

Introduction to QActors and QRobots

Antonio Natali

Alma Mater Studiorum – University of Bologna
via Sacchi 3, 47023 Cesena, Italy,
Viale Risorgimento 2, 40136 Bologna, Italy
`antonio.natali@studio.unibo.it`

Table of Contents

Introduction to QActors and QRobots	1
<i>Antonio Natali</i>	
1 Introduction to QActors	5
1.1 Example: The 'hello world'	5
1.2 QActor specification	5
1.3 The software factory	6
1.4 The work of the application designer	6
1.5 Example: Actor as a finite state machine	6
1.6 Example: Message-based interaction	7
1.6.1 Testing the basicProdCons system	7
2 QActor concept overview	9
2.1 Actions	9
2.1.1 The actor's WorldTheory	9
2.1.2 Logical actions	9
2.1.3 Physical actions	9
2.1.4 Application actions	10
2.1.5 PlanActions	10
2.2 Plans	10
2.3 Messages and message-driven/message based behaviour	10
2.4 Events and event-driven/event-based behaviour	11
2.5 Actor programs as plans	11
2.5.1 Actor programs as finite state machines (FSM)	12
3 The qa language/metamodel	13
3.1 Messages and message based behaviour	13
3.2 Send actions	13
3.2.1 Forward a <i>dispatch</i> : syntax	14
3.2.2 Demand a <i>request</i> : syntax	14
3.3 Receive actions	14
3.3.1 Generic receive with optional message specification: syntax	14
3.3.2 Receive a message with a specified structure: syntax	14
3.3.3 onMsg: syntax	15
3.4 Events and event-based behaviour	15
3.4.1 Emit an event: syntax	15
3.4.2 Sense events: syntax	15
3.4.3 OnEvent action: syntax	15
3.5 Event handlers and event-driven behaviour	16
3.6 Guarded actions	17
3.7 About implementation	18
3.7.1 Sending messages	18
3.7.2 Receiving messages	18
3.7.3 forward implementation (see Subsection 3.7.1)	19
3.7.4 demand implementation (see Subsection 3.7.1)	19
3.7.5 emit implementation	19

3.7.6	sense implementation	19
3.8	QActor knowledge	19
3.8.1	Inspect the state and elaborate in a functional way	20
3.8.2	Change the internal state	20
4	Human interaction with a Qactor	21
4.1	The (remote) Web interface	21
4.2	The (local) GUI user interface	21
5	Building qa models	22
5.0.1	Application designer and System designer.	22
5.1	QActor software factory	22
6	User-defined actions in Prolog	23
6.0.1	Examples of unification	23
6.1	The <code>demo</code> and <code>solve</code> operation.	23
6.2	Loading and using a user-defined theory	23
6.2.1	The initialization directive	24
6.2.2	On backtracking.	25
6.3	Using the actor in Prolog rules	25
6.4	The operator <code>actorOp</code>	26
6.5	Rules at model level	27
6.6	From Prolog to Java again	28
6.6.1	Guards as problem-solving operation.	28
6.6.2	The user-defined <code>select/1</code> operation.	28
6.7	Workflow	32
6.8	Examples of problem solving with <code>tuProlog</code>	32
6.8.1	configuration.	34
6.8.2	family	34
6.8.3	<code>accessData</code>	35
6.8.4	output.	35
7	Advanced Actions (observable, timed, reactive)	36
7.1	Asynchronous Observable Actions	36
7.2	The class <code>ActionObservableGeneric</code>	37
7.2.1	<code>Callable<T></code>	37
7.2.2	<code>Future<T></code>	37
7.2.3	<code>execAsynch()</code>	38
7.2.4	<code>Executors</code>	38
7.2.5	<code>execSynch()</code>	38
7.2.6	<code>T call()</code>	38
7.2.7	<code>startOfAction()</code>	38
7.2.8	<code>endActionInternal()</code>	39
7.2.9	<code>execTheAction()</code> and <code>endOfAction()</code>	39
7.2.10	Fibonacci as Observable	39
7.2.11	Experiments on Fibonacci as Observable	40
7.3	Timed actions	41
7.3.1	<code>ActorTimedAction</code>	41
7.3.2	Fibonacci as a Timed	43
7.3.3	Experiments on Fibonacci as Timed	44
7.4	Reactive actions	46

7.4.1	AsynchActionResult.....	46
7.4.2	Fibonacci as a Reactive.....	47
7.4.3	Experiments on Fibonacci as Reactive	48
8	An interpreter to execute actions.....	49
8.1	Basic actions	49
8.2	Guarded actions.....	50
8.3	Timed actions	50
8.4	Time out	50
8.5	Asynchronous actions	51
8.6	Reactive actions.....	51

1 Introduction to QActors

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

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

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

1.1 Example: The 'hello world'

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

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

Listing 1.1. hello.qa

This example shows that each qactor works within a *Context* that models a computational node associated with a network IP (*host*) and a communication *port* (see Subsection 3.7). The behaviour of a is modelled as a set of macro-actions called *plans* (see Subsection 2.2).

1.2 QActor specification

A *QActor* specification can be viewed as:

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

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

- execute actions (see Subsection 2.1)
- send/receive messages (see Subsection 3.1)
- emit/perceive events (see Subsection 3.4)

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

1.3 The software factory

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

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

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

1.4 The work of the application designer

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

1. Copy in the current workspace the project `it.unibo.iss.libs`.
2. Execute the command `gradle -b build_ctxHello.gradle eclipse` in order to set the required libraries.
3. Modify the code of the actors by introducing in the generated actor class (in the `src` directory) the required application code. In the case of the example, we have no need to modify the generated class `src/it/unibo/qahello/Qahello.java`. In any case, this class is generated only once, so that code changes made by the application designer are not lost if the model is modified.
4. Define a set of JUnit testing operations within a source folder named `test`. Each test unit should start with the string "Test" (see the generated build file). For an example see Subsection 1.6.1.
5. Run the generated main program (`src-gen/it/unibo/ctxHello/MainCtxHello.java`).

1.5 Example: Actor as a finite state machine

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

```
1  /*
2  * basic.qa
3  * A system composed of a qactor working in a context associated with a GUI
4  */
5  System basic //the testing flag avoids automatic termination
6  Context ctxBasic ip [ host="localhost" port=8079 ] -g cyan
7  QActor player context ctxBasic{
8      Plan init normal
9          println("Hello world" ) ;
10         switchToPlan playMusic ;
11         println("Bye bye" )
12     Plan playMusic resumeLastPlan
13         sound time(2000) file('./audio/tada2.wav') ;
14         [ ?? tout(X,Y) ] println( tout(X,Y) ) ;
15         repeatPlan 1
16 }
```

Listing 1.2. `basic.qa`

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

1.6 Example: Message-based interaction

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

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

```

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

```

Listing 1.3. basicProdCons.qa

1.6.1 Testing the basicProdCons system .

Here is an example of testing (written by the application designer) to be run with JUnit4:

```

1 package it.unibo.ctxBasicCons;
2 import static org.junit.Assert.*;
3 import java.util.concurrent.ScheduledThreadPoolExecutor;

```

```

4 import org.junit.After;
5 import org.junit.Before;
6 import org.junit.Test;
7 import alice.tuprolog.SolveInfo;
8 import it.unibo.ctxBasicProd.MainCtxBasicProd;
9 import it.unibo.qactors.QActorUtils;
10 import it.unibo.qactors.akka.QActor;
11
12 public class TestProdCons {
13     private QActor prod;
14     private QActor cons;
15     private ScheduledThreadPoolExecutor sched =
16         new ScheduledThreadPoolExecutor( Runtime.getRuntime().availableProcessors() );
17
18     @Before
19     public void setUp() throws Exception {
20         activateCtxs();
21         //Get a reference to the two actors:
22         //getQActor waits until the actor is active
23         prod = QActorUtils.getQActor("producer_ctrl");
24         cons = QActorUtils.getQActor("consumer_ctrl");
25     }
26
27     @After
28     public void terminate(){
29         System.out.println("==== terminate " );
30     }
31
32     @Test
33     public void execTest() {
34         System.out.println("==== execTest =====" );
35         try {
36             assertTrue("execTest prod", prod != null );
37             assertTrue("actorTest cons", cons != null );
38             System.out.println("==== execTest waits for work completion ... " );
39             Thread.sleep(2000); //give the time to work
40             //Acquire the message stored at the consumer site
41             SolveInfo sol = cons.solveGoal("msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )");
42             System.out.println("execTest CONTENT="+sol.getVarValue("CONTENT"));
43             assertTrue("actorTest info", sol.getVarValue("CONTENT").toString().equals("info(1)" ) );
44         } catch (Exception e) {
45             fail("actorTest " + e.getMessage() );
46         }
47     }
48
49     private void activateCtxs( ){
50         sched.submit( new Runnable(){
51             @Override
52             public void run() {
53                 try {
54                     //QActorContext ctxCons =
55                     MainCtxBasicCons.initTheContext();
56                 } catch (Exception e) { e.printStackTrace(); }
57             }
58         });
59         sched.submit( new Runnable(){
60             @Override
61             public void run() {
62                 try {
63                     //QActorContext ctxProd =
64                     MainCtxBasicProd.initTheContext();
65                 } catch (Exception e) { e.printStackTrace(); }
66             }
67         });
68     }
69 }

```

Listing 1.4. TestProdCons.qa

2 QActor concept overview

2.1 Actions

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

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

1. as changes in the state of the actor (called from now also as the actor's "*mind*", see Subsection 2.1.1);
2. as changes in the actor's working environment.

The first kind of actions are referred here as *logical actions* since they do not affect the physical world. The *actor-mind* is represented by a *tuProlog* theory named *WorldTheory* associated with the actor (see Subsection 3.8)).

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

2.1.1 The actor's WorldTheory. The *WorldTheory* includes facts about the state of the actor and of the world. For example:

- the fact `actorobj/1` memorizes a reference to the *Java/Akka* object that implements the actor (see Subsection 6.3).
- the fact `actorOpDone/2` memorizes the result of the last `actorOp` executed (see Subsection 6.4)
- the fact `goalResult/1` memorizes the result of the last Prolog goal given to a `solve` operation (see Subsection 6.1)
- the fact `result/1` memorizes the result of the last plan action (see Subsection 2.1.5) performed by the actor

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

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

- the rule `actorPrintln/1` prints a given *tuProlog Term* (see Subsection 3.1) in the standard output of the actor;
- the rule `actorOp/1` puts in execution a *Java* method written by the application designer (see Subsection 6.4).
- the rule `assign(KEY,VALUE)` associates to the given *KEY* the given *VALUE*, by removing any previous association.

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

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

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

To execute a *physical action*, a *Qactor* must be equipped with proper (low-level) libraries.

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

2.1.5 PlanActions. A *PlanAction* is a logical or physical action defined by the system or by the application designer. A *PlanAction* can assume different logical forms according to different attributes that can be associated to it. Let us represent the possible sets of attributes with the following notation:

Actions attribute sets	
ACTION	
GUARD ACTION	
ACTION DURATION	
ACTION DURATION ENDEVENT	
ACTION DURATION EVENTLIST PLANLIST	

With reference to the previous notation, we introduce the following terminology:

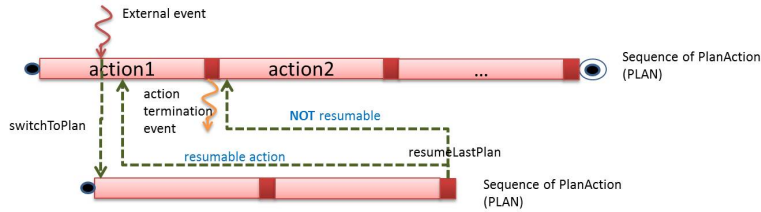
- an action that does not specify any attribute is called **basic action** (see Subsection 8.1);
- an action associated to a **GUARD** is called **guarded action** (see Subsection 8.2);
- an action that specifies a **DURATION** is called **timed action** (see Subsection 8.3);
- an action that specifies a **ENDEVENT** is called (*timed*) **asynchronous action** (see Subsection 8.5);
- an action that specifies **EVENTLIST**, **PLANLIST** is called (*timed,synchronous*) **reactive action** (see Subsection 8.6).

Thus, a *timed action*:

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

2.2 Plans

A *Plan* is a list of *PlanActions* executed in sequence.



2.3 Messages and message-driven/message based behaviour

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

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

Messages are represented as follows:

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

where¹ :

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

The `msg/6` pattern can be used to express *guards* (see Subsection 8.2) to allow conditional evaluation of *PlanActions*.

2.4 Events and event-driven/event-based behaviour

An *event* is defined here as information emitted by some source without any explicit destination. Events can be *emitted* by the *QActors* that compose 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 3.5).

An event can also trigger state transitions in components, usually working as finite state machines that call operations to explicitly *sense* events (see Subsection 3.4.2). We qualify this kind of behaviour as *event-based* behaviour, since the event is 'lost' if no actor is in a state waiting for it.

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

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

2.5 Actor programs as plans

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

Each plan has a name and can be put into execution by a proper *PlanAction* (including an explicit `switchToPlan`, see Subsection 8.1). Plans can be stored in files and dynamically loaded by the user into the *actor-mind* (see Subsection ??).

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

```
Internal representation of plans
plan(ACTIONCOUNTER,PLANNAME,sentence(GUARD,MOVE,EVENTLIST,PLANLIST))
```

For example:

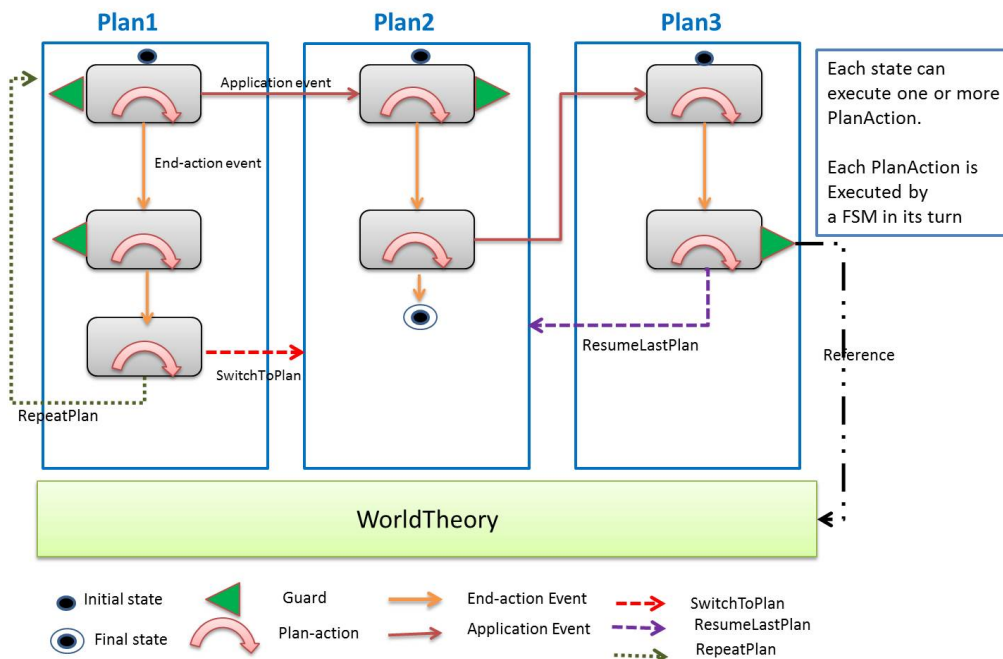
```
Internal representation of a plan
plan(0,p0,sentence(true,move(playsound,'./audio/tada2.wav',1500),'',''))
plan(1,p0,sentence(true,move(playsound,'./audio/music_interlude20.wav',20000),
                    'usercmd,alarm','handleUsercmd,handleAlarm'))
```

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

¹ At the moment only `dispatch`, `request` and `event` are implemented

2.5.1 Actor programs as finite state machines (FSM) .

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



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

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

- a (built-in) normal termination event;
- a user-defined termination event (see Subsection 8.5);
- a time-out termination event;
- a abnormal termination event;

3 The qa language/metamodel.

The *qactor* (qa) language is a custom language built by exploiting the XText technology²; thus, it is also a meta-model. Technically we can say that qa is a 'brother' of UML since it is based on EMOF.

The language-metamodel qa aims at overcoming the *abstraction gap* between the needs of distributed *proactive/reactive* systems and the conventional (object-based) programming language used for implementation (mainly Java, C#, C, etc). In fact, a qa specification aims at capturing main *architectural* aspects of the system by providing a support for *rapid software prototyping*.

Each *Qactor* is able to execute a set of built-in actions to send/receive messages and to emit/sense events. This section is devoted to a brief introduction to the language rules that define the syntax of these operations.

3.1 Messages and message based behaviour

A *QActor* can send/receive *messages* to/from another *Qactor* working in the same or in another *Context*. A *QActor* can also send messages to itself.

The qa language allows us to express send/receive actions as high-level operations that hide at application level the details of the communication support. The *QActor* software factory generates the code required to exploit the *QActor* run time support to implement the message-passing operations.

The qa language defines the following syntax for message declaration:

```
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 syntax 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 ...
```

3.2 Send actions

At qa level, high-level forms of sending-message actions are defined:

- forward a dispatch
- demand a request
- ask an invitation
- ...

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

3.2.1 Forward a *dispatch*: syntax .

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

Example:

```
forward receiver -m info : info(a)
```

3.2.2 Demand a *request*: syntax .

```
1 SendRequest: name="demand" dest=VarOrQactor "-m" msgref=[Message] ":" val = PHead ;
```

Example:

```
demand receiver -m eval : fibo(25,V)
```

3.3 Receive actions

The **qa** language defines several forms of high-level receive-message actions :

- receive any message
- receive a message with optional message specification
- receive a message with a specific structure

3.3.1 Generic receive with optional message specification: syntax .

```
1 ReceiveMsg : name="receiveMsg" duration=TimeLimit (spec=MsgSpec)? ;  
2 TimeLimit : name="time" "(" ( msec=INT | var=Variable ) ")" ;  
3 MsgSpec : "-m" msg=[Message] "sender" sender=VarOrAtomic "content" content=PHead ;
```

Example1: receive a message

```
receiveMsg time( 1000 )
```

Example2: receive a message from a specific **sender** with some specific payload structure:

```
receiveMsg time(100) -m info sender ansa content news(sport(X))
```

3.3.2 Receive a message with a specified structure: syntax .

```
1 OnReceiveMsg: name="receiveTheMsg" "m" "(" msgid=PHead "," msgtype=PHead "," msgsender=PHead  
2 "," msgreceiver=PHead "," msgcontent=PHead "," msgseqnum=PHead ")" duration=TimeLimit ;
```

Example: receive the message whose internal structure **msg/6** is unifiable with the given arguments:

```
receiveTheMsg m( info,dispatch,ansa,R,news(sport(X)),N) time(2000)
```

Once a message has been received, the **onMsg** action allows us to select a message and execute actions according to the specific structure of that message.

3.3.3 onMsg: syntax .

```
1 MsgSwitch : "onMsg" message=[Message] ":" msg = PHead "->" move = Move;  
2 Move      : ActionMove | MessageMove | ExtensionMove | BasicMove | PlanMove | GuardMove | BasicactorMove;
```

Example: print (part of the) content of a message

```
//some receive ...  
onMsg info : news(sport(X))} -> println(X)
```

3.4 Events and event-based behaviour

QActors can emit and sense (perceive) *events* represented as follows:

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

3.4.1 Emit an event: syntax .

The qa language defines an high-level action to emit events

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

Example

```
emit alarm : alarm(fire)
```

3.4.2 Sense events: syntax .

To allow *event-based* behaviour, qa provides a *sense* operation that blocks the execution of a *QActor* until the required event occurs.

```
1 SenseEvent : "sense" duration=TimeLimit events += [Event] ("," events += [Event] )*  
2             "->" plans += Continuation ("," plans += Continuation )* ;  
3 Continuation: plan = [Plan] | name="continue" ;
```

Example

```
sense time(1000) alarm -> continue
```

3.4.3 OnEvent action: syntax . The qa language defines also an event-selection action:

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

Example

```
//sense ...  
onEvent alarm : alarm(fire) -> sound time(2500) file( "./audio/illogical_most2.wav")
```

3.5 Event handlers and event-driven behaviour

The occurrence of an event activates, in *event-driven* way, all the *EventHandlers* 'registered' (i.e. declared in some *Context*) for that event.

In the *qa* language, the declaration of an *EventHandler* must be done within a *Context* with the following syntax:

```
1 EventHandler :
2   "EventHandler" name=ID ( "for" events += [Event] ( "," events += [Event] )* )?
3   ( print ?= "-print" ) ?
4   ( "{ body = EventHandlerBody " }" )?
5   ","
6 EventHandlerBody:
7   op += EventHandlerOperation ( ";" op += EventHandlerOperation)*
8   ;
9 EventHandlerOperation:
10  MemoOperation | SolveOperation | RaiseEvent | SendEventAsDispatch
11  ;
12 MemoOperation:
13  "memo" rule=MemoRule "for" actor=[QActor]
14  | doMemo=MemoCurrentEvent "for" actor=[QActor]
15  ;
16 SolveOperation:
17  "solve" goal=PTerm "for" actor=[QActor]
18  ;
19 SendEventAsDispatch :
20  "forwardEvent" actor=[QActor] "-m" msgref=[Message]
21  ;
22 MemoRule :
23  MemoEvent // | Others memo rules
24  ;
25 MemoEvent :
26  name="currentEvent"
27  ;
```

The syntax shows that, in a *qa* model, we can express only a limited set of actions within an *EventHandler*³:

- memorize and event into the *WorldTheory* of a specific *QActor*
- solve a goal
- forward a dispatch with the content of the event
- emit another event

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

```
1 System eventTracer
2 Event usercmd : usercmd(X)
3 Event alarm : alarm(X)
4 Event obstacle : obstacle(X)
5 Event cmd : cmd(X)
6
7 Context ctxEventTracer ip [ host="localhost" port=8027 ] -g cyan -httpserver
8 EventHandler evh for usercmd,alarm,obstacle,cmd -print {
9   memoCurrentEvent for qaevtracer
10 };
11 EventHandler evhevalfibo for alarm ;
12 EventHandler evhevalother for obstacle ;
13 /*
14  * WARNING: any change in the model modifies the EventHandlers
15  */
16 QActor qaevtracer context ctxEventTracer {
```

³ Of course, other actions can be defined directly in Java by the Application designer.


```

17 Plan init normal
18   println("qaevtracer starts");
19   switchToPlan trace
20 Plan trace
21   [ ?? msg(E,'event',S,none,M,N) ] println(eventm(E,S,M)) else println("noevent");
22   delay time(1000);
23   repeatPlan
24 }

```

Listing 1.5. eventTracer.qa

3.6 Guarded actions

Actions prefixed by a [GUARD] are executed only when the GUARD is evaluated **true**. In the qa language, the GUARD is a boolean condition expressed as a Prolog term that can include unbound variables, possibly bound during the guard evaluation phase. Moreover:

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

For example;

```

1 System actionGuarded
2 Dispatch self : self(X)
3 Context ctxActionGuarded ip [ host="localhost" port=8037 ]
4 QActor qawithguarda context ctxActionGuarded {
5   Rules{
6     execTime(500).
7     data(sonar,10).
8     data(sonar,100).
9   }
10 Plan main normal
11   println("===== " ) ;
12   println("Some guarded actions " ) ;
13   println("===== " ) ;
14   switchToPlan work
15 Plan work resumeLastPlan
16   [ !? execTime(T) ] sound time(T) file( "./audio/tada2.wav" ) ;
17   [ ?? data(sonar,V) ] demo fibo(V,R) ;
18   [ ?? goalResult( R ) ] println( R ) ;
19   [ ?? data(sonar,V) ] println( data(sonar,V) ) ;
20   forward qawithguarda -m self : self("hello") ;
21   switchToPlan end
22 Plan end
23   receiveMsg time(1000);
24   printCurrentMessage -memo ;
25   [ ?? msg(self, 'dispatch',SENDER,RECEIVER,CONTENT,MSGNUM) ]
26     println( mymsg( SENDER,RECEIVER,CONTENT,MSGNUM) ) ;
27   endQActor "bye"
28 }
29 /*
30 fibo(10,55)
31 data(sonar,100)
32 -----
33 qawithguarda_ctrl currentMessage=msg(self,dispatch,qawithguarda_ctrl,qawithguarda,self(hello),1)
34 -----
35 mymsg(qawithguarda_ctrl,qawithguarda,self(hello),1)
36 bye
37 */

```

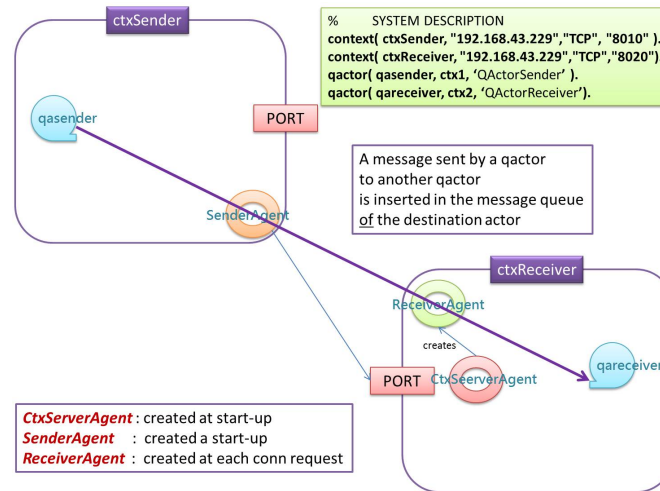
Listing 1.6. actionGuarded.qa

Note that the msg/6 structure is used as guard.

3.7 About implementation

Each *Qactor* must work within a *Context* that models a computational node associated with a network IP (host) and a communication port.

From a model written in the *QActor* language, the *QActor* software factory generates the internal (Akka) actors that allow messages exchanged among *QActor* working on different contexts to flow throw the context ports (using the TCP/IP protocol) and to deliver the message in the message-queue of the destination *QActor*.



3.7.1 Sending messages .

Messages can be sent to a *Qactor* by using the built-in basic `sendMsg` asynchronous, point-to-point action. The API provided by the `qa` run-time support⁴ has the following signature:

```
1 void sendMsg( String msgID, String destActorId, String msgType, String content ) throws Exception
```

3.7.2 Receiving messages .

The `qa` run-time support provides the following operation:

```
1 AsyncActionResult receiveMsg( String msgid, String msgtype, String msgsender, String msgreceiver,
2 String msgcontent, String msgseqnum, int timeout, String events, String plans ) throws Exception
```

`AsyncActionResult` stores the results of the operation (see Subsection 7.4.1).

The `receiveMsg` operation blocks the execution until a messages is received or the specified `timeout` expires. The message must *unify* (with Prolog semantics) with the given arguments.

The `receiveMsg` operation is 'reactive' to the specified *events*. i.e. it transfers the control to the corresponding plan in *plans* when one of the specified events occurs while the operation is still waiting for messages. Thus `receiveMsg` is not a simple procedure, but it is executed by a proper *Finite State Machine*.

⁴ The `qa` run-time support is stored in `qa18Akka.jar`

3.7.3 forward implementation (see Subsection 3.7.1) .

```
1 void forward(String msgId, String dest, String msg) throws Exception{sendMsg(msgId,dest,"dispatch",msg){
2     sendMsg(msgId, dest, QActorContext.dispatch, msg);
3 }
```

3.7.4 demand implementation (see Subsection 3.7.1) .

```
1 void demand(String msgId, String dest, String msg) throws Exception{sendMsg(msgId,dest,"request",msg){
2     sendMsg(msgId, dest, QActorContext.request, msg);
3 }
```

3.7.5 emit implementation . The qa run-time support provides the following operations:

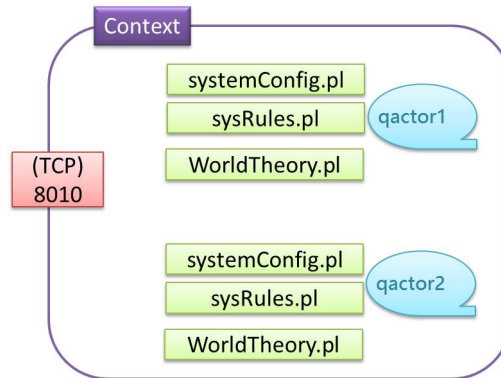
```
1 void emit( String evId, String evContent ) throws Exception
```

3.7.6 sense implementation . The qa run-time support does introduce the following operation:

```
1 AsyncActionResult senseEvents(int tout, String events, String plans,
2     String alarmEvents, String recoveryPlans, ActionExecMode mode) throws Exception{
```

3.8 QActor knowledge

The picture hereunder shows that each actor is associated to a set of tuProlog theories:



- A theory (`systemConfig.pl`) that describes the configuration of the system.
- A theory `sysRules.pl` that describes a set of rules used at system configuration time.
- A theory `WorldTheory.pl` that describes a set of rules and facts that give a symbolic representation of the "world" in which a *QActor* is working.

Let us report here some common functional actions (implemented by the *WorldTheory* associated with each actor) that we can ask a *QActor* to do.

3.8.1 Inspect the state and elaborate in a functional way .

<code>actorPrintln(hello)</code>	prints hello
<code>actorobj(A)</code>	binds A to the name of current actor (robot)
<code>goalResult(A)</code>	binds A to the result of last Prolog goal solved by the actor
<code>result(A)</code>	binds A to the result of last action executed by the actor
<code>fib(10,V),result(A),actorPrintln(r(A))</code>	prints <code>r(fib(10,89))</code>
<code>fib(5,V),goalResult(GR),actorPrintln(r(GR))</code>	prints <code>r(executeInput(do([true],fib(10,89),...))</code>

3.8.2 Change the internal state .

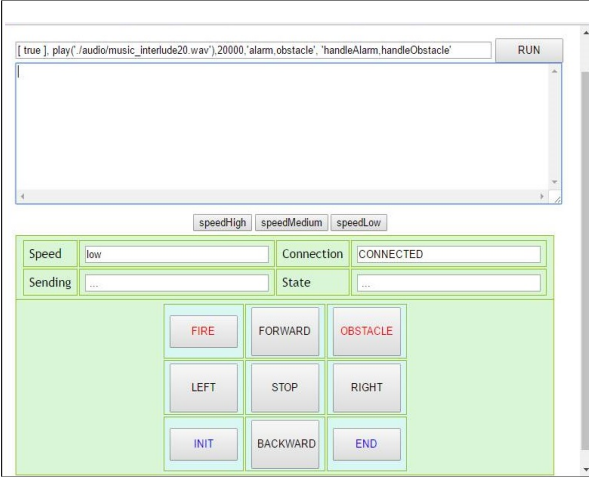
The *WorldTheory* associated with a robot defines also rules that allows an application designer to bind symbols to values and to add/remove rules:

<code>assign(x,3)</code>	set <code>x=3</code> , i.e. store the fact: <code>value(x,3)</code>
<code>getVal(x,V)</code>	binds V with the current value of x
<code>assign(ledstate,false)</code>	set fact: <code>value(ledstate,false)</code>
<code>inc(x,1,V)</code>	binds V to the result of <code>x+1</code> .
<code>assign(x,1),inc(x,3,V),actorPrintln(v(V))</code>	binds V to 4 and prints <code>v(4)</code> .
<code>assign(x,10),getVal(x,VX),actorPrintln(x(VX))</code>	prints <code>x(10)</code> .
<code>addRule(r1(a)),r1(X),actorPrintln(X)</code>	add the fact <code>r1/1</code> to the actor's <i>WorldTheory</i> and prints <code>a</code>

4 Human interaction with a Qactor

A human user can interact with a *Qactor* by using both a remote and a local GUI interface.

4.1 The (remote) Web interface



This web interface is automatically generated in the `srcMore` directory in a package associated with each *Context* when the `-httpserver` flag for a *Context* is set. It is implemented by a HTTP web-socket server working on port 8080.

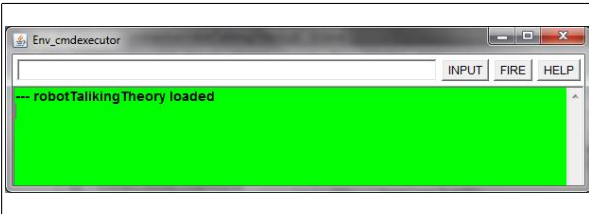
The RUN button at the top of the GUI allow us to ask the robot to execute actions, while the buttons at the bottom allows us to move the (basic) robot and send alarms.

The top-level part of the GUI can be used to inspect and change the state of the robot as represented in the robot's WorldTheory.

This interface emits the following events:

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

4.2 The (local) GUI user interface



This interface is automatically generated as part of the (Abstract) actor class when the `-g` flag for an Actor is set.

The INPUT button generates the event :

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

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

In Section 8 we will introduce an interpreter allowing human users to execute actions expressed as commands using a Web or a GUI actor interface.

5 Building qa models

A goal of **qa** is to help software developers in writing *executable specifications* during the early stages of software development with particular regard to *requirement analysis* and *problem analysis*. More precisely, the main outcome of the problem analysis phase should be the specification of the *logical architecture* of the system, obtained by following a sequence of steps:

- find the main subsystems and define the system *Contexts*;
- define the structure of the *Events* that can occur in the system;
- define the structure of the *Messages* exchanged by the actors;
- define the main *Actors* working in each *Context*;
- define the type of the *logical interaction* among the actors;
- define the *logical behaviour* of each actor according to the interaction constraints.

In several cases these specifications can be refined in the *project phase* by simply 'injecting' application-specific actions (see Section 6) so to reduce the global costs of software development.

5.0.1 Application designer and System designer. In the following, we will name *application designer* the software designer that works to fulfil the functional requirements of system while we will name *system designer* the designer that provides the run-time supports useful to face the business logic without too much involvement in technical problems related to distribution or to other relevant, recurrent, general problems in the application domain.

5.1 QActor software factory

QActor systems run upon a run-time support (built in the project *it.unibo.qactors*) based on the Akka actor system and is deployed (at this moment) in the 'library' *qa18Akka.jar*⁵. The *QActor* run-time support requires in its turn other open-source and custom libraries.

Thanks to the Xtext technology and Eclipse, the *QActor* language/metamodel is associated to a software factory that automatically generates all the files (Prolog theories) and the proper system configuration code, so to allow Application designers to focus on application logic.

⁵ another library *qactors17.jar* is provided for Android that does not support yet java8

6 User-defined actions in Prolog

The user can define application-specific actions in two main ways: (i) by using Java or some other (Java-compatible) programming language or (ii) by using tuProlog.

In this section we will explore how the application designer can exploit tuProlog in order to define business-specific operations.

6.0.1 Examples of unification . Let us recall here that tuProlog does implement occur check (and variable renaming):

Action examples		
<code>a(X,1)=a(1,Y)</code>	<code>'=(a(1,1),a(1,1))</code>	
<code>a(X,1)=a(1,X)</code>	<code>'=(a(1,1),a(1,1))</code>	
<code>a(X,2)=a(1,X)</code>	<code>a(_1,_2)</code>	//tuProlog rewrites variables (occur check)
<code>a(b(X,1),c(Y))=a(b(0,A),c(A))</code>	<code>'=(a(b(0,1),c(1)),a(b(0,1),c(1))</code>	
<code>a(b(X,1),c(Y))=Z</code>	<code>'=(a(b(_2,1),c(_1)),a(b(_2,1),c(_1)))</code>	
<code>a(b(X,1),c(Y))=a(A,B)</code>	<code>'=(a(b(_6,1),c(_1)),a(b(_6,1),c(_1)))</code>	

6.1 The demo and solve operation.

The qa language defines actions to solve Prolog goals:

```
1 Demo: "demo" goal=PHead ("onFailSwitchTo" plan=[Plan])?;
2 SolveGoal: "solve" goal=PHead duration=TimeLimit ("onFailSwitchTo" plan=[Plan])?;
```

In both cases, the result is represented by the fact `goalResult/1` in the actor *WorldTheory* (see Subsection 2.1.2). The duration of a `demo` action is set to 1 day (86400000 msec).

6.2 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 (0) a user-defined theory (stored in file `aTheory.pl`) and then (i) finds a Fibonacci number (plan `compute`), (ii) works with sensor data, for two times in the same way (plan `accessdata`) :

```
1 /*
2  * attheoryUsage.qa in project it.unibo.qactors
3  */
4 System attheoryUsage -testing
5 Context ctxTheoryUsage ip [ host="localhost" port=8049 ] -g cyan
6
7 QActor theoryusage context ctxTheoryUsage{
8   Plan init normal
9   /*0*/ demo consult("./aTheory.pl") onFailSwitchTo prologFailure ;
10    switchToPlan compute ;
11    switchToPlan accessdata ;
12    println( bye )
13   Plan compute resumeLastPlan
14    solve fib(7,V) time(1000);
15    [!? goalResult(G) ] println( G ) //prints fib(7,13)
16   Plan accessdata resumeLastPlan
17    println( "-----" ) ;
```

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

```

18 /*1*/ [ !? data(S,N,V) ] println( data(S,N,V) ) ;
19 /*2*/ [ !? validDistance(N,V) ] println( validDistance(N,V) ) ;
20 /*3*/ demo nearDistance(N,V) onFailSwitchTo prologFailure ;
21 /*4*/ [ !? goalResult(nearDistance(N,V)) ] println( warning(N,V) ) ;
22 /*5*/ demo nears(D) onFailSwitchTo prologFailure ;
23 /*6*/ [ !? goalResult(G) ] println( list(G) ) ;
24 /*
25     demo likes(X,jim) onFailSwitchTo prologFailure ;
26     [ !? goalResult(G) ] println( G ) ;
27     demo character(ulysses, X, Y) onFailSwitchTo prologFailure ;
28     [ !? goalResult(G) ] println( G ) ;
29     demo f(g(a, Z), Y) onFailSwitchTo prologFailure ;
30     [ !? goalResult(G) ] println( G ) ;
31 */
32 repeatPlan 1
33
34 Plan prologFailure
35     println("theoryusage has failed to solve a Prolog goal" )
36 }

```

Listing 1.7. aTheoryUsage.qa

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

```

1 /* =====
2 aTheory.pl in project it.unibo.qactors
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 %% likes(jane,X).
14 %% character(ulysses, Z, king(ithaca, achaeon)).
15 %% f(g(X, h(X, b)), Z).
16
17 initialize :- actorPrintln("initializing the aTheory ...").
18 :- initialization(initialize).

```

Listing 1.8. aTheory.pl

6.2.1 The initialization directive. The following directive:

```
:- initialization(initialize).
```

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

Thus, the output of the theoryusage actor is:

```

1 --- initializing the aTheory ...
2 --- fib(7,13)
3 --- "-----"
4 --- data(sonar,1,10)
5 --- validDistance(2,20)
6 --- warning(2,20)
7 --- nears([d(2,20),d(3,30)])
8 --- "-----"
9 --- data(sonar,1,10)
10 --- validDistance(2,20)
11 --- warning(2,20)
12 --- nears([d(2,20),d(3,30)])
13 --- bye

```


6.2.2 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 (for an example, see Subsection 6.8.3).

6.3 Using the actor in Prolog rules

The predefined rule `actorobj/1` unifies a given variable to a reference to the Java object that implements the actor associated with the current *WorldTheory*. In this way the application designer can access in Prolog to all the public methods of the actor.

For example, the following theory defines a rule (`dance`) to (simulate a) moves of the actor in some planned way:

```

1  /*
2  =====
3  actorDanceTheory.pl in project it.unibo.qactors
4  =====
5  */
6
7  dance :-
8      actorobj( Actor ),
9      actorPrintln(forward),
10     Actor <- delay(2000),
11     actorPrintln(left),
12     Actor <- delay(2000),
13     actorPrintln(right),
14     Actor <- delay(2000),
15     actorPrintln(backward),
16     Actor <- delay(2000).
17
18 initdance :- actorPrintln("initializing the ActorDanceTheory ...").
19 :- initialization(initdance).
```

Listing 1.9. actorDanceTheory.pl

The application designer can use the `dance` rule as a user-defined extension of the actor action-set:

```

1  /*
2  * dancer.qa in project it.unibo.qactors
3  */
4  System dancerSys -testing
5  Event endplay : endplay(X)
6  Event alarm : alarm(X)
7
8  Context ctxDancer ip [ host="localhost" port=8079 ] -g cyan -httpserver
9  EventHandler evh for alarm, endplay -print ;
10
11 QActor dancer context ctxDancer{
12     Plan init normal
13         demo consult("./actorDanceTheory.pl") onFailSwitchTo prologFailure ;
14         switchToPlan playMusic ;
15         switchToPlan compute ;
16         switchToPlan dance ;
17         println("Bye bye" )
18     Plan compute resumeLastPlan
19         demo fib(7,V) ;
20         [?? goalResult(X) ] println( X ) //prints fib(7,13)
21     Plan dance resumeLastPlan
22         solve dance time(12000) onFailSwitchTo prologFailure react event alarm -> handleAlarm ;
```

⁷ Remember from Subsection 2.1.1 that the fact `goalResult/1` is a 'singleton'.

```

23 [ ?? goalResult(X)] println(goalResult(X))
24 Plan playMusic resumeLastPlan
25   sound time(20000) file('./audio/music_interlude20.wav') answerEv endplay
26 Plan handleAlarm resumeLastPlan
27   sound time(1000) file('./audio/tada2.wav') ;
28   println("*** alarm ***")
29 Plan prologFailure
30   println("dancer has failed to solve a Prolog goal" )
31 }

```

Listing 1.10. dancer.qa

6.4 The operator actorOp

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

Here is an example that shows how execute methods that return primitive data and methods that return objects:

```

1 System actorOpdemo
2 Context ctxActorOpdemo ip [ host="localhost" port=8037 ] -g cyan
3 QActor qaactorop context ctxActorOpdemo {
4   Plan main normal
5     println("actionOpdemo STARTS " ) ;
6     switchToPlan testReturnPrimitiveData ;
7     switchToPlan testReturnPojo ;
8     println("actionOpdemo ENDS " )
9   Plan testReturnPrimitiveData resumeLastPlan
10    actorOp intToVoid(5) ;
11    [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
12    actorOp intToString(5) ;
13    [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
14    actorOp intToInt(5) ;
15    [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
16    actorOp floatToFloat(5) ; //qa does not allow to write floats
17    [ ?? actorOpDone( OP,R ) ] println( done(OP,R) )
18   Plan testReturnPojo resumeLastPlan
19     println("actionOpdemo testReturnPojo " ) ;
20     actorOp getDate ;
21     [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
22     demo actionResultOp( toInstant ) ;
23     [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
24     demo actionResultOp( getNano ) ;
25     [ ?? actorOpDone( OP,R ) ] println( done(OP,R) )
26 }

```

Listing 1.11. actorOpdemo.qa

The code written by the application designer is:

```

1 /* Generated by AN DISI Unibo */
2 /*
3  This code is generated only ONCE
4  */
5 package it.unibo.qaactorop;
6 import java.util.Calendar;
7 import java.util.Date;
8
9 import it.unibo.is.interfaces.IOutputEnvView;
10 import it.unibo.qactors.QActorContext;
11
12 public class Qaactorop extends AbstractQaactorop {
13   public Qaactorop(String actorId, QActorContext myCtx, IOutputEnvView outEnvView ) throws Exception{
14     super(actorId, myCtx, outEnvView);
15   }

```

```

16 public int intToInt( int n ) {
17     return n+1;
18 }
19 public Date getDate( ) {
20     Calendar rightNow = Calendar.getInstance();
21     Date d = rightNow.getTime();
22     return d;
23 }
24
25 public void intToVoid( int n ){
26     println( "    Java intToVoid " + n );
27 }
28
29 public String intToString( int n ) {
30     println( "    Java intToString " + n );
31     return ""+n+1;
32 }
33 public float floatToFloat( float n ) {
34     println( "    Java floatToFloat " + n );
35     return n/2;
36 }
37
38 }

```

Listing 1.12. Qaactorop.java

6.5 Rules at model level

Sometimes can be useful to express Prolog facts directly in the model specification, especially when these facts are used for configuration or action-selection purposes. The *Rules* option within a *QActor* allows us to define facts by using a subset of the Prolog syntax⁸

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

```

1  /*
2  * rulesInModel.qa in project it.unibo.qactors
3  */
4  System rulesInModel -testing
5  Event endplay : endplay(X)
6  Event alarm : alarm(X)
7
8  Context ctxRulesInModel ip [ host="localhost" port=8059 ]
9  EventHandler evh for endplay -print ;
10
11 QActor rulebasedactor context ctxRulesInModel -g yellow {
12     Rules{
13         shortSound(1,'./audio/tada2.wav').
14         fastSound(1,'./audio/any_commander3.wav',3000).
15         fastSound(2,'./audio/computer_complex3.wav',3000).
16         fastSound(3,'./audio/illogical_most2.wav',2000).
17         fastSound(4,'./audio/computer_process_info4.wav',4000).
18         longSound(1,'./audio/music_interlude20.wav',15000).
19         longSound(1,'./audio/music_dramatic20.wav',20000).
20     }
21     Plan init normal
22     [ !? longSound(1,F,T)] sound time(T) file(F) answerEv endplay ;
23     delay time(300) ;
24     [ !? fastSound(1,F,T)] sound time(T) file(F) ;
25     [ !? fastSound(4,F,T)] sound time(T) file(F) ;
26     [ !? fastSound(2,F,T)] sound time(T) file(F) ;
27     [ !? longSound(1,F,T)] sound time(T) file(F) react event alarm -> handleAlarm ;
28     [ !? fastSound(3,F,T)] sound time(T) file(F) ;
29     println("Bye bye" )

```

⁸ The extension of this option with full Prolog syntax is a work to do.

```

30 Plan handleAlarm resumeLastPlan
31     println("rulebasedactor hhandleAlarm" )
32 Plan prologFailure
33     println("rulebasedactor has failed to solve a Prolog goal" )
34 }

```

Listing 1.13. rulesInModel.qa

6.6 From Prolog to Java again

Thanks to tuProlog features, the application designer can define rules that can exploit Java to perform the required operation. For example, suppose that we have to solve the following problem:

Build a system that starts by playing a soft music in background. Then the system shows to the user a graphical interface to allow the selection of a sound file (wav). When the user has selected a file, the graphical interface disappears and the system plays the selected (short) sound over the music in background.

The application designer can define a *project model* like the following one:

```

1  /*
2  * userSelect.qa in project it.unibo.qactors
3  */
4  System userSelect -testing
5  Context ctxUserSelect ip [ host="localhost" port=8079 ] -g cyan
6
7  QActor soundselector context ctxUserSelect{
8      Plan init normal
9          demo consult("./userTheory.pl") onFailSwitchTo prologFailure ; //(1)
10 //      sound time(20000) file('./audio/music_interlude20.wav') answerEv endplay;
11      switchToPlan playMusic ;
12      println("Bye bye" )
13
14      Plan playMusic resumeLastPlan
15          [ !? select(FILE) ] sound time(3000) file(FILE)
16
17      Plan prologFailure
18          println("mockbehavior has failed to solve a Prolog goal" )
19 }

```

Listing 1.14. userSelect.qa

6.6.1 Guards as problem-solving operation. In the plan playMusic above, let us consider the sentence:

```
[ !? select(FILE) ] sound time(3000) file(FILE) ;
```

When the guard evaluates to **true**, it should bind (the variable) **FILE** to the file-path selected by the user.

6.6.2 The user-defined select/1 operation. To allow the **select/1** guard to become part of the problem-solution rather than just a test⁹, the application designer writes a proper rule in the tuProlog file **userTheory.pl** loaded into the actor's knowledge base by the **init** plan (at (1)).

⁹ Of course the application designer must assure that a guard computation always terminates and should avoid to write computationally heavy guards.

```

1 testConsult :- actorPrintln("userTheory works").
2
3 mortal(X) :- human(X).
4 human(socrate).
5 human(platone).
6
7 /*
8 Expressions
9 */
10 testOnExp(Z) :- (2 + 3 * 4 / 1) = Z.
11
12 /*
13 A.2.1
14 */
15 character(priam, iliad).
16 character(hecuba, iliad).
17 character(achilles, iliad).
18 character(agamemnon, iliad).
19 character(patroclus, iliad).
20 character(hector, iliad).
21 character(andromache, iliad).
22 character(rhesus, iliad).
23 %% character(ulysses, iliad).
24 %% character(menelaus, iliad).
25 %% character(helen, iliad).
26
27 character(ulysses, odyssey).
28 character(penelope, odyssey).
29 character(telemachus, odyssey).
30 character(laertes, odyssey).
31 character(nestor, odyssey).
32 character(menelaus, odyssey).
33 character(helen, odyssey).
34 character(hermione, odyssey).
35
36 male(priam).
37 male(achilles).
38 male(agamemnon).
39 male(patroclus).
40 male(hector).
41
42 male(rhesus).
43 male(ulysses).
44 male(menelaus).
45 male(telemachus).
46 male(laertes).
47 male(nestor).
48
49 female(hecuba).
50 female(andromache).
51 female(helen).
52 female(penelope).
53
54
55 father(priam, hector).
56 father(laertes, ulysses).
57 father(atreus, menelaus).
58 father(menelaus, hermione).
59 father(ulysses, telemachus).
60
61 mother(hecuba, hector).
62 mother(penelope, telemachus).
63 mother(helen, hermione).
64
65 king(ulysses, ithaca, achaeon).
66 king(menelaus, sparta, achaeon).
67 king(nestor, pylos, achaeon).
68 king(agamemnon, argos, achaeon).
69 king(priam, troy, trojan).
70 king(rhesus, thrace, trojan).

```

```

71 character(priam, iliad, king(troy, trojan)).
72 character(ulysses, iliad, king(ithaca, achaeon)).
73 character(menelaus, iliad, king(sparta, achaeon)).
74
75
76
77
78 /*
79 -----
80 A.2.5
81 -----
82 */
83 % Is the king of LAND also a father? i
84 kingFather(X, LAND) :- king(X, LAND, Y), father(X, _).
85
86 /*
87 -----
88 A.2.7
89 -----
90 */
91 son(X, Y) :- father(Y, X), male(X).
92 son(X, Y) :- mother(Y, X), male(X).
93
94 parent(X, Y) :- mother(X, Y).
95 parent(X, Y) :- father(X, Y).
96
97 grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
98
99 ancestor(X, Y) :- parent(X, Y).
100 ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
101
102 /*
103 -----
104 A.2.7
105 -----
106 */
107 unify(T1, T2, T1) :-
108     actorPrintln(unify(T1, T2)),
109     T1 = T2.
110
111 unify1(X, Y, Z) :- character(ulysses, Z, king(ithaca, achaeon)) = character(ulysses, X, Y).
112
113 /*
114 -----
115 A.5,3
116 -----
117 */
118 p(X) :- q(X), r(X).
119 q(a).
120 q(b).
121 r(b).
122 r(c).
123
124 resolve(X) :- X,
125     actorPrintln(X),
126     fail.
127 resolve(X).
128
129 /*
130 -----
131 A.13.1
132 -----
133 */
134 hero(ulysses).
135 heroin(penelope).
136 daughter(X, Y) :-
137     mother(Y, X),
138     female(X).
139 daughter(X, Y) :-
140     father(Y, X),

```

```

141     female(X).
142
143     /*
144     -----
145     A.15.2
146     -----
147     */
148
149     link(r1, r2).
150     link(r1, r3).
151     link(r1, r4).
152     link(r1, r5).
153     link(r2, r6).
154     link(r2, r7).
155     link(r3, r6).
156     link(r3, r7).
157     link(r4, r7).
158     link(r4, r8).
159     link(r6, r9).
160
161     s(X, Y) :- link(X, Y).
162     s(X, Y) :- link(Y, X).
163
164     minotaur(r8).
165
166     labyrinth(X) :- minotaur(X).
167
168     %% depth_first_search(+Node, -Path)
169     depth_first_search(Node, Path) :-
170         depth_first_search(Node, [], Path).
171     %% depth_first_search(+Node, +CurrentPath, -FinalPath)
172     depth_first_search(Node, Path, [Node | Path]) :-
173         labyrinth(Node).
174     depth_first_search(Node, Path, FinalPath) :-
175         s(Node, Node1),
176         not member(Node1, Path),
177         depth_first_search(Node1, [Node | Path], FinalPath).
178
179
180     /*
181     -----
182     TuProlog
183     -----
184     */
185     selectFile( FileName ) :-
186         %% ACCESS TO JAVA OBJECT INSTANCES
187         java_object('javax.swing.JFileChooser', [], Dialog),
188         Dialog <- showOpenDialog(_),
189         Dialog <- getSelectedFile returns File,
190         File <- getPath returns Path,
191         %% ACCESS TO CLASS STATIC OPERATION
192         class("it.unibo.qactors.QActorUtils") <- adjust( Path ) returns FileName.
193
194     welcome :- actorPrintln("welcome from userTheory.pl").
195     :- initialization(welcome).

```

Listing 1.15. userTheory.pl

In this example the problem can be in large part solved by making reference to objects provided by the standard Java library. The class *it.unibo.utils.Utils* is introduced to solve in Java (rather than in Prolog) a string-substitution problem.

In fact, the `select/1` rule first creates an instance of the Java class *javax.swing.JFileChooser* to show a dialog window to the user. Afterwards, it uses the instance referred by the variable `Dialog` to bind the `File` variable to the file selected by the user and then it uses the object referenced by `File` to bind `Path` to a file-path string. Finally, the rule calls the *static* method `adjust(String path)` of the user-defined class *it.unibo.utils.Utils* to replace all the backslashes with `"/"`.

```

1  /*
2  * LEAVE HERE (line 63) since referenced by Latex
3  */

```

Listing 1.16. QActorUtils.java

6.7 Workflow

The definition of application actions in **tuProlog** is particularly useful during the requirement and problem analysis, since it allows us to introduce in *declarative* style *executable* actions. This promotes *fast prototyping* of qactor-systems, using **tuProlog** as a 'glue' between high-level models (expressed in **qa**) and more detailed operations written in Java.

The workflow of Subsection 5 can be extended with the following steps:

- make reference to the Java classes that constitute the *domain model* expressed as conventional (POJO) objects ;
- use the **actorOp** specification to call Java operations;
- define a set of Prolog rules, each providing (the specification of) a new *operation* to be called in some specific state (**Plan**) of the actor model.

Further examples of this approach will be given in the following.

6.8 Examples of problem solving with **tuProlog**

Let us consider the following model:

```

1  /*
2  * theoryExample.qa in project it.unibo.qactors
3  */
4  System theoryExample -testing
5  Context ctxTheoryExample ip [ host="localhost" port=8099 ]
6
7  QActor theexample context ctxTheoryExample{
8      Plan init normal
9          demo consult("./exampleTheory.pl") onFailSwitchTo prologFailure ;
10         demo consult("./srcMore/it/unibo/ctxTheoryExample/theoryexample.pl") onFailSwitchTo prologFailure ;
11         switchToPlan configuration;
12         switchToPlan family;
13         /*(1)*/ solve assign(n,1) time(0) onFailSwitchTo prologFailure ;
14         switchToPlan accessData ;
15         println( bye )
16         Plan configuration resumeLastPlan
17             demo showSystemConfiguration onFailSwitchTo prologFailure
18         Plan family resumeLastPlan
19             demo son(X,haran) onFailSwitchTo prologFailure ;
20             [ !? goalResult(son(X,haran)) ] println(X) else println(noson(haran));
21             demo daughters(X,haran) onFailSwitchTo prologFailure ;
22             [ !? goalResult(daughters(X,haran)) ] println(X)
23         Plan accessData resumeLastPlan
24             [ !? nextdata(sonar,n,N) ] println( data(sonar,N,V) ) else endPlan "no more data" ;
25             repeatPlan
26         Plan prologFailure
27             println("theexample has failed to solve a Prolog goal" )
28     }

```

Listing 1.17. theoryExample.qa

This model defines a set of plans, each designed to solve a different problem:

- **configuration**: show to the user the configuration of the system.
- **family**: given a knowledge base over a domain (a family) find some relevant information (e.g. a son of a person or all the sons of a person).
- **accessData**: access to sensor data represented in an index-based form by an iterative computation.

The user-defined theory (stored in file `exampleTheory.pl`) is:

```

1  /*
2  =====
3  exampleTheory.pl
4  =====
5  */
6  /*
7  -----
8  Show system configuration
9  -----
10 */
11 showSystemConfiguration :-
12     actorPrintln(' The system is composed of the following contexts'),
13     getTheContexts(CTXS),
14     showElements(CTXS),
15     actorPrintln(' and of the following actors'),
16     getTheActors(A),
17     showElements(A).
18
19 showElements([]).
20 showElements([C|R]):-
21     actorPrintln( C ),
22     showElements(R).
23
24 /*
25 -----
26 Find system configuration
27 -----
28 */
29 getTheContexts(CTXS) :-
30     findall( context( CTX, HOST, PROTOCOL, PORT ), context( CTX, HOST, PROTOCOL, PORT ), CTXS).
31 getTheActors(ACTORS) :-
32     findall( qactor( A, CTX ), qactor( A, CTX ), ACTORS).
33
34 /*
35 -----
36 Family relationship
37 -----
38 */
39 father( abraham, isaac ).
40 father( haran, lot ).
41 father( haran, milcah ).
42 father( haran, yiscah ).
43 male(isaac).
44 male(lot).
45 female(milcah).
46 female(yiscah).
47
48 son(S,F):-father(F,S),male(S).
49 daughter(D,F):- father(F,D),female(D).
50
51 sons(SONS,F) :- findall( son( S,F ), son( S,F ), SONS).
52 daughters(DS,F) :- findall( d( D,F ), daughter( D,F ), DS).
53
54 /*
55 -----
56 Indexed knowledge
57 -----
58 */
59 data(sonar, 1, 10).
60 data(sonar, 2, 20).
61 data(sonar, 3, 30).
62 data(sonar, 4, 40).

```

```

63 data(sonar,length,4).
64
65 data(distance, 1, 100).
66 data(distance, 2, 200).
67 data(distance, 3, 300).
68 data(distance,length,3).
69
70 nextdata( Sensor, I , V):-
71     data(Sensor,length,N),
72     value(I,V),
73     inc(I),
74     V =< N.
75 /*
76 -----
77 Imperative
78 -----
79 */
80 assign( I,V ):-
81     ( retract( value(I,_) ),!; true ),
82     assert( value( I,V ) ).
83 inc(N):-
84     value(N,X),
85     Y is X + 1,
86     assign( N,Y ).
87
88 /*
89 -----
90 initialize
91 -----
92 */
93 initialize :- actorPrintln("initializing the exampleTheory ...").
94 :- initialization(initialize).

```

Listing 1.18. exampleTheory.pl

We note that:

6.8.1 configuration. moreexamples To solve the configuration problem, the application designer loads the generated theory stored in file `../.../it.unibo.qactors.tests/srcMore/it/unibo/ctxTheoryExanple/t`.

```

1 %=====
2 % Context ctxTheoryExanple SYSTEM-configuration: file it.unibo.ctxTheoryExanple.theoryExanple.pl
3 %=====
4 context(ctxtheoryexanple, "localhost", "TCP", "8099" ).
5 %%%
6 qactor( thexample , ctxtheoryexanple, "it.unibo.thexample.MsgHandle_Thexample" ). %%store msgs
7 qactor( thexample_ctrl , ctxtheoryexanple, "it.unibo.thexample.Thexample" ). %%control-driven
8 %%%
9 %%%

```

Listing 1.19. theoryexanple.pl

Then the problem is completely solved (output included) by the tuProlog rule *showSystemConfiguration*.

This example shows that choice to represent the system configuration as a theory allows applications to dynamically inspect the system and to perform actions that could depend on such a configuration.

6.8.2 family. To solve the family problem, the application designer does use the `solve` operation to perform queries to the family knowledge-base. The `findall/3` predicate is very useful to collect a set of solution into a list.

6.8.3 accessData. To solve the family problem, the application designer has introduced some 'imperative' programming style with the rules `assign/3` and `inc/1`. The rule `nextdata/3` is used to access a different `data` value at each iteration of plan `accessData`. The iterations terminate when the `endplan` clause is executed, i.e. when the rule (guard) `nextdata/3` fails.

6.8.4 output. The output of the system execution is:

```
1  *** ctxtheoryexample startUserStandardInput ***
2  Starting the actors ...
3  --- initializing the exampleTheory ...
4  --- The system is composed of the following contexts
5  --- context(ctxtheoryexample,localhost,'TCP','8099')
6  --- and of the following actors
7  --- qactor(theexample,ctxtheoryexample)
8  --- lot
9  --- [d(milcah,haran),d(yiscah,haran)]
10 --- data(sonar,1,10)
11 --- data(sonar,2,20)
12 --- data(sonar,3,30)
13 --- data(sonar,4,40)
14 --- no more data
15 --- bye
```

7 Advanced Actions (observable, timed, reactive)

In Subsection 2.1 we said that an **action** is an activity that must always terminate. The effects of an action can be perceived as changes in the logical or in the physical actor environment.

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 a 'pure computational action', its effects can be perceived (as a result of type **long**) when the action returns the control to the caller.

7.1 Asynchronous Observable Actions

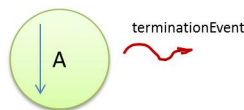
Let us introduce now a new idea of an action¹⁰, with the following properties:

1. the action performs its work in its own thread of control;
2. the action emits an *event* when terminates.

We can define an **Observable Action** is an asynchronous action that emits a termination event¹¹. Moreover:

- the action can be activated in two different ways: synchronous mode and asynchronous mode;
- a Observable Action activated in *synchronous* mode blocks the activator until the action is completed;
- a Observable Action activated in *asynchronous* mode returns immediately the control to the activator;
- the *result* of a Observable Action is an object of some type T.

Thus, an Observable Action works in parallel with its activator that can decide to wait for the termination of the action. The action effects (results) can be perceived by means of an operation (if it is defined) that gives (when the action is terminated) the result of the action or by means of an event-driven or an event-based behaviour.



¹⁰ The growing demand for asynchronous, event-driven, parallel and scalable systems demand for new abstractions.

¹¹ Usually, the name of the termination event starts with the prefix "local_", in order to avoid event propagation over the network.

7.2 The class `ActionObservableGeneric`

The generic, abstract class `ActionObservableGeneric<T>` provides an implementation support for Observable Actions. It implements the following interface ¹²

```
1 package it.unibo.qactors.action;
2 import java.util.concurrent.Callable;
3 import java.util.concurrent.Future;
4
5 public interface IObservableActionGeneric<T> extends Callable<T> {
6     public T execSynch() throws Exception;
7     public Future<T> execASynch() throws Exception;
8     public String getTerminationEventId();
9     public long getExecTime();
10    public String getResultRep();
11    public String getName();
12 }
```

Listing 1.20. `IObservableActionGeneric<T>`

The meaning of some important operations is reported in the following table:

<code>T execSynch()</code>	executes the action and waits for termination
<code>Future<T> execASynch()</code>	starts the action as an asynchronous operation
<code>String getTerminationEventId()</code>	returns the name of the termination event
<code>long getExecTime()</code>	returns the execution time of the action
<code>String getResultRep()</code>	returns (a representation of) the result of the action

7.2.1 `Callable<T>` interface.

Since an Observable Action is a `java.util.concurrent.Callable<T>`, it must define the operation `<T> call()`. The `Callable` interface is similar to `Runnable`, in that both are designed for classes whose instances are potentially executed by another thread. A `Runnable`, however, *does not return a result* and cannot throw a checked exception.

7.2.2 `Future<T>` interface .

The operation `activate` starts the execution of the action (see Subsection 7.2.3) and returns an object that must implement the interface `java.util.concurrent.Future<T>`.

In Java, a `Future` represents the result of an asynchronous computation. It exposes methods allowing a client to monitor the progress of a task being executed by a different thread. Therefore a `Future` object can be used to check the status of a `Callable` and to retrieve the result from the `Callable`. More specifically:

- Check whether a task has been completed or not.
- Cancel a task.
- Check whether task was cancelled or complete normally.

In programming languages, the *future* concept is also known as *promises* or *delays*.

¹² The code is included in the file `qactor18.jar`.

7.2.3 execAsynch() operation.

The operation `execAsynch()` starts the action as an asynchronous operation by using the class `java.util.concurrent.Executors`:

```
1 public Future<T> execAsynch() throws Exception{
2     Future<T> fResult =
3         EventPlatformKb.manyThreadexecutor.submit(this); //invokes T call()
4     return fResult;
5 }
```

7.2.4 Executors class.

Since working with the `Thread` class can be very tedious and error-prone, the *Concurrency API* has been introduced back in 2004 with the release of `Java 5`. The API is located in package `java.util.concurrent` and contains many useful classes for handling concurrent programming, including the concept of an *ExecutorService* as a higher level replacement for working with threads directly. Executors are capable of running asynchronous tasks and typically manage a pool of threads.

7.2.5 execSynch() operation.

The operation `execSynch()` activates the action and forces the caller to wait for the termination of the action execution.

```
1 @Override
2 public T execSynch() throws Exception {
3     Future<T> fResult = execASynch();
4     T fut = fResult.get(); //forces the caller to wait
5     return fut;
6 }
```

7.2.6 T call() operation.

The operation `T call()` is the entry point for the `Executor` and is defined as a sequence of internal operations that starts by taking the current time and ends by calculating the execution time and by emitting the action termination event.

```
1 /* Entry point for the Executor */
2 @Override
3 public T call() throws Exception {
4     startOfAction();
5     execTheAction();
6     result = endActionInternal();
7     return result;
8 }
```

7.2.7 startOfAction() operation.

The operation `startOfAction` takes the current time and retains a reference to the current `Thread`:

```
1 protected Thread myself;
2 protected void startOfAction() throws Exception{
3     tStart = Calendar.getInstance().getTimeInMillis();
4     myself = Thread.currentThread();
5 }
```

7.2.8 endActionInternal() operation.

The operation `endActionInternal` is executed when the action is terminated; it evaluates the action execution time and emits the termination event with payload `result(RES,EXEETIME)`:

The operation `endActionInternal`

```
1  /* Calculate action execution time and emit the termination event */
2  protected T endActionInternal() throws Exception{
3      evalDuration();
4      T res = endOfAction();
5      if(terminationEvId != null && terminationEvId.length()>0) {
6          emitEvent( terminationEvId, res.toString() );
7      }
8      return res;
9  }
10 protected void emitEvent(String event, String res) throws Exception{
11     QActorUtils.raiseEvent((it.unibo.qactors.QActorContext) ctx, getName(), event, outS);
12 }
```

7.2.9 execTheAction() and endOfAction() operations.

The operations `execTheAction` and `endOfAction` are declared abstract in the class *ActionObservableGeneric<T>*.

```
1  /* TO BE DEFINED BY THE APPLICATION DESIGNER */
2  protected abstract void execTheAction() throws Exception;
3  protected abstract T endOfAction() throws Exception;
4  public abstract String getResultRep();
```

These operations must be defined by the application designer with the following semantics:

<code>execTheAction</code>	defines the 'businnes logic' of the action
<code>endOfAction</code>	returns the main result of the action
<code>getResultRep</code>	returns the representation of the result

7.2.10 Fibonacci as Observable Action.

The `fibonacci` computation of Subsection 8.1 is defined in the following as a Observable Action that takes at construction time a goal of the form `fibonacci(N,V)` where `N` is the fibonacci number to evaluate and `V` is the (variable that denotes the) result.

```
1  package it.unibo.actionobsexecutor;
2  import alice.tuprolog.Struct;
3  import alice.tuprolog.Term;
4  import it.unibo.is.interfaces.IOutputEnvView;
5  import it.unibo.qactors.QActorContext;
6  import it.unibo.qactors.action.ActionObservableGeneric;
7
8  public class ActionObservableFibonacci extends ActionObservableGeneric<String> {
9      private String goalTodo = "fibonacci(25,V).";
10     private String myresult = "unknown";
11     private int n;
12     public ActionObservableFibonacci(String name, QActorContext ctx, String goalTodo,
13         String terminationEvId, IOutputEnvView outView) throws Exception {
14         super(name, ctx, terminationEvId, outView);
15         this.goalTodo = goalTodo;
16         println(" %% ActionObservableFibonacci CREATED with terminationEvId=" + terminationEvId );
17     }
18     @Override
19     public void execTheAction() throws Exception {
20         myresult = fibonacci( goalTodo );
21         println(" %% ActionObservableFibonacci execTheAction myresult=" + myresult );
22     }
```

```

23     protected String fibonacci( String goalTodo ){
24         Struct st = (Struct) Term.createTerm(goalTodo);
25         n = Integer.parseInt( ""+st.getArg(0) );
26         return ""+fibonacci(n);
27     }
28     protected long fibonacci( int n ){
29         if( n == 1 || n == 2 ) return 1;
30         else return fibonacci(n-1) + fibonacci(n-2);
31     }
32     @Override
33     public String getResultRep() {
34         return "fibo("+n+", "+this.myresult+"";
35     }
36     @Override
37     protected String endOfAction() throws Exception {
38         return this.myresult;
39     }
40 }

```

Listing 1.21. ActionObservableFibonacci<T>

7.2.11 Experiments on Fibonacci as Observable Action.

The project *it.unibo.qactors.tests* includes the model `actionObsDemo.qa` that shows the effects of the action in the different execution modes:

```

1  /*
2   * actionObsDemo.qa
3   */
4  System actionObsDemo
5  Event run : args(SYNCH,WITHEVH)
6
7  Context ctxActionObsDemo ip [ host="localhost" port=8071 ]
8
9  QActor actionobsexecutor context ctxActionObsDemo -g cyan{
10     Rules{
11         actiontodo( fibo(41,V) ).
12     }
13     Plan init normal
14         println(" *** Fibonacci as Observable Action *** ");
15         [ !? actiontodo(GOAL) ] actorOp createApplicationInterface(GOAL); //emit run
16         switchToPlan work
17     Plan work
18         sense time(600000) run -> continue ;
19         onEvent run : args( true, WITHEVH ) -> actorOp runSynch(WITHEVH) ;
20         onEvent run : args( false,WITHEVH ) -> actorOp runASynch(WITHEVH) ;
21         repeatPlan
22     }

```

Listing 1.22. actionObsDemo.qa



7.3 Timed actions

In several situations, an application designer could activate actions that must always terminate within a prefixed amount of time. In Subsection 2.1.5 we have introduced the idea of *timed action*.

We can define an **Timed Action** as an *Observable Action* whose execution time T satisfies the constraint $T \leq \text{DURATION}$, where DURATION is a prefixed amount of time¹³. Moreover:

- the action can be activated in two different ways: *synchronous mode* and *asynchronous mode*;
- a Timed Action activated in **synchronous** mode: *i*) blocks the activator until the action is completed and *ii*) emits a termination event when it ends its work;
- a Timed Action activated in **asynchronous** mode: *i*) returns immediately the control to the activator, and *ii*) emits a termination event when it ends its work;
- a Timed Action can be **suspended** (interrupted) even if it not already terminated. A Timed Action is called **recoverable** if the action can resume its work after a suspension.

Thus, a Timed Action works in its own thread of control; its activator can decide to wait for the termination of the action (*synchronous mode* activation) or to continue its work. In case of *asynchronous mode* activation, an actor interested in the application result of the action can look at the *termination event* emitted by the action. An actor can also suspend an action, and, if the action is recoverable, it can continue its execution later¹⁴.

Timed Actions implement the following interface:

```
1 package it.unibo.qactors.action;
2 import it.unibo.contactEvent.interfaces.IEventItem;
3 import it.unibo.qactors.QActorUtils;
4
5 public interface IActorAction extends IObservableActionGeneric<String>{
6     public int getMaxDuration();
7     public boolean isSuspended();
8     public IEventItem getInterruptEvent( );
9     public void setMaxDuration(int d);
10    public final String endBuiltinEvent = QActorUtils.locEvPrefix+"end";
11    public final boolean suspendPlan = false;
12    public final boolean continuePlan = true;
13    public enum ActionExecMode{
14        synch(0, "synch"),
15        asynch(1, "aynch");
16        private int value;
17        private String name;
18        private ActionExecMode(int value, String name) {
19            this.value = value;
20            this.name = name;
21        }
22    }
23 }
```

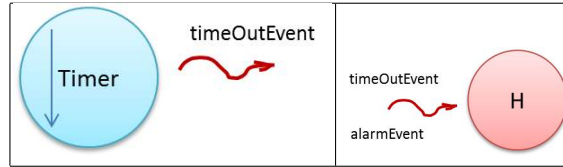
Listing 1.23. IActorAction<T>

7.3.1 ActorTimedAction class.

The class `ActorTimedAction` implements the concept of Timed Action by building a 'subsystem' composed of the action and other two components: a *timer* and a *event handler*:

¹³ Usually, T and DURATION are expressed in millisecs.

¹⁴ This kind of behaviour is useful for example when a robot must execute a move for some time, while being able to reacts to alarms.



Let us report here the code related to action construction:

```

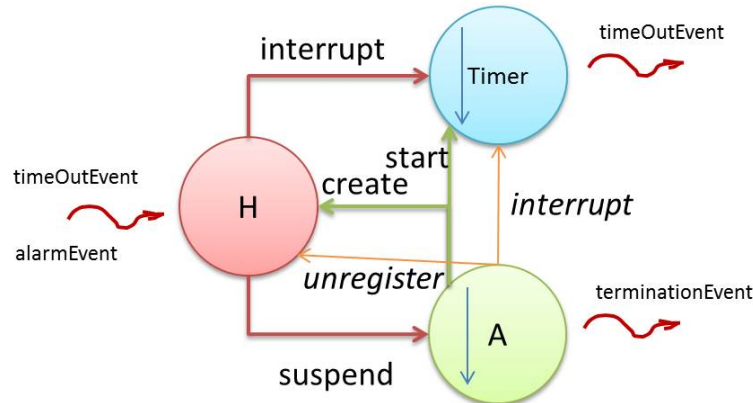
1 package it.unibo.qactors.action;
2 import java.util.concurrent.Callable;
3 import akka.actor.ActorRef;
4 import it.unibo.contactEvent.interfaces.IEventItem;
5 import it.unibo.is.interfaces.IOutputEnvView;
6 import it.unibo.qactors.ActionRegisterMessage;
7 import it.unibo.qactors.QActorContext;
8 import it.unibo.qactors.QActorUtils;
9 import it.unibo.qactors.akka.QActor;
10 import it.unibo.qactors.platform.EventPlatformKb;
11
12 public abstract class ActorTimedAction extends ActionObservableGeneric<String> implements IActorAction{
13     protected ActionTimerWatch timeWatch ;
14     protected String[] alarms ;
15     protected String actionTimedResult = "unknown";
16     protected String toutevId = "";
17     protected String suspendevent = null;
18     protected IEventItem currentEvent;
19     protected int maxduration;
20     protected boolean workingActionSynchFlag = true;
21     private Thread localExecutorThread ;
22     protected boolean cancompensate;
23     protected boolean suspended = false;
24     protected int timeRemained = 0;
25     protected QActor myactor;
26     /*
27     * CONSTRUCTION
28     */
29     public ActorTimedAction(String name, QActor actor, QActorContext ctx, boolean cancompensate,
30         String terminationEvId, String[] alarms,
31         IOutputEnvView outEnvView, int maxduration) throws Exception {
32         super(name, ctx, terminationEvId, outEnvView );
33         this.cancompensate = cancompensate;
34         this.maxduration = maxduration;
35         this.alarms = alarms;
36         myactor = actor;
37     }
38     protected void initActorTimedAction() throws Exception{ //called at startOfAction
39         toutevId = QActorUtils.getNewName(getName()+"tout");
40         //REGISTER THE ACTION
41         registerTheAction();
42         //Create an actor timer related to the action
43         String tough = QActorUtils.getNewName("toutEvh");
44         int timeExtension = 0;
45         timeWatch = new ActionTimerWatch(tough, outEnvView,maxduration+timeExtension,this);
46     }
47
48     protected abstract String getApplicationResult() throws Exception;
49     protected abstract Callable<String> getActionBodyAsCallable();

```

Listing 1.24. ActorTimedAction

The operation *getActionBodyAsCallable* must be defined by the Application designer so to return a *Callable<T>* object that is run within a new Thread, in parallel with a Timer. The Timer is created in a specialized version of the operation *startOfAction*, while the *Callable<T>* is activated in a specialized version of the operation *execTheAction*.

The architecture of the 'subsystem' built by the *initActorTimedAction* operation is shown in the following picture:



The picture shows that the handler H can be activated by a `timeOutEvent` or by an `alarmEvent`. The case of `alarmEvent` is related to the case of reactive actions, introduced in Subsection 7.4.

7.3.2 Fibonacci as a Timed Action.

The `fibonacci` computation of Subsection 8.1 is defined in the following as a Timed Action that takes at construction time a goal of the form `fib(N,V)` where `N` is the fibonacci number to evaluate and `V` is the (variable that denotes the) result.

```

1 package it.unibo.actiontimedexecutor;
2 import java.util.concurrent.Callable;
3 //import java.util.concurrent.Future;
4
5 import alice.tuprolog.Struct;
6 import alice.tuprolog.Term;
7 import it.unibo.is.interfaces.IOutputEnvView;
8 import it.unibo.qactors.QActorContext;
9 import it.unibo.qactors.action.ActorTimedAction;
10 import it.unibo.qactors.akka.QActor;
11
12 public class ActionFibonacciTimed extends ActorTimedAction{
13     private int n = 0;
14     private String myresult="going";
15
16     public ActionFibonacciTimed( QActor actor, QActorContext ctx, IOutputEnvView outEnvView,int n ) throws Exception {
17         this("fib",actor,ctx,n,false,"endFib", new String[] {}, outEnvView,6000000 );
18     }
19
20     public ActionFibonacciTimed(String name, QActor actor, QActorContext ctx, int n, boolean cancompensate,
21         String terminationEvId, String[] alarms, IOutputEnvView outView,
22         int maxduration) throws Exception {
23         super(name, actor, ctx, cancompensate, terminationEvId, alarms, outView, maxduration);
24         this.n = n;
25         println("%%% ActionFibonacciTimed CREATED " + name + " n=" + n + " tout=" + maxduration
26             + " terminationEvId=" + terminationEvId);
27     }
28
29
30     @Override
31     protected Callable<String> getActionBodyAsCallable() {
32         return new Callable<String>(){
33             @Override
34             public String call() throws Exception {
35                 myresult = ""+fibonacci(n);
36                 return myresult;
37             }
38         };

```

```

39 }
40 protected String fibonacci( String goalTodo ) throws Exception{
41     Struct st = (Struct) Term.createTerm(goalTodo);
42     n = Integer.parseInt( ""+st.getArg(0) );
43     return ""+fibonacci(n);
44 }
45 protected long fibonacci( int n ) throws Exception{
46     if( n == 1 || n == 2 ) return 1;
47     else return fibonacci(n-1) + fibonacci(n-2);
48 }
49 @Override
50 public String getApplicationResult() throws Exception {
51     if( this.suspendevent == null ){
52         return "fibo("+n+",val("+this.myresult+"),exectime("+this.getExecTime()+"))";
53     }else{
54         return "fibo("+suspendevent+",exectime("+this.getExecTime()+"))";
55     }
56 }
57 }

```

Listing 1.25. ActionFibonacciTimed

7.3.3 Experiments on Fibonacci as Timed Action.

The project *it.unibo.qactors.tests* includes the model `actionTimedDemo.qa` that shows the effects of the action in the different execution modes:

```

1  /*
2  * actionTimedDemo.qa
3  */
4  System actionTimedDemo
5  Event run : args(SYNCH,WITHEVH)
6  Context ctxActionTimedDemo ip [ host="localhost" port=8077 ]
7
8  QActor actiontimedexecutor context ctxActionTimedDemo -g cyan{
9      Rules{
10         actiontodo( fibo(41,V), 2000 ).
11     }
12     Plan init normal
13         println(" *** Fibonacci as Timed Action *** ");
14         [ !? actiontodo(GOAL, TIME) ] actorOp createApplicationInterface(GOAL, TIME); //emit run
15         switchToPlan waitEvents
16     Plan waitEvents
17         sense time(600000) run -> continue ;
18         onEvent run : args( true,WITHEVH ) -> actorOp runSynch(WITHEVH) ;
19         onEvent run : args( false,WITHEVH ) -> actorOp runASynch(WITHEVH) ;
20         repeatPlan
21     }

```

Listing 1.26. actionTimedDemo.qa



7.4 Reactive actions

We can define a **Reactive Action** as a *Timed Action* that can be suspended (interrupted) by the occurrence of events.

The occurrence of an event E that suspends the execution of a reactive action RA in a plan P, gives raise to the execution of a plan EP associated to the event E. The plan EP, once terminated, can terminate the behaviour of the Actor or resume the plan P, that will be continue from the suspension point of the action RA, if it is recoverable, or from the action that follows RA in P.

7.4.1 AsyncActionResult class.

The **result** of a Reactive Action is an object of type `AsyncActionResult` that wraps the application result into a structure that gives other information about the action (e.g. interrupted or not, execution time remained with respect to the `DURATION`, etc.)

```
1 package it.unibo.qactors.action;
2 import it.unibo.contactEvent.interfaces.IEventItem;
3 /*
4  * Result of a IActorAction executed in asynchronous way
5  * interruptEvent is the event (if any) that has interrupted the action
6  */
7 public class AsyncActionResult implements IAsyncActionResult{
8     protected long timeRemained ;
9     protected String result;
10    protected boolean suspended;
11    protected boolean goon;
12    protected IEventItem interruptEvent;
13    protected IActorAction action;
14
15    public AsyncActionResult(IActorAction action, long time,
16        boolean suspended, boolean goon, String result, IEventItem interruptEvent){
17        this.action = action;
18        this.timeRemained = time;
19        this.goon = goon;
20        this.suspended = suspended;
21        this.result = result;
22        this.interruptEvent = interruptEvent;
23    }
24    public long getTimeRemained(){
25        return (timeRemained >= 0) ? timeRemained : 0 ;
26    }
27    public IActorAction getAction(){
28        return action;
29    }
30    public String getResult(){
31        return result;
32    }
33    public boolean getGoon(){
34        return goon;
35    }
36    public boolean getInterrupted(){
37        return suspended;
38    }
39    public IEventItem getEvent(){
40        return interruptEvent;
41    }
42    public void setResult(String result){
43        this.result = result;;
44    }
45    public void setGoon(boolean goon){
46        this.goon = goon;;
47    }
48    @Override
49    public String toString(){
50        String actionOuts= action==null ? "-" : action.getName();
51        String execTime="" +action.getExecTime();
```

```

52     String maxTime = ""+action.getMaxDuration();
53     String eventInterrupt = interruptEvent==null ? "event(none)": "event("+interruptEvent.getEventId()+")";
54     String resultStr = "result("+result.length()==0?"result(noreresult)":result+")";
55     return "asynchActionResult(ACTION,RESULT,SUSPENDED,TIMES,GOON,EVENT)".
56         replace("ACTION","action("+ actionOuts +")").
57         replace("RESULT", resultStr).
58         replace("SUSPENDED","suspended("+suspended+")").
59         replace("TIMES","times(exec("+execTime+"),max("+maxTime+"))").
60         replace("GOON","goon("+goon+")").
61         replace("EVENT",eventInterrupt)
62         ;
63 }
64 }

```

Listing 1.27. AsynchActionResult<T>

It implements the following interface:

```

1 package it.unibo.qactors.action;
2 import it.unibo.contactEvent.interfaces.IEventItem;
3
4 public interface IAsynchActionResult {
5     //An action can work for a prefixed amount of time DT
6     public boolean getInterrupted(); //true if the action has been interrupted by some event
7     public IEventItem getEvent(); //gives the event that has interrupted the action
8     public long getTimeRemained(); //gives the time TR=DT-TE where TE is the execution time before the interruption
9     public String getResult(); //gives the result of the action
10    public boolean getGoon(); //returns true if the system can continue
11    public void setResult(String result);
12    public void setGoon(boolean goon);
13 }

```

Listing 1.28. IAsynchActionResult.java

7.4.2 Fibonacci as a Reactive Action.

The fibonacci computation of Subsection 7.3.2 can be used to build a Reactive Action:

```

1 package it.unibo.actionreactiveexecutor;
2 import it.unibo.actiontimedexecutor.ActionFibonacciTimed;
3 import it.unibo.is.interfaces.IOutputEnvView;
4 import it.unibo.qactors.QActorContext;
5 import it.unibo.qactors.akka.QActor;
6
7
8 public class ActionReactiveFibonacci extends ActionFibonacciTimed{
9     public ActionReactiveFibonacci(String name, QActor actor, QActorContext ctx, int n, boolean cancompensate,
10         String terminationEvId, String[] alarms, IOutputEnvView outEnvView,
11         int maxduration) throws Exception {
12         super(name, actor, ctx, n, cancompensate, terminationEvId, alarms, outEnvView, maxduration);
13     }
14     /*
15     * Version with (semi)tracing to see if the interrupt takes effect
16     */
17     protected long fibonacci( int n ){
18         if( n == 1 || n == 2 ) return 1;
19         long v1 = fibonacci(n-1);
20         println("fibonacci(" + (n-1) + ")=" + v1);
21         long v2 = fibonacci(n-2);
22         // println("fibonacci(" + (n-2) + ")=" + v2);
23         return v1 + v2;
24     }
25 }

```

Listing 1.29. ActionReactiveFibonacci<T>

7.4.3 Experiments on Fibonacci as Reactive Action.

The project *it.unibo.qactors.tests* includes the model `actionReactiveDemo.qa` that shows the effects of the action in the different execution modes. When the `interrupt` flag is set, an event `usercmd : stop` is generated¹⁵ after `maxtime/2 msecs`.

```
1  /*
2  * actionReactiveDemo.qa
3  */
4  System actionReactiveDemo
5  Event run : args(SYNCH,WITHEVH,WITHINTERRUPT)
6  Event usercmd : stop
7
8  Context ctxActionReactiveDemo ip [ host="localhost" port=8079 ]
9
10 QActor actionreactiveexecutor context ctxActionReactiveDemo -g cyan{
11   Rules{
12     actiontodo( fibo(10,V), 20000 ).
13   }
14   Plan init normal
15     println(" *** Fibonacci as Reactive Action *** ");
16     //solve consult("./actionDemoTheory.pl") time(0) onFailSwitchTo prologFailure ;
17     //[ !? actiontodo(G,T) ] solve createApplicationInterface(G,T) time(0) onFailSwitchTo prologFailure ;
18     [ !? actiontodo(GOAL, TIME) ] actorOp createApplicationInterface(GOAL, TIME); //emit run
19     switchToPlan waitEvents
20   Plan waitEvents
21     sense time(600000) run -> continue ;
22     onEvent run : args( true,WITHEVH,WITHINTERRUPT ) -> actorOp runSynch(WITHEVH,WITHINTERRUPT) ;
23     onEvent run : args( false,WITHEVH,WITHINTERRUPT ) -> actorOp runASynch(WITHEVH,WITHINTERRUPT) ;
24     repeatPlan
25 }
```

Listing 1.30. `actionReactiveDemo.qa`



¹⁵ The operation `QActorUtils.emitEventAfterTime` generates a given event after a given time.

8 An interpreter to execute actions

In this section we introduce a system that provides an interpreter allowing human users to execute actions expressed as commands sent via the interfaces introduced in Section 4. The model of the system is defined in the project *it.unibo.qactors.tests*:

```

1  /*
2  * cmdExecutor.qa
3  *
4  * A system that provides an interpreter to execute actions
5  */
6  System cmdExecutor
7  Event local_inputcmd : usercmd(X) //generated by cmd actor gui-interface
8  Event inputcmd      : usercmd(X) //generated by HTTP cmd user-interface
9  Event alarm         : alarm(X)   //generated by HTTP cmd user-interface
10 Event obstacle      : obstacle(X) //generated by HTTP cmd user-interface
11 Event endplay       : endplay(X) //generated by a user play command
12
13 Context ctxCmdExecutor ip [ host="localhost" port=8039 ] -g cyan -httpserver
14 EventHandler evh for alarm , obstacle , endplay -print ;
15
16 QActor qacmdexecutor context ctxCmdExecutor { // -g yellow
17   Plan main normal
18     println("===== " ) ;
19     println("An actor that executes user commands " ) ;
20     println("===== " ) ;
21     demo consult("./talkTheory.pl") ;
22     sound time(2000) file("./audio/music_interlude20.wav")
23     react event alarm -> handleAlarm ;
24     [ ?? tout(X,Y )] println( tout(X,Y ) ) ;
25     solve fibo(37,V) time(500) ;
26     [ ?? tout(X,Y )] println( tout(X,Y ) ) ;
27     [ ?? goalResult( R )] println( goalResult( R ) ) ;
28     switchToPlan handleInput
29   Plan handleInput
30     sense time(600000) local_inputcmd , inputcmd -> elabInputCmd , elabInputCmd ;
31     repeatPlan
32   Plan elabInputCmd resumeLastPlan
33     printCurrentEvent;
34     onEvent local_inputcmd : usercmd(CMD) -> demo CMD ; //the action (CMD) itself is reactive
35     onEvent inputcmd      : usercmd(CMD) -> demo CMD //the action (CMD) itself is reactive
36   Plan handleAlarm resumeLastPlan
37     println("handleAlarm done" ) ;
38     sound time(1500) file("./audio/tada2.wav")
39   Plan handleObstacle resumeLastPlan
40     println("handleObstacle done" )
41   Plan handleTout resumeLastPlan
42     println("handleTout done" )
43   Plan prologFailure resumeLastPlan
44     println("failure in solving a Prolog goal" )
45 }

```

Listing 1.31. cmdExecutor.qa

The interpreter is written in Prolog in the user theory stored in file *talkTheory.pl*.

It works with reference to a simple syntax that we will introduce in the examples that follows. Since the syntax is at the moment quite limited, only a subset of actions can be executed. For example, since the symbol `' :- '` is not admitted, we cannot execute actions like `addRule(r(X):-q(X))`.

8.1 Basic actions

Basic actions can be expressed as follows:

ACTION

A basic action is a terminating action that implements an algorithm written in Java, tuProlog or in some other executable language (for example C or C++). For example:

Basic actions

```
fib(7,V)
raise(alarm,alarm(fire))
play('./audio/tada2.wav')
```

8.2 Guarded actions

Actions prefixed by a guard can be expressed as follows:

Guarded Action syntax structure

```
[ GUARD ], ACTION , ...
```

They are executed only if the **GUARD** is evaluated **true**. The **GUARD** is a boolean condition expressed as a Prolog term that can include unbound variables, possibly bound during the guard evaluation phase.

For example, the following action plays a sound only if the evaluation of the guard `goon/0` ends with success¹⁶:

guards

```
[ goon ], play('./audio/music_interlude20.wav'),20000,'alarm,obstacle', 'handleAlarm,handleObstacle'
```

The evaluation of the guard will end with success, if we send the command:

guards

```
addRule( goon )
```

8.3 Timed actions

Time actions can be expressed as follows:

Timed Action syntax structure

```
[ GUARD ], ACTION , DURATION
```

For example, the action:

Timed actions

```
play('./audio/tada2.wav') , 1000
```

must terminate within a time $T \leq 1000$ msec. During the time T the actor does not execute any other new action; thus, it cannot accept other commands.

8.4 Time out

If a time-out expires, the fact

Timed actions

```
tout(EVENTID,QACTORID).
```

is asserted in the *WorldTheory* of the working *QActor*. This fact can be used to execute actions under the control of a guard (see Subsection 8.2); for example:

Timed actions

```
[ tout(E,A) ], switchToPlan handleTout
```

¹⁶ No variables are supported in this version of the interpreter.

8.5 Asynchronous actions

Actions of the form:

Asynchronous Action syntax structure
[GUARD] , ACTION , DURATION , ENDEVENT

that specify a ***non-empty*** ENDEVENT atom, are activated in asynchronous way. Each asynchronous action works in a proper *Thread* and emits the specified ENDEVENT at termination. For example:

Asynchronous action
play('./audio/tada2.wav') , 1500 , endplay

is a timed, asynchronous **play** action that returns immediately the control. Thus the robot is able to perform other actions 'in parallel' with the previous one. When the **play** action terminates (after 1500 msecs), the event named **endplay** is raised.

Asynchronous actions cannot be reactive (see Subsection 8.6). This because the idea of reacting to an asynchronous actions must be further explored.

8.6 Reactive actions

Actions of the form:

Reactive Action syntax structure
[GUARD] , ACTION , DURATION , [EVENTLIST], [PLANLIST]

that specify a ***non-empty*** EVENTLIST and PLANLIST are called synchronous ***reactive actions*** since they can be 'interrupted' by one of the events specified in the EVENTLIST. When one of these events occurs, the action is 'interrupted' and the corresponding plan specified in the PLANLIST is put in execution.

For example, the action:

Reactive Action
play('./audio/music_interlude20.wav') , 20000 , [usercmd,alarm], [handleUsercmd,handleAlarm]

is an example of a play action that must terminate within 20 **secs**. During this time, the occurrence of an event named **usercmd** or **alarm** terminates the action and puts in execution the plan **handleUsercmd** or **handleAlarm** respectively.

Reactive actions cannot be activated in asynchronous way, since the idea of reacting to an asynchronous actions must be further explored.