An introduction to the usage of QActors and QRobots

Antonio Natali

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

Table of Contents

1 Introduction to QActors

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

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

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

1.1 Example: The 'hello world'

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

```
/*
 * hello.qa
 */
System helloSystem
Context ctxHello ip [ host="localhost" port=8079 ]

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

Listing 1.1. `hello.qa`

This example shows that each qactor works within a *Context* that models a computational node associated with a network `IP` (`host`) and a communication `port` (see Subsection 3.5). The behaviour of a is modelled as a set of macro-actions called *plans* (see Subsection 2.2).

1.2 QActor specification

A *QActor* specification can be viewed as:

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

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

- execute actions (see Subsection 2.1
- send/receive messages (see Subsection 3.6)
- emit/perceive events (see Subsection 3.11)

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

1.3 The generated code

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

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

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

1.4 The work of the application designer

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

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

1.5 Example: Actor as a finite state machine

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

```
/*
 * basic.qa
 * A system composed of a qactor (player)
 * working in a context named 'ctxBasic' associated with a GUI
 */
System basic   //the testing flag avoids automatic termination
Context ctxBasic ip [ host="localhost" port=8079 ] -g cyan

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

```
        [ ?? tout(X,Y) ] println( tou(X,Y) ) ;
        repeatPlan 1

 }
```

Listing 1.2. `basic.qa`

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

1.6 Example: Message-based interaction

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

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

```
/*
 * basicProdCons.qa
 * A system composed of a producer a consunmer
 */
System basic -testing
Dispatch info : info(X)

Context ctxBasicProd ip [ host="localhost" port=8079 ] -g cyan
Context ctxBasicCons ip [ host="localhost" port=8089 ] -g yellow

QActor producer context ctxBasicProd{
    Plan init normal
        println( producer(starts) ) ;
        //The producer sends a message to itself
        forward producer -m info : info("self message");
        switchToPlan produce ;
        switchToPlan getSelfMessage ;
        println( producer(ends) )
    Plan produce
        //delay time(1500);   //to force a timeout in the consumer
        println( producer(sends) ) ;
        forward consumer -m info : info(1);
        resumeLastPlan
    Plan getSelfMessage
        receiveMsg time(1000);
        printCurrentMessage;
        resumeLastPlan
 }

 QActor consumer context ctxBasicCons{
    Plan init normal
        println( consumer(starts) ) ;
        switchToPlan consume ;
        println( consumer(ends) )
    Plan consume
      receiveMsg time(2000) ;
      [ ?? tout(R,W)] endPlan "consumer timeout";
      printCurrentMessage -memo; //-memo: the current message is stored in the actor KB
      resumeLastPlan
 }
```

Listing 1.3. `basicProdCons.qa`

Here is an example of testing (written by the application designer):

```java
package it.unibo.ctxBasicCons;
import static org.junit.Assert.*;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import alice.tuprolog.SolveInfo;
import it.unibo.ctxBasicProd.MainCtxBasicProd;
import it.unibo.qactors.QActorUtils;
import it.unibo.qactors.akka.QActor;

public class TestProdCons {
      private QActor prod;
      private QActor cons;
      private  ScheduledThreadPoolExecutor sched =
             new ScheduledThreadPoolExecutor( Runtime.getRuntime().availableProcessors() );

    @Before
    public void setUp() throws Exception {
        activateCtxs();
        //Get a reference to the two actors:
        //getQActor waits until the actor is active
        prod  = QActorUtils.getQActor("producer_ctrl");
        cons  = QActorUtils.getQActor("consumer_ctrl");
    }
     @After
     public void terminate(){
        System.out.println("====== terminate " );
     }
     @Test
     public void execTest() {
         System.out.println("====== execTest ==============" );
         try {
                assertTrue("execTest prod", prod != null );
                assertTrue("actorTest cons", cons != null );
                System.out.println("====== execTest waits for work completion ... " );
                Thread.sleep(2000); //give the time to work
                //Acquire the message stored at the consumer site
                SolveInfo sol = cons.solveGoal("msg( MSGID, MSGTYPE, SENDER, RECEIVER, CONTENT, SEQNUM )");
                System.out.println("execTest CONTENT="+sol.getVarValue("CONTENT"));
                assertTrue("actorTest info", sol.getVarValue("CONTENT").toString().equals("info(1)") );
         } catch (Exception e) {
                fail("actorTest " + e.getMessage() );
         }
     }
     private void activateCtxs( ){
         sched.submit( new Runnable(){
            @Override
            public void run() {
                try {
                    //QActorContext ctxCons =
                        MainCtxBasicCons.initTheContext();
                } catch (Exception e) { e.printStackTrace(); }
            }
         });
         sched.submit( new Runnable(){
            @Override
            public void run() {
                try {
                    //QActorContext ctxProd =
                        MainCtxBasicProd.initTheContext();
                } catch (Exception e) { e.printStackTrace(); }
            }
         });
     }
}
```

Listing 1.4. `TestProdCons.qa`

2 QActor concept overview

2.1 Actions

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

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

1. as changes in the state of the "*actor's 'mind'*";
2. as changes in the actor's working environment.

The first kind of actions are referred here as *logical actions* since they do not affect the physical world. The ***actor-mind*** is represented in the following by a Prolog theory named `WorldTheory` associated with the actor (see Subsection 2.1.1 and Subsection 3.3)).

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

2.1.1 The actor's `WorldTheory`.

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

- the rule `actorPrintln/1` prints a given tuProlog `Term` (see Subsection 3.6.1) in the standard output of the actor;
- the fact `actorobj/1` memorizes a reference to the Java/Akka object that implements the actor (see Subsection 6.3).
- the rule `actorOp/1` puts in execution a Java method written by the application designer (see Subsection 6.4).
- the fact `actorOpDone/2` memorizes the result of the last `actorOp` executed (see Subsection 6.4)
- the fact `goalResult/1` memorizes the result of the last Prolog goal given to a `solve` operation (see Subsection 6.1)
- the fact `result/1` memorizes the result of the last plan action (see Subsection 2.1.5) performed by the actor

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

2.1.2 Logical actions.

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

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

2.1.3 Physical actions.

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

2.1.4 Application actions.

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

2.1.5 PlanActions.

A *PlanAction* is a logical or physical action defined by the system or by the application designer. A *PlanAction* can assume different logical forms according to different attributes that can be associated to it:

```
Actions attribute sets
 ACTION
[ GUARD ] , ACTION
[ GUARD ] , ACTION , DURATION
[ GUARD ] , ACTION , DURATION , ENDEVENT
[ GUARD ] , ACTION , DURATION , [EVENTLIST], [PLANLIST]
```

We will use the following terminology:

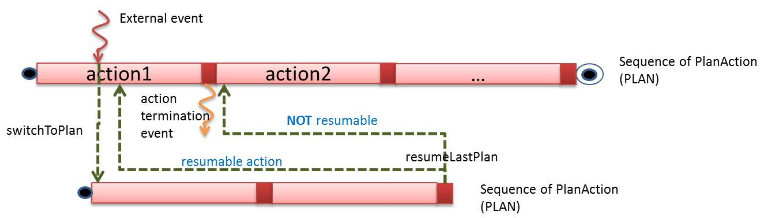
- an action that does not specify any DURATION is called ***basic action*** (see Subsection 5.1);
- an action that does specify a [GUARD] is called ***guarded action*** (see Subsection 5.6);
- an action that specifies a DURATION is called ***timed action*** (see Subsection 5.2);
- an action that specifies a ENDEVENT is called (timed) ***asynchronous action*** (see Subsection 5.4);
- an action that specifies [EVENTLIST],[PLANLIST] is called (timed) ***(synchronous) reactive action*** (see Subsection 5.5).

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.

2.2 Plans

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



2.3 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 said to be ***synchronous***.

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

Messages are represented as follows:

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

where :

MSGID	Message identifier
MSGTYPE	Message type (e.g.:dispatch,request,invitation,event,token)[1]
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 5.6) to allow conditional evaluation of *PlanActions*.

2.4 Events and event-driven/event-based behaviour

An ***event*** is defined here as information emitted by some source without any explicit destination. Events can be ***emitted*** by the *QActors* that compose the *actor-system* or by sources external to the system.

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

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 is 'lost' if no machine is in a state waiting for it.

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

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

2.5 Actor programs as plans

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

Each plan has a name and can be put into execution by a proper *PlanAction* (e.g. `switchToPlan`, see Subsection 5.1). Plans can be stored in files and dynamically loaded by the user into the *actor-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:

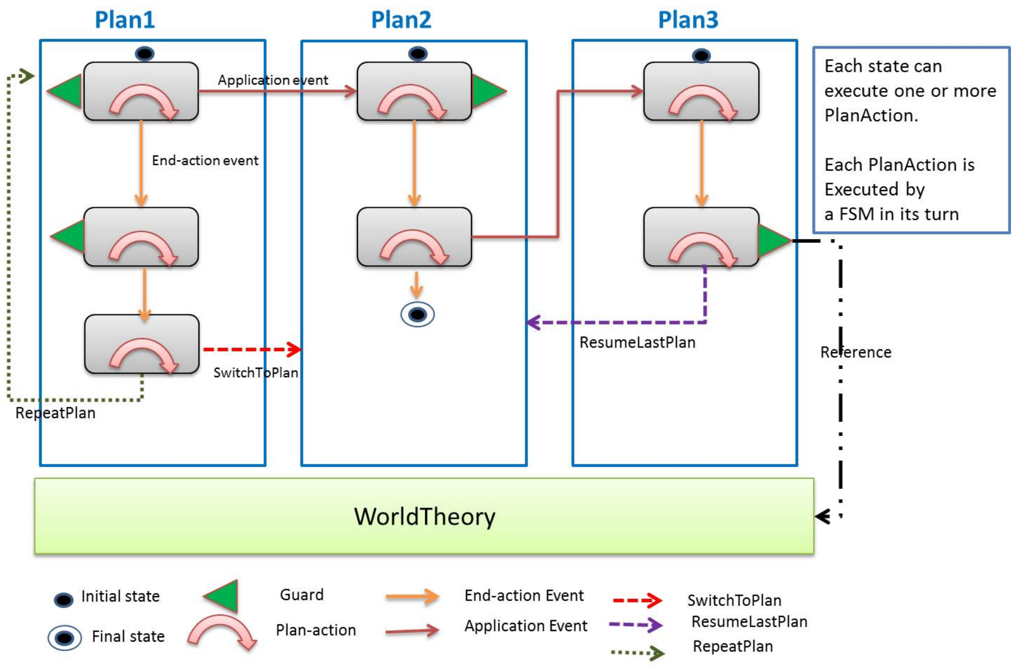
Internal representation of a plan

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

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

2.5.1 Actor programs as finite state machines (FSM)

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



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

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

- a (built-in) normal termination event;
- a user-defined termination event (see Subsection 5.4);
- a time-out termination event;
- a abnormal termination event;

3 The qa language/metamodel.

The *qactor* (qa) language is a custom language built by exploiting the XText technology[2]; thus, it is also a meta-model. Technically we can say that qa is a 'brother' of UML since it is based on EMOF.

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

3.1 Example

As an example of **qa** specifications, we define the behaviour of an actor 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.qa
 * A system composed of a qactor (player)
 * working in a context named 'ctxBasic' associated with a GUI
 */
System basic    //the testing flag avoids automatic termination
Context ctxBasic ip [ host="localhost" port=8079 ] -g cyan

QActor player context ctxBasic{
    Plan init normal
        println("Hello world" ) ;
        switchToPlan playMusic ;
        println("Bye bye" )
    Plan playMusic resumeLastPlan
        sound time(2000) file('./audio/tada2.wav') ;
        [ ?? tout(X,Y) ] println( tou(X,Y) ) ;
        repeatPlan 1

 }
```

Listing 1.5. basic.qa

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

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

3.2 Workflow

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

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

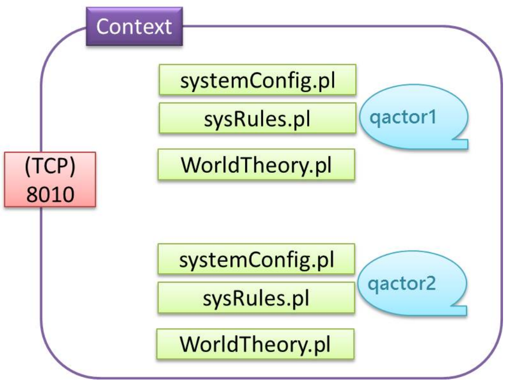
In several cases these specifications can be refined in the *project phase* by simply 'injecting' application-specific actions (see Section 6) so to reduce the global costs of software development.

3.2.1 Application designer and System designer.

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

3.3 QActor knowledge

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



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

3.4 QActor software factory

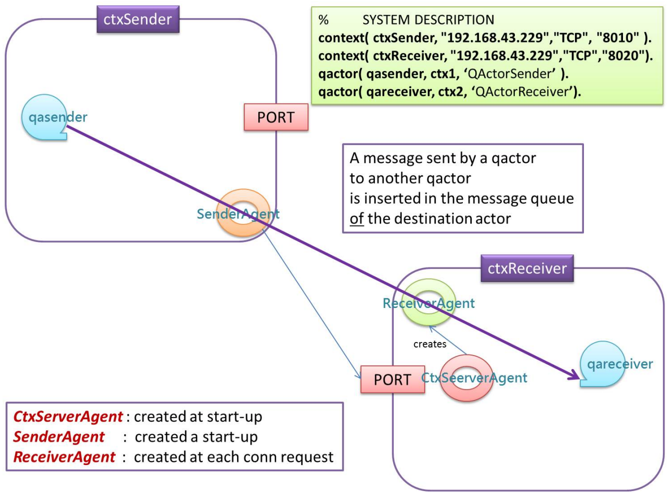
QActor systems run upon a run-time support (built in the project *it.unibo.qactors*) based on the Akka actor system and is deployed (at this moment) in the 'library' *qa18Akka.jar*[3]. The *QActor* run-time support requires in its turn other open-source and custom libraries.

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

3.5 Contexts

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

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



3.6 Messages

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

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

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

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

```

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

```
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 ;
```

3.6.1 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 ...
```

3.7 Send actions

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

3.7.1 Operation that sends a *dispatch* .

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

Example:

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

3.7.2 forward implementation (see Subsection 3.8) .

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

3.7.3 Operation that sends a *request* .

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

Example:

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

3.7.4 demand implementation (see Subsection 3.8) .

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

3.8 Send action implementation

Messages can be sent to a *Qactor* by using the built-in basic `sendMsg` asynchronous, point-to-point action. The `API` provided by the `qa` run-time support[4] has the following signature:

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

3.9 Receive actions

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

3.9.1 Generic receive with optional message 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))
```

3.9.2 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)
```

3.9.3 Select a message and execute .

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

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

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

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

3.10 Receive implementation

The 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:

```java
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 1.6. `IAsynchActionResult.java`

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*.

3.11 Events and event-based behaviour

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

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

3.11.1 Emit action .

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

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

Example

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

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

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

3.11.3 Sense action.

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

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

Example

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

3.11.4 sense implementation

. The 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{
```

3.11.5 OnEvent receive actions.

The qa language defines also an event-selection action:

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

Example

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

3.12 Event handlers and event-driven behaviour

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

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

```
EventHandler :
    "EventHandler" name=ID ( "for" events += [Event] ( "," events += [Event] )* )?
    ( print ?= "-print") ?
    ( "{" body = EventHandlerBody "}" )?
    ";"
EventHandlerBody:
     op += EventHandlerOperation (";" op += EventHandlerOperation)*
;
EventHandlerOperation:
    MemoOperation | SolveOperation | RaiseEvent | SendEventAsDispatch
;
MemoOperation:
    "memo" rule=MemoRule "for" actor=[QActor]
    | doMemo=MemoCurrentEvent "for" actor=[QActor]
;
SolveOperation:
    "solve" goal=PTerm "for" actor=[QActor]
;
SendEventAsDispatch :
    "forwardEvent" actor=[QActor] "-m" msgref=[Message]
;
MemoRule :
    MemoEvent // | Others memo rules
;
MemoEvent :
    name="currentEvent"
;
```

The syntax shows that, in a qa model, we can express only a limited set of actions within an EventHandler[5]:

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

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

```
System eventTracer -testing
Event usercmd : usercmd(X)
Event alarm  : alarm(X)
Context ctxEventTracer ip [ host="localhost" port=8027 ] -g cyan -httpserver
EventHandler evh for usercmd,alarm -print {
    memo currentEvent for qaevtracer
};
QActor qaevtracer context ctxEventTracer {
    Plan init normal
        println("qaevtracer starts");
        switchToPlan work
    Plan work
        [ ?? msg(E,'event',S,none,M,N) ] println(eventm(E,S,M)) else println( noevent );
        delay time(3000);
        repeatPlan
}
```

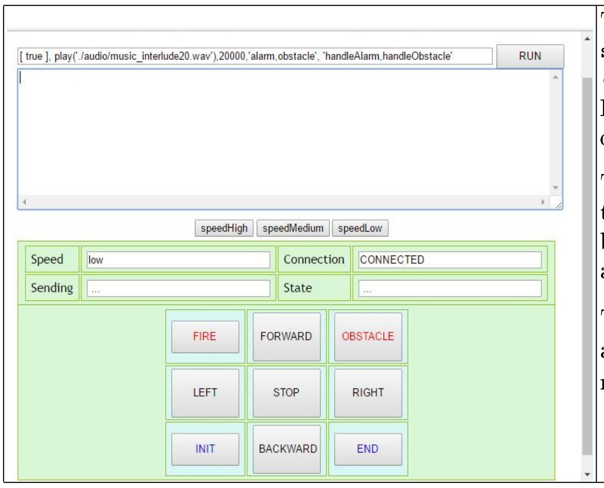
Listing 1.7. eventTracer.qa

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

4 Human interaction with a Qactor

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

4.1 The (remote) Web interface



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

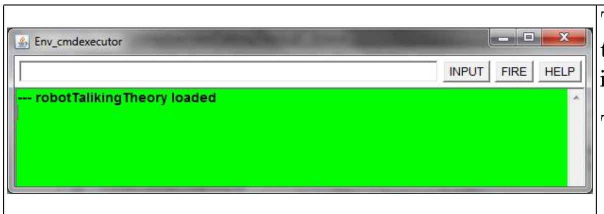
The `RUN` button at the top of the GUI allow us to ask the robot to execute actions, while the buttons at the bottom allows us to move the (basic) robot and send alarms.

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

This interface emits the following events:

`usercmd : usercmd(executeInput(CMD))`	(`RUN` button)
`usercmd : usercmd(robotgui(MOVE)), MOVE=w(low),...,s(high)`	(`MOVE` button)
`alarm : alarm(fire)`	(`FIRE` button)
`obstacle: obstacle(X)`	(`OBSTACLE` button)
`usercmd:usercmd(robotgui(w(low)))`	(`FORWARD` button)
`cmd : cmd(start)`	(`START` button)
`cmd : cmd(stop)`	(`STOP` button)

4.2 The (local) GUI user interface



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

The `INPUT` button generates the event :

`local_inputcmd : usercmd(executeInput(CMD))`

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

Let us report here some common functional actions (implemented by the *WorldTheory* associated with each actor) that we can ask a robot to do[6]. For a demo see the model `cmdExecutor.qa` in project *it.unibo.qactors*.

[6] 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))`.

4.3 Inspect the state and elaborate in a functional way

`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),...))`

4.4 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`, i.e. store the fact: `value(x,3)`
`assign(ledstate,false)`	set fact: `value(ledstate,false)`
`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`

5 About actions

5.1 Basic actions

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

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 )
play a sound                                     play( FILENAME )
emit an event                     raise( EVID,EVCONTENT )
user-defined action               ...
```

Here some example:

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,obstacle", "handleAlarm,handleObstacle"
```

5.2 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

```
fib(41,V), 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.

5.3 Time out

If a time-out expires, the fact

Timed actions

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

is asserted in the *WorldTheory* of the working *QActor*. This fact can be used to execute actions under the control of a guard (see Subsection 5.6); for example:

Timed actions

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

5.4 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.

Asynchronous actions cannot be reactive (see Subsection 5.5). This because the idea of reacting to an asynchronous actions must be further explored.

5.5 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.

5.6 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 qa 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 qa:

- 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*

6 User-defined actions in Prolog

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

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

6.0.1 Examples of unification

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

Action examples

```
a(X,1)=a(1,Y)       '='(a(1,1),a(1,1))
a(X,1)=a(1,X)       '='(a(1,1),a(1,1)))
a(X,2)=a(1,X)       a(_1,_2)                //tuProlog rewrites variables (occur check)

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

6.1 The `solve` operation.

The `qa` language defines actions to `solve` Prolog goals:

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

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

6.2 Loading and using a user-defined theory

The *WorldTheory* of an actor can be extended by the application designer by using the directive[7] `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.qactors
  */
System atheoryUsage -testing
Context ctxTheoryUsage ip [ host="localhost" port=8049 ] -g cyan

QActor theoryusage context ctxTheoryUsage{
    Plan init normal
/*0*/  demo consult("./aTheory.pl") onFailSwitchTo prologFailure ;
        switchToPlan compute ;
        switchToPlan accessdata ;
        println( bye )
    Plan compute resumeLastPlan
        solve fib(7,V) time(1000);
        [!? goalResult(G) ] println( G )  //prints fib(7,13)
    Plan accessdata resumeLastPlan
        println( "------------------------------------" ) ;
```

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

```
/*1*/  [ !? data(S,N,V) ] println( data(S,N,V) ) ;
/*2*/  [ !? validDistance(N,V) ] println( validDistance(N,V) ) ;
/*3*/  demo nearDistance(N,V) onFailSwitchTo prologFailure ;
/*4*/  [ !? goalResult(nearDistance(N,V)) ] println( warning(N,V) ) ;
/*5*/  demo nears(D) onFailSwitchTo prologFailure ;
/*6*/  [ !? goalResult(G) ] println( list(G) )  ;
/*
       demo likes(X,jim) onFailSwitchTo prologFailure ;
       [ !? goalResult(G) ] println( G )  ;
       demo character(ulysses, X, Y) onFailSwitchTo prologFailure ;
       [ !? goalResult(G) ] println( G )  ;
       demo f(g(a, Z), Y) onFailSwitchTo prologFailure ;
       [ !? goalResult(G) ] println( G )  ;
*/
       repeatPlan 1

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

Listing 1.8. aTheoryUsage.qa

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

```
/* ==========================================================
aTheory.pl in project it.unibo.qactors
============================================================== */
 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).

%% likes(jane,X).
%% character(ulysses, Z, king(ithaca, achaean)).
%% f(g(X, h(X, b)), Z).

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

Listing 1.9. aTheory.pl

6.2.1 The initialization directive.

The following directive:

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

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

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

```
--- initializing the aTheory ...
--- fib(7,13)
---  "--------------------------------------"
--- 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
```

6.2.2 On backtracking.

The output shows that the rules `validDistance and nearDistance` exploit *backtracking* in order to return the first valid instance (2), while the repetition of the plan *accessdata* returns always the same data[8]. In fact, *backtracking* is a peculiarity of Prolog and is not included in the computational model of *QActor*. However, an actor could access to different data at each plan iteration, by performing a proper query in which the second argument of `data/3` is used as an index (for an example, see Subsection 6.8.3).

6.3 Using the actor in Prolog rules

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

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

```
/*
===============================================================
actorDanceTheory.pl in project it.unibo.qactors
===============================================================
*/

dance :-
    actorobj( Actor ),
    actorPrintln(forward),
    Actor <- delay(2000),
    actorPrintln(left),
    Actor <- delay(2000),
    actorPrintln(right),
    Actor <- delay(2000),
    actorPrintln(backward),
    Actor <- delay(2000).

initdance :-   actorPrintln("initializing the ActorDanceTheory ...").
:- initialization(initdance).
```

Listing 1.10. `actorDanceTheory.pl`

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

```
/*
 * dancer.qa in project it.unibo.qactors
 */
System dancerSys -testing
Event endplay : endplay(X)
Event alarm  : alarm(X)

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

QActor dancer context ctxDancer{
    Plan init normal
        demo consult("./actorDanceTheory.pl") onFailSwitchTo prologFailure ;
        switchToPlan playMusic ;
        switchToPlan compute ;
        switchToPlan dance ;
        println("Bye bye" )
    Plan compute resumeLastPlan
        demo fib(7,V) ;
        [?? goalResult(X) ] println( X )  //prints fib(7,13)
    Plan dance resumeLastPlan
        solve dance time(12000) onFailSwitchTo prologFailure react event alarm -> handleAlarm ;
```

[8] Remember from Subsection 2.1.1 that the fact goalResult/1 is a 'singleton'.

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

Listing 1.11. dancer.qa

6.4 The operator actorOp

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

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

```
System actorOpdemo
Context ctxActorOpdemo ip [ host="localhost" port=8037 ] -g cyan
QActor qaactorop context ctxActorOpdemo {
    Plan main normal
        println("actionOpdemo STARTS " ) ;
        switchToPlan testReturnPrimitiveData ;
        switchToPlan testReturnPojo ;
        println("actionOpdemo ENDS " )
    Plan testReturnPrimitiveData resumeLastPlan
        actorOp intToVoid(5) ;
        [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
        actorOp intTostring(5) ;
        [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
        actorOp intToInt(5) ;
        [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
        actorOp floatToFloat(5) ;      //qa does not allow to write floats
        [ ?? actorOpDone( OP,R ) ] println( done(OP,R) )
    Plan testReturnPojo resumeLastPlan
        println("actionOpdemo testReturnPojo " ) ;
        actorOp getDate ;
        [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
        demo actionResultOp( toInstant ) ;
        [ ?? actorOpDone( OP,R ) ] println( done(OP,R) ) ;
        demo actionResultOp( getNano ) ;
        [ ?? actorOpDone( OP,R ) ] println( done(OP,R) )
}
```

Listing 1.12. actorOpdemo.qa

The code written by the application designer is:

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

import it.unibo.is.interfaces.IOutputEnvView;
import it.unibo.qactors.QActorContext;

public class Qaactorop extends AbstractQaactorop {
    public Qaactorop(String actorId, QActorContext 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;
    }

    public void intToVoid( int n ){
        println( "     Java intToVoid " + n );
    }

    public String intTostring( int n ) {
        println( "     Java intTostring " + n );
        return ""+n+1;
    }
    public float floatToFloat( float n ) {
        println( "     Java floatToFloat " + n );
        return n/2;
    }

}
```

Listing 1.13. `Qaactorop.java`

6.5 Rules at model level

Sometimes can be useful to express Prolog facts directly in the model specification, especially when these facts are used for configuration or action-selection purposes. The `Rules` option within a *QActor* allows us to define facts by using a subset of the Prolog syntax[9]

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.qactors
 */
System rulesInModel -testing
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) ;
        [ !? longSound(1,F,T)] sound time(T) file(F) react event alarm -> handleAlarm ;
        [ !? fastSound(3,F,T)] sound time(T) file(F) ;
        println("Bye bye" )
```

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

```
    Plan handleAlarm resumeLastPlan
        println("rulebasedactor hhandleAlarm" )
    Plan prologFailure
        println("rulebasedactor has failed to solve a Prolog goal" )
}
```

Listing 1.14. `rulesInModel.qa`

6.6 From Prolog to Java again

Thanks to tuProlog features, the application designer can define rules that can exploit Java to perform the required operation. For example, suppose that we have to solve 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.qactors
 */
System userSelect -testing
Context ctxUserSelect ip [ host="localhost" port=8079 ] -g cyan

QActor soundselector context ctxUserSelect{
    Plan init normal
        demo consult("./userTheory.pl") 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)

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

Listing 1.15. `userSelect.qa`

6.6.1 Guards as problem-solving operation.

In the plan `playMusic` above, let us consider the sentence:

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

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

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

To allow the `select/1` guard to become part of the problem-solution rather than just a test[10] , 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)`).

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

```
testConsult :- actorPrintln("userTheory works").

mortal(X) :- human(X).
human(socrate).
human(platone).

/*
Expressions
*/
testOnExp(Z) :- (2 + 3 * 4 / 1) = Z.

/*
A.2.1
*/
character(priam, iliad).
character(hecuba, iliad).
character(achilles, iliad).
character(agamemnon, iliad).
character(patroclus, iliad).
character(hector, iliad).
character(andromache, iliad).
character(rhesus, iliad).
%% character(ulysses, iliad).
%% character(menelaus, iliad).
%% character(helen, iliad).

character(ulysses, odyssey).
character(penelope, odyssey).
character(telemachus, odyssey).
character(laertes, odyssey).
character(nestor, odyssey).
character(menelaus, odyssey).
character(helen, odyssey).
character(hermione, odyssey).

male(priam).
male(achilles).
male(agamemnon).
male(patroclus).
male(hector).

male(rhesus).
male(ulysses).
male(menelaus).
male(telemachus).
male(laertes).
male(nestor).

female(hecuba).
female(andromache).
female(helen).
female(penelope).


father(priam, hector).
father(laertes,ulysses).
father(atreus,menelaus).
father(menelaus, hermione).
father(ulysses, telemachus).

mother(hecuba, hector).
mother(penelope,telemachus).
mother(helen, hermione).

king(ulysses, ithaca, achaean).
king(menelaus, sparta, achaean).
king(nestor, pylos, achaean).
king(agamemnon, argos, achaean).
king(priam, troy, trojan).
king(rhesus, thrace, trojan).
```

```

character(priam, iliad, king(troy, trojan)).
character(ulysses, iliad, king(ithaca, achaean)).
character(menelaus, iliad, king(sparta, achaean)).



/*
-----------------------------------------
A.2.5
-----------------------------------------
*/
% Is the king of LAND also a father? i
kingFather(X,LAND) :- king(X, LAND, Y), father(X, _).

/*
-----------------------------------------
A.2.7
-----------------------------------------
*/
son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).

parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

/*
-----------------------------------------
A.2.7
-----------------------------------------
*/
unify(T1,T2,T1) :-
    actorPrintln(unify(T1,T2)),
    T1 = T2.

unify1(X,Y,Z) :- character(ulysses, Z, king(ithaca, achaean)) = character(ulysses, X, Y).

/*
-----------------------------------------
A.5,3
-----------------------------------------
*/
p(X) :- q(X), r(X).
q(a).
q(b).
r(b).
r(c).

resolve(X) :- X,
    actorPrintln(X),
    fail.
resolve(X).

/*
-----------------------------------------
A.13.1
-----------------------------------------
*/
hero(ulysses).
heroin(penelope).
daughter(X, Y) :-
    mother(Y, X),
    female(X).
daughter(X, Y) :-
    father(Y, X),
```

```
    female(X).

/*
----------------------------------------
A.15.2
----------------------------------------
*/

link(r1, r2).
link(r1, r3).
link(r1, r4).
link(r1, r5).
link(r2, r6).
link(r2, r7).
link(r3, r6).
link(r3, r7).
link(r4, r7).
link(r4, r8).
link(r6, r9).

s(X, Y) :- link(X, Y).
s(X, Y) :- link(Y, X).

minotaur(r8).

labyrinth(X) :- minotaur(X).

%% depth_first_search(+Node, -Path)
depth_first_search(Node, Path) :-
    depth_first_search(Node, [], Path).
%% depth_first_search(+Node, +CurrentPath,-FinalPath)
depth_first_search(Node, Path, [Node | Path]) :-
    labyrinth(Node).
depth_first_search(Node, Path, FinalPath) :-
    s(Node, Node1),
    not member(Node1, Path),
    depth_first_search(Node1, [Node | Path],FinalPath).


/*
----------------------------------------
TuProlog
----------------------------------------
*/
selectFile( 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.qactors.QActorUtils") <- adjust( Path ) returns FileName.

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

Listing 1.16. `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 `"/"`.

```
    /*
     * LEAVE HERE (line 63) since refenced by Latex
     */
```

Listing 1.17. `QActorUtils.java`

6.7 Workflow

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

The workflow of Subsection 3.2 can be extended with the following steps:

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

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

6.8 Examples of problem solving with tuProlog

Let us consider the following model:

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

QActor thexample context ctxTheoryExanple{
    Plan init normal
        demo consult("./exampleTheory.pl") onFailSwitchTo prologFailure ;
        demo consult("./srcMore/it/unibo/ctxTheoryExanple/theoryexanple.pl") onFailSwitchTo prologFailure ;
        switchToPlan configuration;
        switchToPlan family;
/*(1)*/ solve assign(n,1) time(0) onFailSwitchTo prologFailure ;
        switchToPlan accessData ;
        println( bye )
    Plan configuration resumeLastPlan
        demo showSystemConfiguration onFailSwitchTo prologFailure
    Plan family resumeLastPlan
        demo son(X,haran) onFailSwitchTo prologFailure ;
        [ !? goalResult(son(X,haran)) ] println(X) else println(noson(haran));
        demo daughters(X,haran) 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
    Plan prologFailure
        println("thexample has failed to solve a Prolog goal" )
}
```

Listing 1.18. `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 1.19. `exampleTheory.pl`

We note that:

6.8.1 configuration.

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

```
%==============================================================================
% Context ctxTheoryExanple SYSTEM-configuration: file it.unibo.ctxTheoryExanple.theoryExanple.pl
%==============================================================================
context(ctxtheoryexanple, "localhost", "TCP", "8099" ).
%%% -------------------------------------------
qactor( thexample , ctxtheoryexanple, "it.unibo.thexample.MsgHandle_Thexample" ). %%store msgs
qactor( thexample_ctrl , ctxtheoryexanple, "it.unibo.thexample.Thexample" ). %%control-driven
%%% -------------------------------------------
%%% -------------------------------------------
```

Listing 1.20. `theoryexanple.pl`

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

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

6.8.2 family.

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

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

6.8.4 output. The output of the system execution is:

```
*** 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
```