Exploring the World of CEP: Controlling Robots using CEP - Part 1

By Andy Yuen

Feb 8, 2014.

Table of Contents

1 Introduction

As you all know, CEP stands for Complex Event Processing. According to Wikipedia:

> "**Event processing** is a method of tracking and analyzing (processing) streams of information (data) about things that happen (events), and deriving a conclusion from them."

Drools Fusion is responsible for enabling CEP on the Drools rule engine. This project uses Drools Fusion. It is assumed that you already know how a rule engine like Drools works.

Having nothing to do in my spare time at home (REALLY?), I'd like to create an application small enough for demo purposes and yet mirrors the architecture of Enterprise CEP applications. In other words, I want to build a small CEP application whose structure and architecture can be reused in an Enterprise environment. Many CEP applications examples can be found on the Internet including fraud detection in a banking environment so I am not going to do yet another one of those usual topics. I am going to create a CEP application to control robots. It will be fun as well as educational, for me anyway, in my journey into the world of CEP.

The requirements for the robot is minimal. It must support remote commands for motor control and have 3 distance sensors detecting whether there are obstacles in the front middle, front left or front right direction and be able to send that information to the CEP application as events via an adapter (to be

designed in Part 2). Alternatively, it can have a scanning distance sensor (ie, a distance sensor mounted on top of a servo that moves from side to side) to provide the same information (more will be reviewed in Part 3). The behaviour of the robot is quite simple. It moves in a random direction. When it detects an obstacle in a its path, it just steers around it if the obstacle is not to close. If the obstacle is close-by, it will turn in place to avoid the obstacle. If it gets stuck in a place ie, turn in place left to right and right to left for several times without a clear passage way, it will turn around and move away from the obstacle. You may say, it is so simple that even a $20 Arduino can do it, why use CEP. You are right but remember that this is just the start. This approach can control a swarm of robots instead of just one as long as they have unique names and that they all send events to the CEP application (more on this later). Also more complex behaviours can be added in the form of adding more rules. And most of all, we are exploring what makes up a CEP application.

I am going to document my journey into the CEP world in three parts:

- Part 1 – Drools rules and JUnit tests (this installment)
- Part 2 – CEP Application Architecture (applies to enterprise CEP applications as well)
- Part 3 – Controlling a Real Robot using CEP (the fun part of the project that utilises everything that has been discussed in the earlier installments)

I shall publish the subsequent installments when ready.

2 Desired Robot Behaviour

1. The event initiated by the robot contains a distance value in cm and direction (front left, front middle or front right). It is up to the CEP application to decide whether the event can be ignored or close enough to warrant an action
2. The three events (front left, front middle or front right) are always sent by the robot all at once. These events are assessed to determine what action to take:

   For mid range (determined by the CEP application):

   - if front-middle is clear, continue to move forward
   - if two directions are blocked and a third direction is unblocked, turn to the unblocked direction and move forward
   - if the front middle is blocked and the other directions are unblocked, turn to one of the unblocked directions and move forward

   For close range (determined by the CEP application):

   - if front-middle is clear, continue to move forward
   - if two directions are blocked and a third direction is unblocked, turn in place to

the unblocked direction and move forward
- if the front middle is blocked and the other directions are unblocked, turn in place to one of the unblocked directions and move forward
- if the robot has been turning in place left and right multiple times (to be specified by the CEP application) without being able to get a clear passageway, initiate recovery action eg, turn around and move away

3 Model and Design

In order to implement the desired behaviour, the following events (POJOs) are defined:

- RawObstacleEvents – An event initiated by the robot to report on an obstacle. It contains a name (of the robot), distance (in cm) and direction (front left, front middle or front right)

- MidRangeObstacleEvent - An event generated by rules in the rule engine if the obstacle reported in a RawObstacleEvents is within a certain range. It contains a name and direction.

- MidRangeRecoveryEvent - An event generated by the rule engine to signify the start of recovery for the robot. It is used as a starting point in time for temporal conditions.

- CloseRangeObstacleEvent - An event generated by rules in the rule engine if the obstacle reported in a RawObstacleEvents is within a certain range and that range is closer than that reported by a MidRangeObstacleEvent . It contains a name and direction.

- CloseRangeRecoveryEvent - An event generated by the rule engine to signify the start of recovery for the robot. It is used as a starting point in time for temporal conditions.

One of the events described above is shown in code below (note that it is just a POJO):

```
package org.robot.cep.model;

public class RawObstacleEvent {

   public final String name;
   public final Direction direction;
   public final int distance;

   public String getName() {
                return name;
   }

   public Direction getDirection() {
                return direction;
   }

   public int getDistance() {
                return distance;
   }

   public RawObstacleEvent(String name, Direction direction, int distance)
   {
                this.name = name;
```

```
		this.direction = direction;
		this.distance = distance;
	}

	@Override
	public String toString() {
		// TODO Auto-generated method stub
		return super.toString() + " (" + name + ":" + direction + ":" + distance +")";
	}


}
```

Note that it does not carry a time of occurrence field. The time of occurrence is added by the rule engine when the event is inserted into its working memory. I can provide the time stamp to be used be Drools if I choose to though. Also note that all fields are declared final as an event is immutable.

To tell Drools that RawObstacleEvent is an event, I have to add the following in the drl file:

```
declare RawObstacleEvent
   @role( event )
end
```

The robot class is simple.

```
package org.robot.cep.model;

public class Robot {
   static public final int MID_RANGE = 30;
   static public final int CLOSE_RANGE = 10;

   public enum State {
		NORMAL, AVOID, ESCAPE, DEADLOCK, RETRY
   }

    private String name;

    private State state;

    public RobotControl control;

    public RobotControl getControl() {
		return control;
   }

   public void setControl(RobotControl control) {
		this.control = control;
   }

   public String getName() {
       return this.name;
    }

    public void setName(String name) {
       this.name = name;
    }

    public State getState() {
       return this.state;
    }

    public void setState(State state) {
       this.state = state;
```

```
    }


}
```

The thing to note is that it has a RobotControl object to control the robot. RobotControl is an interface. It is a high level interface to control the movement of a robot. Each RobotControl implementation needs to implement the interface to control the robot remotely using motor control commands (more on this in Part 2). The default implementation provided (DefaultRobotControl) just prints out the method name. Here is the RobotControl interface. The purpose of each method should be self-explanatory.

```
package org.robot.cep.model;

public interface RobotControl {

    public void turnLeft();
    public void turnRight();
    public void turnAround();
    public void spinLeft();
    public void spinRight();
    public void moveForward();
    public void moveBackward();
    public void stop();
    public void performDeadlockManeuver();

}
```

4 Obstacle Avoidance Rules

The project files can be found HERE. The obstacle avoidance rules can be found in the file ceprobot.drl. A robot can be in one of five states:

- NORMAL – everything is OK
- AVOID – mid-range obstacles are detected and the robot performs some action
- ESCAPE – similar to AVOID but for close-range obstacles
- DEADLOCK – the robot is turning in place left and right for several times without being able to get away from the obstacles hence it performs a special maneuver (close-range obstacles)
- RETRY – this is the state the robot enters after performing an action without getting clear from the obstacles. The handling from now on is exactly the same as a robot in the NORMAL state. The reason for creating this state is to allow us to tell if a robot is in trouble

There are rules for the robot to get back to the NORMAL state as you will see later on.

I am going to walk through some of these rules.

```
rule "Event Conversion - close-range"
    salience -10
    when
        $robot : Robot( $myName : name )
                    $e1 : RawObstacleEvent(name == $myName, $direction : direction, distance <= Robot.CLOSE_RANGE)
                        from entry-point TelemetryStream
    then
            System.out.println("converted RawObstacleEvent to CloseRangeObstacleEvent: (" +
                    $myName + ":" + $direction + ":" + $e1.getDistance() + ")");
            retract($e1)
```

```
                entryPoints['TelemetryStream'].insert(new CloseRangeObstacleEvent($myName, $direction));

end
```

This is a simple rule. It converts a RawObstacleEvent initiated from a robot into a CloseRangeObstacleEvent if the distance to the obstacle is less than or equal Robot.CLOSE_RANGE cm and insert it into the entry point named 'TelemetryStream'. There are rules to convert RawObstacleEvent into MidRangeObstacleEvent and simply retract the RawObstacleEvent from the working memory if the obstacle is too far away to have any relevance. A couple of things to note:

- Robot objects are inserted into the working memory as initial facts (See)
- CEP deals with streams of events. Drools generalized the concept of a stream as an "entry point" into the engine. Each entry point defines a partition in the input data storage. Partitioning the stream events reduces
  the match space by allowing patterns to target specific partitions.

```
rule "Close-range - Obstacle - spin left"
    when
        $robot : Robot( (state == State.NORMAL || state == State.RETRY), $myName : name )
        $e1 : CloseRangeObstacleEvent(name == $myName, direction == Direction.MIDDLE)
            from entry-point TelemetryStream
        $e2 : CloseRangeObstacleEvent(name == $myName, direction == Direction.RIGHT,
            this coincides[100ms] $e1) from entry-point TelemetryStream
        not (CloseRangeObstacleEvent(name == $myName, direction == Direction.LEFT,
            this coincides[100ms] $e1) from entry-point TelemetryStream)
    then
        $robot.control.spinLeft();
        modify($robot) {setState(State.ESCAPE)};
        retract($e1)
        retract($e2)
        entryPoints['TelemetryStream'].insert(new CloseRangeRecoveryEvent($myName, State.ESCAPE));

end
```

The above rule says that for any robot which has received a front middle CloseRangeObstacleEvent and a front right CloseRangeObstacleEvent but no front left CloseRangeObstacleEvent at around the same time, then perform a spinLeft (turn left in place), change the robot state to ESCAPE, retract the 2 CloseRangeObstacleEvents from working memory and insert a CloseRangeRecoveryEvent. I shall describe the purpose of the CloseRangeRecoveryEvent in the next rule. Also note that this action is carried out only when the robot is at the NORMAL or RETRY state to prevent a loop.

```
rule "Close-range - Back to NORMAL"
    when
        $robot : Robot( state == State.ESCAPE || state == State.DEADLOCK, $myName : name )
        $e1 : CloseRangeRecoveryEvent(name == $myName) from entry-point TelemetryStream
        not (CloseRangeObstacleEvent(name == $myName, direction == Direction.MIDDLE, this after[1s, 5s] $e1)
    from entry-point TelemetryStream)
    then
        $robot.control.moveForward();
        System.out.println($myName + ": (CloseRangeRecoveryEvent) Resuming NORMAL state after: " + $robot.getState());
        retract($e1)
        modify($robot) {setState(State.NORMAL)};

end
```

This rule says that for any robot which has received a CloseRangeRecoveryEvent and no CloseRangeObstacleEvent for front middle direction is received in the next 5 seconds ie, no more

obstacles ahead, then move forward, retract the CloseRangeRecoveryEvent from working memory and set the state of the robot to NORMAL. Again a guard is put in there to check that the robot state is in either ESCAPE or DEADLOCK state before this rule is activated to prevent a loop. The purpose of the CloseRangeRecoveryEvent is to serve as a marker (time-wise) for the start of recovery as the rule checks that no CloseRangeObstacleEvent occurs within 5 seconds after the CloseRangeRecoveryEvent.

The complementary rule to the above is:

```
rule "Close-range - Retry"
    when
        $robot : Robot( state == State.ESCAPE || state == State.DEADLOCK, $myName : name )
        $e1 : CloseRangeRecoveryEvent(name == $myName) from entry-point TelemetryStream
        CloseRangeObstacleEvent(name == $myName, direction == Direction.MIDDLE, this after[1s, 5s] $e1)
    from entry-point TelemetryStream
    then
        $robot.control.stop();
        System.out.println($myName + ": (CloseRangeRecoveryEvent) Cannot avoid obstacle after state: " + $robot.getState()
    + " - retrying...");
        retract($e1)
        modify($robot) {setState(State.RETRY)};

end
```

It states that if the road ahead is still blocked, enter the RETRY state so that relevant obstacles avoidance rules will be evaluated again when new events come in.

```
rule "Close-range - Deadlock Detected"
    when
                $robot : Robot( state == State.ESCAPE, $myName : name )
                $list : ArrayList( size > 5)
                        from collect(CloseRangeRecoveryEvent(name == $myName) over window:time (10s)
                        from entry-point TelemetryStream)

    then
                System.out.println($myName + ": Close range recovery attempted " + $list.size() + " times within 10 seconds");
                modify($robot) {setState(State.DEADLOCK)};
                $robot.control.performDeadlockManeuver();
                // retract all CloseRangeRecoveryEvents in the list
                // to prevent multiple invocations of the "Close-range - Back to NORMAL" rule
                for (Object event: $list) { retract(event); };
                // insert a new CloseRangeRecoveryEvent for recovery purpose
                entryPoints['TelemetryStream'].insert(new CloseRangeRecoveryEvent($myName, State.DEADLOCK ));

end
```

The above rule uses a sliding window and the collect conditional element to check that there are more than 5 CloseRangeRecoveryEvents inserted within a 10 second interval signifying a possible deadlock condition. This means that the robot is stuck at a place where it turns in place left and right and then right to left and so on but still cannot get out of the situation. When this happens, it will execute a performDeadlockManeuver in which it turns around and go the other way. Again a check is put in there to make sure the robot state is ESACPE before this rule is activated to avoid a loop. Note that it also retracts all CloseRangeRecoveryEvents otherwise this rule will fire multiple times.

5 Differences between Knowledge and KIE APIs

Writing the rules are straight forward but it requires effort to set up the JUnit tests to validate that the rules do what they are created for. I wanted to see how different it is to use the Drools 5 and 6 APIs so I

used both the Knowledge API and the new KIE API for setting up a drools session. The KIE API is much less verbose than the Knowledge API in creating a session by leveraging the kmodule.xml file. Here is a comparison. Using the Knowledge API, I have to do this:

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add(ResourceFactory.newClassPathResource("ceprobot.drl"), ResourceType.DRL);
KnowledgeBuilderErrors errors = kbuilder.getErrors();
if (errors.size() > 0) {
   for (KnowledgeBuilderError error: errors) {
      System.err.println(error);
   }
   throw new IllegalArgumentException("Could not parse knowledge.");
}
KnowledgeBaseConfiguration config = KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
config.setOption( EventProcessingOption.STREAM );
kbase = KnowledgeBaseFactory.newKnowledgeBase(config);
kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());

// load up the knowledge base
KnowledgeSessionConfiguration conf = KnowledgeBaseFactory.newKnowledgeSessionConfiguration();
conf.setOption( ClockTypeOption.get( "pseudo" ) );
// create session
ksession = kbase.newStatefulKnowledgeSession(conf, null);
```

Using the KIE API, it becomes:

```
kcontainer = KIEServices.Factory.get().getKIEClasspathContainer();
// create session
ksession = kcontainer.newKIESession("CepRobotKS");
```

The latter is much less verbose. This is achieved by providing much of that information in the kmodule.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
 <kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
   <kbase name="CepRobotKB" default="true" eventProcessingMode="stream" >
    <ksession name="CepRobotKS" type="stateful" clockType="pseudo"
   default="true" />
   </kbase>
</kmodule>
```

Note that the eventProcessingType, session type and clockType are all specified in the xml file instead of Java code. GetKIEClasspathContainer() scans the directories beneath the classpath for resources including drl files.

Adding an AgendaEventListener in Drools 5:

```
ksession.addEventListener( new DefaultAgendaEventListener()
   { public void afterActivationFired(AfterActivationFiredEvent event)
   {
                    super.afterActivationFired( event );
    System.out.println("->\"" +  event.getActivation().getRule()
.getName() + "\" Rule fired: " );
   }
   });
```

Using Drools 6 (it is a bit confusing as KIE DefaultAgendaEventListener has a different interface from

that in the Knowledge API. Importing the wrong package will cause much confusion):

```
ksession.addEventListener( new DefaultAgendaEventListener()
  {public void afterMatchFired(AfterMatchFiredEvent event)
  {
                super.afterMatchFired( event );
                System.out.println( "->\"" + event.getMatch().getRule().getName() + "\" Rule fired: " );
  }
  });
```

6 Testing the Rules using JUnit

The Drools session and initial facts are set up in setUpBeforeClass() and setUp().

```
  @BeforeClass
  public static void setUpBeforeClass() throws Exception {
     kcontainer = KieServices.Factory.get().getKieClasspathContainer();
  }


  @Before
  public void setUp() throws Exception {
              // create session
              ksession = kcontainer.newKieSession("CepRobotKS");
               clock = ksession.getSessionClock();
               ksession.addEventListener( new DefaultAgendaEventListener() {
                 public void afterMatchFired(AfterMatchFiredEvent event) {
                     super.afterMatchFired( event );
                     System.out.println( "->\"" + event.getMatch().getRule().getName() + "\" Rule fired: " );
                 }
              });

               // insert initial fact
              robot = new Robot();
              robot.setName(ROBOT_NAME);
              robot.setState(State.NORMAL);
              robot.setControl(new DefaultRobotControl(ROBOT_NAME));
              robotHandle = (FactHandle) ksession.insert(robot);
  }
```

One of the tests is shown below for a glimpse of what is being tested and how.

```
  @Test
  public void avoidAndRetryTest() {
     printSeparator();

     entry = ksession.getEntryPoint(STREAM_NAME);
     fireAllAndAssertRobotStateEquals(State.NORMAL);

     // test avoid - turn right
     insertMidRangeObstacleEvent(ROBOT_NAME, Direction.LEFT);
     clock.advanceTime(10, TimeUnit.MILLISECONDS);
     insertMidRangeObstacleEvent(ROBOT_NAME, Direction.MIDDLE);
     clock.advanceTime(10, TimeUnit.MILLISECONDS);
     fireAllAndAssertRobotStateEquals(State.AVOID);
     clock.advanceTime(3, TimeUnit.SECONDS);
     insertMidRangeObstacleEvent(ROBOT_NAME, Direction.MIDDLE);


     // retry
     clock.advanceTime(3, TimeUnit.SECONDS);
     fireAllAndAssertRobotStateEquals(State.RETRY);
```

```
        // test avoid - turn right
        insertMidRangeObstacleEvent(ROBOT_NAME, Direction.MIDDLE);
        clock.advanceTime(10, TimeUnit.MILLISECONDS);
        insertMidRangeObstacleEvent(ROBOT_NAME, Direction.LEFT);
        clock.advanceTime(100, TimeUnit.MILLISECONDS);
        fireAllAndAssertRobotStateEquals(State.AVOID);

        // back to normal
        clock.advanceTime(10, TimeUnit.SECONDS);
        fireAllAndAssertRobotStateEquals(State.NORMAL);

        // test avoid - no turn
        insertMidRangeObstacleEvent(ROBOT_NAME, Direction.RIGHT);
        clock.advanceTime(10, TimeUnit.MILLISECONDS);
        insertMidRangeObstacleEvent(ROBOT_NAME, Direction.LEFT);
        clock.advanceTime(100, TimeUnit.MILLISECONDS);
        fireAllAndAssertRobotStateEquals(State.NORMAL);

    }
```

The test creates events and asserts that the robot is in a particular state before and after their injection into the working memory. Note the use of the pseudo clock in advancing the time. This particular test case covers the scenario where: while the robot is turning right, I inject another event to block its advance resulting in it entering the RETRY state. After that events come in and it reacts to them until it finally avoids the obstacle and gets back to the NORMAL state.

The JUnit test cases are implemented in CepRobotRuleTest.java. Here is the console output of the tests showing the rules that have been fired and the actions taken by the robot:

```
798 [main] INFO org.drools.compiler.kie.builder.impl.ClasspathKIEProject - Found kmodule: file:/home/ayuen/workspace/drools-
projects/ceprobot-6.0/ceprobot/target/META-INF/kmodule.xml
1277 [main] INFO org.drools.compiler.kie.builder.impl.KIERepositoryImpl - KIEModule was added:FileKIEModule[
ReleaseId=org.robotics.cep:CepRobot:1.0.0file=/home/ayuen/workspace/drools-projects/ceprobot-6.0/ceprobot/target]
1280 [main] INFO org.drools.compiler.kie.builder.impl.ClasspathKIEProject - Found kmodule: file:/home/ayuen/workspace/drools-
projects/ceprobot-6.0/ceprobot/target/META-INF/kmodule.xml
1285 [main] INFO org.drools.compiler.kie.builder.impl.KIERepositoryImpl - KIEModule was added:FileKIEModule[
ReleaseId=org.robotics.cep:CepRobot:1.0.0file=/home/ayuen/workspace/drools-projects/ceprobot-6.0/ceprobot/target]
***********************************************
3Pi: Turning left
->"Mid-range Obstacle - turn left" Rule fired:
3Pi: Moving forward
3Pi: (MidRangeRecoveryEvent) Resuming NORMAL state after: AVOID
->"Mid-range - Back to NORMAL" Rule fired:
3Pi: Turning left
->"Mid-range Obstacle - turn left" Rule fired:
3Pi: Moving forward
3Pi: (MidRangeRecoveryEvent) Resuming NORMAL state after: AVOID
->"Mid-range - Back to NORMAL" Rule fired:
3Pi: Turning left
->"Mid-range Obstacle - turn left" Rule fired:
3Pi: Moving forward
3Pi: (MidRangeRecoveryEvent) Resuming NORMAL state after: AVOID
->"Mid-range - Back to NORMAL" Rule fired:
***********************************************
3Pi: Spinning right
->"Close-range Obstacle - spin right" Rule fired:
3Pi: stop
3Pi: (CloseRangeRecoveryEvent) Cannot avoid obstacle after state: ESCAPE - retrying...
->"Close-range - Retry" Rule fired:
3Pi: Spinning right
->"Close-range Obstacle - spin right" Rule fired:
3Pi: Moving forward
3Pi: (CloseRangeRecoveryEvent) Resuming NORMAL state after: ESCAPE
->"Close-range - Back to NORMAL" Rule fired:
***********************************************
3Pi: Close range recovery attempted 6 times within 10 seconds
3Pi: Performing deadlock maneuver
```

```
->"Close-range - Deadlock Detected" Rule fired:
3Pi: Moving forward
3Pi: (CloseRangeRecoveryEvent) Resuming NORMAL state after: DEADLOCK
->"Close-range - Back to NORMAL" Rule fired:
3Pi: Moving forward
3Pi: (CloseRangeRecoveryEvent) Resuming NORMAL state after: NORMAL
->"Close-range - Back to NORMAL" Rule fired:
***********************************************
3Pi: Spinning left
->"Close-range - Obstacle - spin left" Rule fired:
3Pi: Moving forward
3Pi: (CloseRangeRecoveryEvent) Resuming NORMAL state after: ESCAPE
->"Close-range - Back to NORMAL" Rule fired:
3Pi: Spinning left
->"Close-range - Obstacle - spin left" Rule fired:
3Pi: Moving forward
3Pi: (CloseRangeRecoveryEvent) Resuming NORMAL state after: ESCAPE
->"Close-range - Back to NORMAL" Rule fired:
3Pi: Spinning left
->"Close-range - Obstacle - spin left" Rule fired:
3Pi: Moving forward
3Pi: (CloseRangeRecoveryEvent) Resuming NORMAL state after: ESCAPE
->"Close-range - Back to NORMAL" Rule fired:
***********************************************
3Pi: Turning right
->"Mid-range Obstacle - turn right" Rule fired:
3Pi: stop
3Pi: (MidRangeRecoveryEvent) Cannot avoid obstacle after state: AVOID - retrying...
->"Mid-range - Retry" Rule fired:
3Pi: Turning right
->"Mid-range Obstacle - turn right" Rule fired:
3Pi: Moving forward
3Pi: (MidRangeRecoveryEvent) Resuming NORMAL state after: AVOID
->"Mid-range - Back to NORMAL" Rule fired:
***********************************************
3Pi: Spinning left
->"Close-range - Obstacle - spin left" Rule fired:
3Pi: Moving forward
3Pi: (CloseRangeRecoveryEvent) Resuming NORMAL state after: ESCAPE
->"Close-range - Back to NORMAL" Rule fired:
***********************************************
3Pi: Turning right
->"Mid-range Obstacle - turn right" Rule fired:
3Pi: Moving forward
3Pi: (MidRangeRecoveryEvent) Resuming NORMAL state after: AVOID
->"Mid-range - Back to NORMAL" Rule fired:
3Pi: Turning right
->"Mid-range Obstacle - turn right" Rule fired:
3Pi: Moving forward
3Pi: (MidRangeRecoveryEvent) Resuming NORMAL state after: AVOID
->"Mid-range - Back to NORMAL" Rule fired:
***********************************************
Converted RawObstacleEvent to MidRangeObstacleEvent: (3Pi:LEFT:30)
->"Event Conversion - mid-range" Rule fired:
Converted RawObstacleEvent to MidRangeObstacleEvent: (3Pi:MIDDLE:30)
->"Event Conversion - mid-range" Rule fired:
3Pi: Turning right
->"Mid-range Obstacle - turn right" Rule fired:
3Pi: Moving forward
3Pi: (MidRangeRecoveryEvent) Resuming NORMAL state after: AVOID
->"Mid-range - Back to NORMAL" Rule fired:
converted RawObstacleEvent to CloseRangeObstacleEvent: (3Pi:MIDDLE:10)
->"Event Conversion - close-range" Rule fired:
converted RawObstacleEvent to CloseRangeObstacleEvent: (3Pi:LEFT:10)
->"Event Conversion - close-range" Rule fired:
3Pi: Spinning right
->"Close-range Obstacle - spin right" Rule fired:
converted RawObstacleEvent to CloseRangeObstacleEvent: (3Pi:RIGHT:10)
->"Event Conversion - close-range" Rule fired:
3Pi: Moving forward
```

```
3Pi: (CloseRangeRecoveryEvent) Resuming NORMAL state after: ESCAPE
->"Close-range - Back to NORMAL" Rule fired:
************************************************
3Pi: Spinning right
->"Close-range Obstacle - spin right" Rule fired:
3Pi: Moving forward
3Pi: (CloseRangeRecoveryEvent) Resuming NORMAL state after: ESCAPE
->"Close-range - Back to NORMAL" Rule fired:
3Pi: Spinning right
->"Close-range Obstacle - spin right" Rule fired:
3Pi: Moving forward
3Pi: (CloseRangeRecoveryEvent) Resuming NORMAL state after: ESCAPE
->"Close-range - Back to NORMAL" Rule fired:
************************************************
3Pi: Turning left
->"Mid-range Obstacle - turn left" Rule fired:
3Pi: Moving forward
3Pi: (MidRangeRecoveryEvent) Resuming NORMAL state after: AVOID
->"Mid-range - Back to NORMAL" Rule fired:
```

7 Summary

In this installment, I described:

- the problem statement for controlling robots
- the events used
- how a POJO can be specified in drl as an event
- how to write the rules using temporal operators and the collect conditional element and sliding window
- how to write JUnit tests using both the knowledge (Drools 5) and KIE (Drools 6) APIs

They lay the foundation for controlling a robot using CEP in the next installments.

Although the discussion talks about a single robot, the rules developed can actually control a swarm of robots provided each robot has a unique name. At this stage, we have the rules but how do we get the robot initiated events into the Drools working memory? How do we carry out the actions recommended by the rule engine on the robot? What is the application architecture for a CEP application? Can this architecture be used in an enterprise CEP application? All these questions will be asked in the next installments.

Stay tuned for Part 2