



3APL-M

Platform for Lightweight Deliberative Agents
Version: 1.4 (Feb-2005)

Fernando Koch[†]
<http://www.cs.uu.nl/3apl-m>

Table of Contents

Table of Contents	1
What-is 3APL-M?	2
What-is 3APL?	2
Software Requirements	3
Getting started	3
For J2SE	3
For J2ME	4
The 3APL-M architecture	6
Programming in 3APL	7
The Deliberation Cycle	9
Simplified Deliberation Cycle: class Agent	10
Complete Deliberation: class DeliberativeAgent	11
Loading 3APL code for execution	13
Loading Knowledge Bases through Code	13
Load Code from InputStream	14
Load Prolog instructions	15
Creating Sensors	16
Creating Actuators	19
Built-in Capabilities	21
Networking	22
Interfacing (GUI)	23
Memory release control	25
Limitations and known issues	25
In 3APL language	25
Acknowledgements	26
References	26
Appendix I: Complete Code Examples	27
Example of Sensor and Actuator based application	27
Example of Deliberative Agent and BlockWorld	30

[†] Fernando Koch

Intelligent Systems Group

Institute of Information and Computing Sciences

Utrecht University, The Netherlands

What-is 3APL-M?

It is a platform for building applications using Artificial Autonomous Agents Programming Language (3APL) as the enabling logic for the deliberation cycles and internal knowledge representation. It is small enough to be used in Mobile Computing applications. The binary compilations are distributed for Java 2 Micro Edition and Java 2 Standard Edition.



Figure 1 - Examples of 3APL-M applications

What-is 3APL?

3APL is a programming language for implementing cognitive agents. It provides programming constructs for implementing agents' beliefs, goals, basic capabilities (such as belief updates, external actions, or communication actions) and a set of practical reasoning rules through which agents' goals can be updated or revised. The 3APL program is executed by means of an interpreter that deliberates based on the cognitive attitudes of that agent. The Figure 2 presents the conceptual model for a Belief, Desire and Intentions interpreter.

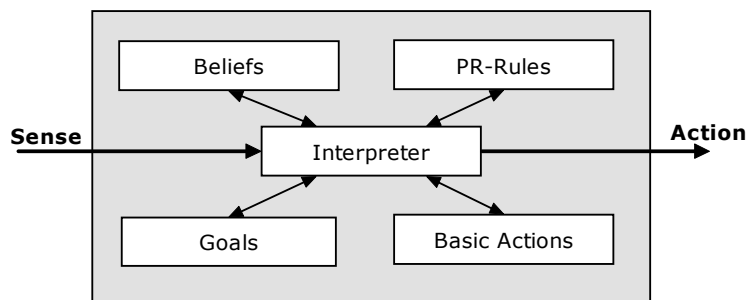


Figure 2 - Conceptual Model for B.D.I interpreters

Software Requirements

3APL-M works in Java 2 Micro Edition and Java 2 Standard Edition environments. It is suitable to build J2ME applications to be deployed in small device. However, it is extensible enough to be used for integrating 3APL agents into desktop- and server-based applications.

The code has been tested using the following environments:

```
java version "1.1.8"

java version "1.4.1_02"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1_02-b06)
Java HotSpot(TM) Client VM (build 1.4.1_02-b06, mixed mode)

J2ME Wireless Toolkit 2.1
Profile: MIDP-2.0
Configuration: CLDC-1.1
Optional: WMA-1.1,MMAPI-1.1,JSR172,JSR185 (JTWI)
```

Getting started

3APL-M works as a library integrated into a Java application and providing an Application Programming Interface (API) to the 3APL-M machine. It supplies a two-way interface between the Java code and the 3APL machinery, allowing the developer to explore the best from the two worlds. The concept is *do the labour work in Java and let 3APL take care of the intelligence (deliberation)*.

The steps to build a “Hello World” application are described below. Notice that the compilation would be largely simplified if the developer uses an integrated development environment (IDE) like Borland JBuilder and MobileSet (<http://www.borland.com>).

For J2SE

1. Download the appropriate library from the web site (<http://www.cs.uu.nl/3apl-m>)
2. Create the Java importing the mTripleApl.* libraries as required and make the calls to the 3APL-M elements. For example:

```
import mTripleApl.Agent;

public class HelloWorldin3APLM {
    public static void main(String[] args) {
        Agent ag = new Agent("myfirstagent"); // create agent
        ag.setTrace(3); // enable tracing
    }
}
```

```

// load knowledge
ag.addCapability("{} Print(X) {write(X)}");
ag.addPlanRule(" <- TRUE | Print('hello '), Print('world'),Print('\n')");
ag.addGoal("print");

// deliberate
ag.deliberate();
}
}

```

3. Compile the code, linking the JAR library taplm-j2se.jar (3APL-M library downloaded from step (1)), such as:

```
$ javac -classpath taplm-j2se.jar HelloWorldin3APLM.java
```

4. Run the application, including the JAR library taplm-j2se.jar in the CLASSPATH:

```
$ java -cp ".:taplm-j2se.jar" HelloWorldin3APLM
```

For J2ME

1. Download the appropriate library from the web site (<http://www.cs.uu.nl/3apl-m>)
2. Create a Java MIDP code, importing the MIDP/CDCL and 3APL-M libraries as required. For example:

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import mTripleApl.*;
import java.util.*;

public class HelloWorldJ2ME
    extends MIDlet
    implements CommandListener {

    Form form;

    public void startApp() {

        // create micro from and display
        this.form = new Form("First mTApI application");
        this.form.setCommandListener(this);
        this.form.addCommand(new Command("Exit", Command.EXIT, 1));
        TextField textField = new TextField("", "", 30, TextField.ANY);
        this.form.append(textField);
        Display.getDisplay(this).setCurrent(this.form);

        // create tApl agent
        Agent ag = new Agent("micro");
        ag.setTrace(3);
    }
}

```

```

// load knowledge
ag.addActuator("Print(X)", new TextFieldActuator(textField));
ag.addPlanRule("print <- TRUE | Print('hello world')");
ag.addGoal("print");

// deliberate
ag.deliberate();
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
    notifyDestroyed();
}

public void commandAction(Command command, Displayable displayable) {
    if (command.getCommandType() == Command.EXIT) {
        destroyApp(true);
    }
}
}

/**
 * ACTUATOR TO PRINT INTO A FIELD
 */
class TextFieldActuator
    implements ActuatorInterface {
    TextField field;

    // create: associate to field
    public TextFieldActuator(TextField field) {
        this.field = field;
    }

    // called from Deliberation engine
    public boolean actuator(String[] data) {
        if (data.length > 0) {
            this.field.setString(data[0]);
        }
        return true;
    }
}

```

3. As per the J2ME specifications, the code needs to be compiled, preverified and a MIDLet MANIFEST file (JAD file) must be assembled. For details on how to build and deploy J2ME applications, check the documentation at <http://java.sun.com/j2me/index.jsp>.

- Run the application using a Phone Emulator or download the compiled JAD/JAR files to a J2ME enabled device for execution. For example, using the J2ME ToolKit Phone Emulator, from Sun Corp:

```
$ emulator.exe -classpath "mTap1\j2me\classes;" -Xdevice:DefaultColorPhone
-Xdescriptor:" mTap1\j2me\jad-temp\HelloWorldin3APLM.jad"
```

The Figure 3 presents the screenshot for this execution.



Figure 3 – Screenshot: HelloWorld application in 3APL-M and J2ME

The 3APL-M architecture

The 3APL-M Architecture, presented in Figure 4 and explained below.

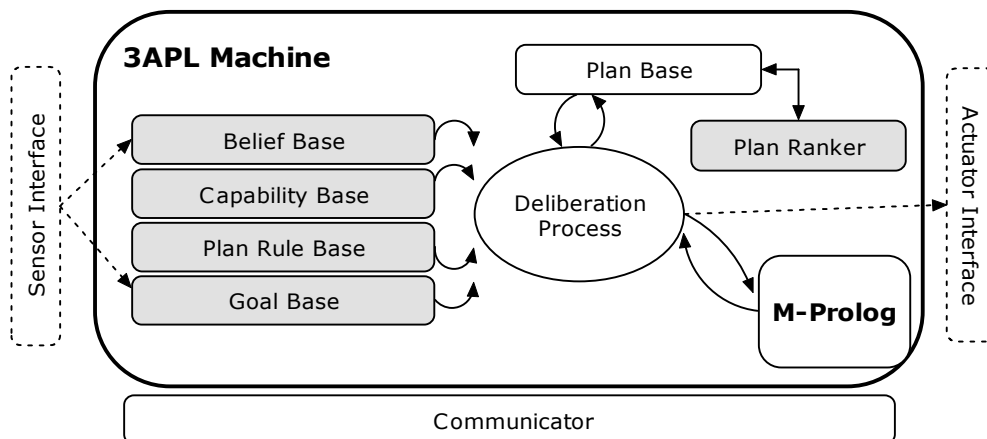


Figure 4 - 3APL-M application architecture

The components are:

- **3APL Machinery** : encapsulates the Agent's components and provides the Application Programming Interface to integrate it to Java applications. For documentation on the API, check the JavaDocs at the 3APL-M WebSite.
- **Belief, Capabilities, Goal and Plan Rules:** are the 3APL structures, as described by Dastani et al. 2003 and Hindricks et al. 1999.
- **Deliberation process** : implement the logic transition rules that create the Plans based on the Goal, Beliefs and PlanRules, as described in Dastani et al. 2003
- **Plan Base** : the list of current Plans generated by the Deliberation Process. This list is available during the processing phase and can be manipulated by an external logic (supplied by the developer). This allows the creation of external Plan Analysers, that can be use to improve the deliberation cycle (for example, classify plans per utility).
- **Plan Ranker:** Internal Class, part of the Planner Sub-system, which classifies the plans in the Plan Base by calculating its UTILITIES. This component drops the Plans that have negative UTILITY from the Plan Base.

$$\text{UTILITY OF PLAN} = \sum \{ \text{WORTH GOALS} \} - \sum \{ \text{COST ACTIONS} \}$$
- **mProlog:** a reduced Prolog engine, optimized for J2ME applications. It is a sub-product of this project and packaged along the 3APL-M application library. The mProlog deliberation cycle is based on the W-Prolog project, from Michael Winikoff, which was heavily re-engineered to accommodate the method requirements for the 3APL-M implementation.
- **Sensor and Actuator interface:** are Java Interfaces that allow the integration of the 3APL-M machinery to the external world. Sensors and Actuators are developed in Java and attached to the 3APL-M machinery through special methods in the API. The idea is: do in Java what is easier to do in Java, and leave the logic part to the 3APL-M Machinery. Information on how to create and attach Sensors and Actuators are available in the How-to pages.
- **Communicator** : is the generic interface for the Communication (networking) infrastructure. The packets exchanged between agents in the same package (local multi-agent system) are handled directly by this infrastructure, whilst packets directed to external agents (out bounding) are queued in a stack and should be handled by a plugged in logic (which is dependent on the device where the application is running, thus not supplied in the distribution package).

Programming in 3APL

To understand the 3APL language, syntax and concepts, check the reference papers:

A 3APL application is composed by four knowledge bases, which are executed by the interpreter during run-time.

The 3APL structures are described as follows:

- **CAPABILITIES:** contains a description of the basic actions an Agent can implement. One capability is represented by the syntax:

{ <pre-condition> } <identifier> { <pos-condition> }

where:

<pre-condition> ::= PrologFact AND PrologFact [OR PrologFact]

<identifier> ::= PrologFactWithUpperCaseStart

<pos-condition> ::= PrologFact, PrologFact, ...

The execution for a Capability Cap(X,Y) is:

- select those Capabilities whose identifier UNIFY Cap(X,Y).
 - from there, select those Capabilities whose <pre-condition> UNIFY to BeliefBase
 - from there, execute each ELEMENT of the <pos-condition> as:
 - if the ELEMENT existis in Prolog (built-in), execute as a Prolog Term
 - if the ELEMENT name starts with upper case letter, executes as a Capability otherwise, ASSERT or RETRACT the fact from the BeliefBase
- **BELIEFBASE:** beliefs describe the situation the agent is in. The belief base can contain information the agent believes about the world and it can contain information that is internal to the agent. The beliefs are represented by a first-order domain language (PrologFacts).
 - **RULEBASE:** The Rule Base contains the PlanRules, which are templates for building the PlanBase (list of expanded Plans) during execution time. Plan is a sequence built from basic elements. The basic elements can be basic actions, tests on the belief base or abstract plans. The effect of execution of a basic action is not a change in the world, but a change in the belief base of the agent. An abstract plan cannot be executed directly in the sense that it updates the belief base of an agent. The PlanRules are represented as:

<Goal> :- <Guard> | <Sequence of Actions>

where:

<Goal > ::= PrologFact

<Guard> ::= PrologFact AND PrologFact [OR PrologFact]

<pos-condition> ::= PrologFact, PrologFact, ...

The execution for a GOAL follows the Deliberationcycle description presented in the Section “How the Deliberation Cycle works?”

- **GOALBASE:** the goals of the agent denote the situation the agent wants to realize. The goals are represented by a first order domain language (Prolog).

A simple example of 3APL code is detailed below.

```
PROGRAM "test3"
```

```
CAPABILITIES:
```

```
{ pos(X) } West() { NOT pos(X), pos(X-1) }
```

```
{ pos(X) } East() { NOT pos(X), pos(X+1) }
```

```
{ } Print() { write('Yeap!') }
```

```
RULEBASE:
```

```
goBase <- pos(X) AND base(X) | SKIP.
```

```
goBase <- pos(X) AND base(Z) AND X>Z | West(), Print(), goBase().
```

```
goBase <- pos(X) AND base(Z) AND X<Z | East(), Print(), goBase().
```

```

BELIEFBASE:
pos(56).
base(0).

GOALBASE:
goBase().

```

- **CAPABILITES “{ pos(X) } West() { NOT pos(X), pos(X-1) }”**: for the capability West() executed with the environmental pre-condition “pos(X)” (assuming X=56 if the BeliefBase contains pos(56)), execute “NOT pos(X->56)”, meaning negate pos(56) from the BeliefBase, and then inferring “pos(X->56-1)”, meaning pos(55) will be asserted in the BeliefBase
- **CAPABILITES “{ } Print() { write('Yeap!') }”**: for the BasicAction Print() will execute the PROLOG action “write('Yeap!')”, which is pre-build in mProlog and causes the print of the String to the standard output
- **RULEBASE “goBase <- pos(X) AND base(X) | SKIP.”**: is a PlanRule, meaning that for the environment where pos(X) is equals base(X) (robot has arraived to the base), it will SKIP the execution.
- **RULEBASE “goBase <- pos(X) AND base(Z) AND X>Z | West(), Print(), goBase()”**: is a PlanRule, meaning that for the environment where pos(X) is after base(Z) (X greater than Z), the Robot should go West. It will generate a Plan in the PlanBase like: “goBase <- TRUE | West(), Print(), goBase()”. If this plan is selected to execution, the three Actions are included in the GOALBASE and executed in sequence
- **BELIEFBASE** contains the environmental settings, in form of Prolog facts
- **GOALBASE** is the list of statements to execute

The Deliberation Cycle

There are two types of Agents provided in the 3APL-M package: Agents with *Simplified Deliberation Cycle* (from Hindriks et al 1999) and Deliberative Agents that implement the *Complete Deliberation Cycle* (from Dastani et al 2003).


```

goBase <- pos(X,Y) AND base(A,B) AND Y<B | North() , goBase().

BELIEFBASE:
pos(10,10)
base(19,19)
base(0,0)

GOALBASE:
goBase()

COSTBASE:
South()=5.
North()=12.
East()=5.
West()=10.

WORTHBASE:
goBase()=10.

```

Following the complete deliberation cycle (Figure 6), the execution flow is:

- **Select GOAL**

goal-base contains: [goBase()]

- **Find PlanRules that match GOAL and BELIEFS**

planrule-find-selected: (1) goBase() <- pos(X, Y) AND base(X, Y) | SKIP()

planrule-find-selected: (2) goBase() <- pos(X, Y) AND base(A, B) AND X > A | West(), goBase()

planrule-find-selected: (3) goBase() <- pos(X, Y) AND base(A, B) AND X < A | East(), goBase()

planrule-find-selected: (4) goBase() <- pos(X, Y) AND base(A, B) AND Y > B | South(), goBase()

planrule-find-selected: (5) goBase() <- pos(X, Y) AND base(A, B) AND Y < B | North(), goBase()

- **Create PLANBASE**

planbase-add: goBase() <- TRUE | West(), goBase()

planbase-add: goBase() <- TRUE | East(), goBase()

planbase-add: goBase() <- TRUE | South(), goBase()

planbase-add: goBase() <- TRUE | North(), goBase()

- **PlanRanker Classifies PLANBASE**

planranker-dropping goBase() <- TRUE | North(), goBase(): utility=-2

plan-classified: (1)= goBase() <- TRUE | East(), goBase(): utility=5

plan-classified: (2)= goBase() <- TRUE | South(), goBase(): utility=5

plan-classified: (3)= goBase() <- TRUE | West(), goBase(): utility=0

- **Select BEST PLAN**

planbase-result: goBase() <- TRUE | East(), goBase()

- **Execute PLAN**

executing plan: goBase() <- TRUE | East(), goBase()

goal-add-in-front: goBase()

goal-add-in-front: East()

When executing a CAPABILITY:

- **Select GOAL**

goal-base contains: [East(), goBase()]

- **Goal is CAPABILITY: Unify pre-condition and ...**

execute-capability: {pos(X, Y)} East() { NOT pos(X, Y), pos(X + 1, Y)}

- **Execute pos-condition**

execute-capability-poscondition: NOT pos(X, Y), pos(X + 1, Y) with X=16,Y=10

belief-delete: pos(16, 10)

belief-add: pos(17, 10)

Loading 3APL code for execution

There are two ways to load 3APL code into the Java application: wire the code in the Java application and load the code from an InputStream.

Loading Knowledge Bases through Code

By wire-in the code in Strings that are compiled in execution time. For that, the developer can use the Agent's methods from the Java class **Agent**.

public final void addBelief(java.lang.String beliefStr)

Add a belief. The String will be compiled by the TAplCompiler.

Parameters: belief String to be added

Example:

```
ag.addBelief("pos(10,10)");
```

public final void addCapability(java.lang.String capabilityStr)

Add a capability. The String will be compiled by the TAplCompiler.

Parameters: capability String to be added

Example:

```
ag.addCapability("{pos(X,Y)} West() { NOT pos(X,Y), pos(X-1,Y),  
BlockMove(west)}");
```

public final void addGoal(java.lang.String goalStr)

Add a goal. The String will be compiled by the TApICompiler.

Parameters: goal String to be added

Example:

```
ag.addGoal("goBase()");
```

public final void addPlanRule(java.lang.String planRuleStr)

Add a plan rule. The String will be compiled by the TApICompiler.

Parameters: planRule String to be added

Example:

```
ag.addPlanRule("goBase <- pos(X,Y) AND base(A,B) AND X>A | West(), goBase().");
```

The complete code example is:

```
// create agent
Agent ag = new Agent("micro");

// add knowledge
ag.addCapability("{pos(X,Y)} West() { NOT pos(X,Y), pos(X-1,Y), BlockMove(west)}");
ag.addCapability("{pos(X,Y)} East() { NOT pos(X,Y), pos(X+1,Y), BlockMove(east)}");
ag.addCapability("{pos(X,Y)} North() { NOT pos(X,Y), pos(X,Y+1), BlockMove(north)}");
ag.addCapability("{pos(X,Y)} South() { NOT pos(X,Y), pos(X,Y-1), BlockMove(south)}");
ag.addPlanRule("goBase <- pos(X,Y) AND base(X,Y) | SKIP.");
ag.addPlanRule("goBase <- pos(X,Y) AND base(A,B) AND X>A | West(), goBase().");
ag.addPlanRule("goBase <- pos(X,Y) AND base(A,B) AND X<A | East(), goBase().");
ag.addPlanRule("goBase <- pos(X,Y) AND base(A,B) AND Y>B | South(), goBase().");
ag.addPlanRule("goBase <- pos(X,Y) AND base(A,B) AND Y<B | North(), goBase().");
ag.addBelief("pos(10,10)");
ag.addBelief("base(19,19)");
ag.addBelief("base(0,0)");
ag.addGoal("goBase()");

// deliberate
ag.deliberate();
```

Load Code from InputStream

The alternative is to load the code from an open InputStream, such as a File, HTTP connection or Local Resource. The developer can use the **Agent.consult(InputStream)** method.

For example:

```
public static void main(String[] args) throws Exception{
```

```

Agent ag = new Agent("second");
ag.setTrace(3);

// add knowledge
FileInputStream file = new FileInputStream("example3.txt");
ag.consult(file);

// deliberate
ag.deliberate();
}

```

And the file example3.txt contains:

```

PROGRAM "test3"

CAPABILITIES:
{ pos(X) } West() { NOT pos(X), pos(X-1) }
{ pos(X) } East() { NOT pos(X), pos(X+1) }
{ } Print() { write('Yeap!') }

RULEBASE:
goBase <- pos(X) AND base(X) | SKIP.
goBase <- pos(X) AND base(Z) AND X>Z | West(), Print(), goBase().
goBase <- pos(X) AND base(Z) AND X<Z | East(), Print(), goBase().

BELIEFBASE:
pos(56).
base(0).

GOALBASE:
goBase().

```

Load Prolog instructions

The version 1.3 introduced the methods to load Prolog instructions directly to the Prolog engine. These instructions are provided as a replacement for the LOAD statement defined in 3APL. As 3APL-M is designed to mobile computing devices and these devices do not have file systems (in general), it would not be possible to create the LOAD statement directly in the language.

```

/**
 * Add Prolog knowledge
 * @param prologStr to add.
 */
public final void addProlog(String prologStr);

/**
 * Load PROLOG application from a Stream.
 * @param stream to consult
 */

```

```
public final void consultProlog(InputStream stream);
```

For example:

```
public class TestCode008 {
    public static void main(String[] args) {

        Agent ag = new Agent("iyad");
        ag.setTrace(5);
        ag.addProlog(
            "print(X) :- write('using print->'), write(X), write('\n').");
        ag.addCapability("{} Print(X) {print(X)}");
        ag.addGoal("Print('Hello world')");

        // deliberate
        ag.deliberate();
    }
}
```

Another example:

```
public static void main(String[] args) throws Exception {

    Agent ag = new Agent("iyad");
    ag.setTrace(5);

    ag.consultProlog(new FileInputStream("test.pl"));
    ag.addCapability("{} Print(X) {print(X)}");
    ag.addGoal("Print('Hello world')");

    // deliberate
    ag.deliberate();
}
```

Creating Sensors

The Figure 7 position the Sensor System within the 3APL-M machine architecture:

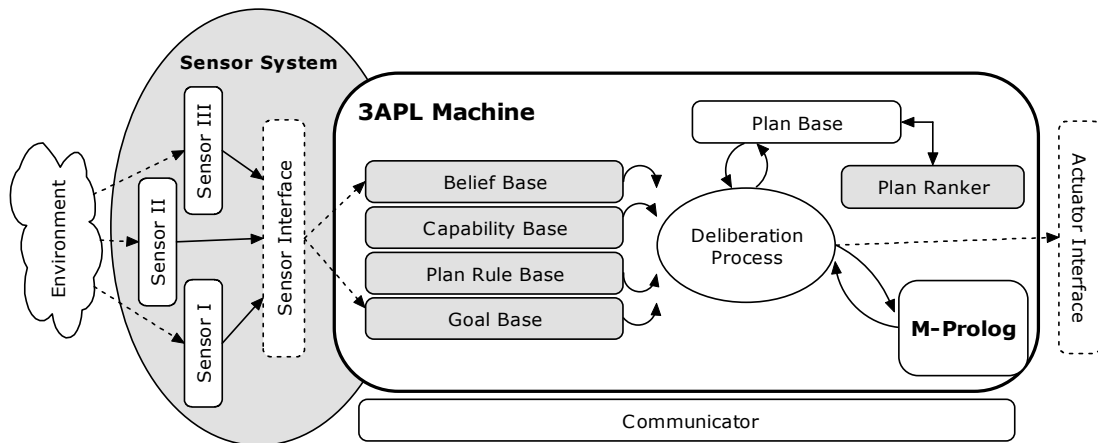


Figure 7 - Sensor subsystem

The 3APL code can be integrated to the environment through Sensors and Actuators. Sensors are Java implementations that collect environmental information and assert this data into the Agent's BELIEFBASE. Optionally, each time a new data is collected the Sensor can add GOAL to process this data into the GOALBASE.

The BELIEF added/replaced in the BELIEFBASE will be:

```
sensor({SensorId}, {CollectedData}).
```

The GOAL added to GOALBASE (if configured) will be:

```
sensor({SensorId}, {CollectedData}).
```

To implement and bind a Sensor:

1. Create a Class that **extends** the **Sensor** abstract class and implements the code for the abstract methods:

```
/**
 * ABSTRACT: Delivered by the implementation.
 * Called each execution to collect current data.
 */
public abstract Object collectData();
```

This method should implement the functionality to collect the data and return in form of a Object (or a Object[]).

2. Bind this class to the Agent using the Agent.addSensor(.) method:

```
/**
 * Add Sensor.
```

```

*
* @param id sensor identification
* @param sensor to add
* @param interval to probe for data
* @param addGoalNotification if true, new
*/
public final void addSensor(String id, Sensor sensor, int interval,
                           boolean addGoalNotification) {

```

For example:

```

/**
 * SENSOR FOR CLOCK
 */
class ClockSensor
    extends Sensor {

    // called during registration
    public void register(String agentName) {}

    // called during reset
    public void reset() {}

    // called from Deliberation engine
    public Object collectData() {
        return new Date(System.currentTimeMillis());
    }
}

```

And bind to an Agent like:

```
ag.addSensor("clock", new ClockSensor(), 2, true);
```

This means that the “clock” instance of ClockSensor will be probed each 2 seconds and in case new data is found it will add/replaces:

```

TO BELIEFBASE: sensor(clock, 'Tue 20 Aug 2004 11:59:32pm').
TO GOALBASE: sensor(clock, 'Tue 20 Aug 2004 11:59:32pm').

```

Alternatively, the Sensor can be added as no automatically collection and the data grabbing is then implemented using the Sensor({sensorId}, Result) CAPABILITY. For example:

```

// attach sensors
ag.addSensor("clock", new ClockSensor(), 0, false);

```

And in the 3APL code (Capability) do:

```
ag.addCapability("{} Print(X) {write([X,'\n'])}");
```

```

ag.addPlanRule("doIt <- TRUE | Sensor(clock, Result), Print(Result)");
ag.addGoal("doIt");

```

Creating Actuators

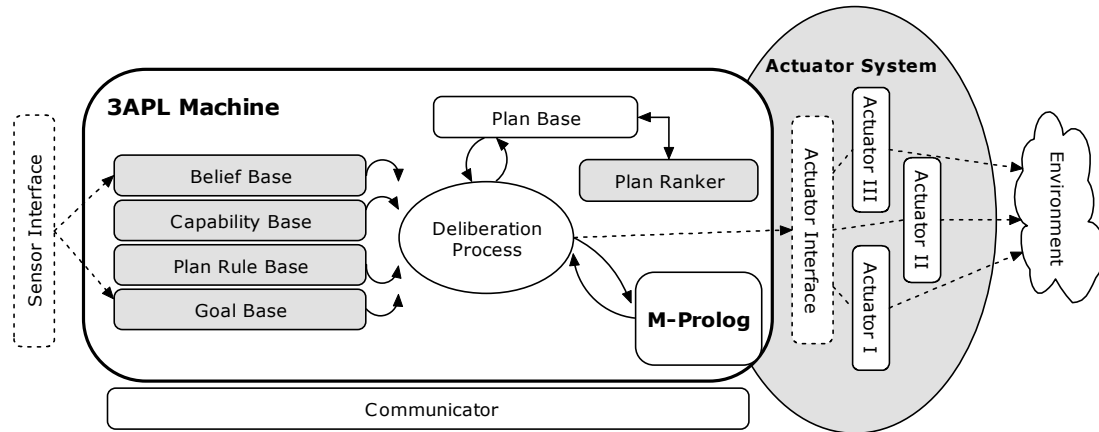


Figure 8 - Actuator sub-system

The 3APL code can be integrated to the environment through Sensors and Actuators. Actuators are Java implementations that allow the developer to implement ACTIONS from the 3APL-M Machinery to the external world using Java code. For example, this can be used to control robot movements, graphical interfaces (like the BlockWorld example) or text fields in Forms.

To implement and bind a Sensor:

1. Create a Class that **implements** the interface **ActuatorInterface** and implements the code for the abstract method:

```

/**
 * Called during registration phase
 *
 * @param agentName registering this actuator
 */
public void register(String agentName);

/**
 * Called during reset
 */
public void reset();

/**
 * Called each request to act.
 *
 * @param parameters to use for act

```

```

    * @return true if act went OK
    */
    public boolean actuator(String[] parameters);

```

This method should implement the functionality, by acting on the environment considering the passed-in arguments.

2. Bind this class to the Agent using the Agent.addActuator(.) method:

```

/**
 * Add actuator.
 *
 * @param actionStr identification for the actuator.
 * @param actuator to add
 */
public final void addActuator(String actionStr,
                               ActuatorInterface actuator)

```

For example:

```

/**
 * ACTUATOR TO PRINT INTO A FORM FIELD
 */
class TextFieldActuator
    implements ActuatorInterface {
    TextField field;

    // create: associate to field
    public TextFieldActuator(TextField field) {
        this.field = field;
    }

    // called during registration
    public void register(String agentName){}

    // called during reset
    public void reset(){}

    // called from Deliberation engine
    public boolean actuator(String[] data) {
        if (data.length > 0) {
            this.field.setString(data[0]);
        }
        return true;
    }
}

```

And bind to an Agent like:

```

ag.addActuator("PrintClock(X)", new TextFieldActuator(fieldClock));

```

The complete example can be found at the Appendix I.

Built-in Capabilities

Built-in Capabilities are those treated internally in the Capability execution code thus not requiring explicit declaration in the 3APL Capability base.

@command Sensor(SensorId, Result)

Where SensorId is the name of an allocated sensor

Returns the value collected from the sensor

@command System(SystemParameter, Result)

Where SystemParameter is:

memused: memory used by application

timestamp: current time stamp

any of the System.getProperty(.) values

Example code for using the built-in Capabilities:

```
// create tApl agent
Agent ag = new Agent("micro");
ag.setTrace(4);

// load knowledge
ag.addCapability("{} Print(X) {write([X,'\n'])}");
ag.addPlanRule("doIt <- TRUE | Sensor(clock, Result), Print(Result)");
ag.addPlanRule("doIt <- TRUE | System(memused, Result), Print(Result)");
ag.addGoal("doIt");

// attach sensors
ag.addSensor("clock", new ClockSensor(), 0, false);

// deliberate
ag.deliberate();
```

@command STOP

End deliberation cycle (clear remaining goals)

@command Assert(Knowledge)

Adds knowledge to beliefBase

```
ag2.addPlanRule("goZero <- pos(X) AND X==0 AND System(timestamp, Time) |
    Assert(reachedBaseZero(Time)).
```

@command Retract(Knowledge)

Remove knowledge to beliefBase

```
ag2.addPlanRule("goZero <- pos(X) AND X==0 AND System(time, Time) |
    Assert(reachedBaseZero(Time)).
```

@command AddGoal(Goal)

Adds new Goal to Goalbase

```
ag2.addPlanRule("goZero <- pos(X) AND X==0 | AddGoal(goBack).
```

Networking

The version 1.2 includes the built-in capabilities Send() and Receive() design for networking. The methods to integrate this structure to external network handlers will be released in a later version.

@command Send(MsgID, Destination, Performative, Data);

Where:

MsgID is the message identification

Destination is the receiver's address

Performative to execute (FIPA standard)

Data to send

@command Receive(MsgID, Sender, Performative, Data, [,Timeout])

Where:

MsgID is the message identification waiting for (for WILDCARD)

Sender is the message originator waiting for (for WILDCARD)

Performative is the received message's performative

Data is the received message's data

Timeout in seconds (optional)

Example code:

```
// CREATE AGENT FOR THE MOBILE
final Agent micro = new Agent("micro");

// load knowledge
micro.addBelief("location(near, storeA).");
micro.addBelief("shoppingList([productA, productB]).");

micro.addPlanRule("displayQuote(Shopping, Quote) <- TRUE |"+
    "Print(['Quote received from ',Shopping,' is $',Quote]).");
micro.addPlanRule("getQuote(Shopping, List, Result) <- TRUE |"+
    "Send(MsgId, Shopping, 'query-ref', quote(List)), " +
    "Receive(MsgId, Shopping, Performative, Result, 4).");
micro.addPlanRule("resolve <- location(near, Shopping) AND shoppingList(List)
|"+ "getQuote(Shopping, List, Result), " +
```

```

        "Assert(receivedQuote(Shopping, List, Result)), " +
        "displayQuote(Shopping, Result).");

micro.addGoal("resolve");

// CREATE AGENT FOR THE STORE
final Agent storeA = new Agent("storeA");

storeA.addCapability("{} Print(Value) { write(Value) }");
storeA.addCapability("{} CalculateQuote(Request, Value)
    { random(50, 100,Value) }");
storeA.addPlanRule("resolve <- TRUE |"+
    "Receive(MsgID, User, request, Request)," +
    "CalculateQuote(Request, Value)," +
    "Send(MsgID, User, response, Value)");
storeA.addGoal("resolve");

// deliberate in parallel

new Thread(new Runnable() {
public void run() {
    micro.deliberate();
    System.out.println(micro.toString());
}
}).start();

new Thread(new Runnable() {
public void run() {
    storeA.deliberate();
}
}).start();

```

Interfacing (GUI)

The J2ME library contains the class *mTripleApl..micro.J2MEGUIActuator (MIDlet)* that can be used for simple interfacing creation.

```

/**
 * Create the GUIActuator attached to a Midlet
 * (use default Image for interfaces)
 *
 * @param midlet to be attached
 */
public J2MEGUIActuator(MIDlet midlet);

/**
 * Create the GUIActuator attached to a Midlet
 * (use alternative image file)

```

```

*
* @param midlet to be attached
* @param imageFileName of image file to be used
*/
public J2MEGUIActuator(MIDlet midlet, String imageFileName);

```

The procedure is:

(1) add J2MEGUIActuator to the Agent:

```
micro.addActuator("GUI(Type,Message)", new J2MEGUIActuator(this));
```

(2) call the actuator using promptOk, promptYesNo

```
GUI(promptYesNo, Message)
```

```
GUI(promptOk, Message)
```

Example code:

```

final Agent micro = new Agent("micro");
micro.addActuator("GUI(Type,Message)", new J2MEGUIActuator(this));

micro.addCapability("{} AskConfirmation(Message)
    { GUI(promptYesNo, Message) }");
micro.addCapability("{} Display(Message)
    { GUI(promptOk, Message) }");

micro.addPlanRule("resolve <- location(near, Shopping) AND shoppingList(List)
|"+
    "AskConfirmation(['Near ',Shopping, '. Request for quote?'])" +
    "getQuote(Shopping, List, Result), " +
    "Assert(receivedQuote(Shopping, List, Result)), " +
    "displayQuote(Shopping, Result).");
micro.addGoal("resolve");
micro.deliberate();

```

The resulting interface is:



Memory release control

The version 1.3.3 introduces a method to control the frequency of memory garbage collection implemented between deliberation steps. From implementation experiences, we learned that memory release is a major performance factor in J2ME applications. The Agent.MODERATED level will result in more memory being used but better performance. In the other extreme, if Agent.AGGRESSIVE is set, the memory consumption will be reduced significantly on expenses of processor.

```
/**
 * Set level of Garbage Collection.
 * @param level for GC activation (Agent.GC_NONE, Agent.GC_AGGRESSIVE,
 Agent.GC_MEDIUM, Agent.GC_MODERATED)
 */
public final void setGCLevel(int level)
```

Example code:

```
public static void main(String[] args) {
    Agent ag = new Agent("micro");
    ag.setTrace(4);
    ag.setGCLevel(Agent.GC_MODERATED);
}
```

Limitations and known issues

There are some limitations introduced in the 3APL-M platform, for the sake of simplicity. This section will present these reductions.

In 3APL language

The following predicates, defined in the 3APL BNF syntax model, ARE NOT IMPLEMENTED in this version of the application:

```
<Sequencegoal> ::= 'BEGIN' <goal> ; <goal> 'END'
<javagoal> ::= 'Java(" <identifier> "', <methodcall> ', <varname> ')"
<ifgoal> ::= 'IF' <wff> 'THEN' <goal> | 'IF' <wff> 'THEN' <goal> 'ELSE' <goal>
```

$\langle \textit{whilegoal} \rangle ::= \textit{'WHILE'} \langle \textit{wff} \rangle \textit{'DO'} \langle \textit{goal} \rangle$

Acknowledgements

3APL-M code is based on original **3APL Platform** code, developed at Intelligent Systems Group, Institute of Information and Computing Sciences, Utrecht University. Thanks for Prof. John-Jules Meyer, Prof. Frank Dignum, Prof. Mehdi Dastani and Meindert Kroese, whose explanations and publications were indispensable in providing the theoretical basis for this development. Special thanks to Meindert for patiently answering my long thread of explanation requests on the original source code. The **mProlog** engine was developed based on the W-Prolog code from Michael Winikoff. That code was heavily re-engineered, by introducing the required methods for the 3APL integration. Thanks Michael for letting us use your code as reference and support our project.

3APL-M has been developed when the author was working at University of Melbourne, Australia. The author is thankful for the colleagues in Melbourne, specially Anton Katan, Iyad Rahwan and Liz Sonenberg, for the collaboration and input.

References

3APL Web Site at <http://www.cs.uu.nl/3apl>

3APL-M Web Site at <http://www.cs.uu.nl/3apl-m>

Koen V. Hindriks and Frank S. De Boer and Wiebe Van Der Hoek and John-Jules Ch. Meyer. Agent Programming in 3APL, Autonomous Agents and Multi-Agent Systems. Autonomous Agents and Multi-Agent Systems. v. 2. n. 4. pp. 357--401. Kluwer Academic Publishers. 1999.

Mehdi Dastani, Frank Dignum and John-Jules Meyer. Autonomy and Agents Deliberation. The First International Workshop on Computational Autonomy - Potential, Risks, Solutions (Autonomous 2003). pp. 23—35. Melbourne, Australia. July, 2003.

Mehdi Dastani, Frank de Boer, Frank Dignum and John-Jules Meyer. Programming Agent Deliberation: An Approach Illustrated Using the 3APL Language. Autonomous Agents and Multi-Agent Systems 2003 (AAMAS'03). Melbourne, Australia. July, 2003.

Frank Dignum and Rosaria Conte. Intentional agents and goal formation. Intelligent Agents IV: 4th International Workshop on Agent Theories, Architectures and Languages. Springer Verlag. LNAI 1365. pp. 231—244. 1998.

Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, John-Jules Meyer. A Programming Language for Cognitive Agents: Goal Directed 3APL. Proceedings of the First Workshop on Programming

Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS03) to be held at AAMAS'03, Melbourne, July 2003, 2003.

Mehdi Dastani, Frank Boer, Frank Dignum, Wiber van der Hoek, Meindert Kroese, John-Jules. Meyer, Programming the Deliberation Cycle of Cognitive Robots, Proceedings of the The Third International Cognitive Robotics Workshop, Held in conjunction with AAI-2002, Edmonton, Canada 2002

Appendix I: Complete Code Examples

More examples can be found in the web site at <http://www.cs.uu.nl/3apl-m/demo.html>.

Example of Sensor and Actuator based application.

```
[00008656] tap1 1 micro => goal-add-in-front: PrintClock(Wed Sep 08 05:42:00 UTC 2004)
[00008656] tap1 1 micro => === deliberation-step ===
[00008672] tap1 3 micro => goal-base contains: [PrintClock(Wed Sep 08 05:42:00 UTC 2004)]
[00009344] tap1 1 micro => belief-delete: sensor(status, 118524 bytes used)
[00009391] tap1 1 micro => belief-add: sensor(status, 118520 bytes used)
[00009406] tap1 1 micro => goal-add: sensor(status, 118520 bytes used)
[00009422] tap1 1 micro => deliberating: [sensor(status, 118520 bytes used)]
[00009422] tap1 1 micro => === deliberation-step ===
[00009438] tap1 3 micro => goal-base contains: [sensor(status, 118520 bytes used)]
[00009438] tap1 3 micro => planrule-find-rejected: sensor(clock(), X) <- TRUE | PrintClock
[00009438] tap1 2 micro => planrule-find-selected: (1) sensor(status(), X) <- TRUE | Print
[00009453] tap1 1 micro => planbase-add: sensor(status(), 118520 bytes used) <- TRUE | Pr
[00009484] tap1 2 micro => planbase-result: sensor(status(), 118520 bytes used) <- TRUE |
[00009500] tap1 1 micro => goal-add-in-front: PrintStatus(118520 bytes used)
[00009500] tap1 1 micro => === deliberation-step ===
[00009547] tap1 3 micro => goal-base contains: [PrintStatus(118520 bytes used)]
[00010516] tap1 1 micro => belief-delete: sensor(clock, Wed Sep 08 05:42:00 UTC 2004)
[00010547] tap1 1 micro => belief-add: sensor(clock, Wed Sep 08 05:42:02 UTC 2004)
[00010578] tap1 1 micro => goal-add: sensor(clock, Wed Sep 08 05:42:02 UTC 2004)
[00010578] tap1 1 micro => deliberating: [sensor(clock, Wed Sep 08 05:42:02 UTC 2004)]
[00010594] tap1 1 micro => === deliberation-step ===
[00010594] tap1 3 micro => goal-base contains: [sensor(clock, Wed Sep 08 05:42:02 UTC 2004)
[00010609] tap1 2 micro => planrule-find-selected: (1) sensor(clock(), X) <- TRUE | PrintC
[00010641] tap1 3 micro => planrule-find-rejected: sensor(status(), X) <- TRUE | PrintSta
[00010656] tap1 1 micro => planbase-add: sensor(clock(), Wed Sep 08 05:42:02 UTC 2004()) <-
[00010672] tap1 2 micro => planbase-result: sensor(clock(), Wed Sep 08 05:42:02 UTC 2004C
[00010672] tap1 1 micro => goal-add-in-front: PrintClock(Wed Sep 08 05:42:02 UTC 2004)
[00010688] tap1 1 micro => === deliberation-step ===
[00010688] tap1 3 micro => goal-base contains: [PrintClock(Wed Sep 08 05:42:02 UTC 2004)]
```



```
package mTripleApl.micro;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import mTripleApl.*;
import java.util.*;

/**
 * Example: Clock SENSOR, MicroEdition Form ACTUATOR.
 *
 * Create a agent with:
 * - one Sensor for time
 * - one Actuator to print in a form field
 * - one rule that handles notifications from the Sensor and forwards the
 * data to the Actuator
 */
```

```

*
* @author Fernando Koch
*/

public class TestMicro001
    extends MIDlet
    implements CommandListener {

    Form form;

    /**
     * MIDLET INTERFACE.
     * Start MIDlet run.
     */
    public void startApp() {

        // create micro from and display
        this.form = new Form("First mTApI application");
        this.form.setCommandListener(this);
        this.form.addCommand(new Command("Exit", Command.EXIT, 1));
        TextField fieldClock = new TextField("Clock Sensor", "", 30, TextField.ANY);
        this.form.append(fieldClock);
        TextField fieldStatus = new TextField("Status Sensor", "", 30,
TextField.ANY);
        this.form.append(fieldStatus);
        Display.getDisplay(this).setCurrent(this.form);

        // create tApI agent
        Agent ag = new Agent("micro");
        ag.setTrace(4);

        // load knowledge
        ag.addActuator("PrintClock(X)", new TextFieldActuator(fieldClock));
        ag.addActuator("PrintStatus(X)", new TextFieldActuator(fieldStatus));
        ag.addPlanRule("sensor(clock,X) <- TRUE | PrintClock(X)");
        ag.addPlanRule("sensor(status,X) <- TRUE | PrintStatus(X)");

        // attach sensors
        ag.addSensor("clock", new ClockSensor(), 2, true);
        ag.addSensor("status", new StatusSensor(), 3, true);

        // append tApI agent to form
        this.form.append(ag.toString());
    }

    /**
     * MIDLET INTERFACE.
     * Handle pausing the MIDlet
     */
    public void pauseApp() {
    }
}

```

```

/**
 * MIDLET INTERFACE.
 * Handle destroying the MIDlet
 *
 * @param unconditional for no cry
 */
public void destroyApp(boolean unconditional) {
    notifyDestroyed();
}

/**
 * Handle command events
 *
 * @param command incoming
 * @param displayable element
 */
public void commandAction(Command command, Displayable displayable) {
    if (command.getCommandType() == Command.EXIT) {
        destroyApp(true);
    }
}

/**
 * ACTUATOR TO PRINT INTO A FORM FIELD
 */
class TextFieldActuator
    implements ActuatorInterface {
    TextField field;

    // create: associate to field
    public TextFieldActuator(TextField field) {
        this.field = field;
    }

    // called during registration
    public void register(String agentName){}

    // called during reset
    public void reset(){

    }

    // called from Deliberation engine
    public boolean actuator(Hashtable parameters) {
        this.field.setString((String) parameters.get("X"));
        return true;
    }
}

/**
 * SENSOR FOR CLOCK

```

```

*/
class ClockSensor
    extends Sensor {

    // called during registration
    public void register(String agentName){}

    // called during reset
    public void reset(){

    // called from Deliberation engine
    public Object collectData() {
        return new Date(System.currentTimeMillis());
    }
}

/**
 * SENSOR FOR KVM STATUS
 */
class StatusSensor
    extends Sensor {

    // called during registration
    public void register(String agentName){}

    // called during reset
    public void reset(){

    // called from Deliberation engine
    public Object collectData() {
        System.gc();
        long usedMemory = Runtime.getRuntime().totalMemory() -
            Runtime.getRuntime().freeMemory();
        return Long.toString(usedMemory) + " bytes used";
    }
}

```

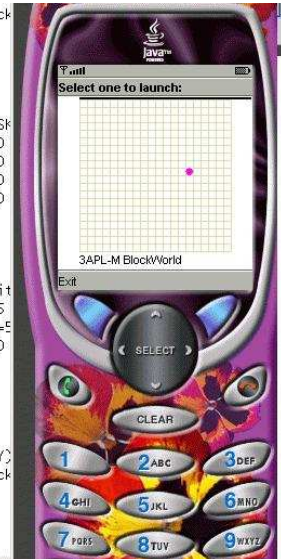
Example of Deliberative Agent and BlockWorld.

The BlockWorld code can be downloaded from the 3APL-M web site.

```

[00005078] tap1 3 micro => execute-capability-poscondition: NOT pos(X, Y), pos(X + 1, Y), Block
[00005078] tap1 1 micro => belief-delete: pos(13, 10)
[00005125] tap1 1 micro => belief-add: pos(14, 10)
Order to go east()
[00005640] tap1 1 micro => === deliberation-step ===
[00005640] tap1 3 micro => goal-base contains: [goBase()]
[00005656] tap1 2 micro => planrule-find-selected: (1) goBase() <- pos(X, Y) AND base(X, Y) | Sk
[00005671] tap1 2 micro => planrule-find-selected: (2) goBase() <- pos(X, Y) AND base(A, B) AND
[00005671] tap1 2 micro => planrule-find-selected: (3) goBase() <- pos(X, Y) AND base(A, B) AND
[00005687] tap1 2 micro => planrule-find-selected: (4) goBase() <- pos(X, Y) AND base(A, B) AND
[00005718] tap1 2 micro => planrule-find-selected: (5) goBase() <- pos(X, Y) AND base(A, B) AND
[00005812] tap1 1 micro => planbase-add: goBase() <- TRUE | West(), goBase()
[00005859] tap1 1 micro => planbase-add: goBase() <- TRUE | East(), goBase()
[00005953] tap1 1 micro => planbase-add: goBase() <- TRUE | South(), goBase()
[00006015] tap1 1 micro => planbase-add: goBase() <- TRUE | North(), goBase()
[00006062] tap1 2 planranker => planranker-dropping goBase() <- TRUE | North(), goBase(): utilit
[00006093] tap1 3 micro => plan-classified: (1)= goBase() <- TRUE | East(), goBase(): utility=5
[00006125] tap1 3 micro => plan-classified: (2)= goBase() <- TRUE | South(), goBase(): utility=5
[00006125] tap1 3 micro => plan-classified: (3)= goBase() <- TRUE | West(), goBase(): utility=0
[00006140] tap1 2 micro => planbase-result: goBase() <- TRUE | East(), goBase()
[00006187] tap1 1 micro => goal-add-in-front: goBase()
[00006187] tap1 1 micro => goal-add-in-front: East()
[00006187] tap1 1 micro => === deliberation-step ===
[00006187] tap1 3 micro => goal-base contains: [East(), goBase()]
[00006218] tap1 2 micro => execute-capability: {pos(X, Y)} East() { NOT pos(X, Y), pos(X + 1, Y), Block
[00006281] tap1 3 micro => execute-capability-poscondition: NOT pos(X, Y), pos(X + 1, Y), Block
[00006281] tap1 1 micro => belief-delete: pos(14, 10)
[00006312] tap1 1 micro => belief-add: pos(15, 10)
Order to go east()

```



```

package mTripleApl.micro;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import mTripleApl.*;
import java.util.*;

/**
 * Example: BlockWorld ACTUATOR, Deliberative cycle.
 *
 * Create a agent with:
 * - one Actuator to print in a BlockWorld (Canvas) object
 *
 * @author Fernando Koch
 */

public class TestMicro003
    extends MIDlet
    implements CommandListener {

    BlockWorld blockWorld;

    /**
     * MIDLET INTERFACE.
     * Start MIDlet run.
     */
    public void startApp() {

        // create BlockWorld
        this.blockWorld = new BlockWorld(20, 10, 10);
        this.blockWorld.setCommandListener(this);
        this.blockWorld.addCommand(new Command("Exit", Command.EXIT, 1));
        Display.getDisplay(this).setCurrent(this.blockWorld);
        this.blockWorld.repaint();
    }
}

```

```

// create agent
DeliberativeAgent ag = new DeliberativeAgent("micro");
ag.setTrace(6);

// load knowledge bases
ag.addCapability(
    "{pos(X,Y)} West() { NOT pos(X,Y) , pos(X-1,Y) , BlockMove(west)}");
ag.addCapability(
    "{pos(X,Y)} East() { NOT pos(X,Y) , pos(X+1,Y) , BlockMove(east)}");
ag.addCapability(
    "{pos(X,Y)} North() { NOT pos(X,Y) , pos(X,Y+1) , BlockMove(north)}");
ag.addCapability(
    "{pos(X,Y)} South() { NOT pos(X,Y) , pos(X,Y-1) , BlockMove(south)}");
ag.addPlanRule("goBase <- pos(X,Y) AND base(X,Y) | SKIP.");
ag.addPlanRule(
    "goBase <- pos(X,Y) AND base(A,B) AND X>A | West() , goBase().");
ag.addPlanRule(
    "goBase <- pos(X,Y) AND base(A,B) AND X<A | East() , goBase().");
ag.addPlanRule(
    "goBase <- pos(X,Y) AND base(A,B) AND Y>B | South() , goBase().");
ag.addPlanRule(
    "goBase <- pos(X,Y) AND base(A,B) AND Y<B | North() , goBase().");
ag.addBelief("pos(10,10)");
ag.addBelief("base(19,19)");
ag.addBelief("base(0,0)");
ag.addGoal("goBase()");

// set costs/values
ag.setCost("South()", 5);
ag.setCost("North()", 4);
ag.setCost("East()", 5);
ag.setCost("West()", 10);
ag.setWorth("goBase()", 10);

// attach actuator
ag.addActuator("BlockMove(X)", new BlockWorldActuator2(this.blockWorld));

// deliberate
ag.deliberate();

// print agent mental state
System.out.println(ag.toString());

// print agent mental state
System.out.println(ag.getStatistics());
}

/**
 * MIDLET INTERFACE.
 * Handle pausing the MIDlet

```

```

    */
    public void pauseApp() {
    }

    /**
     * MIDLET INTERFACE.
     * Handle destroying the MIDlet
     *
     * @param unconditional for no cry
     */
    public void destroyApp(boolean unconditional) {
        notifyDestroyed();
    }

    /**
     * Handle command events
     *
     * @param command incoming
     * @param displayable element
     */
    public void commandAction(Command command, Displayable displayable) {
        if (command.getCommandType() == Command.EXIT) {
            destroyApp(true);
        }
    }
}

/**
 * ACTUATOR TO BLOCK WORLD
 */
class BlockWorldActuator2
    implements ActuatorInterface {
    BlockWorld blockWorld;

    // create: associate to world
    public BlockWorldActuator2(BlockWorld blockWorld) {
        this.blockWorld = blockWorld;
    }

    // called during registration
    public void register(String agentName){}

    // called during reset
    public void reset(){}

    // called from Deliberation engine
    public boolean actuator(Hashtable parameters) {
        String command = (String) parameters.get("X");
        if (command != null) {
            System.out.println("Order to go " + command);
        }
    }
}

```

```
    if (command.startsWith("west")) {
        this.blockWorld.moveX( -1);
    }
    else if (command.startsWith("east")) {
        this.blockWorld.moveX(1);
    }
    else if (command.startsWith("north")) {
        this.blockWorld.moveY(1);
    }
    else if (command.startsWith("south")) {
        this.blockWorld.moveY( -1);
    }
}
return true;
}
}
```