An introduction to the usage of 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

1 Introduction

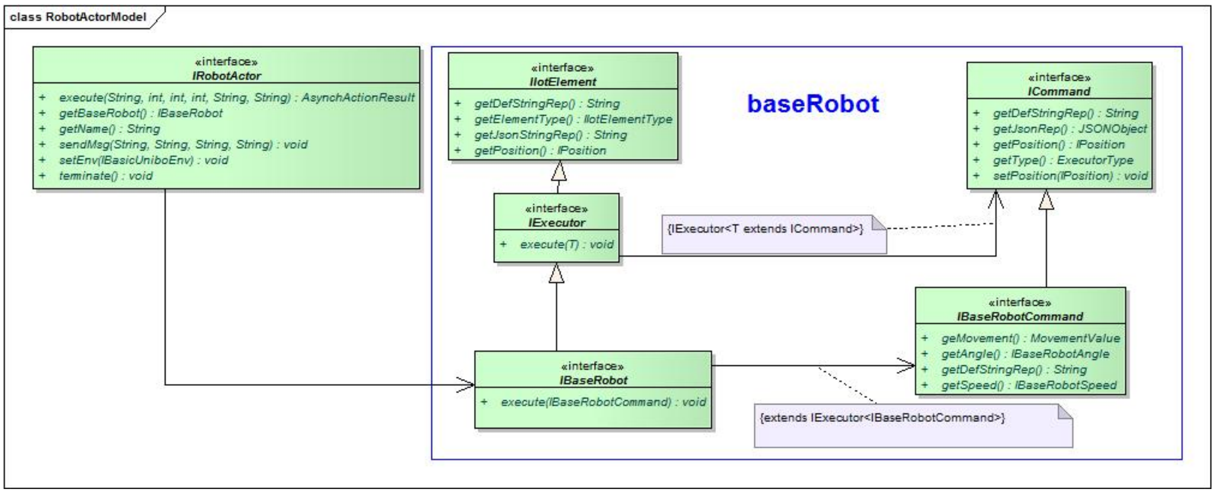
A *QRobot*[1] is defined here as an active entity (more precisely as a *QActor*[2]) that can execute both physical and logical actions and that is able to acquire information from its environment by means of sensors.

In fact, a physical *QRobot* is a normally equipped with two motors that allow it to move according to the *differential drive* pattern and a set of sensors (e.g. sonar, magnetometer, accelerometer, line-detector, etc.) each modelled as an observable object (in the sense of GOF) and/or as an emitter of *events* (see Subsection 2.6).

To facilitate application testing, we usually replace physical robots with a ***mock*** robot that simulates movements and sensors.

1.1 QRobot API

A *QRobot* is implemented in Java and in tuProlog and provides the set of API defined by the `IRobotActor` interface:


class RobotActorModel
«interface»
IRobotActor
+ execute(String, int, int, int, String, String) : AsynchActionResult
+ getBaseRobot() : IBaseRobot
+ getName() : String
+ sendMsg(String, String, String, String) : void
+ setEnv(IBasicUniboEnv) : void
+ terminate() : void
«interface»
IIotElement
+ getDefStringRep() : String
+ getElementType() : IIotElementType
+ getJsonStringRep() : String
+ getPosition() : IPosition
baseRobot
«interface»
ICommand
+ getDefStringRep() : String
+ getJsonRep() : JSONObject
+ getPosition() : IPosition
+ getType() : ExecutorType
+ setPosition(IPosition) : void
«interface»
IExecutor
+ execute(T) : void
{IExecutor<T extends ICommand>}
«interface»
IBaseRobotCommand
+ geMovement() : MovementValue
+ getAngle() : IBaseRobotAngle
+ getDefStringRep() : String
+ getSpeed() : IBaseRobotSpeed
«interface»
IBaseRobot
+ execute(IBaseRobotCommand) : void
{extends IExecutor<IBaseRobotCommand>}

However, software applications usually does not make use of these interfaces, since a software component usually interacts with a *QRobot* by using:

- messages (see Subsection 2.5)
- events (see Subsection 2.6)

[1] The leading *Q* means 'quasi' since the *QRobot* is not related to robotics but it is intended as a low-cost device to be used for case-studies in software engineering and in the `IOT` (*Internet of Things*) application context.

[2] A *QActor* is the basic component of a custom programming meta-model inspired to the actor model, extended with even-driven programming; thus it is a 'quasi' actor.

1.2 Actions

A *QRobot* can execute a set of predefined *actions* (e.g movement actions) that must always terminate. A *timed action* (see Subsection 3.4) 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 robot's 'mind';
2. as changes in the robot's physical behaviour or in the robot's working environment.

The first kind of actions are referred here as *logical actions* since they do not affect the physical world. Actions that change the robot's physical state or the robot's working environment are called *physical actions.*

1.2.1 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 *QRobot* is (being a *QActor*) 'natively' able to compute the `n-th` Fibonacci's number in two ways: in a fast way (`fib/2` Prolog rule) and in a slow way (`fibo/2` Prolog rule).

The ***robot-mind*** is a knowledge base (named `WorldTheory`) associated with each actor.

1.2.2 The actor's `WorldTheory`.

The `WorldTheory` includes computational rules written in tuProlog and facts about the state of the robot and of the world. For example:

– the rule `actorPrintln/1` prints a given tuProlog `Term` (see Subsection 2.5.1) in the standard output of the actor;
– the fact `actorobj/1` memorizes a reference to the Java object that implements the actor (see Subsection 5.3).
– the fact `result/1` memorizes the result of the last plan action (see Subsection 1.2.5) performed by the robot while the fact `goalResult/1` memorizes the result of the last Prolog query. Facts of this kind can be used to express *guards* (see Subsection 3.7) related to action evaluation.

1.2.3 Physical actions.

Physical actions are implemented by using low-cost devices such as `RaspberryPi` and `Arduino`.

1.2.4 Application actions.

Besides the predefined actions, a *QRobot* can execute actions defined by an application designer according to the constraints imposed by its logical architecture. More on this in Section 5.

1.2.5 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 type of attributes that can be associated to it:

Actions attribute sets

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

We will use the following terminology:

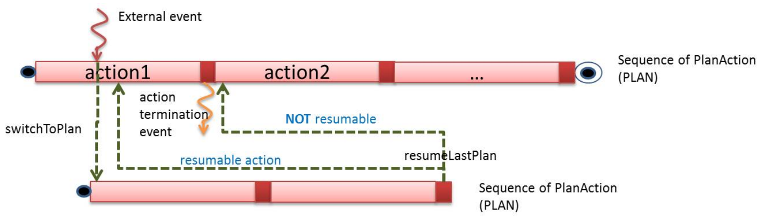
- an action that does not specify any DURATION is called ***basic action*** (see Subsection 3.1);
- an action that does specify a [GUARD] is called ***guarded action*** (see Subsection 3.7);
- an action that specifies a DURATION is called ***timed action*** (see Subsection 3.4);
- an action that specifies a ENDEVENT is called (timed) ***asynchronous action*** (see Subsection 3.5);
- an action that specifies [EVENTLIST],[PLANLIST] is called (timed) ***(synchronous) reactive action*** (see Subsection 3.6).

A *timed action*:

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

1.3 Plans

A ***Plan*** is a sequence of *PlanActions*.



1.4 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 the *robot-system* or by sources external to the system.

The occurrence of an event can put in execution (in mutually exclusive way) 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.

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

Events are represented as messages (see Subsection 1.5) with no destination (RECEIVER=none):

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

1.5 Messages and (reactive) message

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 ***synchronous***.

Messages can be ***sent*** and/or ***received*** by the *QActors* that compose the *robot-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 *massage-based* behaviour only, by excluding *massage-drive* behaviour.

Messages are represented as follows:

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

where [3] :

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 3.7) to allow conditional evaluation of *PlanActions*.

1.6 Robot programs as plans

A ***robot-program*** consists in a set of *Plans*; the (unique, mandatory) *Plan* qualified as `normal` is executed as the robot 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* (e.g. `switchToPlan`, see Subsection 3.1). Plans can be stored in files and dynamically loaded by the user into the *robot-mind*.

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:

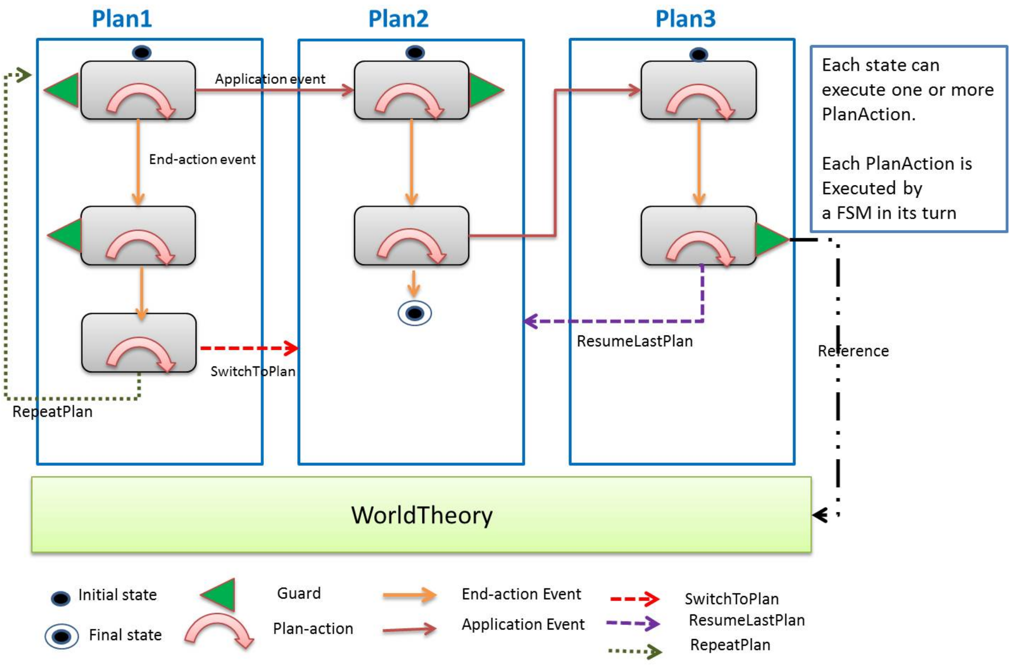
Internal representation of a plan
```
plan(0,p0,sentence(true,move(playsound,'./audio/tada2.wav',1500,''),'',''))
plan(1,p0,sentence(true,move(robotmove,moveforward,100,3000,0),'',''))
plan(2,p0,sentence(true,move(playsound,'./audio/music_interlude20.wav',20000,''),'usercmd,alarm','handleUsercmd,handleAlarm'))
```

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

1.6.1 Robot programs as finite state machines (FSM)

A *Plan* is 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 1.4) and returns the control to the 'calling' plan only when it declares to '`resumeLastPlan`' (otherwise the computation ends).

[3] At the moment only `dispatch` and `request` are implemented



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

Each *PlanAction* is in their 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 3.5);
- a time-out termination event;
- a abnormal termination event;

QRobot plans can be expressed in two ways:

- statically; as sentences of the **ddr** custom language, defined as a meta-model by using the `XText` technology (see Section 2).
- dynamically: as commands sent by an (human) user to the robot by using a proper command interface (see Section 3);

Examples of plans are given in Section 4.

2 The ddr language/metamodel.

The *differential drive robot* (ddr) language is a custom language built by exploiting the XText technology[4]; thus, it is also a meta-model. Technically we can say that ddr is a 'brother' of UML since it is based on EMOF.

2.1 drr as an extension of qa

The ddr language-metamodel is an extension of another, more general language-metamodel, called qa[5], that 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).

A qa 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 qa specification aims at capturing main *architectural* aspects of the system by providing a support for *rapid software prototyping.*

2.2 Workflow

A goal of qa/ddr 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*, including specialized actors such as *robots* (one robot for 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 5) so to reduce the global costs of software development.

2.2.1 The application and the 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 heterogeneity.

[4] The ddr language is defined in the project *it.unibo.xtext.qactor.robot.*
[5] The qa language is defined in the project *it.unibo.xtext.qactor.*

2.3 Example

As an example of **ddr** specifications, we define the behaviour of a ***mock*** robot[6] as a *QActor* able to execute two plans: an initial `init` plan (qualified as '`normal`') that calls another plan named `playMusic` that, once terminated, returns the control to the previous one:

```
/*
 * basic.ddr
 * A robot system named 'basic' is composed of a 'mock' robot working in the 'ctxBasic' context
 */
RobotSystem basic -regeneratesrc

Context ctxBasic ip [ host="localhost" port=8079 ]

Robot mock QActor mockbehavior context ctxBasic{
   Plan init normal
      println("Hello world" ) ;
      switchToPlan playMusic ;
      println("Bye bye" )
   Plan playMusic resumeLastPlan
      sound time(2000) file('./audio/tada2.wav')
 }
```

Listing 1.1. `basic.ddr`

The **ddr** specification shows that a *QRobot* is an element of a distributed software system (*robot-system* from now-on); the robot can work in cooperation/competition with other robots and other components, each modelled as a *QActor*.

2.4 Contexts

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

2.5 Messages

A *QRobot* can `send`/`receive` ***messages*** to/from another *QRobot* (including itself) or *QActor* working in the same or in another *Context*.

The `ddr` language allows us to express `send/receive` actions as high-level operations that hide at application level the details of the communication support.

Messages exchanged among components working on different contexts flow throw the context ports (at the moment using the `TCP/IP` protocol) and are stored in the component (*QActor*) local message-queue.

The `qa` language defines the following syntax for message declaration:

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

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

[6] A ***mock*** robot is a 'virtual' entity that simulates movements and sensors.

2.5.1 PHead.

The PHead syntax rule defines a subset of Prolog syntax:

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

2.5.2 Send actions.

Messages can be sent to a *QRobot* by using the built-in basic `sendMsg` asynchronous, point-to-point action. The `API` provided by the `ddr/qa` run-time support[7] has the following signature:

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

At `ddr` level, higher-level forms of sending-message actions are defined:

– operation that sends a *dispatch*:

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

it is implemented as follows:

```
void forward(String msgId, String dest, String msg) throws Exception{sendMsg(msgId,dest,"dispatch",msg );}
```

– operation that sends a *request*

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

it is implemented as follows:

```
void demand(String msgId, String dest, String msg) throws Exception{sendMsg(msgId,dest,"request",msg);}
```

2.5.3 Receive actions.

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

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

The `AsynchActionResult` is an object that stores the results of the operation; it implements the following interface:

```
package it.unibo.qactors.action;
import it.unibo.contactEvent.interfaces.IEventItem;

public interface IAsynchActionResult {
   //An action can work for a prefixed amount of time DT
   public boolean getInterrupted(); //true if the action has been interrupted by some event
   public IEventItem getEvent(); //gives the event that has interrupted the action
   public long getTimeRemained(); //gives the time TR=DT-TE where TE is the execution time before the interruption
   public String getResult();    //gives the result of the action
   public boolean getGoon();     //returns true if the system can continue
   public void setResult(String result);
   public void setGoon(boolean goon);
}
```

Listing 1.2. `IAsynchActionResult.java`

[7] The `ddr` run-time support is stored in the files *qactors18.jar* and *uniboQactorRobot.jar*

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*.

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

– A) generic *receive* with optional specification

```
ReceiveMsg : name="receiveMsg" duration=TimeLimit (spec=MsgSpec)? ;
   TimeLimit : name="time" "(" ( msec=INT | var=Variable ) ")" ;
   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))
```

– B) receive a message with a specified structure:

```
OnReceiveMsg: name="receiveTheMsg" "m" "(" msgid=PHead "," msgtype=PHead "," msgsender=PHead
   "," 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)
```

– C) select a message and execute:

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

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

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

2.6 Events and event-driven/event-based behaviour

QActors can `emit` and `sense` (perceive) *events* (see Subsection 2.6) represented as follows:

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

2.6.1 Emit action.

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

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

The `qa` language defines

– A) high-level emit-event action

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

Example: emit an event

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

2.6.2 Event handlers.

The occurrence of an event activates, in ***event-driven*** way, all the ***EventHandlers*** 'registered' (i.e.declared in some *Context*) for that event. The *EventHandlers* are activated one at the time by a single-thread scheduler.

2.6.3 Sense action.

To allow ***event-based*** behaviour, qa provides ***sense*** operation that blocks the execution of a *QActor* until the required event occurs. The ddr/qa run-time support does introduce the following operation:

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

The qa language defines:

– A) high-level sense-event action

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

Example: sense an event

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

2.6.4 OnEvent receive actions.

The qa language defines also :

– B) high-level event-selection action

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

Example: perceive an event

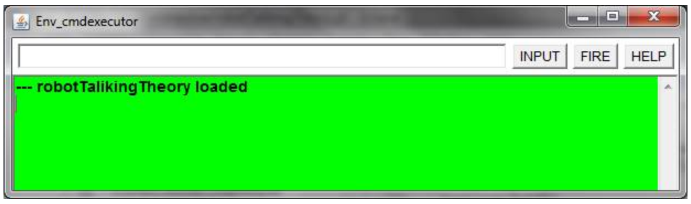
```
//sense ...
onEvent alarm :  alarm(fire} -> sound time(2500) file( "./audio/illogical_most2.wav")
```

3 Human interaction with a QRobot

A human user can interact with a robot by using:

– a system console (based on `System.in`) that emits the following event:
  `local_inputcmd : usercmd(executeInput( CMD ))`
  This input device is automatically created by each *Context*.

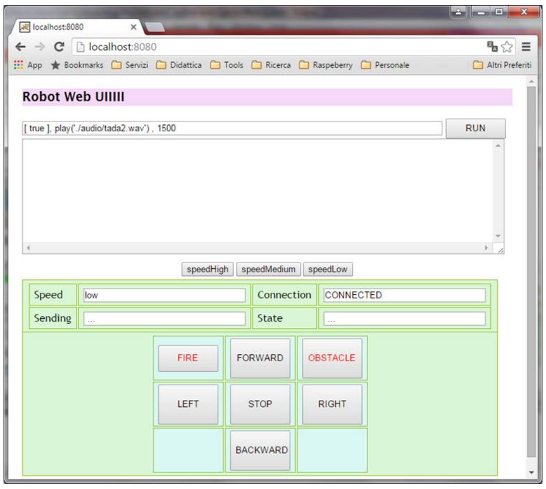
– a user GUI that emits:
  `local_inputcmd : usercmd(executeInput( CMD ))` (`INPUT` button)
  `alarm : alarm(fire)` (`FIRE` button)



```
--- robotTalikingTheory loaded
```

  This input device is created when the `-g` option is included in a *QActor* declaration. However the application designer must handle the events, like done for example in the `userCmdInterpreter.qa` system of the project *it.unibo.robot.interactive*.

– a web interface (implemented by a `HTTP` web-socket server working on port `8080`) that emits:
  `usercmd : usercmd(executeInput( CMD ))` (`RUN` button)
  `usercmd : usercmd(robotgui(MOVE))` with `MOVE=w(low),...,s(high)` (`MOVE` button)
  `alarm : alarm(fire)` (`FIRE` button)
  `alarm : alarm(obstacle)` (`OBSTACLE` button)



Robot Web UIIIII

[true]. play('./audio/tada2.wav') , 1500

Speed	low	Connection	CONNECTED
Sending	...	State	...

  This web interface is automatically generated in the `srcMore` directory in a package associated with each *Context*.

Through these interfaces the human user can execute Prolog sentences or predefined *PlanActions*. According to the *PlaAction* types introduced in Subsection 1.2.5, user commands can take one of the following forms (the prefix `[GUARD],` is optional; if omitted the system automatically includes `[true],`):

User commands syntax structure

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

Let us give some examples of *PlaActions* according to the terminology of Subsection 1.2.5.

3.1 Basic actions

A basic action can be one of the actions implemented as Prolog rules. The predefined Prolog rule-set is defined at the moment in the file `robotTalkTheory.pl` of the project *it.unibo.robot.interactive* as follows:

Predefined robot actions

evaluate fibonacci slow	fibo(N,V)
evaluate fibonacci fast	fib(N,V)
print a sentence	println(TERM)
show the plan	showPlan
show a plan	showPlan(PLANNAME)
store the pdefault plan in a file	storePlan(FILENAME,PLANNAME)
clear the current plan (pdefault)	clearPlan
remove all msg/6 facts	clean
load a plan from file	loadPlan(FILENAME)
run a plan	runPlan(PLANNAME)
user-defined action	...

The content of file `robotTalkTheory.pl` can be modified by the application designer.

3.2 Robot movement actions

The robot movement actions take the form:

`move( MOVE,SPEED,ANGLE )`

For example, let us suppose to move a robot as follows:

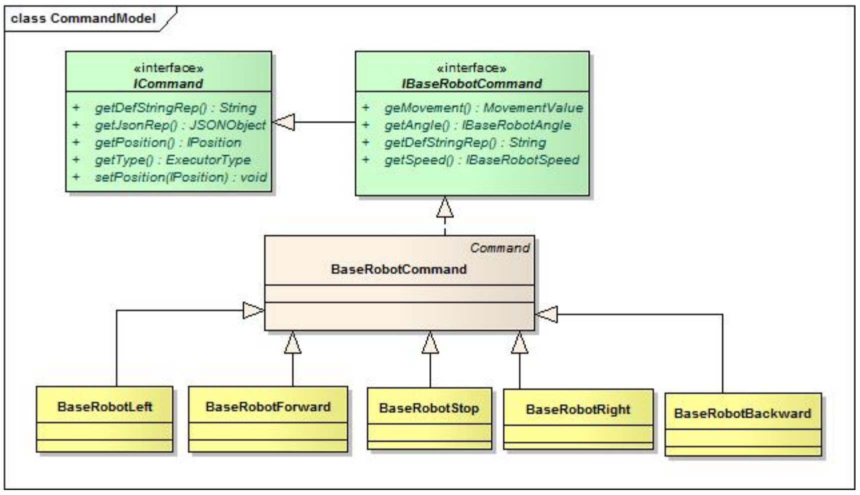
```
-------------->
              |
<--------------
```

If the robot front is looking at East, the moves could be:

Robot moves

```
move(mf,100,0), 3000   //mf = moveforward
move(mr,50,0), 100     //mr = moveright
move(mf,100,0), 1000
move(mr,50,0), 100
move(mf,100,0), 3000
```

Robot movement actions are a high-level representation of the base-robot (see Subsection 1.1) operations shown hereunder:


class CommandModel
«interface»
ICommand
+ getDefStringRep() : String
+ getJsonRep() : JSONObject
+ getPosition() : IPosition
+ getType() : ExecutorType
+ setPosition(IPosition) : void
«interface»
IBaseRobotCommand
+ geMovement() : MovementValue
+ getAngle() : IBaseRobotAngle
+ getDefStringRep() : String
+ getSpeed() : IBaseRobotSpeed
Command
BaseRobotCommand
BaseRobotLeft
BaseRobotForward
BaseRobotStop
BaseRobotRight
BaseRobotBackward

Robot movement actions are implemented as ***resumable*** actions.

3.3 Basic actions examples

The predefined robot action set is defined at the moment as follows:

Predefined robot actions
```
play a sound          play( FILENAME )
emit an event         raise( EVID,EVCONTENT )
move the robot        move( MOVE,SPEED,ANGLE )  with MOVE=mf|mb|ml|mr|ms
send a dispatch       forward( DEST, MSGID, MSGCONTENTTERM )
```

Here some example[8]:

Robot action examples
```
a basic logical action:         fib(7,V)
a actor logical action:         raise(alarm,alarm(fire))
a robot physical,timed action:  play('./audio/tada2.wav') , 1500
a timed,asynchronous action:    play('./audio/tada2.wav') , 1500 , endplay
a timed,reactive action:        play('./audio/music_interlude20.wav'),20000,[alarm],[handleAlarm]
```

3.4 Timed actions

Actions of the form:

Timed Action syntax structure
```
[ GUARD ] , ACTION , DURATION
```

that specify a `DURATION`, are called ***timed actions***, since they must terminate within a `DURATION` time. For example, the actions:

Timed actions
```
move(mr,100,0), 1000
play('./audio/tada2.wav') , 1000
```

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

[8] These examples can be tested by executing the application `MainCtxUserCmdInterpret.java` defined in project `it.unibo.robot.interactive`.

3.5 Asynchronous actions

Actions of the form:

```
Asynchronous Action syntax structure
[ GUARD ] , ACTION , DURATION , ENDVENT
```

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 (see Section ??).

Asynchronous actions cannot be reactive (see Subsection 3.6). This because the idea of reacting to an asynchronous actions must be further explored.

3.6 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:

```
Action syntax structure
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.

If an action is *resumable*, it can be continue its execution after an interruption.

3.7 Guarded actions

Actions prefixed by a [GUARD] :

```
Guarded Action syntax structure
[ GUARD ] , ACTION , ...
```

are executed only when 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 for a time T given by the evaluation of the guard execTime/1:

```
A guarded action
[ execTime(T) ] , play('./music_interlude20.wav') , T
```

In the next example, the msg/6 structure is used as guard;

msg/5 as guard
```
[ msg(alarm,"event",SENDER,none,alarm(A),MSGNUM) ] , println(  alarm(A) )
```

In the ddr language the previous examples should be expressed as follows:

guard in ddr
```
[ !? execTime(T) ]  sound time(T) file('./music_interlude20.wav')
[ ?? msg(alarm,"event",SENDER,none,alarm(A),MSGNUM) ]  println(  alarm(A) )
```

Important to note that, in ddr:

– the prefix !? before the guard condition means that the knowledge (Prolog fact or rule) that makes the guard true must ***not*** be removed form 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*

3.8 Action examples

We report here some example that can be tested by executing the application MainCtxUserCmdInterpret.java defined in project it.unibo.robot.interactive:

Action examples
```
a basic logical action:         fib(7,V)
a basic logical,timed action:   fibo(25,V) , 5000
a actor logical action:         raise(alarm,alarm(fire))
a robot movement action:        move(mr,100,0)
a robot movement,timed action:  [true] , move(mb,100,0) , 1000
a robot physical,timed action:  play('./audio/tada2.wav') , 1500
a timed,asynchronous action:    play('./audio/tada2.wav') , 1500 , endplay
a timed,reactive action:        play('./audio/music_interlude20.wav'),20000,[usercmd,alarm],[handleUsercmd,handleAlarm]
```

Examples on fundamental concept of ***unification***:

Action examples
```
a(X,1)=a(1,Y)                      result('='(a(1,1),a(1,1)))
a(X,1)=a(1,X)                      result('='(a(1,1),a(1,1)))
a(X,2)=a(1,X)                      result(failure)

a(b(X,1),c(Y))=Z                      result('='(a(b(_2,1),c(_1)),a(b(_2,1),c(_1))))
a(b(X,1),c(Y))=a(A,B)              result('='(a(b(_6,1),c(_1)),a(b(_6,1),c(_1))))
a(b(X,1),c(Y))=a(b(0,A),c(A))      result('='(a(b(0,1),c(1)),a(b(0,1),c(1))))
```

4 Actions sequences as plans

Let us suppose to send to the robot the following sequence of commands:

```
Parallel actions
1) play('./audio/music_interlude20.wav'),5000,endplay
2) play('./audio/computer_process_info4.wav'),4000,endplay
3) play('./audio/music_dramatic20.wav'), 15000,  [ alarm ] , [ handleAlarm ]
4) move(mf,100,0), 5000 ,  [ alarm ] , [ handleAlarm ]
```

This action sequence can be statically defined as a robot plan in the `ddr` custom language as follows:

```
/*
 * planexample.ddr
 */
RobotSystem planexample -regeneratesrc
Event endplay : endplay(X)
Event alarm  :  alarm(X)

Context ctxPlanExample ip [ host="localhost" port=8022 ]
EventHandler evh for alarm, endplay -print ;

QActor alarmemitter context ctxPlanExample{
    Plan init normal
        delay time(2000) ;
        println("emit first alarm");
        /*a1*/ emit alarm : alarm(fire) ;
        delay time(3000) ;
        println("emit second alarm");
        /*a2*/ emit alarm : alarm(fire)
}

Robot mock QActor planbehaviour context ctxPlanExample{
    Plan pnormal normal
        /*1*/ sound time(5000) file('./audio/music_interlude20.wav' ) answerEv endplay ;
        /*2*/ sound time(4000) file('./audio/computer_process_info4.wav' ) answerEv endplay ;
        /*3*/ sound time(20000) file('./audio/music_dramatic20.wav' ) react event alarm -> handleAlarm ;
        /*4*/ robotForward speed(100) time(4000) react event alarm -> handleAlarmSlow ;
              println( end )
     Plan handleAlarm resumeLastPlan
        onEvent alarm : alarm(A) -> println( reactionfast(alarm,A) )
     Plan handleAlarmSlow resumeLastPlan
        memoCurrentEvent;
        [ ?? msg(alarm,"event",SENDER,RECEIVER,alarm(A),MSGNUM) ] println( reactionslow(alarm,A) ) ;
        solve fibo(25,V) time(10000) ;
        [ ?? goalResult(X)] println( resultFibo(X) )
   }
```

Listing 1.3. `planexample.ddr`

In this system, the application designer has introduced (for testing purposes) the *QActor* `alarmemitter` as a source of alarms. Moreover, a very simple `handleAlarm` plan is explicitly defined.

A 'conventional' way to explain the behaviour of this software system could be the following:

> The system activates in parallel three play actions: (1), (2) and (3).
> When the first **alarm** event occurs (a1) , the last launched music (3) is interrupted and the **handleAlerm** plan is executed; afterwards, the robot starts to move forward (4).
> When the second **alarm** event occurs (a2), the robot stops and, when alarm handling is terminated, it completes the action).

Of course, the `alarm` events can be generated by using the `GUI` or `Web` user interfaces introduced in Section 3 but the *QActor* `alarmemitter` is more useful for automated testing.

4.1 Defining and running plans via the user interface

If we interact with the robot via some interface of Section 3, we can start by defining and storing the `handleAlarm` plan:

Define a store a handleAlarm plan
```
[ msg(alarm,"event",SENDER,RECEIVER,alarm(A),MSGNUM) ] , println(  alarm(A) )
[ not msg(alarm,"event",SENDER,RECEIVER,M,MSGNUM) ] , println( noevent(alarm) )
storePlan('handleAlarm.txt', handleAlarm)
```

Now, if we load and run the plan, then the `no alarm` message should be displayed):

Load and test the handleAlarm plan
```
loadPlan('handleAlarm.txt')
showPlan(handleAlarm)
runPlan(handleAlarm)
```

In the same way we can define and store a 'normal' behaviour plan named `pnormal`:

Define a store a pnormal plan
```
clearPlan
play('./audio/music_interlude20.wav'),12000,endplay                    (1)
play('./audio/computer_process_info4.wav'),4000,endplay                (2)
play('./audio/music_dramatic20.wav'),20000, [ alarm ] , [ handleAlarm ]   (3)
move(mf,100,0), 5000 ,  [ alarm ] , [ handleAlarm ]                    (4)
storePlan('pnormal.txt', pnormal)
```

Finally we can load and execute the normal plan:

Execute pnormal plan
```
loadPlan( 'pnormal.txt' )
showPlan( pnormal )
runPlan( pnormal )
```

While this plan is running, the user can raise the `alarm` event to stop the robot movement:

Emit the event
```
raise( alarm, alarm(fire) )
```

Note that the system works even if the *synchronous* actions (last `play (3)` and `move (4)`) should not allow any further interaction with the robot until completed. In fact, the implementation of the user-commands interpreter (`robotCmdInterpreter.ddr` in project *it.unibo.robot.interactive*) does introduce a specialized *QActor* (`userinputforevents`) dedicated to the handling of (`alarm`) events.

4.2 Automatic generation of interpretable plans

Our software factory generates a file named `plan.txt` that included a set of Prolog 'facts' that represent the actions that compose the plans of a ddr/qa model of an actor. This file is inclide in the directory `srcMore/xxx` where `xxx` is the name of the package related to the actor.

These plans can be executed by writing an interpreter like that provided by the `talkTheory.pl`.

5 User-defined actions in Prolog

The user can define application-specific actions in two main way:

- using Java or some other (Java-compatible) programming language
- using tuProlog

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

5.1 The `solve` operation.

The qa language defines an action that allows application designers to `solve` Prolog goals :

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

The result is represented as the fact `goalResult/1` in the actor *WorldTheory* (see Subsection 1.2.1).

5.2 Loading and using a theory

The *WorldTheory* of an actor can be extended by the application designer by using the directive[9] `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`) :

```
/*
 * atheoryUsage.qa in project it.unibo.robot.interpreter
  */
System aTheoryUsage -regeneratesrc
Context ctxTheoryUsage ip [ host="localhost" port=8049 ]

QActor theoryusage context ctxTheoryUsage{
    Plan init normal
/*0*/  solve consult("./aTheory.pl") time(0) onFailSwitchTo prologFailure ;
       switchToPlan compute ;
       switchToPlan accessdata ;
       println( bye )
    Plan compute resumeLastPlan
       solve fib(7,V) time(0); //time(0) => no timed and no reactive action
       [!? goalResult(X) ] println( X )  //prints fib(7,21)
    Plan accessdata resumeLastPlan
       println( "------------------------------------" ) ;
/*1*/  [ !? data(S,N,V) ] println( data(S,N,V) ) ;
/*2*/  [ !? validDistance(N,V) ] println( validDistance(N,V) ) ;
/*3*/  solve nearDistance(N,V) time(0) onFailSwitchTo prologFailure ;
/*4*/  [ !? goalResult(nearDistance(N,V)) ] println( warning(N,V) ) ;
/*5*/  solve nears(D) time(0) onFailSwitchTo prologFailure ;
/*6*/  [ !? goalResult(nears(D)) ] println( nears(D) ) ;
       repeatPlan 1
    Plan prologFailure
       println("theoryusage has failed to solve a Prolog goal" )
}
```

Listing 1.4. `aTheoryUsage.qa`

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

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

```
/* =========================================================
aTheory.pl in project it.unibo.robot.interpreter
========================================================== */
 data(sonar, 1, 10).
 data(sonar, 2, 20).
 data(sonar, 3, 30).
 data(sonar, 4, 40).

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

initialize :-  actorPrintln("initializing the aTheory ...").
:- initialization(initialize).
```

Listing 1.5. `aTheory.pl`

5.2.1 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:

```
--- initializing the aTheory ...
--- fib(7,21)
--- "-------------------------------------"
--- data(sonar,1,10)
--- validDistance(2,20)
--- warning(2,20)
--- nears([d(2,20),d(3,30)])
--- "-------------------------------------"
--- data(sonar,1,10)
--- validDistance(2,20)
--- warning(2,20)
--- nears([d(2,20),d(3,30)])
--- bye
```

5.2.2 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 data. 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 5.7.3).

5.3 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 move the robot in some planned way:

```
/*
===============================================================
robotDanceTheory.pl in project it.unibo.robot.interpreter
===============================================================
*/
dance :-
    actorobj( Robot ),
    actorPrintln(forward),
    Robot <- execute( forward, 100, 0, 4000, "", "" ),
    actorPrintln(left),
    Robot <- execute( left, 100, 0, 2000, "", "" ),
    actorPrintln(right),
    Robot <- execute( right, 100, 0, 2000, "", "" ),
    actorPrintln(backward),
    Robot <- execute( backward, 100, 0, 8000, "", "" ).

initialize :-  actorPrintln("initializing the robotDanceTheory ...").
:- initialization(initialize).
```

Listing 1.6. robotDanceTheory.pl

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

```
/*
 * dancer.ddr in project it.unibo.robot.interpreter
 */
RobotSystem dancerSys -regeneratesrc
Event endplay : endplay(X)
Event alarm  : alarm(X)

Context ctxDancer ip [ host="localhost" port=8079 ] -httpserver
EventHandler evh for alarm, endplay -print ;

Robot mock QActor dancer context ctxDancer{
    Plan init normal
        solve consult("./robotDanceTheory.pl") time(0) onFailSwitchTo prologFailure ;
//          switchToPlan playMusic ;
        switchToPlan dance ;
        println("Bye bye" )
    Plan dance
        solve dance time(2000) onFailSwitchTo prologFailure react event alarm -> handleAlarm
    Plan playMusic resumeLastPlan
        sound time(20000) file('./audio/music_interlude20.wav') answerEv endplay
    Plan handleAlarm
        println("*** alarm ***" )
    Plan prologFailure
        println("dancer has failed to solve a Prolog goal" )
}
```

Listing 1.7. dancer.ddr

5.4 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 syntax[10]

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:

```
/*
 * rulesInModel.qa in project it.unibo.robot.interpreter
```

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

```
 */
System rulesInModel -regeneratesrc
Event endplay : endplay(X)
Event alarm  : alarm(X)

Context ctxRulesInModel ip [ host="localhost" port=8059 ]
EventHandler evh for endplay -print ;

QActor rulebasedactor context ctxRulesInModel -g yellow {
   Rules{
      shortSound(1,'./audio/tada2.wav').
      fastSound(1,'./audio/any_commander3.wav',3000).
      fastSound(2,'./audio/computer_complex3.wav',3000).
      fastSound(3,'./audio/illogical_most2.wav',2000).
      fastSound(4,'./audio/computer_process_info4.wav',4000).
      longSound(1,'./audio/music_interlude20.wav',15000).
      longSound(1,'./audio/music_dramatic20.wav',20000).
   }
   Plan init normal
      [ !? longSound(1,F,T)] sound time(T) file(F) answerEv endplay ;
      delay time(300) ;
      [ !? fastSound(1,F,T)] sound time(T) file(F) ;
      [ !? fastSound(4,F,T)] sound time(T) file(F) ;
      [ !? fastSound(2,F,T)] sound time(T) file(F) ;
      [ !? fastSound(3,F,T)] sound time(T) file(F) ;
      println("Bye bye" )
   Plan prologFailure
      println("mockbehavior has failed to solve a Prolog goal" )
}
```

Listing 1.8. `rulesInModel.qa`

5.5 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 to 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:

```
/*
 * userSelect.qa in project it.unibo.robot.interpreter
 */
System userSelect -regeneratesrc
Context ctxUserSelect ip [ host="localhost" port=8079 ]

QActor soundselector context ctxUserSelect{
   Plan init normal
      solve consult("./userTheory.pl") time(0) onFailSwitchTo prologFailure ; //(1)
      sound time(20000) file('./audio/music_interlude20.wav') answerEv endplay;
      switchToPlan playMusic ;
      println("Bye bye" )

   Plan playMusic resumeLastPlan
      [ !? select(FILE) ] sound time(3000) file(FILE) ;
      repeatPlan 1  //the plan is executed two times

   Plan prologFailure
      println("mockbehavior has failed to solve a Prolog goal" )
}
```

Listing 1.9. `userSelect.qa`

5.5.1 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.

5.5.2 The user-defined `select/1` operation.

To allow the `select/1` guard to become part of the problem-solution rather than just a test[11] , 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)`).

```
/*
==============================================================
userTheory.pl in project it.unibo.robot.interpreter
==============================================================
*/
select( FileName ) :-
/*
----------------------------------------
ACCESS TO JAVA OBJECT INSTANCES
----------------------------------------
*/
   java_object('javax.swing.JFileChooser', [], Dialog),
   Dialog <- showOpenDialog(_),
   Dialog <- getSelectedFile returns File,
   File  <- getPath returns Path,
/*
----------------------------------------
ACCESS TO CLASS STATIC OPERATION
----------------------------------------
*/
   class("it.unibo.utils.Utils") <- adjust( Path ) returns FileName.

welcome :- actorPrintln("welcome from userTheory.pl").
:- initialization(welcome).
```

Listing 1.10. `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 `"/"`.

```
package it.unibo.utils;
public class Utils {
   public static String adjust(String fname ){
      return fname.replace("\\", "/").replace("'", "");
   }
}
```

Listing 1.11. `Utils.pl`

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

5.6 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 robot-systems, using tuProlog as a 'glue' between high-level models (expressed in qa / ddr) and more detailed operations written in Java.

The workflow of Subsection 2.2 can be extended with the following steps:

– make reference to the Java classes that constitute the *domain model* expressed as conventional (POJO) objects ;
– define one or more utility classes that can help the usage of the domain objects by means of Prolog rules;
– 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.

5.7 Examples of problem solving with tuProlog

Let us consider the following model:

```
/*
 * theoryExanple.qa in project it.unibo.robot.interpreter
  */
System theoryExanple -regeneratesrc
Context ctxTheoryExanple ip [ host="localhost" port=8099 ]

QActor thexample context ctxTheoryExanple{
    Plan init normal
        solve consult("./exampleTheory.pl") time(0) onFailSwitchTo prologFailure ;
        solve consult("./srcMore/it/unibo/ctxTheoryExanple/theoryexanple.pl") time(0) onFailSwitchTo prologFailure ;
        switchToPlan configuration;
        switchToPlan family;
/*(1)*/ solve assign(n,1) time(0) onFailSwitchTo prologFailure ;
        switchToPlan accessData ;
        println( bye )
    Plan configuration resumeLastPlan
        solve showSystemConfiguration time(0) onFailSwitchTo prologFailure
    Plan family resumeLastPlan
        solve son(X,haran) time(0) onFailSwitchTo prologFailure ;
        [ !? goalResult(son(X,haran)) ] println(X) else println(noson(haran));
        solve daughters(X,haran) time(0) onFailSwitchTo prologFailure ;
        [ !? goalResult(daughters(X,haran)) ] println(X)
    Plan accessData resumeLastPlan
        [ !? nextdata(sonar,n,N) ] println( data(sonar,N,V) ) else endPlan "no more data" ;
        repeatPlan 0
    Plan prologFailure
        println("thexample has failed to solve a Prolog goal" )
}
```

Listing 1.12. theoryExanple.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:

```
/*
================================================================
exampleTheory.pl
================================================================
*/
 /*
 ----------------------------------
Show system configuration
 ----------------------------------
 */
showSystemConfiguration :-
    actorPrintln(' The system is composed of the following contexts'),
    getTheContexts(CTXS),
    showElements(CTXS),
    actorPrintln(' and of the following actors'),
    getTheActors(A),
    showElements(A).

showElements([]).
showElements([C|R]):-
    actorPrintln( C ),
    showElements(R).

 /*
 ----------------------------------
Find system configuration
 ----------------------------------
 */
getTheContexts(CTXS) :-
    findall( context( CTX, HOST, PROTOCOL, PORT ), context( CTX, HOST, PROTOCOL, PORT ), CTXS).
getTheActors(ACTORS) :-
    findall( qactor( A, CTX ), qactor( A, CTX ), ACTORS).

 /*
 ----------------------------------
 Family relationship
 ----------------------------------
 */
 father( abraham, isaac ).
 father( haran, lot ).
 father( haran, milcah ).
 father( haran, yiscah ).
 male(isaac).
 male(lot).
 female(milcah).
 female(yiscah).

 son(S,F):-father(F,S),male(S).
 daughter(D,F):- father(F,D),female(D).

 sons(SONS,F) :- findall( son( S,F ), son( S,F ), SONS).
 daughters(DS,F) :- findall( d( D,F ), daughter( D,F ), DS).

 /*
 ----------------------------------
 Indexed knowledege
 ----------------------------------
 */
 data(sonar, 1, 10).
 data(sonar, 2, 20).
 data(sonar, 3, 30).
 data(sonar, 4, 40).
 data(sonar,length,4).

 data(distance, 1, 100).
 data(distance, 2, 200).
 data(distance, 3, 300).
 data(distance,length,3).

 nextdata( Sensor, I , V):-
```

```
   data(Sensor,length,N),
   value(I,V),
   inc(I),
   V =< N.
 /*
 ---------------------------------
 Imperative
 ---------------------------------
 */
assign( I,V ):-
   ( retract( value(I,_) ),!; true ),
   assert( value( I,V ) ).
inc(N):-
   value(N,X),
   Y is X + 1,
   assign( N,Y ).

/*
-----------------------------------------------------------
initialize
-----------------------------------------------------------
*/
initialize :-  actorPrintln("initializing the exampleTheory ...").
:- initialization(initialize).
```

Listing 1.13. `exampleTheory.pl`

We note that:

5.7.1 configuration.

moreexamples To solve the `configuration` problem, the application designer loads the generated theory stored in file `./srcMore/it/unibo/ctxTheoryExanple/theoryexanple.pl`:

```
%====================================================================================
% Context ctxTheoryExanple SYSTEM-configuration: file it.unibo.ctxTheoryExanple.theoryExanple.pl
%====================================================================================
context(ctxtheoryexanple, "localhost", "TCP", "8099" ).
%%% -------------------------------------------
qactor( thexample , ctxtheoryexanple ).
%%% -------------------------------------------
```

Listing 1.14. `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.

5.7.2 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.

5.7.3 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.

5.7.4 output. The output of the system execution is:

```
*** ctxtheoryexanple startUserStandardInput ***
Starting the actors ....
--- initializing the exampleTheory ...
---    The system is composed of the following contexts
--- context(ctxtheoryexanple,localhost,'TCP','8099')
---     and  of the following actors
--- qactor(thexample,ctxtheoryexanple)
--- lot
--- [d(milcah,haran),d(yiscah,haran)]
--- data(sonar,1,10)
--- data(sonar,2,20)
--- data(sonar,3,30)
--- data(sonar,4,40)
--- no more data
--- bye
```

6 About actions

In Subsection 1.2 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):

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

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

6.1 Asynchronous Observable Actions

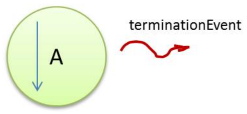
Let us introduce now a new idea of an action[12] , with the following properties:

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 event[13]. 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.



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

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

6.2 The class `ActionObservableGeneric`

The generic, abstract class `ActionObservableGeneric<T>` provides an implementation support for Observable Actions. It implements the following interface [14]

```java
package it.unibo.qactors.action;
import java.util.concurrent.Callable;
import java.util.concurrent.Future;

public interface IObservableActionGeneric<T> extends Callable<T> {
    public Future<T> activate() throws Exception;
    public void execAction() throws Exception ;
    public long getExecTime();
    public String getTerminationEventId();
    /*
     * Return a representation of the action and/or result
     */
    public String getActionRep();
    public String getResultRep();
    public String getActionAndResultRep() throws Exception;
}
```

Listing 1.15. `IObservableActionGeneric<T>`

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

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

6.2.1 The interface `Callable<T>`.

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.

6.2.2 The interface `Future<T>`.

The operation `activate` starts the execution of the action (see Subsection 6.2.3) 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*.

[14] The code is included in the file *qactor18.jar*.

6.2.3 The operation activate.

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

```
protected Future<T> fResult;
   public Future<T> activate() throws Exception{
      Result = EventPlatformKb.manyThreadexecutor.submit(this);
      return fResult;
   }
```

6.2.4 The class Executors.

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.

6.2.5 The operation execAction.

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

```
   @Override
   public void execAction() throws Exception {
      activate();
      fResult.get(); //to force the caller to wait
   }
```

6.2.6 The operation T call.

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.

```
   /* Entry point for the Executor */
   @Override
   public T call() throws Exception {
      startOfAction();
      execTheAction();
      result = endActionInternal();
      return result;
   }
```

6.2.7 The operation startOfAction.

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

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

6.2.8 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,EXECTIME)`:

The operation `endActionInternal`

```
    /* Calculate action execution time and emit the termination event */
    protected T endActionInternal() throws Exception{
        T res = endOfAction();
        emitEvent( terminationEvId, res );
        return res;
    }
    protected void emitEvent(String event, T res){
        long tEnd = Calendar.getInstance().getTimeInMillis();
        durationMillis = tEnd - tStart ;
        platform.raiseEvent( getName(), event, "result("+res+",exectime("+durationMillis+"))" );
    }
```

6.2.9 The operations `execTheAction` and `endOfAction` .

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

```
    /* TO BE DEFINED BY THE APPLICATION DESIGNER */
    protected abstract void execTheAction() throws Exception;
    protected abstract T endOfAction() throws Exception;
```

These operations must be defined by the application designer in the following way:

`execTheAction`	defines the 'businnes logic' of the action
`endOfAction`	returns the main result of the action

6.3 Fibonacci as Observable Action

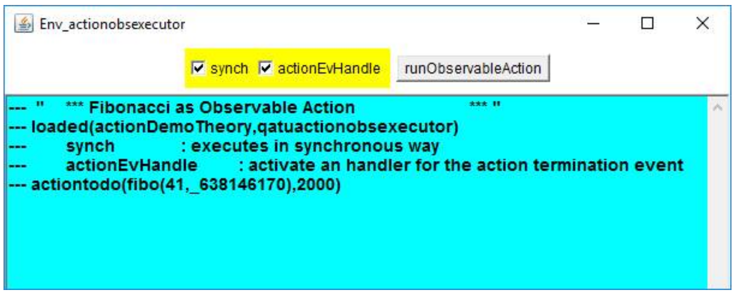
The `fibonacci` computation of Section 6 is defined in the following as a Observable Action that takes at construction time a goal of the form `fibo(N,V)` where `N` is the fibonacci number to evaluate and `V` is the (variable that denotes the) result

```
package it.unibo.actionobsexecutor;
import alice.tuprolog.Struct;
import alice.tuprolog.Term;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.action.ActionObservableGeneric;

public class ActionObservableFibonacci extends ActionObservableGeneric<String> {
private String goalTodo = "fibo(25,V).";
private String myresult = "unknown";
private int n;
    public ActionObservableFibonacci(String name, String goalTodo,
            String terminationEvId, IOutputEnvView outView) throws Exception {
        super(name, terminationEvId, outView);
        this.goalTodo = goalTodo;
        println("%%% ActionObservableFibonacci CREATED with terminationEvId=" + terminationEvId );
    }
    @Override
    public void execTheAction() throws Exception {
      myresult = fibonacci( goalTodo );
    }
    protected String fibonacci( String goalTodo ){
        Struct st = (Struct) Term.createTerm(goalTodo);
        n = Integer.parseInt( ""+st.getArg(0) );
        return ""+fibonacci(n);
    }
    protected long fibonacci( int n ){
        if( n<0 || n==0 || n == 1 ) return 1;
        else return fibonacci(n-1) + fibonacci(n-2);
    }
    @Override
    public String getResultRep() {
        return "fibo("+n+","+this.myresult+")";
    }
    @Override
    protected String endOfAction() throws Exception {
        return this.myresult;
    }
}
```

Listing 1.16. `ActionObservableFibonacci<T>`

6.3.1 Experiments on Fibonacci as Observable Action.

The project *it.unibo.robot.interactive*, package `it.unibo.actionobsexecutor` shows the effects of the action in the different execution modes:



```
--- "    *** Fibonacci as Observable Action                ***  "
--- loaded(actionDemoTheory,qatuactionobsexecutor)
---       synch              : executes in synchronous way
---       actionEvHandle         : activate an handler for the action termination event
--- actiontodo(fibo(41,_638146170),2000)
```

6.4 Timed actions

In several situations, an application designer could activate actions that must always terminate within a prefixed amount of time. In Subsection 1.2.5 we have introduced the idea of *timed action*.

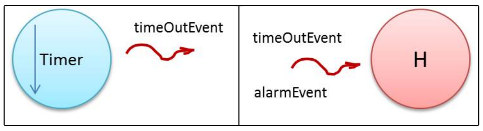
We can define an ***Timed Action*** as an *Observable Action* whose execution time `T` satisfies the constraint `T<=DURATION`, where `DURATION` is a prefixed amount of time[15]. Moreover:

- 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, *ii)* emits immediately the termination event and *iii)* emits another event (called ***answer*** *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.
- the ***result*** of a Timed Action is an object (of type `AsynchActionGenericResult<T>`, see Subsection 6.7) that the 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.)
-

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 *answer event* emitted by the action. An actor can also suspend an action, and, if the action is recoverable, continue its execution later[16].

6.5 The class `ActorTimedAction`

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:



```java
package it.unibo.qactors.action;
import it.unibo.contactEvent.interfaces.IEventItem;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.QActor;

public abstract class ActorTimedAction extends AsynchActionGeneric<String> implements IActorAction{
   protected ActionTimer at;
   protected QActor myactor;
   protected ActionTimedEventHandler evh;
   protected String[] alarms ;
   protected String actionTimedResult = "unknown";
   protected String toutevId ="";
```

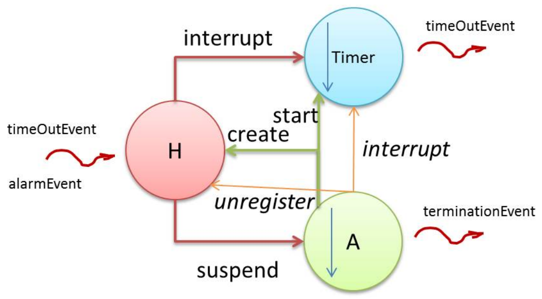
[15] Usually, `T` and `DURATION` are expressed in millisecs.

[16] 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.

```java
    protected String suspendevent = "notyetevaluated";

    public ActorTimedAction(String name, QActor myactor, boolean cancompensate,
            String terminationEvId, String answerEvId, String[] alarms,
            IOutputEnvView outEnvView, long maxduration) throws Exception {
        super(name, cancompensate, terminationEvId, answerEvId, outEnvView, maxduration);
        this.myactor  = myactor;
        this.alarms  = alarms;
        initActorTimedAction();
    }
    public ActorTimedAction(String name, QActor myactor, boolean cancompensate,
            String terminationEvId, String answerEvId, String alarms,
            IOutputEnvView outEnvView, long maxduration) throws Exception {
        this( name, myactor, cancompensate, terminationEvId, answerEvId, QActor.createArray(alarms), outEnvView,
             maxduration);
    }
    /*
     * 1) create a ActionTimer that emits a toutevId event
     * 2) create an EventHandler for the events alarms + toutevId that (when 'fires') calls suspendAction
     */
    protected void initActorTimedAction() throws Exception{
        toutevId = getName()+"tout";
        String[] events = new String[alarms.length + 1];
        if( alarms.length > 0 ){
            System.arraycopy(alarms,0,events,0,alarms.length );
            System.arraycopy(new String[]{toutevId},0,events,alarms.length,1 );
        }
        else events = new String[]{toutevId};
        at  = new ActionTimer(getName()+"Timer",this.outEnvView,this.maxduration,toutevId);
        evh = new ActionTimedEventHandler(getName()+"Evh", myactor.getContext(), this, at, events, outEnvView);
//      myactor.removeRule("tout( X,Y)"); //clean
    }
    protected void startOfAction() throws Exception{
//          println("%%% " + getName() + " startOfAction evhname=" + evh.getName() );
        suspended = false;
        at.start();
        super.startOfAction();
    }
    protected AsynchActionGenericResult<String> endActionInternal() throws Exception{
         at.interrupt();
         evh.unregister();
         return super.endActionInternal();
    }
    public void suspendAction(){
        suspended = true;
        IEventItem ev = evh.getEvent();
        if(ev!=null){
//          println("%%% ActorTimedAction " + getName() + " suspendAction for event=" + ev.getEventId() ); //+ " " +
     myself
            if( ev.getEventId().equals(toutevId)){
                myactor.removeRule("tout( X,Y)"); //clean
                myactor.addRule("tout("+ toutevId +","+ this.getName() + ")");
                suspendevent ="timeOut("+maxduration+")";
            }
            else suspendevent = "interrupted("+ev.getEventId()+")";
        }
//      println("%%% ActorTimedAction " + getName() + " interrupt myself=" + myself.getName() ); //
        myself.interrupt();
    }

    @Override
    public IEventItem getInterruptEvent(){
        return evh.getEvent();
    }

 }
```

Listing 1.17. ActorTimedAction

The architecture of the 'subsystem' built by the `init` operation of `ActorAction` is shown in the following picture:



6.6 The class `AsynchActionGeneric`

The generic, abstract class `AsynchActionGeneric<T>` provides the `API` for Timed Actions, as defined by the following interface [17]

```java
package it.unibo.qactors.action;
import java.util.concurrent.Future;

import it.unibo.contactEvent.interfaces.IEventItem;
import it.unibo.qactors.action.IActorAction.ActionRunMode;

public interface IAsynchAction<T> extends IObservableActionGeneric<T> {
    public T execSynch() throws Exception;
    public Future<T> execASynch() throws Exception;
    public T waitForTermination() throws Exception ;
    public void suspendAction();
    public void showMsg(String msg);

    public ActionRunMode getExecMode();
    public boolean isSuspended();
    public boolean canBeCompensated();
    public String getActionName();
    public String getTerminationEventId();
    public String geAnswerEventId();
    public int getMaxDuration();

    public IEventItem getInterruptEvent( );

    public void setTheName(String name);
    public void setAnswerEventId(String evId);
    public void setCanCompensate(boolean b);
    public void setTerminationEventId(String evId);
    public void setMaxDuration(int d);
}
```

Listing 1.18. `IAsynchAction<T>`

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

`AsynchActionGenericResult<T> execSynch()`	executes the action in synchronous way (waits for termination)
`Future<AsynchActionGenericResult<T» execAsynch`	executes the action in asynchronous way
`AsynchActionGenericResult<T> waitForTermination()`	waits until the action is terminated (mainly for execAsynch)
`void suspendAction()`	suspends the execution of the action if not already terminated
`ActionRunMode getExecMode()`	returns an instance of ActionRunMode (termination with answer or not)

[17] The code is included in the file *qactor18.jar*.

6.7 The class `AsynchActionGenericResult`

The generic class `AsynchActionGenericResult<T>` provides objects that represent the result of a Timed Action:

```java
package it.unibo.qactors.action;

public class AsynchActionGenericResult<T> {
protected AsynchActionGeneric<T> action;
protected long timeRemained ;
protected T result;
protected boolean suspended;
    public AsynchActionGenericResult( AsynchActionGeneric<T> action, T result, long time, boolean suspended){
        this.action       = action;
        this.result       = result;
        this.timeRemained = time;
        this.suspended    = suspended;
    }
    public long getTimeRemained(){
        return timeRemained;
    }
    public T getResult(){
        return result;
    }
    public boolean getInterrupted(){
        return suspended;
    }
    public void setResult(T result){
        this.result = result;;
    }
    @Override
    public String toString(){
        String execTime=""+action.getExecTime();
        String maxTime =""+action.getMaxDuration();
        return "asynchActionResult(ACTION, RESULT,SUSPENDED,TIMES)".
              replace("ACTION","action("+action.name+")").
              replace("RESULT","result("+result+")").
              replace("SUSPENDED","suspended("+suspended+")").
              replace("TIMES","times(exec("+execTime+"),max("+maxTime+"))");
    }
}
```

Listing 1.19. `AsynchActionGenericResult<T>`

6.8 Fibonacci as a Timed Action

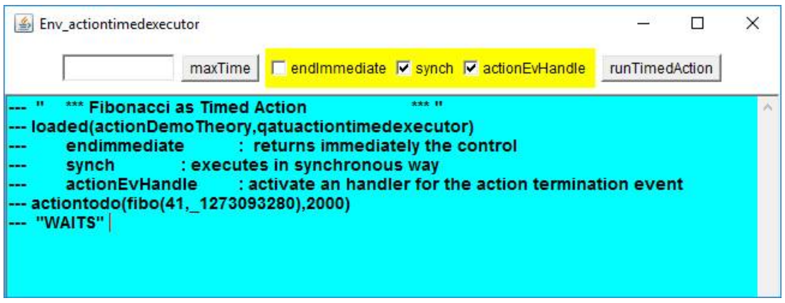
The `fibonacci` computation of Section 6 is defined in the following as a Timed Action that takes at construction time a goal of the form `fibo(N,V)` where `N` is the fibonacci number to evaluate and `V` is the (variable that denotes the) result.

```java
package it.unibo.actiontimedexecutor;
import alice.tuprolog.Struct;
import alice.tuprolog.Term;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.QActor;
import it.unibo.qactors.action.ActorTimedAction;

public class ActionTimedFibonacci extends ActorTimedAction{
private String goalTodo = "fibo(25,V).";
private int n = 0;
    public ActionTimedFibonacci(String name, QActor myactor, String goalTodo, boolean cancompensate,
            String terminationEvId, String answerEvId, String[] alarms, IOutputEnvView outView,
            long maxduration) throws Exception {
        super(name, myactor, cancompensate, terminationEvId, answerEvId, alarms, outView, maxduration);
        this.goalTodo = goalTodo;
//      println("%%% ActionTimedFibonacci CREATED goalTodo " + goalTodo );
    }
    /*
     * This operation is called by ActionObservableGeneric.call
     */
    @Override
    public void execTheAction() throws Exception {
     try{
         actionTimedResult = fibonacci( goalTodo );
         //The real result is set by endOgAction() -> getResult() -> getApplicationResult
//       println("%%% ActionTimedFibonacci ENDS result=" + actionTimedResult );
     }catch(Exception e){
         println("%%% ActionTimedFibonacci EXIT " + e.getMessage() );
     }
    }
    protected String fibonacci( String goalTodo ) throws Exception{
        Struct st = (Struct) Term.createTerm(goalTodo);
        n = Integer.parseInt( ""+st.getArg(0) );
        return ""+fibonacci(n);
    }
    protected long fibonacci( int n ) throws Exception{
        if( n<0 || n==0 || n == 1 ) return 1;
        else return fibonacci(n-1) + fibonacci(n-2);
    }
    @Override
    public String getApplicationResult() throws Exception {
        if( this.suspended)
            return "fibo(" + n + ", " + suspendevent +")" ;
        else
            return "fibo("+n+",val("+this.actionTimedResult+"),time("+this.durationMillis+"))";
    }
 }
```

Listing 1.20. `ActionTimedFibonacci<T>`

6.8.1 Experiments on Fibonacci as Timed Action.

The Project *it.unibo.robot.interactive*, package `it.unibo.actiontimedexecutor` shows the effects of the action in the different execution modes:



[ ] endImmediate [✓ synch] [✓ actionEvHandle]

```
--- "    *** Fibonacci as Timed Action                    *** "
--- loaded(actionDemoTheory,qatuactiontimedexecutor)
---       endimmediate          :  returns immediately the control
---       synch             : executes in synchronous way
---       actionEvHandle        : activate an handler for the action termination event
--- actiontodo(fibo(41,_1273093280),2000)
---  "WAITS"
```

6.9 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.

6.10 Fibonacci as a Reactive Action

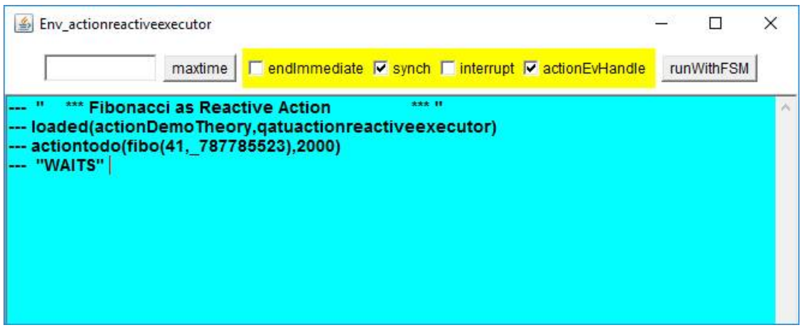
The `fibonacci` computation of Subsection 6.8 can be used to build a Reactive Action, by introducing a test to 'interrupt' the action if it has been suspended by some event.

```java
package it.unibo.actionreactiveexecutor;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.QActor;
import it.unibo.actiontimedexecutor.ActionTimedFibonacci;

public class ActionReactiveFibonacci extends ActionTimedFibonacci{
   public ActionReactiveFibonacci(String name, QActor myactor, String goalTodo, boolean cancompensate,
         String terminationEvId, String answerEvId, String[] alarms, IOutputEnvView outView,
         long maxduration) throws Exception {
      super(name, myactor, goalTodo, cancompensate, terminationEvId, answerEvId, alarms, outView, maxduration);
   }
   protected long fibonacci( int n ) throws Exception{
      if( n<0 || n==0 || n == 1 ) return 1;
      else{
         if( this.suspended ) return 0;
         long v1 = fibonacci(n-1);
         if( this.suspended ) return 0;
         long v2 = fibonacci(n-2);
         return v1 + v2;
      }
   }
}
```

Listing 1.21. `ActionReactiveFibonacci<T>`

6.10.1 Experiments on Fibonacci as Reactive Action.

The Project *it.unibo.robot.interactive*, package `it.unibo.actionreactiveexecutor` shows the effects of the action in the different execution modes. When the `interrupt` flag is set, an event `usercnd : stop` is generated[18] after `maxtime/2` *msecs*.



```
--- "   *** Fibonacci as Reactive Action          *** "
--- loaded(actionDemoTheory,qatuactionreactiveexecutor)
--- actiontodo(fibo(41,_787785523),2000)
---  "WAITS"
```

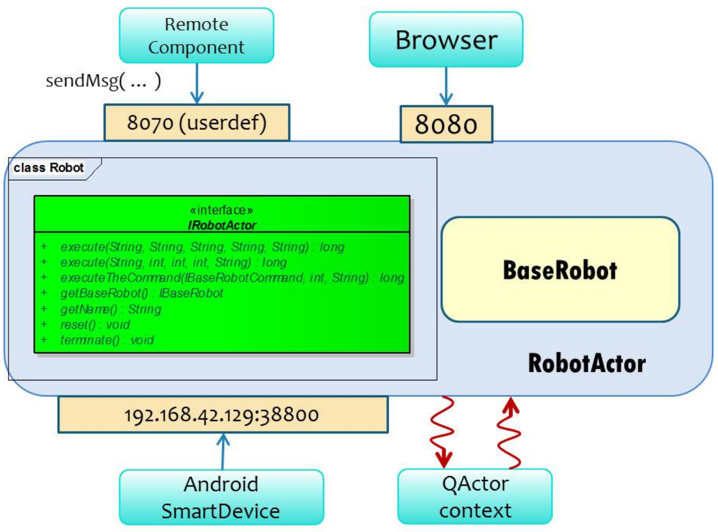
[18] The `QActor` operation `emitEventAsynch` performs the task to generate a given event after a given time.

7 Working with physical robots

The following table summarizes the projects that will be used for the development of software applications involving the usage of physical robots.

it.unibo.lab.baseRobot	The basic software for differential drive robots (`ddr`) that are able to move and to acquire sensor data. Library: *labbaseRobotSam.jar*.
it.unibo.xtext.robot.base	The metamodel (file extension `.baseddr`) for ddr-robot configuration. Plugin: *it.unibo.xtext.robot.base_1.0.0.jar* Plugin: *it.unibo.xtext.robot.base.ui_1.0.0.jar*
it.unibo.xtext.qactor.robot	The metamodel (file extension `.baseddr`) that extends the `QActor` metamodel with concepts related to a `ddr`. Plugin: *it.unibo.xtext.qactor.robot_1.2.5.jar* Plugin: *it.unibo.xtext.qactor.robot.ui_1.2.5.jar*.
it.unibo.qactor.robot	The run time support for `ddr` robots as qactors. Library: *uniboQactorRobot.jar*
it.unibo.xtext.qactor	The metamodel (file extension `.qa`) that defines the `QActor` metamodel . Plugin: *it.unibo.xtext.qactor_1.2.5.jar* Plugin: *it.unibo.xtext.qactor.ui_1.2.5.jar*.
it.unibo.qactor	The run time support for qactors. Library: *qactors18.jar*

A robot is implemented as a `QActor` able to interact with an instance of a *BaseRobot*, as shown in the following picture:



it.unibo.lab.robotUsage	Example of the usage of the `API` of a basic robot: a robot that executes some moves and handles a distance sensor.
it.unibo.qactor.robot.avatar	Examples of the usage of the `ddr` language/metamodel.

7.1 Description of robots

The physical configuration of a ddr-robot (with name `nano1`) must be described in a file named *iotRobot.properties* stored in a directory named `configuration/nano1`:

```
# =================================
# Generated by AN for SAM config lanaguage
# Pay attention to the spaces
# =================================
motor.left=gpio.motor
motor.left.pin.cw=8
motor.left.pin.ccw=9
motor.left.private=false
#
motor.right=gpio.motor
motor.right.pin.cw=12
motor.right.pin.ccw=13
motor.right.private=false
#
distance.front_top=hcsr04
distance.front_top.trig=0
distance.front_top.echo=2
distance.front_top.private=false
#
distance.back=serial
distance.back.private=false
#
line.bottom=serial
line.bottom.private=false
#
magnetometer.top=serial
magnetometer.top.private=false
#
#--------------------------------------------
#   COMPOSED COMPONENT
#--------------------------------------------
actuators.bottom=ddmotorbased
actuators.bottom.name=motors
actuators.bottom.comp=motor.left,motor.right
actuators.bottom.private=true
#--------------------------------------------
#   MAIN ROBOT
#--------------------------------------------
baserobot.bottom=differentialdrive
baserobot.bottom.name=nano1
baserobot.bottom.comp=actuators.bottom
baserobot.bottom.private=false
```

Listing 1.22. `iotRobot.properties` for nano1

This file is consulted by the basic robot support (*qactors18.jar*).

7.2 High Level Description of robot configuration

A file *configuration/nano1/iotRobot.properties* is automatically generated (project *it.unibo.xtext.robot.base*) by a custom DSL that provides a more high-level form of description:

7.2.1 The Mock robot.

```
/*
 * ==================================
 * mock
 * ==================================
 */
RobotBase mock
 //BASIC
```

```
motorleft = Motor [ simulated 0 ] position: LEFT
motorright = Motor [ simulated 0 ] position: RIGHT
l1Mock    = Line  [ simulated 0 ] position: BOTTOM
distFrontMock= Distance [ simulated 0 ] position: FRONT
mgn1 = Magnetometer [ simulated 0 ] private position: FRONT
//COMPOSED
rot   = Rotation [ mgn1 ] private position: FRONT
motors = Actuators [ motorleft , motorright ] private position: BOTTOM
Mainrobot mock [ motors,rot ]
;
```

Listing 1.23. `uniboRobots.baseddr`: rules for mock

In the *Mock* robot, all the actuators (motors) and the sensors are simulated.

7.2.2 The nano1 robot.

```
 /*
 * ==================================
 * nano1
 * ==================================
 */
RobotBase  nano1
//BASIC
motorleft = Motor [ gpiomotor pincw 8 pinccw 9 ] position: LEFT
motorright = Motor [ gpiomotor pincw 12 pinccw 13 ] position: RIGHT
distanceRadar = Distance [ sonarhcsr04 pintrig 0 pinecho 2] position: FRONT_TOP
//line = Line  [ gpioswitch pin 15 activelow ] position: BOTTOM
distanceBack = Distance [ serial rate 9600 ] position: BACK
linefBottom = Line [ serial rate 9600 ] position: BOTTOM
magneto = Magnetometer [ serial rate 9600 ] position: TOP
//COMPOSED
motors = Actuators [ motorleft , motorright ] private position: BOTTOM
Mainrobot nano1 [ motors ]
;
```

Listing 1.24. `uniboRobots.baseddr`: rules for nano1

In the *nano1* robot, the motors and a sonar are directly connected to a RaspberryPi, while another sonar, a magnetometer and a lime detector are connected to an Arduino (Micro).

7.3 Avatar

The project *it.unibo.qactor.robot.avatar* gives an example of a ddr-robot working as a device that executes commands sent from a user console (embedded in a browser):

```
RobotSystem avatar
/*
 * ===========================================================
 * avatar
 * FEATURE : A Robot is a QActor that can execute remote commands
 * TO NOTE :
 * ===========================================================
 */
Event usercmd : usercmd(X)  //from robot GUI; X=robotgui(CMD) CMD=s(low)
Event inputcmd : usercmd(X) //from input GUI; X=executeInput( do(G,M) )
Event alarm   : alarm(X)    //from red buttons X = fire |
Event sensordata : sensordata(X) //from SensorObserver

Context ctxAvatar ip [ host="localhost" port=8070 ]  -httpserver
EventHandler evh for alarm , sensordata -print ;

Robot smilzomock QActor avatar context ctxAvatar {
    Plan init normal
        println("A robot performs the command move sent by the user via http GUI" ) ;
```

```
      //sudo /home/pi/pi-blaster/pi-blaster
//        solve consult("./talkTheory.pl") time(0) onFailSwitchTo prologFailure ;
      solve consult("talkTheory.pl") time(0) onFailSwitchTo prologFailure ;
      switchToPlan cmdDriven;
      println("ENDS" )
   Plan cmdDriven resumeLastPlan
      println("wait for a command " ) ;
      sense time(600000) usercmd , inputcmd -> evalRobotCmd , evalInputCmd ;
      [ ?? tout(X,Y)] switchToPlan stopTheRobot ;
      repeatPlan 0
   Plan evalRobotCmd resumeLastPlan
      printCurrentEvent ;
      onEvent usercmd : usercmd( X ) -> solve actorOp( execCmdGui(X) ) time(0) ;
      [ !? actorOpResult(R)] println( resultRobotCmd(R) )
   Plan evalInputCmd resumeLastPlan
      printCurrentEvent ;
      onEvent inputcmd : usercmd( X ) -> solve X time(0) ;
      [ !? result(R)] println(resultInputCmd(R))

   Plan doWork resumeLastPlan
      switchToPlan doWorkViaProlog ;
      [ !? repeat(N) ] repeatPlan 4
   Plan doWorkViaProlog resumeLastPlan
      println("wait for user command " ) ;
      sense time(600000) usercmd -> continue ; //the command is executed by the handler
      [ ?? tout(X,Y)] switchToPlan stopTheRobot ;
//    printCurrentEvent -memo;
//    [ ?? msg(usercmd, _ , EMITTER, none , usercmd( X ), T) ] solve actorOp( execCmdGui( X ) ) time(0) ;
      onEvent usercmd : usercmd( X ) -> solve actorOp( execCmdGui(X) ) time(0) ; //react event alarm -> handleAlarm;
      [ ?? tout(X,Y)] switchToPlan stopTheRobot ;
      repeatPlan 0
   Plan stopTheRobot resumeLastPlan
      println("Stop the robot" )
//        solve actorOp( execCmdGui(robotgui( h(low) ) ) ) time(0)
   Plan prologFailure resumeLastPlan
      println("Prolog goal FAILURE" )
   Plan handleAlarm resumeLastPlan
      println("handleAlarm" )
}
```

Listing 1.25. `avatar.ddr`

By 'pressing' one of the buttons presented by the robot console, an event is generated. Please, look at the different kinds of events.

7.4 The operation actorOp

The built-in operation `actorOp/1` is implemented in Prolog (in the actor's *WorldTheory*) and accepts in input a term (`OP`) that should correspond to a public method defined by the actor. The operations calls the method `OP` and creates a (singleton) fact `actorOpDone/2` to remember the result `ROP` returned by the call. More, precisely, the result of `actorOp(OP)`is represented as a String (`ROPStr`) in the fact `actorOpDone(OP,ROPStr)`.

If the result `ROP` is bound to a non-primitive Java object, the system 'binds' such an object[19] to an internal identifier (`actoropresult`) so that it can be referenced in the application model.

To use the non-primitive Java object returned by `actorOp/1`, the application designer can exploit the built-in operation `actionResultOp(OP)`; it calls the method `OP` in the object currently referenced by the identifier `actoropresult`.

For example (see the project *it.unibo.robot.interactive*):

[19] The 'binding' is done by using a tuProlog *register* operation.

```
System actionOpdemo
Context ctxActionOpdemo ip [ host="localhost" port=8037 ] -g cyan
QActor qaactionop context ctxActionOpdemo {
   Plan main normal
      println("qadebug STARTS " ) ;
      switchToPlan testReturnPrimitiveData ;
      switchToPlan testReturnPojo ;
      println("qadebug ENDS " )
   Plan testReturnPrimitiveData resumeLastPlan
      solve actorOp( intToInt(5) ) time(0) ;
      [ !? actorOpDone( OP,R ) ] println( done(OP,R) )
   Plan testReturnPojo resumeLastPlan
      solve actorOp( getDate ) time(0) ;
      [ !? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
      solve actionResultOp( toInstant ) time(0) ;
      [ !? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
      solve actionResultOp( getNano ) time(0) ;
      [ !? actorOpDone( OP,R ) ] println( done(OP,R) )
}
```

Listing 1.26. actionOpdemo.qa

The operation getDate is defined by the application designer as follows:

```
/* Generated by AN DISI Unibo */
/*
This code is generated only ONCE
*/
package it.unibo.qaactionop;
import java.util.Calendar;
import java.util.Date;
import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.ActorContext;
public class Qaactionop extends AbstractQaactionop {
   public Qaactionop(String actorId, ActorContext myCtx, IOutputEnvView outEnvView ) throws Exception{
      super(actorId, myCtx, outEnvView);
   }
   public int intToInt( int n ) {
      return n+1;
   }
   public Date getDate( ) {
      Calendar rightNow = Calendar.getInstance();
      Date d = rightNow.getTime();
      return d;
   }
```

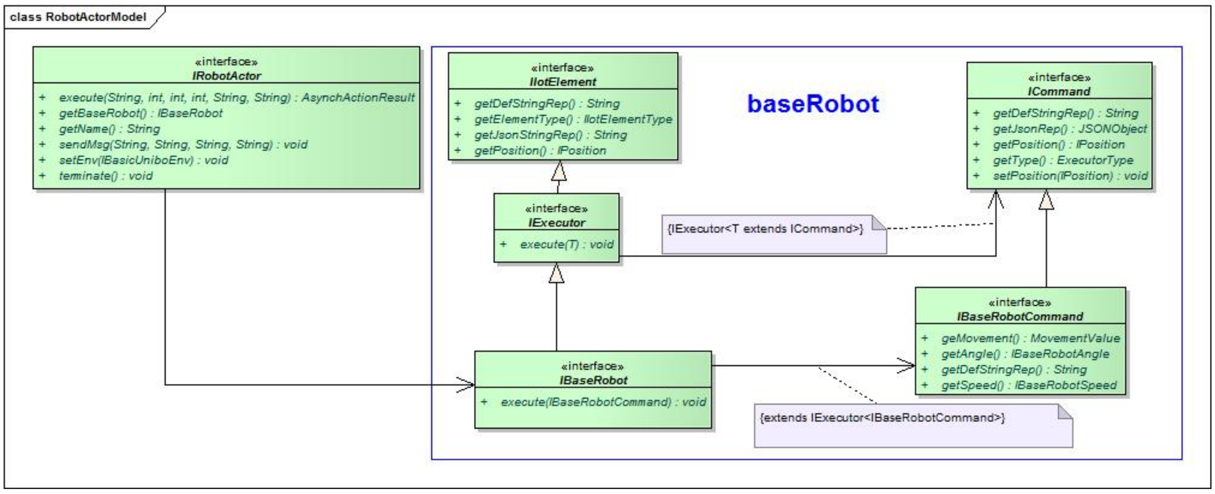
Listing 1.27. actionOpdemo.qa

The result is:

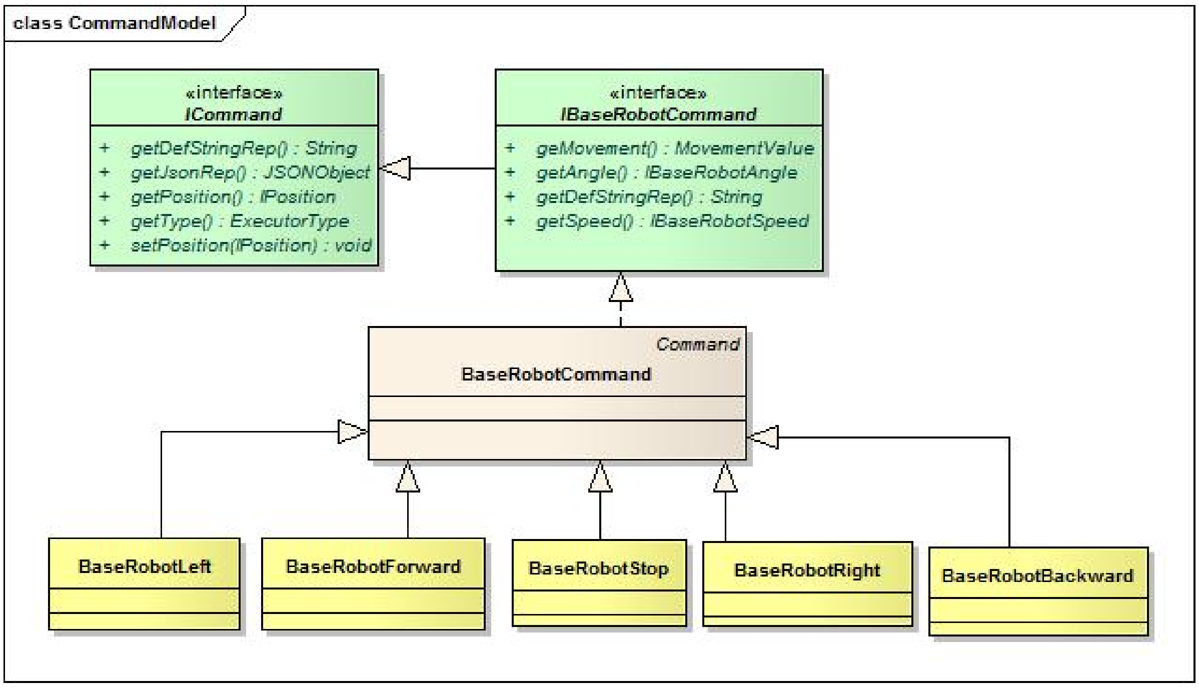
```
...
Starting the actors ....
---  "qadebug STARTS "
--- done(intToInt(5),'6')
--- done(getDate,'Mon May 16 18:03:15 CEST 2016')
--- done(toInstant,'2016-05-16T16:03:15.181Z')
--- done(getNano,'181000000')
---  "qadebug ENDS "
```

7.5 Models

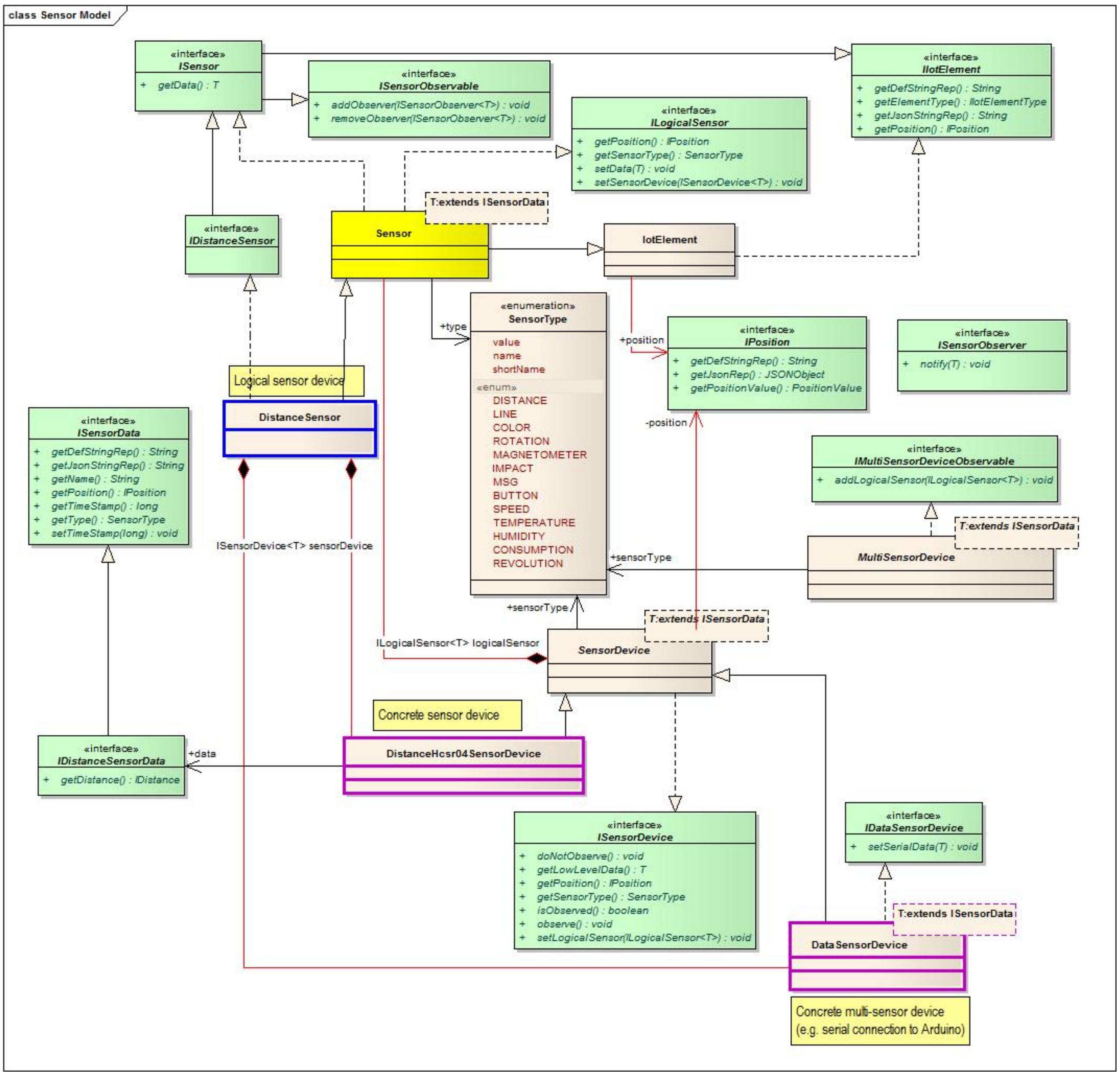
7.5.1 A model of the robot.


class RobotActorModel
«interface»
IRobotActor
+ execute(String, int, int, int, String, String) : AsynchActionResult
+ getBaseRobot() : IBaseRobot
+ getName() : String
+ sendMsg(String, String, String, String) : void
+ setEnv(IBasicUniboEnv) : void
+ teminate() : void
baseRobot
«interface»
IIotElement
+ getDefStringRep() : String
+ getElementType() : IIotElementType
+ getJsonStringRep() : String
+ getPosition() : IPosition
«interface»
ICommand
+ getDefStringRep() : String
+ getJsonRep() : JSONObject
+ getPosition() : IPosition
+ getType() : ExecutorType
+ setPosition(IPosition) : void
«interface»
IExecutor
+ execute(T) : void
{IExecutor<T extends ICommand>}
«interface»
IBaseRobotCommand
+ geMovement() : MovementValue
+ getAngle() : IBaseRobotAngle
+ getDefStringRep() : String
+ getSpeed() : IBaseRobotSpeed
«interface»
IBaseRobot
+ execute(IBaseRobotCommand) : void
{extends IExecutor<IBaseRobotCommand>}

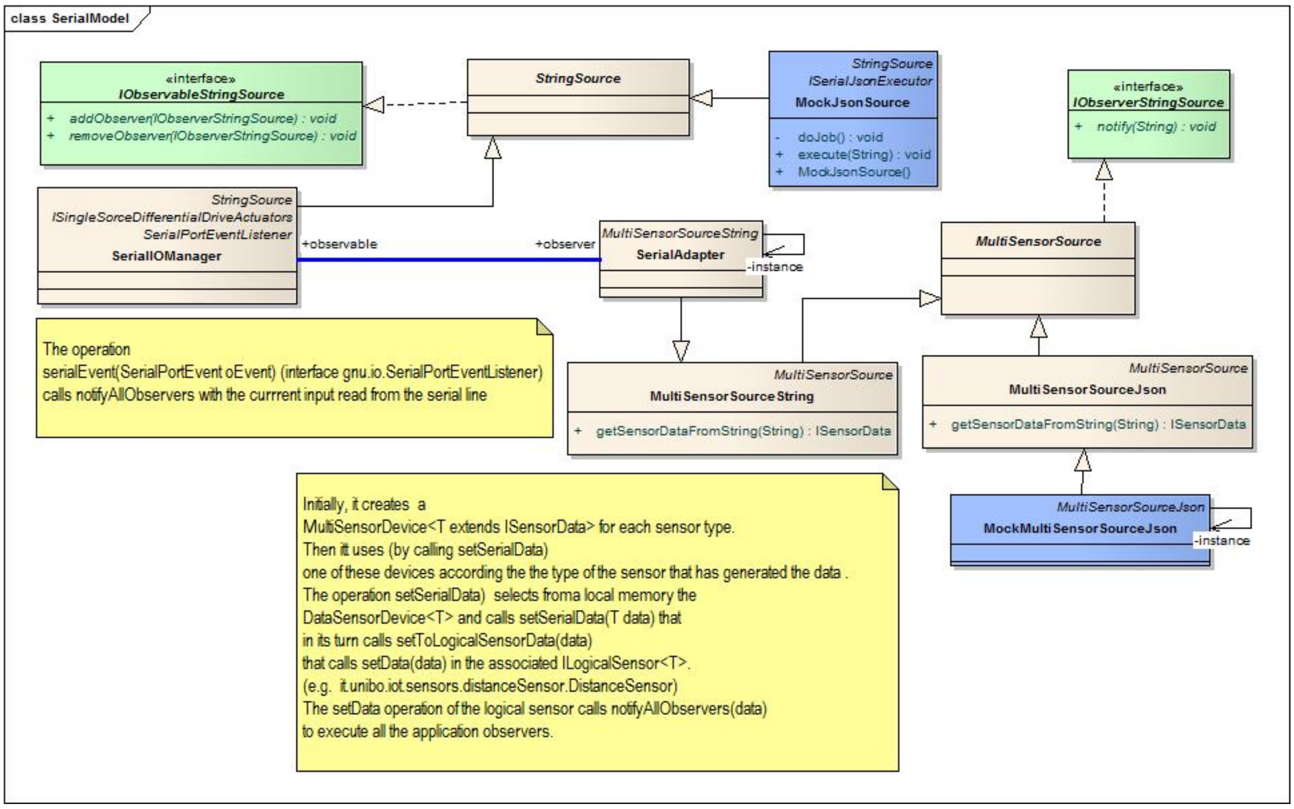
7.5.2 A model of the robot commands.


class CommandModel
«interface»
ICommand
+ getDefStringRep() : String
+ getJsonRep() : JSONObject
+ getPosition() : IPosition
+ getType() : ExecutorType
+ setPosition(IPosition) : void
«interface»
IBaseRobotCommand
+ geMovement() : MovementValue
+ getAngle() : IBaseRobotAngle
+ getDefStringRep() : String
+ getSpeed() : IBaseRobotSpeed
Command
BaseRobotCommand
BaseRobotLeft
BaseRobotForward
BaseRobotStop
BaseRobotRight
BaseRobotBackward

7.5.3 A model of sensors.


class Sensor Model
«interface» ISensor
+ getData() : T
«interface» ISensorObservable
+ addObserver(ISensorObserver<T>) : void
+ removeObserver(ISensorObserver<T>) : void
«interface» IIotElement
+ getDefStringRep() : String
+ getElementType() : IIotElementType
+ getJsonStringRep() : String
+ getPosition() : IPosition
«interface» ILogicalSensor
+ getPosition() : IPosition
+ getSensorType() : SensorType
+ setData(T) : void
+ setSensorDevice(ISensorDevice<T>) : void
«interface» IDistanceSensor
Sensor
T:extends ISensorData
IotElement
«enumeration» SensorType
value
name
shortName
«enum»
DISTANCE
LINE
COLOR
ROTATION
MAGNETOMETER
IMPACT
MSG
BUTTON
SPEED
TEMPERATURE
HUMIDITY
CONSUMPTION
REVOLUTION
+type
+position
«interface» IPosition
+ getDefStringRep() : String
+ getJsonRep() : JSONObject
+ getPositionValue() : PositionValue
«interface» ISensorObserver
+ notify(T) : void
Logical sensor device
DistanceSensor
-position
«interface» ISensorData
+ getDefStringRep() : String
+ getJsonStringRep() : String
+ getName() : String
+ getPosition() : IPosition
+ getTimeStamp() : long
+ getType() : SensorType
+ setTimeStamp(long) : void
«interface» IMultiSensorDeviceObservable
+ addLogicalSensor(ILogicalSensor<T>) : void
T:extends ISensorData
MultiSensorDevice
ISensorDevice<T> sensorDevice
+sensorType
+sensorType
T:extends ISensorData
ILogicalSensor<T> logicalSensor
SensorDevice
Concrete sensor device
«interface» IDistanceSensorData
+ getDistance() : IDistance
+data
DistanceHcsr04SensorDevice
«interface» ISensorDevice
+ doNotObserve() : void
+ getLowLevelData() : T
+ getPosition() : IPosition
+ getSensorType() : SensorType
+ isObserved() : boolean
+ observe() : void
+ setLogicalSensor(ILogicalSensor<T>) : void
«interface» IDataSensorDevice
+ setSerialData(T) : void
T:extends ISensorData
DataSensorDevice
Concrete multi-sensor device
(e.g. serial connection to Arduino)

7.5.4 A model of serial.


class SerialModel
«interface» IObservableStringSource
+ addObserver(IObserverStringSource) : void
+ removeObserver(IObserverStringSource) : void
StringSource
StringSource ISerialJsonExecutor MockJsonSource
- doJob() : void
+ execute(String) : void
+ MockJsonSource()
«interface» IObserverStringSource
+ notify(String) : void
StringSource ISingleSorceDifferentialDriveActuators SerialPortEventListener SerialIOManager
+observable
+observer
MultiSensorSourceString SerialAdapter
-instance
MultiSensorSource
MultiSensorSource MultiSensorSourceString
+ getSensorDataFromString(String) : ISensorData
MultiSensorSource MultiSensorSourceJson
+ getSensorDataFromString(String) : ISensorData
MultiSensorSourceJson MockMultiSensorSourceJson
-instance

The operation
serialEvent(SerialPortEvent oEvent) (interface gnu.io.SerialPortEventListener)
calls notifyAllObservers with the currrent input read from the serial line

Initially, it creates a
MultiSensorDevice<T extends ISensorData> for each sensor type.
Then it uses (by calling setSerialData)
one of these devices according the the type of the sensor that has generated the data .
The operation setSerialData) selects froma local memory the
DataSensorDevice<T> and calls setSerialData(T data) that
in its turn calls setToLogicalSensorData(data)
that calls setData(data) in the associated ILogicalSensor<T>.
(e.g. it.unibo.iot.sensors.distanceSensor.DistanceSensor)
The setData operation of the logical sensor calls notifyAllObservers(data)
to execute all the application observers.

7.6 Sensors

Sensors are implemented as observable `POJO` (see Subsection 7.5.3). The generated class `SensorObserver` delegates to the rule `sensor/1` of *sensorTheory* (see Subsection 7.6.1) the policy to handle the sensor data.

```
/* Generated by AN DISI Unibo */
package it.unibo.avatar;
import it.unibo.contactEvent.interfaces.IContactEventPlatform;
import it.unibo.contactEvent.platform.ContactEventPlatform;
import it.unibo.iot.models.sensorData.ISensorData;
import it.unibo.iot.sensors.ISensorObserver;
import it.unibo.is.interfaces.IOutputView;
import it.unibo.qactors.QActor;
import it.unibo.system.SituatedPlainObject;

public class SensorObserver<T extends ISensorData> extends SituatedPlainObject implements ISensorObserver<T>{
protected  IContactEventPlatform platform;
protected QActor actor;
    public SensorObserver(QActor actor, IOutputView outView) {
        super(outView);
        this.actor = actor;
    }
    @Override
    public void notify(T data) {
        try {
            //println("SensorObserver: " + data.getClass().getName() );
            //println("SensorObserver: " + data.getDefStringRep() );
            /*
               We delegate to sensor/1 of sensorTheory the policy to handle the sensor DATA
               inlcluding raising of events like sensordata:sensordata(DATA)
               SENSORDATA raw
                  {"p":"b","t":"l","d":{"detection":0}},"tm":16124773}
                  {"p":"t","t":"m","d":{"raw3axes":{"x":308, "y":-649, "z":120}}}
                  {"p":"b","t":"l","d":{"detection":1}},"tm":520200}
               SENSORDATA high level
                  distance( VALUE, DIRECTION, POSITION )
                  magnetometer(x(VX),y(VY),z(VZ),POSITION)
            */
//          platform.raiseEvent("sensor", "sensordata", "sensordata("+data.getDefStringRep()+")" );
            String goal = "sensor( DATA )".replace("DATA", data.getDefStringRep());
            actor.solveGoal( goal , 0, "","" , "" );
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Listing 1.28. SensorObserver.java

7.6.1 The sensor Theory. The *sensorTheory* is generated (only once) in the src directory in order to allow the application designer to introduce sensor handling rules related to the application logic. Its initial content is defined as follows:

```
%============================================
% sensorTheory.pl for actor avatar
%============================================
%% **** SONAR ****
sensor( distance(DIST,DIR,POS) ):-
    !, actorobj(Actor),
    ( DIST < 20,!,
      %% Actor <- emit( sensordata, sensordata(distance(DIST,DIR,POS)) ),
      actorPrintln( distance(DIST,DIR,POS) ) ;
      true ).
%% **** MAGNETOMETER ****
sensor( magnetometer(x(VX),y(VY),z(VZ),POS) ):-
     !, actorobj(Actor),
     %% Actor <- emit( sensordata, sensordata(magnetometer(x(VX),y(VY),z(VZ),POS)) ),
     actorPrintln( magnetometer(x(VX),y(VY),z(VZ),POS) ).
%% **** LINE DETECTO ****
sensor( line( V,POS) ):-
     !, actorobj(Actor),
     %% Actor <- emit( sensordata, sensordata(line(V,POS)) ),
     actorPrintln( line( V,POS) ).

sensor( X ):-
    actorPrintln( X ).

/*
------------------------------------------------------------
initialize
------------------------------------------------------------
*/
initSensorTheory :-
    actorobj(Actor),
    ( Actor <- isSimpleActor returns R, R=true, !,
      actorPrintln(" *** sensorTheory loaded FOR ACTORS ONLY *** ");
      actorPrintln(" *** sensorTheory loaded FOR ROBOTS *** ")
    ).

:- initialization(initSensorTheory).
```

Listing 1.29. sensorTheory.pl

7.6.2 Sensors handled by Arduino.

The file `uniboArduinoBaseSupport.ino` in project *it.unibo.qactor.robot* provides a support to handle motors and sensors coherent with the assumptions of *it.unibo.lab.baseRobot*:

```
/*
 * ARDUINO MICRO
 * it.unibo.actor.robot/arduino
 */
#include "GY_85.h"
#include <ArduinoJson.h>
#include<Wire.h>      //For MPU8050 and GY85

#define usingGY85  true    //Arduino connected to GY85
#define using6050  false   //Arduino connected to MPU6050
#define jsonFormat true
#define serialFormat false

boolean gyroscope  = false ;
boolean accelerom  = false ;
boolean temperature = true ;
boolean compass    =  true ;
boolean sonar      =  true ;
boolean line       =  true ;

#define aceeleromData   0
#define gyroData        1
#define magnetomData    2
#define temperatureData 3

void explain(){
  Serial.println("------------------------------------------------------------");
  Serial.println("uniboArduinoBaseSupport in it.unibo.qactor.robot/arduino");
  Serial.println("By AN from the Sam's original version bbb");
  if( usingGY85 ) explainGy85();
  else explain6050();
  explainInput();
  Serial.println("------------------------------------------------------------");
}

void explainInput(){
  Serial.println("INPUT r: (reset) no sensor data");
  Serial.println("INPUT a: accelerometer data");
  Serial.println("INPUT g: gyroscope data");
  Serial.println("INPUT l: line data");
  Serial.println("INPUT m: compass data");
  Serial.println("INPUT s: sonar data");
  Serial.println("INPUT t: temperature data");
}
void setup()
{
  Serial.begin(9600);
  setupSensors();
```

Listing 1.30. `uniboArduinoBaseSupport.ino`

7.7 Motors

Let us report here an example of using the `GPIO` library to control a motor via `PWM`

```
#!/bin/bash
# -----------------------------------------------------------------
# nanoMotorDrive.sh
# test for nano0
# Key-point: we can manage a GPIO pin by using the GPIO library.
# On a OC, edit this file as UNIX
# -----------------------------------------------------------------

in1=2 #WPI 8 BCM 2 PHYSICAL 3
```

```
in2=3 #WPI 9 BCM 3 PHYSICAL 5
inwp1=8
inwp2=9

if [ -d /sys/class/gpio/gpio2 ]
then
 echo "in1 gpio${in1} exist"
 gpio export ${in1} out
else
 echo "creating in1 gpio${in1}"
 gpio export ${in1} out
fi

if [ -d /sys/class/gpio/gpio3 ]
then
 echo "in2 gpio${in2} exist"
 gpio export ${in2} out
else
 echo "creating in2 gpio${in2}"
 gpio export ${in2} out
fi

gpio readall

echo "run 1"
 gpio write ${inwp1} 0
 gpio write ${inwp2} 1
 sleep 1.5

echo "run 2"
 gpio write ${inwp1} 1
 gpio write ${inwp2} 0
 sleep 1.5

echo "stop"
 gpio write ${inwp1} 0
 gpio write ${inwp2} 0

gpio readall
```

Listing 1.31. `nanoMotorDriveA.sh`

7.7.1 Servo

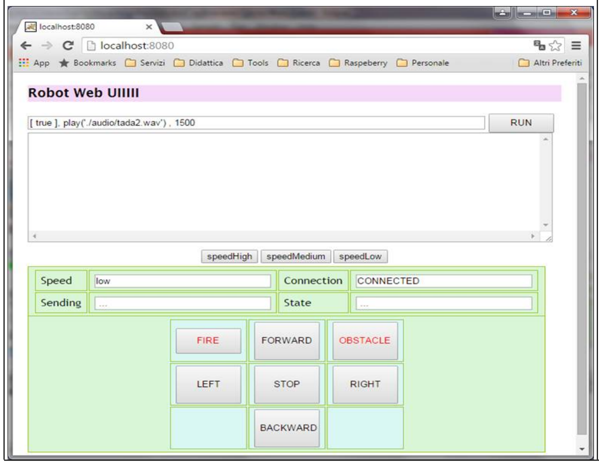
7.7.2 The pi-blaster. The pi-blaster project enables `PWM` on the `GPIO` pins. If we start pi-blaster without any parameters, it will enable `PWM` on the default pins;

```
cd /home/pi/pwm/piBlaster/
sudo ./pi-blaster
Channel number    GPIO number   Pin in P1 header
      0                4            P1-7
      1               17            P1-11
      2               18            P1-12
      3               21            P1-13
      4               22            P1-15
      5               23            P1-16
      6               24            P1-18
      7               25            P1-22
Set GPIO pin 17 to a PWM of 20%
echo "17=0.2" > /dev/pi-blaster
```

8 Interacting with physical robots

In Section 3 we said that a human user can remotely interact with a robot by using a a web interface. Moreover, each robot (being an actor) can be equipped with a local GUI.

8.1 The (remote) web user interface



[true], play('./audio/tada2.wav') , 1500

Speed: low
Connection: CONNECTED
Sending: ...
State: ...

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.

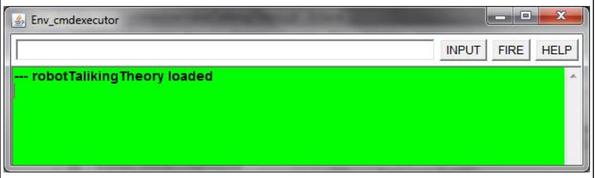
The `RUN` button at the top of the GUI allow us to ask the robot to execute actions, while the buttons at the bottom allows us to move the (basic) robot and send alarms.

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:

`usercmd : usercmd(executeInput(CMD))`	(`RUN` button)
`usercmd : usercmd(robotgui(MOVE))`, `MOVE=w(low),...,s(high)`	(`MOVE` button)
`alarm : alarm(fire)`	(`FIRE` button)
`alarm : alarm(obstacle)`	(`OBSTACLE` button)

8.2 The (local) GUI user interface



--- robotTalkingTheory loaded

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.

8.3 Inspect the state and elaborate in a functional way

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

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

8.4 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:

`assign(x,3)`	set `x=3`
`inc(x,1,V)`	binds V to the result of `x+1`.
`assign(x,1),inc(x,3,V),actorPrintln(v(V))`	binds V to 4 and prints `v(4)`.
`assign(x,10),getVal(x,VX),actorPrintln(x(VX))`	prints `x(10)`.
`addRule(r1(a)),r1(X),actorPrintln(X)`	add the fact `r1/1` to the actor's WorldTheory and prints `a`

Note that with the built-in (Web)GUI interface we work with a limited syntax. For example, since the symbol `':-'` ' is not admitted, we cannot write `addRule(r(X):-q(X))`.

8.5 Robot move actions

A robot is defined as an actor able to perform physical moves and to pereceive senosr data from the physical environment. Thus the user should be able to ask the robot to execute basic move actions in a 'naive' way or as timed or reactive actions.

`move(mf,100,0)`	move forward with full speed
`move(mf,100,0),1000`	move forward with full speed for 1 sec
`move(mf,100,0),3000,endmove`	move forward with full speed for 3 secs in asynchronous way and at the end emits the event `endmove`
`move(mf,100,0),3000,[alarm],[handleAlarm]`	move forward with full speed for 10 sec and reacts to the 'alarm' event

8.6 Reactive actions

`play('./audio/music_interlude20.wav'),5000, [alarm],[handleAlarm]`	play a sound in reactive way
`move(mf,100,0),3000,[alarm],[handleAlarm]`	move forward with full speed for 10 sec and reacts to the 'alarm' event

8.7 Building an interpreter

The `talkTheory.pl`.

9 Interactions using MQTT

The MQ *Telemetry Transport* (MQTT) is an ISO standard (ISO/IEC PRF 20922) publish-subscribe based "light weight" messaging protocol for use on top of the TCP/IP protocol. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium.

The *Eclipse Paho* project provides open-source client implementations of MQTT and MQTT-SN (*MQTT For Sensor Networks*[20]) messaging protocols aimed at new, existing, and emerging applications for *Machine-to-Machine* (M2M) and *Internet of Things* (IoT). There are already MQTT C and Java libraries with Lua, Python, C++ and JavaScript at various stages of development.

The usage of the MQTT protocol can be delegated to rules defined by the application designer in some theory, e.g. a mqttTheory.pl.

9.1 The mqttTheory

The mqttTheory that 'extends' the qa action-set with new operations for the usage of the MQTT protocol can be defined as follows:

```
/*
===============================================================
mqttTheory.pl
===============================================================
*/
connect( Name, BrokerAddr, Topic ):-
   java_object("it.unibo.mqtt.utils.MqttUtils", [], UMQTT),
   actorobj(A),
   actorPrintln( connect(UMQTT, A, Name, BrokerAddr, Topic ) ),
   UMQTT <- connect(A, Name, BrokerAddr, Topic ).
disconnect :-
   actorPrintln( disconnect ),
   class("it.unibo.mqtt.utils.MqttUtils") <- getMqttSupport returns UMQTT,
   %% actorPrintln( disconnect( UMQTT ) ),
   UMQTT <- disconnect.

publish( Name, BrokerAddr, Topic, Msg, Qos, Retain ):-
   actorPrintln( publish( Name, BrokerAddr, Topic, Msg, Qos, Retain ) ),
   actorobj(A),
   %% java_object("it.unibo.mqtt.utils.MqttUtils", [], UMQTT),
   class("it.unibo.mqtt.utils.MqttUtils") <- getMqttSupport returns UMQTT,
   %% actorPrintln( publish( UMQTT ) ),
   UMQTT <- publish(A, Name, BrokerAddr, Topic, Msg, Qos, Retain).
subscribe( Name, BrokerAddr, Topic ):-
   actorPrintln( subscribe( BrokerAddr, Topic ) ),
   actorobj(A),
   %% java_object("it.unibo.paho.utils.MqttUtils", [], UMQTT),
   class("it.unibo.mqtt.utils.MqttUtils") <- getMqttSupport returns UMQTT,
   %% actorPrintln( subscribe( UMQTT ) ),
   UMQTT <- subscribe(A, Name, BrokerAddr, Topic ).

mqttinit :-  actorPrintln("mqttTheory started ...") .
:- initialization(mqttinit).
```

Listing 1.32. The mqttTheory.pl

These rules in their turn make use of a Java utility object of class MqttUtils.

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

9.2 The MqttUtils

The Java utility class to be used as a support for MQTT interaction can be defined as follows:

```java
package it.unibo.mqtt.utils;
import it.unibo.contactEvent.interfaces.IContactEventPlatform;
import it.unibo.contactEvent.platform.ContactEventPlatform;
import it.unibo.qactors.QActor;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;

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

    public static MqttUtils getMqttSupport( ){
        return myself ;
    }
    public MqttUtils(){
        try {
            platform = ContactEventPlatform.getPlatform();
            myself  = this;
            System.out.println("        %%% MqttUtils created "+ myself );
        } catch (Exception e) {
            System.out.println("        %%% MqttUtils WARNING: "+ e.getMessage() );
        }
    }
//  public void connect( String clientid, String brokerAddr, String topic ) throws MqttException{
//      connect(actor,clientid,brokerAddr, topic);
//      System.out.println("        %%% MqttUtils connect done " );
//  }
    public void connect(QActor actor, String brokerAddr, String topic ) throws MqttException{
        System.out.println("        %%% MqttUtils connect/3 " );
        clientid = MqttClient.generateClientId();
        connect(actor, clientid, brokerAddr, topic);
    }
    public void connect(QActor actor, String clientid, String brokerAddr, String topic ) throws MqttException{
        System.out.println("        %%% MqttUtils connect/4 "+ clientid );
        this.actor = actor;
        client = new MqttClient(brokerAddr, clientid);
        MqttConnectOptions options = new MqttConnectOptions();
        options.setWill("unibo/clienterrors", "crashed".getBytes(), 2, true);
        client.connect(options);
    }
    public void disconnect( ) throws MqttException{
        System.out.println("        %%% MqttUtils disconnect "+ client );
        if( client != null ) client.disconnect();
    }
    public void publish(QActor actor, String clientid, String brokerAddr, String topic, String msg, int qos, boolean
         retain) throws MqttException{
        MqttMessage message = new MqttMessage();
        message.setRetained(retain);
        if( qos == 0 || qos == 1 || qos == 2){//qos=0 fire and forget; qos=1 at least once(default);qos=2 exactly once
            message.setQos(0);
        }
        message.setPayload(msg.getBytes());
        System.out.println("        %%% MqttUtils publish "+ message );
        client.publish(topic, message);
    }
//  public void subscribe( String clientid, String brokerAddr, String topic) throws Exception {
//      subscribe(actor,clientid,brokerAddr, topic);
//     System.out.println("        %%% MqttUtils subscribe done " );
// }
   public void subscribe(QActor actor, String clientid, String brokerAddr, String topic) throws Exception {
       try{
           this.actor = actor;
           client.setCallback(this);
           client.subscribe(topic);
       }catch(Exception e){
               System.out.println("        %%% MqttUtils subscribe error "+ e.getMessage() );
               eventMsg = "mqtt(" + eventId +", failure)";
               System.out.println("        %%% MqttUtils subscribe error "+ eventMsg );
               //platform.raiseEvent("mqttutil", eventId, eventMsg );
               if( actor != null ) actor.sendMsg("mqttmsg", actor.getName(), "dispatch", "error");
               throw e;
       }
   }
   @Override
   public  void connectionLost(Throwable cause) {
       System.out.println("        %%% MqttUtils connectionLost = "+ cause.getMessage() );
   }
   @Override
   public  void deliveryComplete(IMqttDeliveryToken token) {
       System.out.println("        %%% MqttUtils deliveryComplete token= "+ token );
   }
   @Override
   public void messageArrived(String topic, MqttMessage msg) throws Exception {
       System.out.println("messageArrived on "+ topic + "="+msg.toString());
       String mqttmsg = "mqttmsg(" + topic +"," + msg.toString() +")";
       System.out.println("        %%% MqttUtils messageArrived mqttmsg "+ mqttmsg);
//     platform.raiseEvent("mqttutil", eventId, mqttmsg ); //events are here 'equivalent' to messages
       if( actor != null ) actor.sendMsg("mqttmsg", actor.getName(), "dispatch", mqttmsg);
   }

   /*
    * ================================================================
    * TESTING
    * ================================================================
    */

       public void test() throws Exception{
           System.out.println("        %%% MqttUtils test " );
           connect(null,"qapublisher_mqtt", "tcp://m2m.eclipse.org:1883", "unibo/mqtt/qa");
           publish(null,"qapublisher_mqtt","tcp://m2m.eclipse.org:1883", "unibo/mqtt/qa", "sensordata(aaa)", 1, true);

           connect( null,"observer_mqtt", "tcp://m2m.eclipse.org:1883", "unibo/mqtt/qa");
           subscribe(null,"observer_mqtt", "tcp://m2m.eclipse.org:1883", "unibo/mqtt/qa");

       }

       public static void main(String[] args) throws Exception{
           new MqttUtils().test();
       }
}
```

Listing 1.33. The utility class `MqttUtils.java`

An object of class `MqttUtils` is used as a *singleton* and works as the support for the actor that calls the operation `connect` (that creates a `MqttClient`).

The `subscribe` operation sets this singleton support as the object that provides the callback (`messageArrived`) to be called when the `MqttClient` is a subscriber. The callback is defined so to map a `MqttMessage` into a dispatch of the form:

```
mqttmsg : mqttmsg( TOPIC,PAYLOAD )
```

This dispatch is then sent to the actor that uses the singleton support, i.e. that works as a `MqttClient` (subscriber).