

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

Introduction to QActors (2018)	1
<i>Antonio Natali</i>	
1 Introduction to QActors	5
1.1 Overview	5
1.2 Example: the 'hello world'	7
1.3 Example: no-input transitions	8
1.4 Example: repeat/resume plans	8
1.5 How a Plan/State works	9
2 From specifications to execution	10
2.1 The custom language qa	10
2.2 The qa software factory	11
2.2.1 Example of generated files	12
3 The actor's WorldTheory	12
3.0.2 Facts about the state	12
3.0.3 Guarded actions	12
3.0.4 The <code>demo</code> operator	13
3.0.5 Built-in Prolog rules	14
3.0.6 Rules at model level	14
3.0.7 Example: using built-in tuProlog rules	15
3.1 Loading and using a user-defined theory	16
3.1.1 The initialization directive	16
3.1.2 On backtracking	17
3.2 Introspection	17
4 Actions (built-in and user-defined)	19
4.0.1 Action results	19
4.0.2 Built-in and User-defined actions	19
4.0.3 The operation <code>javaRun</code>	19
4.0.4 The operation <code>javaOp</code>	20
4.0.5 The operation <code>nodeOp</code>	20
4.1 Asynchronous actions	20
4.1.1 A base-actor for asynchronous action implementation	21
4.1.2 <code>onReceive</code>	21
4.1.3 <code>endActionInternal</code>	21
4.1.4 <code>ActionActorFibonacci</code>	22
5 Messages and events	23
5.1 Messages	23
5.1.1 Sending/Receiving messages	24
5.1.2 Example: a producer-consumer system	24
5.2 State transitions	26
5.2.1 Switch part	26
5.3 Guarded transitions	27
5.3.1 Example: action after a self-message	27
5.4 Events	28

5.4.1	Example: event-based behavior	29
5.5	Event handlers and event-driven behaviour	30
5.6	The <code>qa-infrastructure</code>	32
5.7	Example: a distributed producer-consumer system	33
6	Automated Testing	34
6.1	Types of testing	34
6.2	Testing of the producer-consumer system	35
7	Built-in GUI interfaces	37
7.1	Built-in web server	37
7.1.1	Input string prefix <code>m-</code>	38
7.1.2	Input string prefix <code>i-</code>	38
7.1.3	Input string without special prefix	38
7.2	The local actor-GUI interface	39
8	Components written in other languages	40
8.1	Using Node.js	40
8.2	The operation <code>runNodeJs</code>	41
8.2.1	The <code>NodeJs</code> client	41
8.2.2	The result	42
9	Reply to a sender	44
10	Dynamic systems: the flag <code>-standalone</code>	46
10.1	Example of a pivot	46
10.1.1	Rules to show the current system configuration	46
10.2	Adding a new context	47
10.3	Adding another context	48
10.4	Dynamic extension of the configuration knowledge	49
10.5	Removing a node	49
11	The <code>MqttUtils</code>	50
11.1	MQTT support	50
11.2	MQTT built-in operations	52
11.3	Example: a producer-consumer system	52
11.3.1	Running the system	54
12	Application-specific actions	56
13	A Custom GUI	57
13.0.2	Using the <code>uniboEnvBaseAwt</code> framework	58
13.0.3	Observable POJO objects	58
13.0.4	Environment interfaces	59
13.0.5	Command interfaces	59
13.0.6	Adding a command panel	60
13.0.7	Adding an input panel	60
13.0.8	Adding a new panel	60
13.0.9	Built-in GUI	61
13.0.10	Built-in commands	61
13.1	A Command interpreter	62
13.2	Reactive actions	64
13.3	Built-in Actions	66
13.3.1	The syntax of a Plan	66
13.4	The operator <code>actorOp</code>	67

14 File watcher	69
-----------------------	----

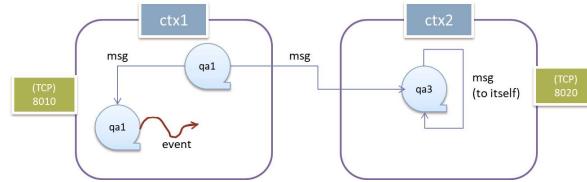
1 Introduction to QActors

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.

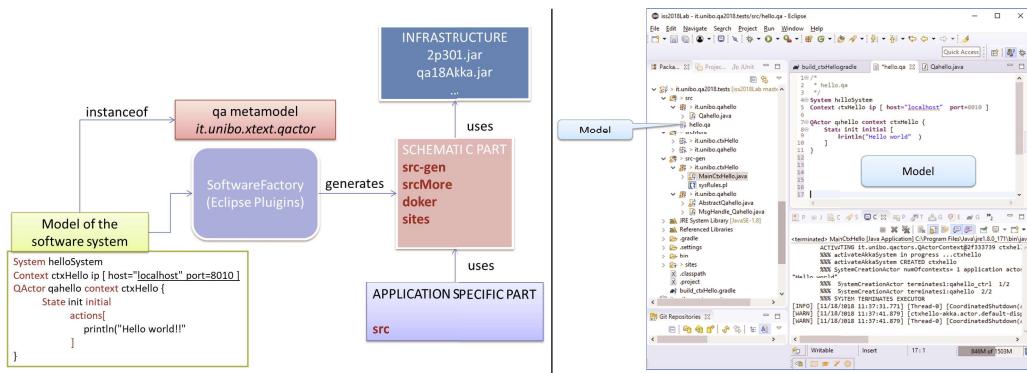
1.1 Overview

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](#).

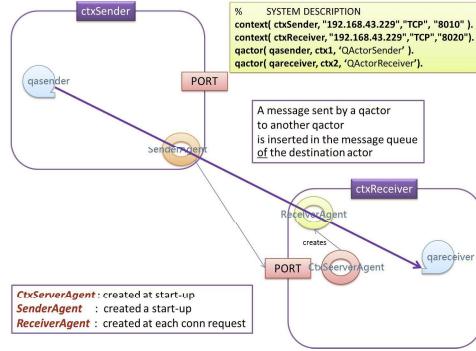


- A [QA-System](#) can be specified in textual form by means of the *QActor* language/metamodel. The *QActor* language is associated to a [software factory](#) (Subsection 2.2) that automatically generates the proper system *run-time support* so to allow Application designers to focus on application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

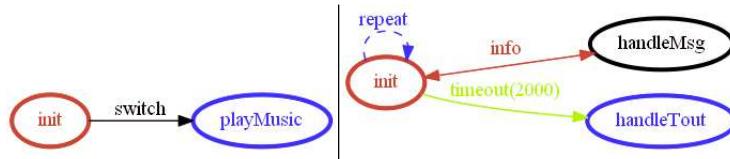


- The [configuration](#) of a [QA-System](#) is explicitly represented by a set of 'facts' written in [tuProlog](#) syntax (Subsection 5.6) replicated in each Context (Context Knowledge Base or simply [ContextKB](#)). A [QA-System](#) can be configured in a static or in a dynamic way. In case of [dynamic configuration](#), the knowledge about the configuration is dynamically updated in each [Context](#) of the system (see Subsection 10).
- The [start-up](#) of a distributed *QActor* system (i.e. a system made of two or more Contexts) is handled by the run-time support. In particular, the Application code (i.e. the code written into the actors) begins to run only when all the Contexts are activated.

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



- To deliver information among the Contexts, the **QA-Infrastructure** can use pairwise TCP connections between the Contexts or a **MQTT** broker. The choice is up to the Application designer.
- Each **QActor** behaves as a (Moore's) *Finite State Automaton* (**FSM**). While in a state, a **QActor** can execute both **synchronous** and asynchronous actions. An **asynchronous** action terminates immediately and emits an event (see Subsection 4.1) when it terminates.
- A **state-transition** from a state S1 to another state S2 can be done (triggered) only when all the actions activated in S1 are terminated. A state-transition can be triggered by a message or by an event or as a choice of the **QActor** (no-input transition).



- A state-transition can be associated to a boolean condition (**guard**) and can be triggered only when the guard is true (see Subsection 3.0.3). When two or more transitions are possible, the systems checks the possibility to trigger one of them by following the order in which they are written. If no transition can be triggered, the **QActor** remains in the state until the occurrence of some message or event that allows a transition.
- A **QActor** is able to execute a set of **pre-defined actions** (see Subsection 13.3, Subsection 3.0.5). 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[javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveForward', 'arg': 60000 }");
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 5.6).

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

The example shows also an older version in which:

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

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

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

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

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

State transitions can be performed when:

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

1.3 Example: no-input transitions

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

```
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     ]
```

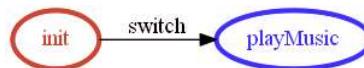
Listing 1.2. noinputTansition.qa

switchTo is a predefined operation of the *QActor* language (see Subsection 2.1).

sound is a *QActor* built-in action (see Subsection 13.3, Subsection 3.0.5) that ends after 2 seconds.

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

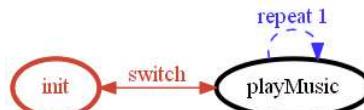
The *QActor* performs a state-switch from its **init** state to the **playMusic** state with a **no-input transition** **switchTo**. The behaviour of the actor can be represented by a state diagram like that shown in the following picture:



This state diagram is automatically generated¹ by the *QActor* software factory (see Subsection 2.2).

1.4 Example: repeat/resume plans

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



```
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)
```

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

```

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

```

Listing 1.3. repeatResume.qa

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

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

1.5 How a Plan/State works

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

1. the *QActor* executes, in sequential way, all the actions specified in the *Plan*. Each action must be defined as an `algorithm`, i.e. it must terminate;
2. if the *State/Plan* specification ends with the sentence `finally repeatPlan N` (*N* natural number $N \geq 1$), 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 5.2;
4. when a state has terminated its work (i.e. its actions, transition and repetition), it can resume the execution of its *oldState*. This happens if the `resumeLastPlan` keyword is inserted after the state name. Otherwise, the state is considered a `termination state` and the *QActor* does not perform any other work.

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

```

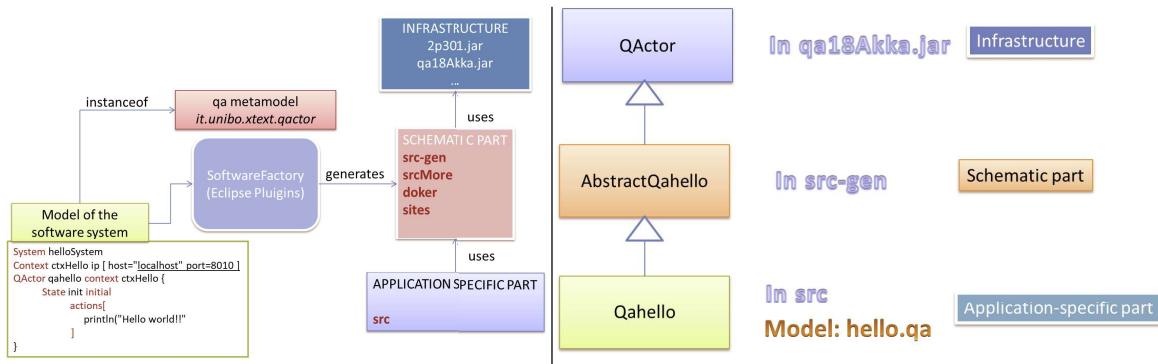
1 "player STARTS"
2 "player ENDS"
3 playSomeMusic
4 playSomeMusic
5 "player STARTS"
6 "player ENDS"
7 playSomeMusic
8 playSomeMusic

```

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

2 From specifications to execution

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



2.1 The custom language qa

The *qa* language is a custom 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 ;

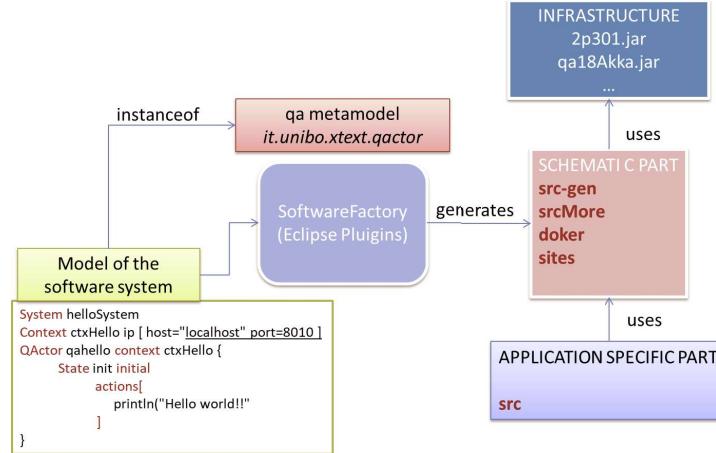
```

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.

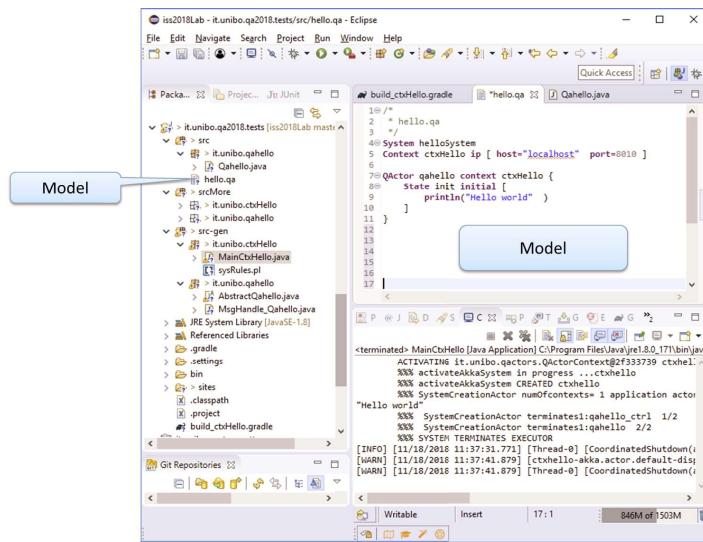
² The *qa* language/metamodel is defined in the project *it.unibo.xtext.qactor*.

2.2 The qa software factory

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



For each `QActor` and for each `Context`, the `QActor` software factory generates (Java) code in the directories `src-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.

2.2.1 Example of generated files .

In the case of the example of Subsection 1.2:

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

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

3 The actor's WorldTheory

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

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

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

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

3.0.2 Facts about the state .

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

<code>actorobj/1</code>	memorizes a reference to the 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 3.0.4)

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

3.0.3 Guarded actions .

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

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

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

Let us consider the following example;

```
1 System naiveGuardedActions
2 Context ctxNaiveGuardedAction ip [ host="localhost" port=8037 ]
3 QActor qanaiiveguarded 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
20 fib(6,8)
21 */
```

Listing 1.4. naiveGuardedActions.qa

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

3.0.4 The demo operator .

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

```
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(qatuqademoexample_ctrl)
23 qademoexample(actorobj(qatuqademoexample_ctrl))
24 fibo(6,8)
25 fibo(6,8)
26 failure
27 "qademoexample ENDS"
28 */
```

Listing 1.5. demoExample.qa

3.0.5 Built-in Prolog rules .

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

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

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

3.0.6 Rules at model level .

Sometimes can be useful to express tuProlog facts and rules directly in the model specification, especially for configuration or action-selection purposes. The `Rules` option within a *QActor* allows us to define facts and rules by using a [subset](#) of the tuProlog syntax⁵

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

```

1 System rulesInModel
2 Context ctxRulesInModel ip [ host="localhost" port=8059 ]
3 QActor rulebasedactor context ctxRulesInModel {
4   Rules{
5     music(1, './audio/tada2.wav',2000).
6     music(2,'./audio/any_commander3.wav',3000).
7     // music(3,'./audio/computer_complex3.wav',3000).
8     // music(4,'./audio/illogical_most2.wav',2000).
9     // music(5,'./audio/computer_process_info4.wav',4000).
10    // music(6,'./audio/music_interlude20.wav',3000).
11    // music(7,'./audio/music_dramatic20.wav',3000).
12  }
13  State init initial[
14    [ !? actorobj(X) ] println( myName(X) )
15  ]
16  switchTo work
17  State work [
18    [ ?? music(N,F,T) ] sound time(T) file(F) else endPlan "bye"
19  ]
20  finally repeatPlan
21 }
```

Listing 1.6. rulesInModel.qa

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

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

3.0.7 Example: using built-in tuProlog rules .

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

```
1  /*
2   * builtinPrologExample.qa
3   */
4  System builtinPrologExample
5  Context ctxBuiltInExample ip [ host="localhost" port=8079 ]
6
7 QActor qabuiltinxexample context ctxBuiltInExample{
8     Rules{
9         r1 :- assign(x,10),getVal(x,V1),eval(plus,V1,3,RV),actorPrintln(r1(RV)).
10        r2 :- distance(X),actorPrintln(r2(X) ).
11        r3 :- assert( distance(200) ).  

12    }
13    Plan init normal
14    [
15        println("execute built-in rules" );
16        demo assign(x,30) ;
17        demo getVal(x,V) ;
18        [ ?? goalResult(getVal(x,V)) ] println( valueOfx(V) ) ;
19        println("execute a user-defined rule r1 (in Rules)" );
20        demo r1 ;
21        println("add a distance/1 fact" );
22        addRule distance(100);
23        println("execute a user-defined rule r2 that refers to distance/1" );
24        demo r2 ;
25        println("execute a user-defined rule r3 that adds another distance/1" );
26        demo r3 ;
27        println("conditional execution using distance/1 facts as guards" );
28        [ ?? distance(D) ] println( distance(D) ) else println( nodistance );
29        [ ?? distance(D) ] println( distance(D) ) else println( nodistance );
30        [ ?? distance(D) ] println( distance(D) ) else println( nodistance );
31        println("remove the rule r1" );
32        removeRule r1 ;
33        println("attempt to run the removed rule r1" );
34        demo r1 ;
35        println("END" )
36    ]
}
```

Listing 1.7. builtinPrologExample.qa

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

```
1 "execute built-in rules"
2 valueOfx(30)
3 "execute a user-defined rule r1 (in Rules)"
4 r1(13)
5 "add a distance/1 fact"
6 "execute a user-defined rule r2 that refers to distance/1"
7 r2(100)
8 "execute a user-defined rule r3 that adds another distance/1"
9 "conditional execution using distance/1 facts as guards"
10 distance(100)
11 distance(200)
12 nodistance
13 "remove the rule r1"
14 "attempt to run the removed rule r1"
15 "END"
16 */
```

Listing 1.8. builtinPrologExample.qa

3.1 Loading and using a user-defined theory

The *WorldTheory* of an actor can be extended by the application designer by using the directive⁶ `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`) :

```
1  /*
2   * atheoryUsage.qa
3   */
4 System atheoryUsage
5 Context ctxTheoryUsage ip [ host="localhost" port=8049 ]
6
7 QActor qatheyrouser context ctxTheoryUsage{
8     Plan init normal
9     [   println( "qatheyrouser STARTS" ) ;
10    /*0*/  demo consult("./src/it/unibo/qatheyrouser/aTheory.pl")
11    ]
12    switchTo accessdata
13
14    Plan accessdata resumeLastPlan
15    [   println( "-----" ) ;
16    /*1*/  [ !? data(S,N,V) ] println( data(S,N,V) ) ;
17    /*2*/  [ !? validDistance(N,V) ] println( validDistance(N,V) ) ;
18    /*3*/  demo nearDistance(N,V) ;
19    /*4*/  [ !? goalResult(nearDistance(N,V)) ] println( warning(N,V) ) ;
20    /*5*/  demo nears(D) ;
21    /*6*/  [ !? goalResult(G) ] println( list(G) )
22    ]
23    finally repeatPlan 1
24 }
```

Listing 1.9. aTheoryUsage.qa

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

```
1  /* =====
2  aTheory.pl
3  ===== */
4  data(sonar, 1, 10).
5  data(sonar, 2, 20).
6  data(sonar, 3, 30).
7  data(sonar, 4, 40).
8
9  validDistance( N,V ) :- data(sonar, N, V), V>10, V<50.
10 nearDistance( N,X ) :- validDistance( N,X ), X < 40.
11 nears( D ) :- findall( d( N,V ), nearDistance(N,V), D).
12
13 aTheoryInit :- output("initializing the aTheory ...").
14 :- initialization(aTheoryInit).
```

Listing 1.10. aTheory.pl

3.1.1 The initialization directive .

The following directive:

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

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

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

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

```

1  /*
2  OUTPUT
3  -----
4  "qatheoryuser STARTS"
5  initializing the aTheory ...
6  -----
7  data(sonar,1,10)
8  validDistance(2,20)
9  warning(2,20)
10 list(nears([d(2,20),d(3,30)]))
11 -----
12 data(sonar,1,10)
13 validDistance(2,20)
14 warning(2,20)
15 list(nears([d(2,20),d(3,30)]))
16 */

```

Listing 1.11. aTheoryUsage.qa

3.1.2 On backtracking .

The output shows that the rules `validDistance` and `nearDistance` exploit *backtracking* in order to return the first valid instance (2), while the repetition of the plan `accessdata` returns always the same data⁷. 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.

3.2 Introspection

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

```

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

```

Listing 1.12. introspection.qa

```

1 getTheContexts(Ctx,CTXS) :-
2   Ctx <- solvegoal("getTheContexts(X)","X") returns CTXS .
3 getTheActors(Ctx,ACTORS) :-
4   Ctx <- solvegoal("getTheActors(X)","X") returns ACTORS.
5
6 /**
7 -----Show system configuration
8 -----*/

```

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

```

10  /*
11  showSystemConfiguration :-
12      actorPrintln('-----'),
13      actorObj(A),
14      A <- getQActorContext returns Ctx,
15      %% Ctx <- getName returns CtxName, actorPrintln( CtxName ),
16      getTheContexts(Ctx,CTXS),
17      actorPrintln('CONTEXTS IN THE SYSTEM:'),
18      showElements(CTXS),
19      actorPrintln('ACTORS IN THE SYSTEM:'),
20      getTheActors(Ctx,ACTORS),
21      showElements(ACTORS),
22      actorPrintln('-----').
23
24 showElements(ElementListString) :-
25     text_term( ElementListString, ElementList ),
26     %% actorPrintln( list(ElementList) ),
27     showListOfElements(ElementList).
28 showListOfElements([]).
29 showListOfElements([C|R]) :-
30     actorPrintln( C ),
31     showElements(R).

```

Listing 1.13. curConfigTheory.pl

These rules are currently included as built-in Prolog rules both in the `actorKb` (file `WordTheory.pl`) and in the context rules (file `sysRules.pl`). Moreover, the *QActor* language provides the built-in operation `showSystemConfiguration`. For an example see Subsection 10.1.1.

4 Actions (built-in and user-defined)

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

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

As a general statement, we can say that⁸:

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

4.0.1 Action results .

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

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

4.0.2 Built-in and User-defined actions .

Each *QActor* is able to execute:

- a set of built-in operations, defined by the **qa** language as key-words. Examples are: `println`, `sound`, `forward`, `emit`, `addRule`, `javaRun`, `javaOp`, `nodeOp`, etc.
- a set of actions implemented in Java by the *QActor* System designer in the class **QActor** of the library **qaAkka18.jar**. Examples are: `debugStep`, `publish`, `subscribe`, `sendPost`, `doCoapPost`, etc.. This kind of actions can be executed by means of the built-in operation `javaOp` (see Subsection 4.0.4).
- a set of actions implemented in some Java class written by the *Application designer*. These actions can be executed by means of the built-in action `javaRun` (see Subsection 4.0.3).
- a set of actions written by the *Application designer* in **tuProlog** (see Subsection 3.0.5, Subsection 3.0.7).

4.0.3 The operation `javaRun` .

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

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

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

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

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

4.0.4 The operation `javaOp`.

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

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

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

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

The set of action currently defined in the in the `QActor` class are reported in Subsection 13.3.

4.0.5 The operation `nodeOp`.

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

Let us consider the following sentence:

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

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

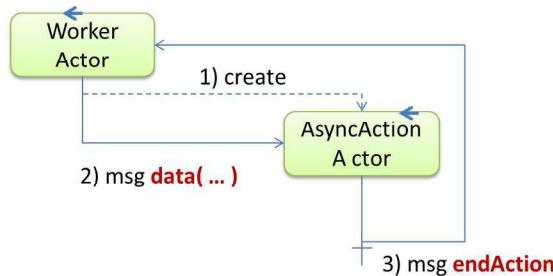
4.1 Asynchronous actions

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

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

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

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



Let us introduce here a possible implementation of asynchronous actions.

4.1.1 A base-actor for asynchronous action implementation .

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

```
1 public abstract class ActionObservableGenericActor<T> extends UntypedAbstractActor{
2     protected String name = "noname";
3     protected IOutputEnvView outEnvView = null;
4     protected String terminationEvId = "endAction";
5     protected long tStart = 0;
6     protected long durationMillis = -1;
7     protected QActorContext ctx = null;
8     protected T result;
9     protected QActor myactor = null;
10
11    public ActionObservableGenericActor(String name, QActor qa, IOutputEnvView outEnvView) {
12        this.name      = name.trim();
13        this.myactor   = qa;
14        this.ctx       = (qa != null) ? qa.getQActorContext() : null;
15        this.outEnvView = outEnvView;
16    }
17    /*
18     * TO BE DEFINED BY THE APPLICATION DESIGNER
19     */
20    protected abstract void execTheAction(Struct actionInput) throws Exception;
21    protected abstract T endOfAction() throws Exception;
```

Listing 1.14. ActionObservableGenericActor<T>

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

4.1.2 onReceive .

```
1 public void onReceive(Object msg) {
2     // System.out.println(" %% ActionObservableGenericActor onReceive: " + msg + " from " + getSender().path() );
3     try{
4         Struct msgt = (Struct) Term.createTerm(msg.toString()); //check syntax data( ... )
5         tStart = Calendar.getInstance().getTimeInMillis();
6         execTheAction( msgt );
7         result = endActionInternal();
8     }catch(Exception e){
9         System.out.println(" %% ActionObservableGenericActor onReceive ERROR " + e.getMessage());
10        //stop the actor. we could propagate to the parent (SystemCreationActor)
11        getContext().stop( getSelf() ); //LOCAL RECOVERY POLICY
12    }
13 }
```

Listing 1.15. onReceive<T>

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

4.1.3 endActionInternal .

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

8 }

Listing 1.16. endActionInternal

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

```
1 System asynchFibo
2 Dispatch endAction : endAction(A,R)
3 Dispatch data : data(X)
4
5 Context ctxAsynchFibo ip[ host="localhost" port=8018 ] // -g cyan
6
7 QActor asynchcallerfibo context ctxAsynchFibo{
8     Plan init normal[
9         javaOp "doAsynAction(\"asynchfibo\",
10             \"it.unibo.actionAsynch.ActionActorFibonacci\", \"30\")"
11     ]
12     switchTo handleActionEnd
13
14     State handleActionEnd [ println("WAIT FOR ASYNCH ACTION TERMINATION") ]
15     transition stopAfter 30000
16         whenMsg endAction -> callActionResult
17
18     Plan callActionResult resumeLastPlan[
19         //onMsg endAction : endAction(actionFiboAsych,V) -> println( V ) ;
20         onMsg endAction : endAction(A,V) -> println( endAction(A,V) )
21     ]
22 }
23 /*
24 * OUTPUT
25 "WAIT FOR ASYNCH ACTION TERMINATION"
26 endaction(asynchfibo,fibo(30,832040,timemsec(58)))
27 */
```

Listing 1.17. asynchFibo.qa

4.1.4 ActionActorFibonacci .

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

```
1 package it.unibo.actionAsynch;
2 import alice.tuprolog.Struct;
3 import it.unibo.is.interfaces.IOutputEnvView;
4 import it.unibo.qactors.action.ActionObservableGenericActor;
5 import it.unibo.qactors.akka.QActor;
6
7 public class ActionActorFibonacci extends ActionObservableGenericActor<String> {
8     private long myresult = 0;
9     private int n = 0;
10    public ActionActorFibonacci(String name, QActor actor, IOutputEnvView outEnvView) {
11        super(name, actor, outEnvView);
12    }
13    @Override
14    public void execTheAction(Struct actionInput) throws Exception {
15        //actionInput : data( n )
16        n = Integer.parseInt( actionInput.getArg(0).toString() );
17        myresult = fibonacci( n );
18        // throw new Exception("simulateFault"); // (1)
19    }
20    protected long fibonacci( int n ){
21        if( n == 1 || n == 2 ) return 1;
22        else return fibonacci(n-1) + fibonacci(n-2);
23    }
24    @Override
25    protected String endOfAction() throws Exception {
26        return "fibo(\"+n\",""+this.myresult+"",timemsec("durationMillis")+"")";
27    }
```

Listing 1.18. ActionActorFibonacci

5 Messages and events

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

```

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

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

```

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

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

```

1 System basicProdCons
2 Dispatch info : info(X)
3 Dispatch autoMsg : autoMsg(X)
```

Listing 1.19. An example of message declaration

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

5.1 Messages

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

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

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

At application-level, we say that a *QActor* works according to a **message-based** behaviour, while at the lower level (in the **qa-infrastructure**, see Subsection 5.6) 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 example¹⁰:

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

At low level (in the qa-infrastructure, see Subsection 5.6) 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 3.0.3) to allow conditional evaluation of *Actions*.

5.1.1 Sending/Receiving messages .

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

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

```
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 ;
```

5.1.2 Example: a producer-consumer system .

As an example of a message-based transition, let us introduce a very simple producer-consumer system¹¹, in which the producer sends two times a *dispatch* to the consumer:

```
1 System basicProdCons
2 Dispatch info : info(X)
3 Dispatch autoMsg : autoMsg(X)
4
5 Context ctxBasicProdCons ip [ host="localhost" port=8019 ]
6 QActor producer context ctxBasicProdCons{
7     Rules{
8         item(1).
9         item(2).
10    }
11    State init initial [
12        [ !? item(X) ] println( producerSending( item(X) ) );
13        [ !? item(X) ] addRule produced( item(X) );
14        [ ?? item(X) ] selfMsg autoMsg : autoMsg( item(X) );
15        delay 500
16    ]
17    transition stopAfter 6000
```

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

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

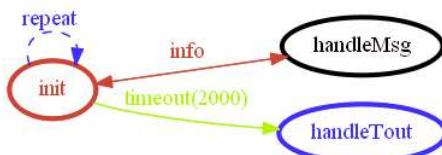
```

18     whenMsg autoMsg -> handleAutoMsg
19     finally repeatPlan
20
21 Plan handleAutoMsg [
22     printCurrentMessage;
23     onMsg autoMsg : autoMsg(M) -> forward consumer -m info : info( M )
24 ]
25 transition whenTime 20 -> init
26     whenMsg autoMsg -> handleAutoMsg
27 }
28 QActor consumer context ctxBasicProdCons{

```

Listing 1.20. basicProdCons.qa

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



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

The output is:

```

1 State init normal [ println( consumer(waiting) ) ]
2 transition whenTime 6000 -> handleTout
3     whenMsg info -> handleMsg
4 finally repeatPlan
5 State handleMsg resumeLastPlan [
6     printCurrentMessage ;
7     onMsg info : info(X) -> addRule consumed( X );
8     onMsg info : info(X) -> println( consumerReceiving(X) )
9 ]
10 State handleTout [ println( consumerTout ) ]
11 }
12 /*
13 OUTPUT
14 -----
15 producerSending(item(1))
16 consumer(waiting)
17 -----
18 consumer_ctrl currentMessage=msg(info,dispatch,producer_ctrl,consumer,info(item(1)),1)
19 -----
20 consumerReceiving(item(1))
21 consumer(waiting)
22 producerSending(item(2))
23 -----
24 consumer_ctrl currentMessage=msg(info,dispatch,producer_ctrl,consumer,info(item(2)),2)
25 -----
26 consumerReceiving(item(2))
27 consumer(waiting)
28 consumerTout
29 */

```

Listing 1.21. basicProdCons.qa

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

5.2 State transitions

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

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

Thus, a state transition sentence can start:

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

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

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

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

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

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

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

The meaning is:

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

5.2.1 Switch part .

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

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

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

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

```

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.

5.3 Guarded transitions

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

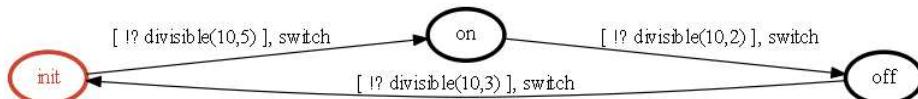
```

1 System guardedTransitions
2 Context ctxGuardedTransitions ip [ host="localhost" port=8037 ]
3 QActor qaguardedtrans context ctxGuardedTransitions {
4     Plan init normal
5         [ _println("qaguardedtrans STARTS" ) ]
6         switchTo [ !? divisible(10,5) ] on
7     Plan on
8         [ _println( on ) ]
9         switchTo [ !? divisible(10,2) ] off
10    Plan off
11        [ _println( off ) ]
12        switchTo [ !? divisible(10,3) ] init //possible loop!
13    }
14 /*
15 OUTPUT
16 -----
17 "qaguardedtrans STARTS"
18 on
19 off
*/

```

Listing 1.22. `guardedTransitions.qa`

The generated state diagram is:



More interesting examples will be given in the following.

5.3.1 Example: action after a self-message .

A more interesting example is given in the next section

```

1 System msgToSelf
2 Dispatch execute : code( CODE )
3 Dispatch eval   : code( CODE )
4
5 Context ctxMsgToSelf ip [ host="localhost" port=8019 ]

```

```

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

```

Listing 1.23. msgToSelf.qa

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

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

5.4 Events

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

The occurrence of an event can put in execution some code devoted to the management of that event. We qualify this kind of behaviour as `event-driven` behaviour, since the event 'forces' the execution of code (see Subsection 5.5).

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

At application level, an event is denoted by specifying the key-word `Event`, a name and a payload. For example¹²:

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

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

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

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

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

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

```
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 ;
```

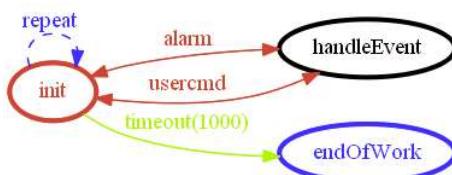
5.4.1 Example: event-based behavior .

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

```
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 {
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 1.24. 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 1.25. basicEvents.qa

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

5.5 Event handlers and event-driven behaviour

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

```

1 EventHandler :
2   "EventHandler" name=ID ( "for" events += [Event] ( "," events += [Event] )* )?
3   ( print ?= "-print" )?
4   ( "{" body = EventHandlerBody "}" )?
5   ";"?
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`¹³:

```

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

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

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

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

```

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
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 1.26. eventTracer.qa

The output is:

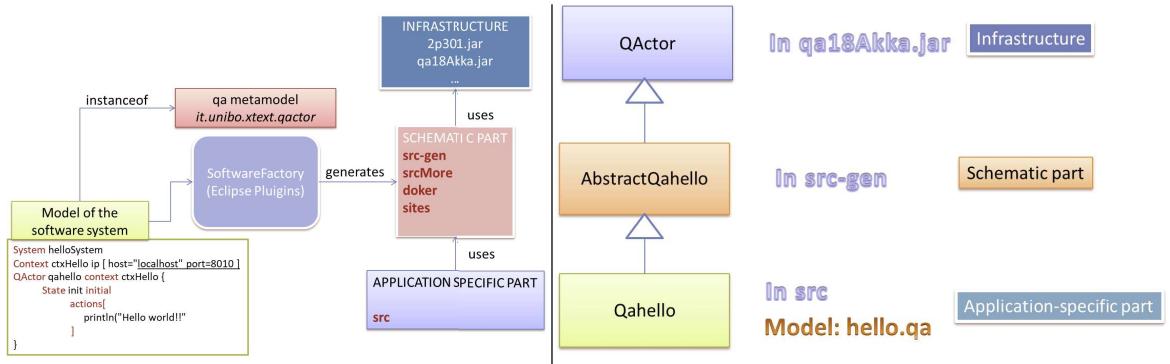
```

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 1.27. eventTracer.qa

5.6 The qa-infrastructure

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



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

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

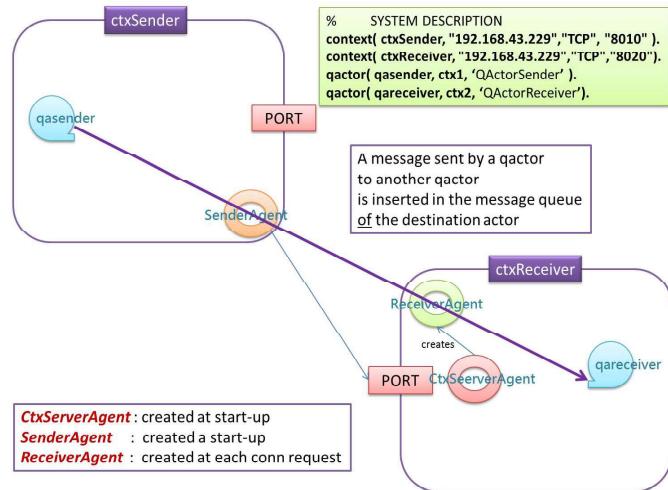
For each *Context*, the *QActor* software factory generates:

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

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

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

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



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

5.7 Example: a distributed producer-consumer system

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

```

1 System prodConsDistributed
2 Dispatch info : info(X)
3 Dispatch autoMsg : autoMsg(X)
4
5 Context ctxBasicProdDistr ip [ host="localhost" port=8079 ]
6 Context ctxBasicConsDistr ip [ host="localhost" port=8089 ]
7
8 QActor producerdistr context ctxBasicProdDistr{
9 Rules{
10   item(1).
11   item(2).
12 }
13 State init initial [ println( producerdistr(waiting) ) ]
14 switchTo doWork
15
16 State doWork [
17   [ !? item(X) ] println( producer(sends(X)) ) ;
18   [ !? item(X) ] addRule produced( item(X) );
19   [ ?? item(X) ] selfMsg autoMsg : autoMsg( item(X) )
20 ]
21 transition stopAfter 3000
22   whenMsg autoMsg -> handleAutoMsg
23 finally repeatPlan

```

Listing 1.28. prodConsDistributed.qa

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

6 Automated Testing

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

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

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

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

6.1 Types of testing

The main types of testing can be summarized as follows:

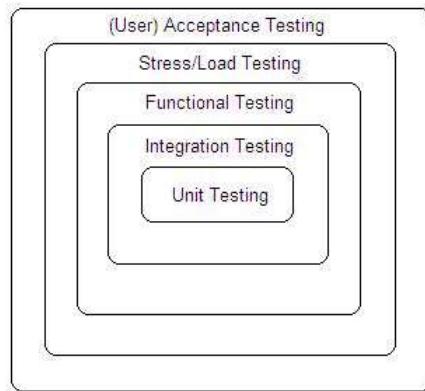


Fig. 1. Test types

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

With reference to our motto:

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

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

6.2 Testing of the producer-consumer system

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

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

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

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

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

```

42     solProd = engineProd.solveNext();
43     solCons = engineCons.solveNext();
44     checkSolution(solProd, solCons);
45     itemProd = solProd.getVarValue("ITEM").toString();
46     itemCons = solCons.getVarValue("ITEM").toString();
47     System.out.println("test-2 itemProd="+itemProd + " itemCons="+ itemCons );
48     assertTrue( "test", itemProd.equals(itemCons) );
49 }
50 boolean b1 = engineProd.hasOpenAlternatives();
51 boolean b2 = engineProd.hasOpenAlternatives();
52 if( (b1 && ! b2) || (! b1 && b2) ) fail("test: DIFFERENT NUMBER OF ITEMS ");
53 } catch (Exception e) {
54     System.out.println("test ERROR: "+ e.getMessage());
55     fail("test ERROR: "+ e.getMessage());
56 }
57 }
58 }
59 }
60 }
```

Listing 1.29. TestProdCons.java

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

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

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

7 Built-in GUI interfaces

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

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

7.1 Built-in web server

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



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

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

This interface emits the following events:

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

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

When the `-httpserver` flag is set, a simple `HTTPserver`¹⁴ is created and started during the Context configuration phase. This server is based on the `WebSocket` technology¹⁵ and answers to HTTP requests on

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

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

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

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

7.1.1 Input string prefix `m-` .

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

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

The `QActorHttpServer` sends the message to the `RECEIVER` actor.

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

7.1.2 Input string prefix `i-` .

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

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

where `MOVE` can take one of the following forms:

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

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

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

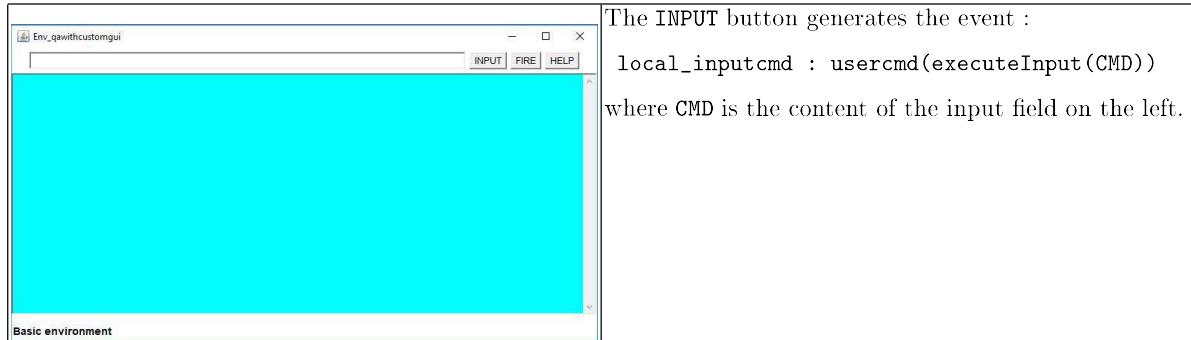
7.1.3 Input string without special prefix .

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

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

7.2 The local actor-GUI interface

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



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

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

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

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

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

8 Components written in other languages

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

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

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

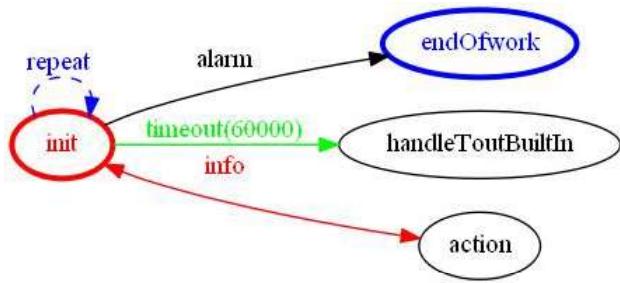
8.1 Using Node.js

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

```
1 System jsToQa
2 Event alarm : alarm(X)
3 Dispatch info : info(X)
4
5 Context ctxJsToQa ip[ host="localhost" port=8031]
6 EventHandler evh for alarm -print;
7 /*
8  * This actor activates a NodeJs process that sends messages to the qareceiver
9  */
10 QActor qajaactoivator context ctxJsToQa {
11     Plan init normal [
12         println("qajaactoivator START") ;
13         nodeOp "./nodejsCode/TcpClientToQaNode.js localhost 8031" -o ;
14     //    actorOp runNodeJs("./nodejsCode/TcpClientToQaNode.js localhost 8031", "true") ;
15     // [? actorOpDone(OP,R)] println( rrr(R) );
16         println("qajaactoivator END")
17     ]
18 }
19
20 /*
21  * This actor handles message sent by the NodeJs process by distinguishing between
22  * messages (dispatch info : info(X)) and events (alarm : alarm(X)).
23  */
24 QActor qareceiver context ctxJsToQa {
25     Plan init normal [
26         println("qareceiver WAITS")
27     ]
28     transition stopAfter 60000
29         whenEvent alarm -> endOfwork ,
30         whenMsg info : info(X) do printCurrentMessage
31     finally repeatPlan
32
33     Plan endOfwork [
34         onEvent alarm : alarm(X) -> println( endOfwork(X) )
35     ]
36
37 }
```

Listing 1.30. jsToQa.qa

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



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

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

8.2 The operation `runNodeJs`

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

```

1 protected String runNodeJs(String prog, String showOutput){
2     try {
3         String cmd = "node "+ prog;
4         java.lang.Process nodeExecutor = runtimeEnvironment.exec("node "+prog);
5         if( showOutput.equals("false") ) return "";
6         InputStream nodeInputStream = nodeExecutor.getInputStream();
7         return getOutput(prog,nodeInputStream);
8     } catch (Exception e) {
9         println("      " + prog+ " WARNING >" + e.getMessage() );
10    }
11 }
12

```

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

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

8.2.1 The `NodeJs` client .

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

```

1 /**
2 * =====
3 * TcpClientToQaNode.js
4 * =====
5 */
6 var net = require('net');
7 var host = process.argv[2];
8 var port = Number(process.argv[3]); //23 for telnet
9
10 console.log('connect to ' + host + ":" + port);
11 var socket = net.connect({ port: port, host: host });
12 console.log('connect socket allowHalfOpen= ' + socket.allowHalfOpen );
13 socket.setEncoding('utf8');
14
15 // when receive data back, print to console
16 socket.on('data',function(data) {
17     console.log(data);
18 });
19 // when server closed

```

```

20 | socket.on('close',function() {
21 |     console.log('connection is closed');
22 | });
23 | socket.on('end',function() {
24 |     console.log('connection is ended');
25 | });
26 | /*
27 | * TERMINATION
28 | */
29 | process.on('exit', function(code){
30 |     console.log("Exiting code= " + code );
31 | });
32 | //See https://coderwall.com/p/4yis4w/node-js-uncought-exceptions
33 | process.on('uncaughtException', function (err) {
34 | cursor.reset().fg.yellow();
35 |     console.error('got uncaught exception:', err.message);
36 | cursor.reset();

```

Listing 1.31. TcpClientToQaNode.js: connect

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

```

1 =====
2 */
3 function sendMsg( msg ){
4     try{
5         socket.write(msg+"\n");
6     }catch(e){
7         console.log(" ----- EVENT " + e );
8     }
9 }
10 function sendMsgAfterTime( msg, time ){
11     setTimeout(function(){
12         //println("SENDING..." + msg );
13         sendMsg( msg );
14     });

```

Listing 1.32. TcpClientToQaNode.js: utilities

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

```

1 =====
2 */
3 var msgNum=1;
4 sendMsgAfterTime("msg(info,dispatch,jsSource,qareceiver,info(ok1)," + msgNum++ +"")", 200);
5 sendMsgAfterTime("msg(info,dispatch,jsSource,qareceiver,info(ok2)," + msgNum++ +"")", 500);
6 sendMsgAfterTime("msg(alarm,event,jsSource,none,alarm(obstacle)," + msgNum++ +"")", 700);
7 sendMsgAfterTime("msg(info,dispatch,jsSource,qareceiver,info(ok3)," + msgNum++ +"")", 1000);

```

Listing 1.33. TcpClientToQaNode.js: application

8.2.2 The result .

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

```

1 "qajaactoivator START"
2 "qareceiver WAITS"
3     *** nodjs> connect to localhost:8031
4     *** nodjs> connect socket allowHalfOpen= false
5     %%% CtxServerAgent ctxjstoqa_Server WORKING on port 8031
6 -----
7 qareceiver_ctrl currentMessage=msg(info,dispatch,jsSource,qareceiver,info(ok1),1)
8 -----
9 "qareceiver WAITS"
10 -----
11 qareceiver_ctrl currentMessage=msg(info,dispatch,jsSource,qareceiver,info(ok2),2)
12 -----

```

```
13 "qareceiver WAITS"
14 endOfwork(obstacle)
15    %% SystemCreationActor terminates1:qareceiver_ctrl 1/4
16    %% SystemCreationActor terminates1:qareceiver 2/4
17 >>> evh      (defaultState, TG=01:00:00)      || msg(alarm,event,jsSource,none,alarm(obstacle),3)
18 ...
19     *** nodjs> SOCKET END
20 "qajaactoivator END"
21 ...
```

9 Reply to a sender

In the *QActor* language the built-in operation `replyToCaller` can be used by an actor `a1` that receives a dispatch, to forward a dispatch to the sender. It is important to day that the sender that will receive the 'reply' will be:

- the sender specified in the `currentMessage`, if `a1` is still in a state that 'remembers' the current message;
- otherwise, the sender that will receive the 'reply' will be the actor denoted in the message stored by the last execution of the built-in operation `storeCurrentMessageForReply`.

Here an example:

1. the actor `caller` forwards a dispatch to the actor `called1` that immediately replies to it. In this case, the operation `storeCurrentMessageForReply` is not called, since the system has still access to the current message that specifies the sender;
2. after having handled the answer, the `caller` forwards another dispatch to to the actor `called1` that now executes `storeCurrentMessageForReply` since it, in its turn, forwards a dispatch to the actor `called2`, by expecting a reply from if. After having received the reply, the actor `called1` forwards a reply to the original `caller`.

```
1 Dispatch autoMsg : autoMsg(X)
2
3 Context ctxSendAnswer ip[ host="localhost" port=8028 ] // -g cyan
4
5 QActor caller context ctxSendAnswer{
6     Plan init normal[
7         println("caller STARTS")
8     ]
9     switchTo firstRequest
10
11    Plan firstRequest [
12        println("firstRequest send request 1");
13        forward called1 -m arequest : arequest( r1,d(1) )
14    ]
15    transition stopAfter 30000
16    whenMsg answer -> handleAnswer
17
18    State handleAnswer [
19        onMsg answer : answer(R,D) -> println(answer1_at_caller(R,D))
20    ]
21    switchTo secondRequest
22
23    State secondRequest[
24        println("firstRequest send request 2");
25        forward called1 -m arequest : arequest( r2,d(2) )
26    ]
27    transition stopAfter 30000
28    whenMsg answer : answer(R,D) do println(answer2_at_caller(R,D))
29
30 }
31 /*
32 */
33 */
34 QActor called1 context ctxSendAnswer{
35     Plan init normal[
36         println("called1 STARTS")
37     ]
38     switchTo waitRequest
39
40     Plan waitRequest[
41         println("called1 waiting ...")
42     ]
43     transition stopAfter 5000
44     whenMsg arequest -> handleRequest
```

```

45
46 Plan handleRequest[
47     printCurrentMessage;
48     storeCurrentMessageForReply;
49     //javaOp "storeCurrentMessageForReply()";
50     onMsg arequest : arequest( r1,DD ) ->
51         replyToCaller -m answer : answer( called1(r1),D );
52     onMsg arequest : arequest( r2,D ) ->
53         selfMsg autoMsg : autoMsg( arequest( r2,D ) )
54 ]
55 transition whenTime 500 -> waitRequest
56     whenMsg autoMsg -> handleSecondRequest
57
58 Plan handleSecondRequest[
59     println("called1 handleSecondRequest ");
60     onMsg autoMsg : autoMsg( M ) ->
61         forward called2 -m arequest : M
62 ]
63 transition stopAfter 5000
64     whenMsg answer -> sendAnswerToCaller
65
66 Plan sendAnswerToCaller[
67     println("called1 send the answer from called2 to its original caller ");
68     printCurrentMessage;
69     onMsg answer : answer( R,D ) ->
70         replyToCaller -m answer : answer( called1(R), D )
71 ]
72 transition stopAfter 3000
73 }
74
75 /*
76 *
77 */
78 QActor called2 context ctxSendAnswer{
79     Plan init normal[
80         println("called2 waiting ...")
81     ]
82     transition stopAfter 5000
83         whenMsg arequest -> handleRequest
84
85     Plan handleRequest[
86         printCurrentMessage;
87         onMsg arequest : arequest( R,D )->
88             replyToCaller -m answer : answer( called2(R),D )
89     ]
90     switchTo init
91 }
92
93 /*
94 * OUTPUT
95 */

```

Listing 1.34. sendAnswer.qa

10 Dynamic systems: the flag `-standalone`

A dynamic system is defined here as a *QActor* system composed by a set of Contexts (at least one) that can be extended by adding new Contexts by means of partial system specifications.

A **partial specification** is the specification of a *QActor* system in which a Context (just one!) is declared with the flag `-standalone`. This means that our system will interact with actors working in a computational node already in existence.

For example, a dynamically extendible system can be introduced in the following way:

1. Define a start-up system, working in a Context that will be used as a '**pivot**' for other Contexts.
2. Define partial specifications of others systems in which the pivot-Context is qualified as `-standalone`.

10.1 Example of a pivot

Here is an example of a 'pivot' that shows the initial configuration of the system and then waits for a message (an dispatch named `info`) or for an event (`alarm`). When the pivot receives (from a dynamically added actor) the `info` message, it emits the `alarm` that could be perceived by all the dynamically added actors.

```
1 System dynamic
2
3 Dispatch info : info(X)
4 Event alarm : alarm(X)
5
6 Context ctxPivot ip [ host="localhost" port=8030 ] -g cyan
7
8 QActor qa0 context ctxPivot{
9     State init initial [
10         demo consult("curConfigTheory.pl")
11     ]
12     switchTo work
13
14     Plan work [
15         demo showSystemConfiguration;
16         println( "qa0 in ctxPivot READY" )
17     ]
18     transition stopAfter 600000
19         whenEvent alarm : alarm(X) do println( alarm(X) ),
20         whenMsg info -> handleInfo
21         finally repeatPlan
22
23     State handleInfo resumeLastPlan [
24         printCurrentMessage;
25         delay 1000;
26         onMsg info : info(X) -> emit alarm : alarm( fire )
27     ]
28 }
```

Listing 1.35. pivot.qa

10.1.1 Rules to show the current system configuration .

The initial configuration of the system is described in the file `it.unibo.ctxPivot.dynamic.pl` generated by the qa software factory in the `srcMore` directory. A program that shows such a configuration can be easily written in tuProlog as done in Subsection 3.2. Since these rules are now built-in the `actorKb`, the output of the pivot is:

```
1 -----,
2 'CONTEXTS IN THE SYSTEM:',
3 context(ctxpivot,localhost,'TCP','8030')
4 'ACTORS IN THE SYSTEM:'
```

```

5 | qactor(qa0,ctxpivot,'it.unibo.qa0.MsgHandle_Qa0')
6 | qactor(qa0_ctrl,ctxpivot,'it.unibo.qa0.Qa0')
7 | -----

```

10.2 Adding a new context

New nodes (Contexts) dynamically added to the system will use the **pivot-Context** as a reference point. If an actor of a partial system must send messages to actors working in the **pivot-Context**, the partial specification must include a 'dummy' declaration of the actors of the pivot-Context (without the specification of the behaviour). If we use events rather than messages, we do not need any dummy declaration, since the events are automatically propagated to all the nodes by the *QActor* runtime.

As an example, let us dynamically add a new node (Context **ctx2Pivot**) that includes an actor that forwards the **info** dispatch to the actor **qa0** in the pivot.

```

1 System dynamic
2
3 Dispatch info : info(X)
4 Event alarm : alarm(X)
5
6 Context ctxPivot ip [ host="localhost" port=8030 ] -standalone
7 Context ctx1Pivot ip [ host="localhost" port=8040 ]
8
9 EventHandler evh1 for alarm -print;
10
11 QActor qa1 context ctx1Pivot{
12     Plan init initial [
13         println( "qa1 READY" );
14         forward qa0 -m info : info("hello form qa1 in ctx1Pivot")
15     ]
16     switchTo observe
17
18     Plan observe[
19         println( "qa1 observing ..." )
20     ]
21     transition stopAfter 600000
22         whenEvent alarm -> handleAlarm
23
24     Plan handleAlarm [
25         onEvent alarm : alarm(X) -> println( alarm(X) )
26     ]
27     transition whenTime 3000 -> endOfJob
28         whenEvent alarm -> handleAlarm
29
30     Plan endOfJob[
31         println( "qa1 endOfJob ..." );
32         disconnectFromStandalone ctxPivot;
33         delay 3000
34     ]
35 }
36 /*
37 * Dummy declaration
38 */
39 QActor qa0 context ctxPivot{
40     State init initial [
41         println( "qa0 dummy : never here" )
42     ]
43 }

```

Listing 1.36. ctx1Pivot.qa

Such a 'dummy' declaration is required for reference purposes (in fact message interaction requires a reference to an actor name).

10.3 Adding another context

In order to avoid (inelegant) 'dummy' declarations we can exploit the built-in `sendTo` operation, according to the following syntax:

```
1 "sendto" dest=VarOrAtomic "in" ctx=[Context] "-m" msgref=[Message] ":" val = PHead
```

The `dest` argument must denote the name of the destination actor, while `ctx` is a reference to the Context in which it works. Here is an example:

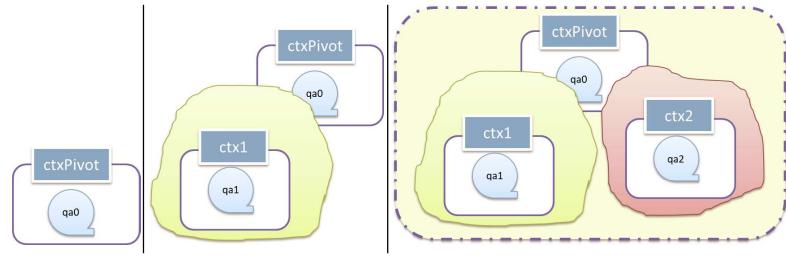
```
1 System dynamic
2
3 Dispatch info : info(X)
4 Event alarm : alarm(X)
5
6 Context ctxPivot ip [ host="localhost" port=8030 ] -standalone
7 Context ctx2Pivot ip [ host="localhost" port=8050 ]
8
9 EventHandler evh1 for alarm -print;
10
11 QActor qa2 context ctx2Pivot{
12     Plan init normal [
13         demo consult("curConfigTheory.pl");
14         demo showSystemConfiguration;
15         println( "qa2 READY" )
16     ]
17     switchTo doWork
18
19     Plan doWork[
20         sendto qa0 in ctxPivot -m info : info( helloctx2Pivot )
21     ]
22     transition whenTime 60000 -> stopTheWork
23     whenEvent alarm -> handleAlarm
24
25     Plan handleAlarm [
26         printCurrentEvent;
27         demo showSystemConfiguration;
28         onEvent alarm : alarm(X) -> println( alarm(X) )
29     ]
30     transition whenTime 60000 -> stopTheWork
31     whenEvent alarm -> handleAlarm
32
33     Plan stopTheWork [
34         println( "BYE form qa2" );
35         disconnectFromStandalone ctxPivot;
36         delay 3000
37     ]
38 }
```

Listing 1.37. ctx2Pivot.qa

10.4 Dynamic extension of the configuration knowledge

Each time that a new component is dynamically added to the system, the system knowledge-base stored in each Context¹⁶ is dynamically extended by the `qa` run-time support with the configuration related to the new component. In this way, each actor in the system becomes able to send messages to other remote actors (dynamically introduced in the system), if it is able to know their names

¹⁶ The static part of the system knowledge is stored in a file `projectname.pl` generated in the Context package of the directory `srcMore`.



The 'evolution' of the system knowledge-base is shown by the system (rule `showSystemConfiguration`).

10.5 Removing a node

When all the actors working in a dynamically added Context are terminated, that Context should be removed from the system (and the system knowledge-base properly updated).

In the current implementation the removal of a context must explicitly triggered by the Application designer through the built-in operation `disconnectFromStandalone`.

11 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 Networks¹⁷) 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.

11.1 MQTT support

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

```
1 package it.unibo.qactors.mqtt;
2 import it.unibo.qactors.akka.QActor;
3 import org.eclipse.paho.client.mqttv3.MqttCallback;
4 import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
5 import org.eclipse.paho.client.mqttv3.MqttClient;
6 import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
7 import org.eclipse.paho.client.mqttv3.MqttException;
8 import org.eclipse.paho.client.mqttv3.MqttMessage;
9 import alice.tuprolog.Struct;
10 import alice.tuprolog.Term;
11
12 public class MqttUtils implements MqttCallback{
13     private static MqttUtils myself = null;
14     protected String clientid = null;
15     protected String eventId = "mqtt";
16     protected String eventMsg = "";
17     protected QActor actor = null;
18     protected MqttClient client = null;
19
20     public static MqttUtils getMqttSupport( QActor qa ){
21         // System.out.println(" %% MqttUtils getMqttSupport qa="+ qa );
22         if( myself == null ) myself = new MqttUtils();
23         return myself ;
24     }
25     public MqttUtils(){
26         try {
27             myself = this;
28         } catch (Exception e) {
29             println(" %% MqttUtils WARNING: "+ e.getMessage() );
30         }
31     }
32     public boolean connect(QActor actor, String clientid, String brokerAddr, String topic ) {
33         try{
34             this.actor = actor;
35             client = new MqttClient(brokerAddr, clientid);
36             println(" %% MqttUtils doing connect/4 "+ clientid + " " + brokerAddr + " " + topic );//+ " " + client
37             MqttConnectOptions options = new MqttConnectOptions();
38             options.setKeepAliveInterval(480);
39             options.setWill("unibo/clienterrors", "crashed".getBytes(), 2, true);
40             client.connect(options);
41             println(" %% MqttUtils connect DONE "+ clientid + " to:" + brokerAddr + " " + topic );//+ " " + client
42             return true;
43         }catch(Exception e){
44             actor.println(" %% MqttUtils connect ERROR for:" + brokerAddr);
45             return false;
46         }
47     }
48     public void disconnect( ) {
```

¹⁷ MQTT-SN is a protocol derived from MQTT, designed for connectionless underlying network transports such as UDP

```

49     try{
50         println(" %% MqttUtils disconnect "+ client );
51         if( client != null ) client.disconnect();
52     }catch(Exception e){
53         actor.println(" %% MqttUtils disconnect ERROR " + e.getMessage());
54     }
55 }
56
57 public void subscribe(QActor actor, String clientid, String topic) throws Exception {
58     try{
59         this.actor = actor;
60         client.setCallback(this);
61         client.subscribe(topic);
62     }catch(Exception e){
63 //         println(" %% MqttUtils subscribe error "+ e.getMessage() );
64 //         eventMsg = "mqqt(" + eventId +", failure)";
65 //         println(" %% MqttUtils subscribe error "+ eventMsg );
66         if( actor != null ) actor.sendMsg("mqttmsg", actor.getName(), "dispatch", "error");
67         throw e;
68     }
69 }
70 @Override
71 public void connectionLost(Throwable cause) {
72     println(" %% MqttUtils connectionLost = "+ cause.getMessage() );
73 }
74 @Override
75 public void deliveryComplete(IMqttDeliveryToken token) {
76 //     println(" %% MqttUtils deliveryComplete token= "+ token );
77 }
78 /*
79 * sends to a topic a content of the form
80 *      msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
81 */
82 public void publish(QActor actor, String clientid, String topic, String msg,
83                     int qos, boolean retain) throws MqttException{
84     MqttMessage message = new MqttMessage();
85     message.setRetained(retain);
86     if( qos == 0 || qos == 1 || qos == 2){//qos=0 fire and forget; qos=1 at least once(default);qos=2 exactly once
87         message.setQos(0);
88     }
89     message.setPayload(msg.getBytes());
90     try{
91         client.publish(topic, message);
92 //         println(" %% MqttUtils published "+ message + " on topic=" + topic);
93     }catch(MqttException e){
94         println(" %% MqttUtils publish ERROR "+ e );
95     }
96 }
97 /*
98 * receives a message of the form
99 *      msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )
100 * and sends it to the RECEIVER
101 */
102 private String msgSender = null;
103
104 @Override //MqttCallback
105 public void messageArrived(String topic, MqttMessage msg) {
106     String msgID      = null;
107     String msgType    = null;
108     String dest       = null;
109     String msgcontent = null;
110     try {
111 //         println(" %% MqttUtils messageArrived on "+ topic + ": "+msg.toString());
112         Struct msigt   = (Struct) Term.createTerm(msg.toString());
113 //         println(" %% MqttUtils messageArrived msigt "+ msigt + " actor=" + actor.getName() );
114         msgID      = msigt.getArg(0).toString();
115         msgType    = msigt.getArg(1).toString();
116         msgSender  = msigt.getArg(2).toString();
117         dest       = msigt.getArg(3).toString();
118         msgcontent = msigt.getArg(4).toString();

```

```

119     if( actor != null ) { //send a msg to itself (named without _ctrl)
120         if( msgType.equals("event") ) {
121             actor.emitFromMqtt(msgID, msgcontent);
122         }else {
123             actor.sendMsgFromMqtt( msgSender, msgID, actor.getName().replace("_ctrl", ""), msgType,
124                                     msgcontent);
125         }
126     } catch (Exception e) {
127         println("messageArrived ERROR "+e.getMessage());
128     }
129 }
```

Listing 1.38. 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 MQTT 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.

11.2 MQTT built-in operations

The `QActor` language-metamodel defines a set of built-in operation for interactions based on MQTT:

— MQTT —	
<code>pubSubServer addrString</code>	specifier the name of a MQTT server to be used in the application.
<code>connectAsPublisher topicString</code>	connect as publisher to the given topic of the specifier MQTT server.
<code>connectAsSubscriber topicString</code>	connect as subscriber to the given topic of the specifier MQTT server.
<code>publishMsg</code>	publish a <code>QActor</code> message to for the specified destination actor.
<code>topicStr for actorNameStr -m msgRef:PHead</code>	
<code>publishEvent topicString -e msgRef : PHead</code>	publish a <code>QActor</code> event.

These operations are implemented by using the class `MqttUtils` of Subsection 11.1.

11.3 Example: a producer-consumer system

```

1 System prodConsDistributedMqtt
2 Dispatch info : info(X)
3 Dispatch autoMsg : autoMsg(X)
4
5 Event trace : trace(X)
6
7 //pubSubServer "tcp://m2m.eclipse.org:1883"
8 pubSubServer "tcp://test.mosquitto.org:1883"
9
10
11 Context ctxProdDistrMqtt ip [ host="localhost" port=8079 ]
12 Context ctxConsDistrMqtt ip [ host="localhost" port=8089 ]
13 Context ctxCons2DistrMqtt ip [ host="localhost" port=8099 ] -g yellow
14
```

```

15 | QActor producerdistrmqtt context ctxProdDistrMqtt -pubsub{
16 |   Rules{
17 |     item(1).
18 |     item(2).
19 |     item(3).
20 |   }
21 |   State init initial [
22 |     println( producerdistrmqtt( starts ) )
23 |   ]
24 |   switchTo cleanTheQaTopic
25 |   State cleanTheQaTopic[
26 |     //clearTopic "unibo/qasys"
27 |     javaOp "clearQaTopic()"
28 |   ]
29 |   switchTo doWork
30 |
31 |   State doWork [
32 |     delay 2000;
33 |     [ !? item(X) ] println( producer(sends(X)) ) ;
34 |     [ !? item(X) ] emit trace : trace( log( producerdistrmqtt(item(X)) ) ) ;
35 |     [ ?? item(X) ] selfMsg autoMsg : autoMsg( item(X) )
36 |   ]
37 |   transition stopAfter 60000
38 |     whenMsg autoMsg -> handleAutoMsg
39 |   finally repeatPlan
40 |
41 |   Plan handleAutoMsg [
42 |     onMsg autoMsg : autoMsg(M) -> forward consumerdistrmqtt -m info : info( M )
43 |   ]
44 |   transition whenTime 200 -> doWork
45 |     whenMsg autoMsg -> handleAutoMsg
46 |   }
47 |
48 |
49 | QActor consumerdistrmqtt context ctxConsDistrMqtt -pubsub{
50 |   State init initial [
51 |     println( consumerdistrmqtt(waiting) )
52 |   ]
53 |   switchTo doWork
54 |
55 |   State doWork [
56 |     //      println( consumerdistrmqtt(waiting) )
57 |   ]
58 |   transition whenTime 30000 -> handleTout
59 |     whenMsg info -> handleMsg
60 |     finally repeatPlan
61 |
62 |   State handleMsg resumeLastPlan [
63 |     printCurrentMessage ;
64 |     onMsg info : info(X) -> emit trace : trace( log( consumerdistrmqtt(item(X)) ) ) ;
65 |     onMsg info : info(X) -> println( consumerdistrmqttReceiving(X) )
66 |   ]
67 |   State handleTout [
68 |     println( consumerdistr(tout) )
69 |   ]
70 | }
71 |
72 | QActor consumer2distrmqtt context ctxCons2DistrMqtt -pubsub{
73 |   State init initial [
74 |     println( consumer2distrmqtt(starts) )
75 |   ]
76 |   switchTo doWork
77 |
78 |   State doWork [
79 |     //      println( consumer2distrmqtt(waiting) )
80 |   ]
81 |   transition whenTime 4000 -> showTrace
82 |     whenEvent trace -> handleTraceevent
83 |     finally repeatPlan
84 |

```

```

85     State handleTraceevent resumeLastPlan[
86         onEvent trace : trace(X) -> println( ccccccccconsumer2mqtt_trace(X) );
87         onEvent trace : trace(X) -> addRule trace(X)
88     ]
89
90     State showTrace[
91         println("-----");
92         [?? trace(M) ] println( M ) else{
93             javaOp "clearQaTopic()";
94             delay 5000;
95             endQActor "consumer2distrmqtt END"
96         }
97     ]
98     finally repeatPlan
99 
```

Listing 1.39. prodConsDistributedMqtt.qa

Note that the producer and the consumer emit the event `trace` to make observable their job. These events are handled by the actor named `consumer2distrmqtt` that can be used an 'observer' useful to automate the (acceptance) testing of the system according to Subsection 6.2.

11.3.1 Running the system .

```

1  /*
2  * The system works correctly if each produced item is consumed in the same order.
3  */
4
5 package it.unibo.qa2018.prodcons;
6
7 import static org.junit.Assert.fail;
8 import org.junit.Before;
9 import org.junit.Test;
10
11 public class TestProdConsDistrMqtt {
12
13
14     @Before
15     public void setUp(){
16         System.out.println("===== setUp =====");
17         try {
18             new Thread() {
19                 public void run() { startProducer(); }
20             }.start();
21             new Thread() {
22                 public void run() { startConsumer(); }
23             }.start();
24             new Thread() {
25                 public void run() { startConsumer2(); }
26             }.start();
27             Thread.sleep(1000);
28         } catch (Exception e) {
29             fail( e.getMessage() );
30         }
31     }
32
33     protected void startProducer() {
34         try {
35             it.unibo.ctxProdDistrMqtt.MainCtxProdDistrMqtt.initTheContext();
36             System.out.println("setUp ENDS producer");
37         } catch (Exception e) {
38             fail( e.getMessage() );
39         }
40     }
41     protected void startConsumer() {
42         try { 
```

```
43         it.unibo.ctxConsDistrMqtt.MainCtxConsDistrMqtt.initTheContext();
44         System.out.println("setUp ENDS consumer" );
45     } catch (Exception e) {
46         fail( e.getMessage() ) ;
47     }
48 }
49 protected void startConsumer2() { //TESTER
50     try {
51         it.unibo.ctxCons2DistrMqtt.MainCtxCons2DistrMqtt.initTheContext();
52         System.out.println("setUp ENDS consumer2 " );
53     } catch (Exception e) {
54         fail( e.getMessage() ) ;
55     }
56 }
57 @Test
58 public void test(){
59     System.out.println("||||||||||||||||| test |||||||||||||||||||");
60     try {
61         Thread.sleep( 60000 );
62     } catch (InterruptedException e) {
63         e.printStackTrace();
64     }
65 }
66 }
67 }
68 }
69 }
```

Listing 1.40. TestProdConsDistrMqtt.java