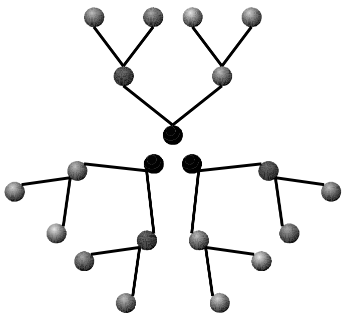

Multiple **H**ypothesis **L**ibrary

Programming Manual

revision 1.0

David Miguel Antunes
`davidmiguel [ at ] antunes.net`

May 10, 2011

Website

You can obtain more information and resources on the MHL website.

Please visit `http://www.multiplehypothesis.com`.

Contents

Chapter 1

The MHL API

All the classes in the MHL API will now be described. Additional information can be found in the javadoc.

1.1 The first classes

The description of the MHL's API classes will begin by those classes which are used to access the library and most of its functionality. Their UML class diagram is depicted in figure 1.1.

1.1.1 MHLService

The `MHLService` class represents the service which provides the MHL implementation. This service can be found using the Java class `ServiceLoader`. The `MHLService` class provides a `Factory` with which other objects of the library are created.

Code Example 1.1: Finding an MHL implementation.

```
ServiceLoader<MHLService> serviceLoader =
      ServiceLoader.load(MHLService.class);
MHLService mhl = serviceLoader.iterator().next();
```

In code example 1.1, variable `mhl` is set to the first `MHLService` implementation returned by the `ServiceLoader` class.

1.1.2 Factory

After obtaining a `Factory` from a `MHLService` implementation the library should now use it to create an `HypothesesManager`, which will be perhaps the most important class to deal with the library. This factory also provides the `PrunerFactory`, used to create pruners. The `Factory` also allows the application to create instances of classes such as the `GeneratedHypothesis` and the `GeneratedHypotheses` classes when generating new hypotheses.

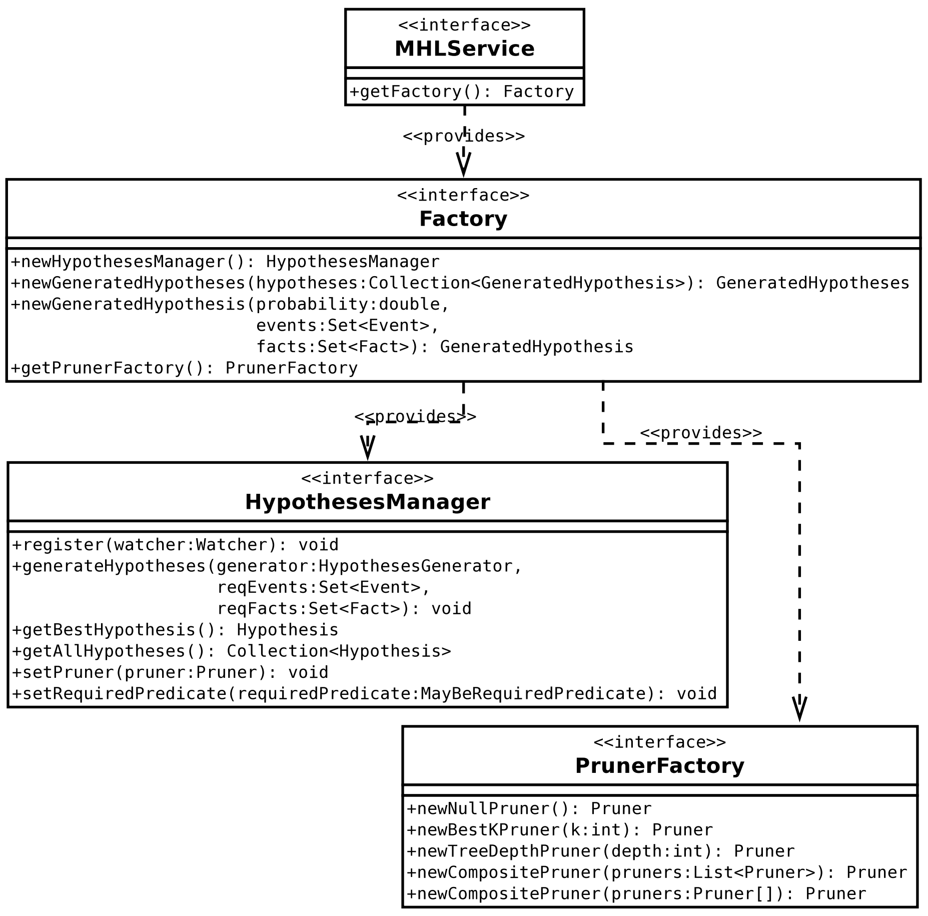
<<interface>>
MHLService
+getFactory(): Factory
<<provides>>
<<interface>>
Factory
+newHypothesesManager(): HypothesesManager
+newGeneratedHypotheses(hypotheses:Collection<GeneratedHypothesis>): GeneratedHypotheses
+newGeneratedHypothesis(probability:double,
events:Set<Event>,
facts:Set<Fact>): GeneratedHypothesis
+getPrunerFactory(): PrunerFactory
<<provides>>
<<provides>>
<<interface>>
HypothesesManager
+register(watcher:Watcher): void
+generateHypotheses(generator:HypothesesGenerator,
reqEvents:Set<Event>,
reqFacts:Set<Fact>): void
+getBestHypothesis(): Hypothesis
+getAllHypotheses(): Collection<Hypothesis>
+setPruner(pruner:Pruner): void
+setRequiredPredicate(requiredPredicate:MayBeRequiredPredicate): void
<<interface>>
PrunerFactory
+newNullPruner(): Pruner
+newBestKPruner(k:int): Pruner
+newTreeDepthPruner(depth:int): Pruner
+newCompositePruner(pruners:List<Pruner>): Pruner
+newCompositePruner(pruners:Pruner[]): Pruner

Figure 1.1: The classes used to access most of the library's functionality.

1.1.3 HypothesesManager

This class is responsible for the most important interactions with the library. A tracker usually only needs one `HypothesesManager` instance.

Usually, after creating an `HypothesesManager`, a watcher is registered, and the `Pruner` and `MayBeRequiredPredicate` are set.

1.1.4 PrunerFactory

The pruner factory allows the creation of different types of pruners. The null pruner does not perform any pruning. The best k pruner, and the depth pruner limit the number of leaves and the maximum depth of the hypothesis trees, respectively. The composite pruner creates a pruner which applies other pruners in sequence to the hypothesis tree.

Code Example 1.2: Setting the pruning strategy.

```
PrunerFactory prunerFactory = factory.getPrunerFactory();
hypothesesManager.setPruner(prunerFactory.newCompositePruner(
                new Pruner[]{
                    prunerFactory.newBestKPruner(6),
                    prunerFactory.newTreeDepthPruner(5)}));
```

In code example 1.2, the pruning strategy is set to a maximum of 6 leaves and 5 tree levels.

1.2 Events and Facts

The description of the MHL API will now focus on the `Event` and `Fact` classes and the classes which depend upon and use them.

1.2.1 Event

An event represents something which occurred in the world at some point in time. It usually contains a timestamp. An example of an event could be: “Ship 29384 moved from (183.23, 330.19) to (187.94, 334.82) at 22:43 26/07/2015”. When the probability of an event is 1, it is sent to the application via the `confirmedEvent` method of the `Watcher` interface.

Events should never be changed after their creation.

Code Example 1.3: Example of an event.

```
public class NewTargetEvent implements Event {
    private final long timestamp;
    private final long id;
    private final double x, y;

    // Constructor

    // Getters
}
```

1.2.2 Fact

A fact is a "piece" of the state of the world. For example when the application generates an event like "Ship 29384 moved from (183.23, 330.19) to (187.94, 334.82) at 22:43 26/07/2015", it may also generate a fact like "Ship 29384 is in (187.94, 334.82)"

Contrary to events, facts are never sent to the application. They are kept indefinitely by the library. However, when a fact is provided by the library to the application in the context of an hypothesis generation, by the method `generate` of the `HypothesesGenerator` class the fact is deleted from the library (in that hypothesis). Even so, it can be restated by the application by including it in the `GeneratedHypothesis`.

Facts should never be changed after their creation.

Code Example 1.4: Example of a fact.

```java
public class TargetFact implements Fact {
    private final long id, lastDetection;
    private final KalmanFilter kf;

    // Constructor

    // Getters
}
```

Because `Fact` is an interface and does not declare any methods, any Java class may be a fact, hence it being simply a "piece" of information.

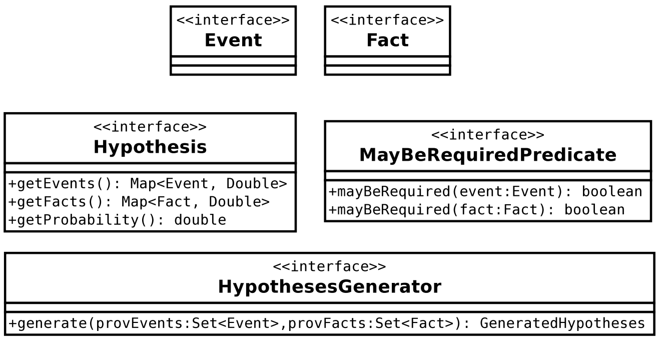
<<interface>>
Event
<<interface>>
Fact
<<interface>>
Hypothesis
+getEvents(): Map<Event, Double>
+getFacts(): Map<Fact, Double>
+getProbability(): double
<<interface>>
MayBeRequiredPredicate
+mayBeRequired(event:Event): boolean
+mayBeRequired(fact:Fact): boolean
<<interface>>
HypothesesGenerator
+generate(provEvents:Set<Event>,provFacts:Set<Fact>): GeneratedHypotheses

Figure 1.2: The classes related to events and facts.

1.2.3 Hypothesis

The `Hypothesis` class instances are created by the library when it is reporting on the best hypothesis or on all the hypothesis being maintained (when requested by the methods `getBestHypothesis` or `getAllHypotheses` of the `HypothesesManager` class).

The `Hypothesis` contains a set of events, a set of facts, and the probability of the hypothesis. It also contains the individual probability of each event and fact.

1.2.4 MayBeRequiredPredicate

Following the cluster splitting operation, it is possible that some clusters are created containing only events and/or facts which will never be requested for an hypothesis generation. These clusters will consume memory space, and will never be deleted.

The `MayBeRequiredPredicate` predicate should be provided to the library, via the `setRequiredPredicate` method of the `HypothesesManager` class. The predicate is used to determine if a cluster contains any events or facts which may be requested by the application, otherwise, the cluster will be deleted.

Code Example 1.5: Example of a MayBeRequiredPredicate.

```
hypothesesManager.setRequiredPredicate(new MayBeRequiredPredicate()
    {

    public boolean mayBeRequired(Event event) {
        return false;
    }

    public boolean mayBeRequired(Fact event) {
        return true;
    }
});
```

Consider the code example 1.5. If the application only defined one fact, only requires that fact and no event, it may set a `MayBeRequiredPredicate` as in the example. This predicate states that the application will never require any event, and may only require facts.

1.2.5 HypothesisGenerator

When asking for a new hypotheses generation, using the `generateHypotheses` method of the `HypothesesManager` class, the application must provide an `HypothesisGenerator` which will actually generate the hypotheses. The method `generate` accepts the provided events and facts, and returns the `GeneratedHypotheses` (created with the `Factory` instance).

1.3 The Watcher interface

Only three interfaces are left to reference in the MHL API. Two of them are the `GeneratedHypothesis` and the `GeneratedHypotheses`. These need no further attention, as they are empty interfaces, and instantiated with the `Factory`, which was already described.

The last interface which is left to mention is the `Watcher` interface. This interface allows the application to receive notifications on changes on the events and facts being maintained in the library, notifications of confirmed events (with probability 1, and no longer being kept in the library), and notifications of a change in the best hypothesis.

Code Example 1.6: A simple Watcher implementation.

```
public class SimpleWatcher implements Watcher {
    private HashSet<Event> events = new HashSet<Event>();
    private HashSet<Fact> facts = new HashSet<Fact>();

    public void newFact(Fact fact) { facts.add(fact); }
    public void removedFact(Fact fact) { facts.remove(fact); }

    public void newEvent(Event event) { events.add(event); }
    public void removedEvent(Event event) { events.remove(event); }

    public void confirmedEvent(Event event) { events.remove(event);
        }

    public void newFacts(Collection<Fact> facts) {
        for (Fact fact : facts)
            newFact(fact);}

    public void removedFacts(Collection<Fact> facts) {
        for (Fact fact : facts)
            removedFact(fact);}

    public void newEvents(Collection<Event> events) {
        for (Event event : events)
            newEvent(event);}

    public void removedEvents(Collection<Event> events) {
        for (Event event : events)
            removedEvent(event);}

    public void confirmedEvents(Collection<Event> events) {
        for (Event event : events)
            confirmedEvent(event);}

    public HashSet<Event> getEvents() { return events; }

    public HashSet<Fact> getFacts() { return facts; }

    public void bestHypothesis(Hypothesis hypothesis) {}
}
```

Code example 1.6 presents a simple `Watcher` implementation. This implementation does not save confirmed events to persistent storage, or any other kind of storage, and it does nothing with new best hypotheses. But it is sufficient to keep track of the events and facts saved by the library, which it keeps in two sets (one with all the events maintained by the library, and another one with the facts).

A mature `Watcher` implementation would save confirmed events to persistent storage, present the new best hypotheses to the user, and organize the events and facts in such a way that it speeds up searches.

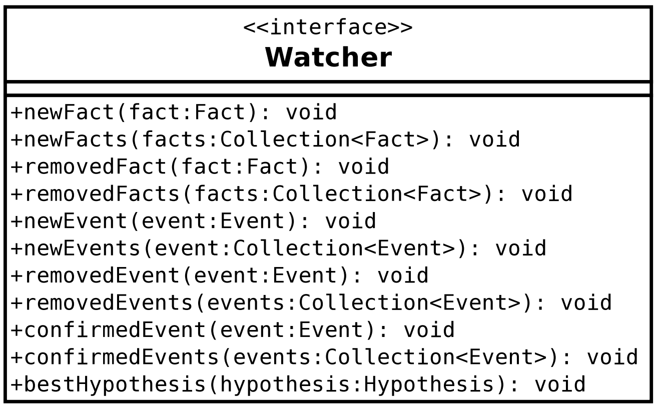
<<interface>>
Watcher
+newFact(fact:Fact): void
+newFacts(facts:Collection<Fact>): void
+removedFact(fact:Fact): void
+removedFacts(facts:Collection<Fact>): void
+newEvent(event:Event): void
+newEvents(event:Collection<Event>): void
+removedEvent(event:Event): void
+removedEvents(events:Collection<Event>): void
+confirmedEvent(event:Event): void
+confirmedEvents(events:Collection<Event>): void
+bestHypothesis(hypothesis:Hypothesis): void

Figure 1.3: The Watcher interface.

Chapter 2

Practical Example: Simple Tracker Application

Now a practical example of a very simple, yet operational, tracking application will be presented. The application will track a set of simulated targets moving around in a closed space. The tracker will receive the measurements of the positions of the targets (with some missing detections) and random false measurements simulating sensor noise.

For simplicity sake, only one fact will be used by the tracker. A complete application should include events to save the tracking history.

2.1 Classes

The application will be implemented with only five files:

Target Represents a simulated target.

Main Creates the display, the targets, the tracker, and generally conducts the overall simulation.

SimpleWatcher A `Watcher` implementation.

TargetFact A fact representing a target being tracked.

Tracker The class which uses the library to perform the actual tracking.

Each of these classes will now be described in detail.

2.1.1 The Simulated Target

The `Target` class will contain all the information necessary to the simulation of each target.

Code Example 2.1: The simulated target class.

```
public class Target {

    private double x, y, heading, velocity;
    public Target() {
        x = Math.random() * 400;
        y = Math.random() * 400;
        heading = Math.random() * 2d * PI;
        velocity = Math.random() * 5;
    }

    public void update() {
        heading += (Math.random() - 0.5) * PI * 0.4;
        velocity = min(5, max(0, velocity + Math.random() - 0.5));

        x = min(400, max(0, (x + velocity * cos(heading))));
        y = min(400, max(0, (y + velocity * sin(heading))));
    }

    public double getX() { return x; }
    public double getY() { return y; }
}
```

The target has a position, a velocity, and a heading. They are initialized with random values in the constructor. The `update` method introduces random changes to the target's heading and velocity, and updates the target position according to its velocity and heading.

2.1.2 Main class

The `main` method of the `Main` class starts with the creation of a `Tracker` instance, the simulated targets, and measurements lists.

Code Example 2.2: The main class part 1/3: initializations.

```
public static void main(String[] args) throws InterruptedException,
    InvocationTargetException {
    final Tracker tracker = new Tracker();
    final List<Target> targets = new ArrayList<Target>();
    for (int i = 0; i < 50; i++) {
        targets.add(new Target());
    }
    final List<Point2D> realMeasurements = new ArrayList<Point2D>();
    final List<Point2D> noise = new ArrayList<Point2D>();
```

Afterwards a `JFrame` is created, with 400px by 400px which will display the simulated targets and the predictions from the tracker.

Code Example 2.3: The main class part 2/3: graphic display.

```
    final JFrame frame = new JFrame() {

        @Override
        public void paint(Graphics g) {
            synchronized (targets) {
                g.clearRect(0, 0, 400, 400);
                g.setColor(Color.blue.darker());
                for (Target target : targets) {
                    g.fillRect((int) target.getX(), (int) target.
                        getY() - 5, 5, 10);
                }

                g.setColor(Color.red);
```

```
                    for (Fact fact : tracker.getBestHypothesis().
                        getFacts().keySet()) {
                        g.fillRect(
                                (int) ((TargetFact) fact).getX() - 5,
                                (int) ((TargetFact) fact).getY() - 5,
                                5, 10);
                    }
                }
            }
        };
        frame.setSize(400, 400);
        frame.setResizable(false);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
```

And in the end, the main loop. This loop continuously updates the simulated targets, creates the measurements (missing some detections and introducing false detections) and calls the appropriate method in the `Tracker` class. The measurements sent to the tracker are represented by 2D points, using the `Point2D` class which is already defined in the Java language.

Code Example 2.4: The main class part 3/3: main loop.

```
        while (true) {
            synchronized (targets) {
                for (Target target : targets) { target.update(); }

                realMeasurements.clear();
                for (Target t : targets) {
                    if (Math.random() < 0.95)
                        realMeasurements.add(new Point2D.Double(t.getX()
                            , t.getY())); }

                noise.clear();
                for (int i = 0; i < 80; i++) {
                    noise.add(new Point2D.Double(Math.random() * 400,
                        Math.random() * 400)); }

                List<Point2D> finalMeasurements = new ArrayList<Point2D
                    >();
                finalMeasurements.addAll(noise);
                finalMeasurements.addAll(realMeasurements);
                tracker.newScan(finalMeasurements);
            }

            EventQueue.invokeAndWait(new Runnable() {

                public void run() {
                    synchronized (targets) {
                        doubleFrame.repaint();
                    }
                }
            });
        }
```

2.1.3 SimpleWatcher class

The `SimpleWatcher` class is very similar to the one described in section 1.3, for this reason it won't be described again.

2.1.4 TargetFact class

The `TargetFact` represents the state of a particular target being tracked.

Code Example 2.5: The TargetFact class.

```
public class TargetFact implements Fact {

    private long id, lastDetection;
    private double x, y, velocityX, velocityY;

    // Constructor

    // Getters

    boolean measurementInGate(Point2D measurement) {
        if ((measurement.distance(x + velocityX, y + velocityY)) <
            7) {return true;}
        return false;
    }

    double measurementInGateProbability(Point2D measurement) {
        if (!measurementInGate(measurement)) { return 0; }
        double distance = (measurement.distance(x + velocityX, y +
            velocityY));
        return distance == 0d ? 1 : Math.min(1, (1d / distance));
    }
}
```

The target has several characteristics, as follows. A unique identifier. The last time of the last detection, which is used to delete the target if it is not detected for several consecutive scans. A (x, y) position, and a velocity.

The method `measurementInGate` returns true if a measurement is inside the gate of the target. The gate of the target is being modeled as a circular region of radius=7 around the target's expected position (assuming it maintains a constant velocity). The method `measurementInGateProbability` returns the probability of the measurement having its origin on the target (which is zero if the measurement falls outside the target's gate).

2.1.5 Tracker class

The tracker class is naturally the most complex class of the application. Thus, before detailing the class, a general view will be given.

Code Example 2.6: Overview of the Tracker class.

```
public class Tracker {

    private MHLService mhl;
    private Factory factory;
    private HypothesesManager hm;
    private SimpleWatcher watcher = new SimpleWatcher();
    private long time = 0;
    private long targetIdGen = 0;

    public Tracker() { /*...*/ }

    private boolean intersect(Set s1, Set s2) { /*...*/ }
    public Hypothesis getBestHypothesis() {
      return hm.getBestHypothesis(); }

    public void newScan(List<Point2D> measurements) { /*...*/ }

    private void createMeasurementGroups(
      List<Point2D> measurements,
      HashSet<Fact> isolatedTargets,
      Map<Set<Point2D>, Set<Fact>> map) { /*...*/ }
    private void generateHypothesesOnMeasurements(
      Map<Set<Point2D>,
      Set<Fact>> map) { /*...*/ }
    private void generateHypothesesForIsolatedTargets(
      HashSet<Fact> isolatedTargets) { /*...*/ }
}
```

The class has instances of the several basic classes needed to used the library, which where described in section 1.1. Additionally it has a variable representing time, which is incremented by 1 in each scan, and a `targetIdGen` variable which is incremented when new targets are created to generate new, unique, target identifiers.

The method `intersect` receives two sets as arguments and tests if they intersect. Its implementation is straightforward, and will not be detailed here. The implementation of the `getBestHypothesis` method is also very simple, and does not require explanation.

The three last methods are auxiliary methods of the `newScan` method, and will be detailed along with it.

The constructor

The constructor searches for an implementation of the `MHLService` and, using it, creates the `Factory` and the `HypothesesManager`. Also in the constructor the pruning strategy is set, as described in section 1.1.4, and the `SimpleWatcher` instance is registered in the `HypothesesManager`.

Code Example 2.7: Tracker class constructor.

```
public Tracker() {
    ServiceLoader<MHLService> serviceLoader = ServiceLoader.load(
        MHLService.class);
    mhl = serviceLoader.iterator().next();
    factory = mhl.getFactory();
    hm = factory.newHypothesesManager();

    PrunerFactory prunerFactory = factory.getPrunerFactory();
    hm.setPruner(prunerFactory.newCompositePruner(
            new Pruner[]{
                prunerFactory.newBestKPruner(6),
                prunerFactory.newTreeDepthPruner(5)}));

    hm.register(watcher);
}
```

The newScan method

Before describing the `newScan` method an explanation of the way in which it must process the measurements will be given.

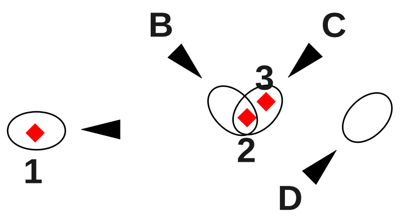

Figure 2.1: Three targets and two measurements example.

Measurement processing Consider the figure 2.1. There are three facts representing the targets A, B, C, and D in the library. They may be in different hypotheses and even in different clusters. The when generating hypotheses about the measurement which was fallen in the gate of target A (measurement 1), the fact associated with target A must be previously required from the library, so that it is provided to the application while generating hypotheses. The same for the measurement 2. But hypotheses for a measurement can only be generated once, otherwise it could result in some hypotheses which would assign the measurement to two targets, and other incorrect behavior. Thus, when generating hypothesis for measurement 2, both facts B and C must be required. But, as measurement 3 shares a target gate with measurement 2, so their hypothesis generation must be made together, otherwise it could result in some hypotheses which would assign the same target to different measurements, and other incorrect behavior. In summary, if a measurement falls in the gate of a target, it must be required from the library (in the method `generateHypotheses` of the `HypothesesGeneration` class). If two methods must require the same fact, their hypothesis generation must be made together.

Method newScan The body of the `newScan` method is the following:

Code Example 2.8: The newScan method.

```
public void newScan(List<Point2D> measurements) {
    time++;
    Map<Set<Point2D>, Set<Fact>> map = new HashMap<Set<Point2D>, Set
        <Fact>>();
    HashSet<Fact> isolatedTargets = new HashSet<Fact>(watcher.
        getFacts());

    createMeasurementGroups(measurements, isolatedTargets, map);

    generateHypothesesOnMeasurements(map);

    generateHypothesesForIsolatedTargets(isolatedTargets);
}
```

The variable `map` is a map from sets of measurements which must be processed together to the set of facts which must be required when processing them. The set `isolatedTargets` contains the targets on which gates no measurement has fallen.

Method createMeasurementGroups This method receives the `map` and the `isolatedTargets` variables, along with the `measurements` variable, and fills `map` and `isolatedTargets` according to what has been described before.

Code Example 2.9: The createMeasurementGroups method.

```
private void createMeasurementGroups(List<Point2D> measurements,
    HashSet<Fact> isolatedTargets, Map<Set<Point2D>, Set<Fact>> map)
     {

  for (Point2D measurement : measurements) {
    Set<Fact> reqFacts = new HashSet<Fact>();
    Set<Point2D> measurementGroup = new HashSet<Point2D>();
    measurementGroup.add(measurement);
    for (Fact f : watcher.getFacts()) {
      if (((TargetFact) f).measurementInGate(measurement)) {
        reqFacts.add(f);
        isolatedTargets.remove(f);
      }
    }
    map.put(measurementGroup, reqFacts);
  }
  outter:
  while (true) {
    for (Map.Entry<Set<Point2D>, Set<Fact>> entry1 : map.entrySet())
         {
      for (Map.Entry<Set<Point2D>, Set<Fact>> entry2 : map.entrySet
          ()) {
        if (!entry1.equals(entry2)) {
          if (intersect(entry1.getValue(), entry2.getValue())) {
            Set<Fact> reqFacts = new HashSet<Fact>();
            Set<Point2D> measurementGroup = new HashSet<Point2D>();
            reqFacts.addAll(entry1.getValue());
            reqFacts.addAll(entry2.getValue());
            measurementGroup.addAll(entry1.getKey());
            measurementGroup.addAll(entry2.getKey());
            map.put(measurementGroup, reqFacts);
            map.remove(entry1.getKey());
            map.remove(entry2.getKey());
            continue outter;
          }
        }
      }
    }
    break;
  }
}
```

Method generateHypothesesOnMeasurements This method generates new hypotheses for each set of measurements and corresponding set of facts in the `map` variable. It uses the murty algorithm to avoid generating high numbers of inconsequent, low probability, hypotheses.

Code Example 2.10: The generateHypothesesOnMeasurements method.

```
private void generateHypothesesOnMeasurements(Map<Set<Point2D>, Set<
    Fact>> map) {
  for (final Map.Entry<Set<Point2D>, Set<Fact>> measurementGroup :
      map.entrySet()) {
    hm.generateHypotheses(new HypothesesGenerator() {

      public GeneratedHypotheses generate(Set<Event> provEvents, Set
          <Fact> provFacts) {
        List<Point2D> measurements =
            new ArrayList<Point2D>(measurementGroup.getKey());
```

```
List<TargetFact> targets =
    new ArrayList<TargetFact>((Set<TargetFact>) (Object)
        provFacts);
double[][] costMatrix =
    new double[measurements.size()][targets.size() +
        measurements.size() * 2];
for (int i = 0; i < measurements.size(); i++) {
  for (int j = 0; j < targets.size(); j++) {
    costMatrix[i][j] = targets.get(j).
        measurementInGateProbability(measurements.get(i));
  }
  costMatrix[i][targets.size() + i] = 0.1;
  costMatrix[i][targets.size() + measurements.size() + i] =
      0.01;
}
MurtyAlgorithm.MurtyAlgorithmResult result = MurtyAlgorithm.
    solve(costMatrix, 10);
HashSet<GeneratedHypothesis> generatedHypothesesSet = new
    HashSet<GeneratedHypothesis>();
for (int solution = 0; solution < result.getCustomer2Item().
    length; solution++) {
  int[] assignment = result.getCustomer2Item()[solution];
  Set<Fact> newFacts = new HashSet<Fact>();
  double hypProb = 1;
  for (int m = 0; m < assignment.length; m++) {
    Point2D measurement = measurements.get(m);
    int item = assignment[m];
    hypProb *= costMatrix[m][item];
    if (item < targets.size()) {
      TargetFact target = targets.get(item);
      TargetFact targetFactUpdate =
          new TargetFact(
          target.getId(),
          time,
          measurement.getX(),
          measurement.getY(),
          measurement.getX() - target.getX(),
          measurement.getY() - target.getY());
      newFacts.add(targetFactUpdate);
    } else if (item < targets.size() + measurements.size())
        {
      // False Alarm
    } else {
      TargetFact newTargetFact =
          new TargetFact(
          targetIdGen++,
          time,
          measurement.getX(),
          measurement.getY(),
          0,
          0);
      newFacts.add(newTargetFact);
    }
  }
  int[] item2Customer = result.getItem2Customer()[solution];
  for (int i = 0; i < targets.size(); i++) {
    if (item2Customer[i] == -1) {
      if (time - targets.get(i).getLastDetection() > 2) {
        // Target removed (TargetFact not added to the
        // new facts).
      } else {
        TargetFact target = targets.get(i);
```

```
                  TargetFact targetFactUpdate =
                      new TargetFact(target.getId(),
                      time,
                      target.getX() + target.getVelocityX(),
                      target.getY() + target.getVelocityY(),
                      target.getVelocityX(),
                      target.getVelocityY());
                  newFacts.add(targetFactUpdate);
                }
                hypProb *= 0.001;
              }
            }
            generatedHypothesesSet.add(factory.newGeneratedHypothesis(
                hypProb, new HashSet<Event>(), newFacts));
          }
          GeneratedHypotheses generatedHypotheses = factory.
              newGeneratedHypotheses(generatedHypothesesSet);
          return generatedHypotheses;
        }
      }, new HashSet<Event>(), measurementGroup.getValue());
    }
  }
```

Method generateHypothesesForIsolatedTargets This method generates new hypotheses for the isolated targets. These targets are deleted if they remain undetected for several scans.

Code Example 2.11: The generateHypothesesForIsolatedTargets method.

```
private void generateHypothesesForIsolatedTargets(HashSet<Fact>
    isolatedTargets) {
  for (Fact fact : isolatedTargets) {
    HashSet<Fact> reqFacts = new HashSet<Fact>();
    reqFacts.add(fact);
    hm.generateHypotheses(new HypothesesGenerator() {

      public GeneratedHypotheses generate(Set<Event> provEvents, Set
          <Fact> provFacts) {
        HashSet<GeneratedHypothesis> generatedHypothesesSet = new
            HashSet<GeneratedHypothesis>();
        if (provFacts.isEmpty()) {
          generatedHypothesesSet.add(factory.newGeneratedHypothesis
              (1, new HashSet<Event>(), new HashSet<Fact>()));
          return factory.newGeneratedHypotheses(
              generatedHypothesesSet);
        } else {
          HashSet<Fact> newFacts = new HashSet<Fact>();
          TargetFact target = (TargetFact) provFacts.iterator().next
              ();
          TargetFact targetFactUpdate = new TargetFact(
                  target.getId(),
                  time,
                  target.getX() + target.getVelocityX(),
                  target.getY() + target.getVelocityY(),
                  target.getVelocityX(),
                  target.getVelocityY());
          if (time - target.getLastDetection() <= 2) {
            newFacts.add(targetFactUpdate);
          }
          generatedHypothesesSet.add(factory.newGeneratedHypothesis
              (0.01, new HashSet<Event>(), newFacts));
```

```
                return factory.newGeneratedHypotheses(
                    generatedHypothesesSet);
            }
          }
        }, new HashSet<Event>(), reqFacts);
      }
    }
```

Revisions

Number	**Date**	**Description**
1.0	Monday, May 02 2011	Initial revision.