Example: a producer-consumer system .

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* to the consumer:

```
1 System basicProdCons
2 Dispatch info : info(X)
3 Context ctxBasicProdCons ip [ host="localhost" port=8019 ]
4 QActor producer context ctxBasicProdCons{
5     State init initial [
6         println( "producer sends info(1)" );
7         forward consumer -m info : info(1);
8         delay 500;
9         println( "producer sends info(2)" );
10        forward consumer -m info : info(2)
11    ]
12 }
13 QActor consumer context ctxBasicProdCons{
14     State init normal [ println( consumer(waiting) ) ]
15     transition whenTime 2000 -> handleTout
16     whenMsg info -> handleMsg
17     finally repeatPlan
18     State handleMsg resumeLastPlan [
19         printCurrentMessage;
20         onMsg info : info(X) -> println( msgcontent(X) )
21     ]
22     State handleTout
23     [ println( consumerTout ) ]
24 }
```

Listing 1.7. basicProdCons.qa

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



The output is:

```
1 /*
2  * OUTPUT
3  * -----
4  * consumer(waiting)
5  * "producer sends info(1)"
```

```

6 | -----
7 | consumer_ctrl currentMessage=msg(info,dispatch,producer_ctrl,consumer,info(1),1)
8 | -----
9 | msgcontent(1)
10| consumer(waiting)
11| "producer sends info(2)"
12| -----
13| consumer_ctrl currentMessage=msg(info,dispatch,producer_ctrl,consumer,info(2),2)
14| -----
15| msgcontent(2)
16| consumer(waiting)
17| consumerTout
18| */

```

Listing 1.8. basicProdCons.qa

The meaning of a **transition** is discussed in Subsection ??.

1.12 State transitions

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

PlanTransition	-----
PlanTransition : SwitchTransition MsgTransition ;	

Thus, a state transition sentence can start:

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

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

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

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

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

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

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*.

1.12.1 Switch part .

A **TransSwitch** grammar rule allows us to specify

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

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

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

```
1 System demoActionSwitch
2 Dispatch info : payload( CONTENT )
3 Context ctxDemoActionSwitch ip [ host="localhost" port=8019 ]
4
5 QActor sendertoself context ctxMsgToSelf{
6     Plan init initial [
7         selfMsg info : payload( "helloWorld" )
8     ]
9     transition stopAfter 100
10    whenMsg info : payload(V) do println(V)
11 }
```

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

1.12.2 Example: action after a self-message .

A more interesting example is given in the next section

```
1 System msgToSelf
2 Dispatch execute : code( CODE )
3 Dispatch eval : code( CODE )
4
5 Context ctxMsgToSelf ip [ host="localhost" port=8019 ]
6
7 QActor sendertoself context ctxMsgToSelf{
8     Plan init initial [
9         selfMsg eval : code( fibo(10,V) )
10    ]
11    transition whenTime 100 -> handleTout
12        whenMsg execute -> execProg ,
13        whenMsg eval : code(P) do selfMsg execute : code(P)
14        finally repeatPlan 1
15
16     State execProg resumeLastPlan [
17         onMsg execute : code(P) -> demo P;
18         [ ?? goalResult(V) ] println(V)
19     ]
20     State handleTout
21         [ println( consumerTout ) ]
22 }
23 /*
24 OUTPUT
25 -----
26 fibo(10,55)
27 -----
```

Listing 1.9. msgToSelf.qa

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

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

1.13 Events

In the *QActor* metamodel, an **event** is intended as information emitted by some source without any explicit destination. Events can be *emitted* by the *QActors* that compose a *actor-system* or by sources external to the system.

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

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

At application level, an event is denoted by specifying the key-word **Event**, a name and a payload. For example⁶:

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

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

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

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

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

```
1 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.

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

1.13.1 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:

```
1 /*
2  * basicEvents.qa
3  */
4 System basicEvents
5 Event usercmd : usercmd(X)
6 Event alarm : alarm(X)
7
8 Context ctxBasicEvents ip [ host="localhost" port=8037 ]
9
10 QActor qaeventemitter context ctxBasicEvents {
```

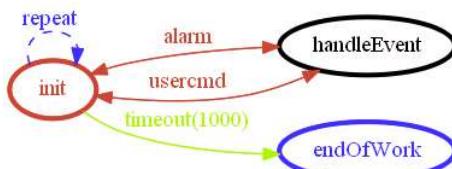
⁶ For the syntax of **PHead**, see Subsection ??.

```

11     State init initial
12     [  println("qaeventemitter STARTS ") ;
13        delay 500 ; // (1)
14        println("qaeventemitter emits alarm(fire)") ;
15        emit alarm : alarm(fire) ;
16        delay 500 ; // (2)
17        println("qaeventemitter emits usercmd(hello)") ;
18        emit usercmd : usercmd(hello) ;
19        println( "qaeventemitter ENDS" )
20    ]
21 }
22 QActor qaeventperceptor context ctxBasicEvents {
23     State init initial
24     [  println("qaeventperceptor STARTS ") ]
25     transition whenTime 1000 -> endOfWork
26         whenEvent alarm -> handleEvent,
27         whenEvent usercmd -> handleEvent
28         finally repeatPlan
29
30     State handleEvent resumeLastPlan
31     [  println("ex4_perceptor handleEvent " ) ;
32        printCurrentEvent ;
33        onEvent alarm : alarm(X) -> println( handling(alarm(X)) ) ;
34        onEvent usercmd : usercmd(X) -> println( handling(usercmd(X)) )
35    ]
36     State endOfWork
37     [  println("qaeventperceptor ENDS (tout) " ) ]
38 }
```

Listing 1.10. basicEvents.qa

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



The output is

```

1  /*
2  OUTPUT
3  -----
4  "qaeventemitter STARTS "
5  "qaeventperceptor STARTS "
6  "qaeventemitter emits alarm(fire)"
7  "ex4_perceptor handleEvent "
8
9  qaeventperceptor_ctrl currentEvent=msg(alarm,event,qaeventemitter_ctrl,none,alarm(fire),3)
10 -----
11 handling(alarm(fire))
12 "qaeventperceptor STARTS "
13 "ex4_alarmemitter emits usercmd(hello)"
14 "ex4_perceptor handleEvent "
15 -----
16 qaeventperceptor_ctrl currentEvent=msg(usercmd,event,qaeventemitter_ctrl,none,usercmd(hello),7)
17 -----
18 handling(usercmd(hello))
19 "qaeventperceptor STARTS "
20 "qaeventemitter ENDS"
21 "qaeventperceptor ENDS (tout)
22 */
```

Listing 1.11. 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.

1.14 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:

```

1 EventHandler :
2   "EventHandler" name=ID ( "for" events += [Event] ( "," events += [Event] )* )?
3   ( print ?= "-print" )?
4   ( "{"
5     "}" body = EventHandlerBody "}" )?
6   ","
6 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:

```

1 EventHandlerOperation: MemoOperation | SolveOperation | RaiseOtherEvent | SendEventAsDispatch ;
2
3 MemoOperation: doMemo=MemoCurrentEvent "for" actor=[QActor] ;
4 MemoCurrentEvent : "memoCurrentEvent" (lastonly?="-lastonly")? ;
5
6 SolveOperation: "demo" goal=PTerm "for" actor=[QActor] ;
7
8 RaiseOtherEvent: "emit" ev=[Event] ("fromContent" content = PHead "to" newcontent=PHead )? ;
9
10 SendEventAsDispatch: "forwardEvent" actor=[QActor] "-m" msgref=[Message] ;

```

- `MemoOperation`: memorize an event into the *WorldTheory* of a specific *QActor*
- `SolveOperation`: ‘tell’ to a specific *QActor* to solve a goal
- `RaiseOtherEvent`: emit another event with the content of the event
- `SendEventAsDispatch`: forward a dispatch with the content of the event

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

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

```

1 /*
2  * eventTracer.qa
3  */
4 System eventTracer
5 Event usercmd : usercmd(X)
6 Event alarm : alarm(X)
7
8 Context ctxEventTracer ip [ host="localhost" port=8027 ]
9 EventHandler evh for usercmd,alarm -print {
10   memoCurrentEvent for qaevtracer
11 };
12 //WARNING: any change in the model modifies the EventHandlers
13 QActor qaevtracer context ctxEventTracer {
14   Plan init normal
15   [ println("qaevtracer starts") ;
16     [ ?? msg(E,'event',S,none,M,N) ]
17       println(qaevtracer(E,S,M)) else println("noevent") ;
18     delay 300
19   ]
20   finally repeatPlan 5
21 }
22 QActor qatraceremitter context ctxEventTracer {
23   Plan init normal

```

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

```

24     [  println("qatraceremitter STARTS " ) ;
25      delay 500 ;    //(1)
26      println("qatraceremitter emits alarm(fire)" ) ;
27      emit alarm : alarm(fire) ;
28      delay 500 ;    //(2)
29      println("qatraceremitter emits usercmd(hello)" ) ;
30      emit usercmd : usercmd(hello) ;
31      println( "qaeventemitter ENDS" )
32   ]
33 }
```

Listing 1.12. eventTracer.qa

The output is:

```

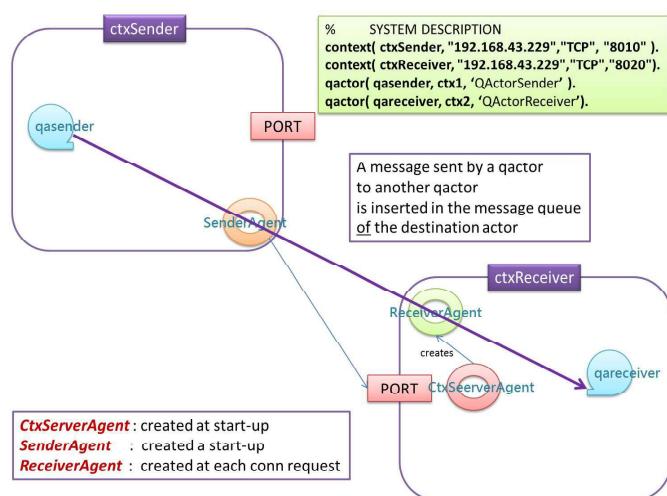
1 /*
2 OUTPUT
3 -----
4 "qatraceremitter STARTS "
5 "qaevtracer starts"
6 "noevent"
7 "qaevtracer starts"
8 "noevent"
9 "qatraceremitter emits alarm(fire)"
10 >>> evh      (defaultState, TG=01:00:00)      || msg(alarm,event,qatraceremitter_ctrl,none,alarm(fire),5)
11 "qaevtracer starts"
12 qaevtracer(alarm,qatraceremitter_ctrl,alarm(fire))
13 "qaevtracer starts"
14 "noevent"
15 "qatraceremitter emits usercmd(hello)"
16 >>> evh      (defaultState, TG=01:00:00)      || msg(usercmd,event,qatraceremitter_ctrl,none,usercmd(hello),7)
17 "qaeventemitter ENDS"
18 "qaevtracer starts"
```

Listing 1.13. eventTracer.qa

1.15 The qa-infrastructure

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

- a high-level description of the Contexts and of the actors that work in each of them. This *Context Knowledge Base* or simply *ContextKB* is replicated for each context and automatically consulted at the system start-up.
Example: for a Context named *aCtx*, the description is written in a file named *aCtx.pl* within the directory *srcMore/it.unibo.aCtx*. In the same directory is generated also the file, named *sysRules.pl*, that includes the *tuProlog* rules used by the *qa-infrastructure* (see Subsection ??).
- an Akka actor-system and a Akka *SystemCreationActor* that work as the 'father' of all the other actors in that *Context*.



More specifically, the *SystemCreationActor* 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 through 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*.

1.16 Example: a distributed producer-consumer system

```

1 System prodConsDistributed
2 Dispatch info : info(X)
3
4 Context ctxBasicProd ip [ host="localhost" port=8079 ]
5 Context ctxBasicCons ip [ host="localhost" port=8089 ] -g green
6
7 QActor producer1 context ctxBasicProd
8 Rules{
```