```java
package org.seamcat.epp;

 * @author karl koch (adhoc@heiseka.de)

import java.util.ArrayList;

public class BlockingIntegral5 implements EventProcessingPlugin {

    private Map<String, List<Double>> iRSSblockingValue;
    private Map<String, List<Double>> iRSSblockingValueModified;
    private LinkedHashMap<String, List<Double>> iltFrequencies;
    private List<Double> vlrFrequencies;
    private List<DiscreteFunction> blockingMasksModified;
    private Function2D blockingMaskOrigin;
    private Function2D blockingAttenuationOrigin;
    private List<Distribution> iltFrequencyDistributions;
    private List<String> iltFrequencyDistributionTypes;
    private List<Double> iltBW;
    private List<Double> blockingModifiedSum;
    private LinkedHashMap<String, List<Double>> blockingMasksModifiedOffsets;
    private LinkedHashMap<String, List<Double>> blockingMasksModifiedValues;
    private ArrayList<Point2D> blockMask;
    private DiscreteFunction modified;

    @Override
    public List<ParameterDefinition> getParameterDefinitions() {
        List<ParameterDefinition> parameters = new ArrayList<ParameterDefinition>();
        /* parameters to be added here */
        return parameters;
    }

    @Override
    public String getDescription() {
        return "calculates for each type of ILT a frequency dependent blocking mask";
    }

    @Override
    public ResultTypes evaluate(Scenario scenario, Iterable<EventResult> results, List<Parameter> params) {
        List<SingleValueTypes<?>> resultTypes = new ArrayList<SingleValueTypes<?>>();
        List<VectorResultType> vectors = new ArrayList<VectorResultType>();
        ResultTypes types = new ResultTypes(resultTypes, vectors);

        initPlugin(scenario, params); // as an option

        /* define the output vectors */
        iRSSblockingValue = new LinkedHashMap<String, List<Double>>();
        iRSSblockingValueModified = new LinkedHashMap<String, List<Double>>();
        iltFrequencies = new LinkedHashMap<String, List<Double>>();
        vlrFrequencies = new ArrayList<Double>();
        blockingMasksModified = new ArrayList<DiscreteFunction>();
        blockingMasksModifiedOffsets = new LinkedHashMap<String, List<Double>>();
        blockingMasksModifiedValues = new LinkedHashMap<String, List<Double>>();
        iltBW =  new ArrayList<Double>();

        /* collect data */
        for (EventResult res : results) {
            int linkCount = 0;
            vlrFrequencies.add(scenario.getVictimSystem().getFrequency().trial());
            for (InterferenceLink link : scenario.getInterferenceLinks()) {
                double sumBlock = 0, block = 0;
                String linkDescription = "interferingLink: " + linkCount + " - ";
                InterferenceLinkResults linkResults = res.getInterferenceLinkResult(link);
                for (int i = 0; i < linkResults.getInterferenceLinkResults().size(); i++) {
                    InterferenceLinkResult interferenceLinkSubLinkResult = linkResults.getInterferenceLinkResults().get(i);
                    block += Math.pow(10, interferenceLinkSubLinkResult.getRiRSSBlockingValue() / 10.);
                }
                sumBlock = 10 * Math.log10(block);
                ensureName(linkDescription, iRSSblockingValue).add(sumBlock);
                ensureName(linkDescription, iltFrequencies).add(scenario.getInterferenceLinks().get(linkCount).getInterferingSystem().getFrequency().trial());
                double bw = scenario.getInterferenceLinks().get(linkCount).getInterferingSystem().getTransmitter().getBandwidth();
                iltBW.add(bw);
                linkCount++;
            }
        }

        /* not clear whether it waits for the end of the simulation, but it seems to do so */
        if (scenario.numberOfEvents() == vlrFrequencies.size()) {
            blockMask = new ArrayList<Point2D>();
            doBlockingCalculations(scenario, results);
            createMaskVectors();
        }

        /* store results */
        addResultVector(vectors, "iRSSblockingValue", "dBm", iRSSblockingValue);
        addResultVector(vectors, "iRSSblockingValueModified", "dBm", iRSSblockingValueModified);
        vectors.add(new VectorResultType("iRSSblockingValueModifiedSum", "dBm", blockingModifiedSum));
        addResultVector(vectors, "blockingMasksModifiedOffsets", "MHz", blockingMasksModifiedOffsets);
        addResultVector(vectors, "blockingMasksModifiedValues", "dB", blockingMasksModifiedValues);
        addResultVector(vectors, "iltFrequencies", "MHz", iltFrequencies);

        return types;
    }

    /**
     * in the absence of an appropriate output vector, this exports the offsets and values of the generated masks separately
     */
    private void createMaskVectors() {
        for (int i = 0; i < blockingMasksModified.size(); i++) {
            String linkDescription = "interferingLink: " + (i) + " - ";
            List<Point2D> points = blockingMasksModified.get(i).getPoints();
            for (int j = 0; j < points.size(); j++) {
                ensureName(linkDescription, blockingMasksModifiedOffsets).add(points.get(j).getX());
                ensureName(linkDescription, blockingMasksModifiedValues).add(points.get(j).getY());
            }
        }
    }

    /**
     * trial to start the additional methods after finishing the simulation
     * @param scenario
     * @param results
     */
    private void doBlockingCalculations(Scenario scenario, Iterable<EventResult> results) {
        /* frequency and ILT bandwidth distributions */
        Distribution vlrFrequencyDistribution = scenario.getVictimSystem().getFrequency();
        String vlrFrequencyDistributionType = scenario.getVictimSystem().getFrequency().toString();
        iltFrequencyDistributions = new ArrayList<Distribution>();
        iltFrequencyDistributionTypes = new ArrayList<String>();
        for (int i = 0; i < scenario.getInterferenceLinks().size(); i++) {
            iltFrequencyDistributions.add(scenario.getInterferenceLinks().get(i).getInterferingSystem().getFrequency());
            iltFrequencyDistributionTypes.add(scenario.getInterferenceLinks().get(i).getInterferingSystem().getFrequency().toString());

        }

        /* origin VLR blocking mask  and bandwidth*/
        blockingMaskOrigin = scenario.getVictimSystem().getReceiver().getBlockingMask();
        blockingAttenuationOrigin = getBlockingAttenuation(scenario, results, blockingMaskOrigin);
        double vlrBandwidth = scenario.getVictimSystem().getReceiver().getBandwidth();

        /* create frequency dependent masks */
        generateMasks(vlrFrequencyDistributionType, vlrFrequencyDistribution, iltFrequencyDistributionTypes, iltFrequencyDistributions, iltBW, vlrBandwidth);

        /* modify blocking results of the interfering links*/
        modifyBlockingResults(scenario);

        /* calculate sum of blocking */
        calculateSumBlockingModified(scenario);

    }

    /**
     *builds the sum of all blocking vectors
     * @param scenario
     */
    private void calculateSumBlockingModified(Scenario scenario) {
        blockingModifiedSum = new ArrayList<Double>();
        for (int i = 0; i < vlrFrequencies.size(); i++) {
            double sum = 0;
            for (int j = 0; j < scenario.getInterferenceLinks().size(); j++) {
                String linkDescription = "interferingLink: " + (j) + " - ";
                sum += Math.pow(10, iRSSblockingValueModified.get(linkDescription).get(i) / 10);
            }
            blockingModifiedSum.add(10 * Math.log10(sum));
        }
    }

    /**
     * applies the frequency dependent masks by considering the delta of
     * attenuation
     *
     * @param scenario
     */
    private void modifyBlockingResults(Scenario scenario) {
        for (int i = 0; i < scenario.getInterferenceLinks().size(); i++) {
            for (int j = 0; j < vlrFrequencies.size(); j++) {
                String linkDescription = "interferingLink: " + i + " - ";
                List<Double> blockOrigin = iRSSblockingValue.get(linkDescription);
                double offset = Math.abs(vlrFrequencies.get(j) - iltFrequencies.get(linkDescription).get(j));
                double attOrigin = blockingAttenuationOrigin.evaluate(offset);
                double attModified = blockingMasksModified.get(i).evaluate(offset);
                double blockingModified = blockOrigin.get(j) + (attOrigin - attModified);
                double blockingOrigin = blockOrigin.get(j);
                ensureName(linkDescription, iRSSblockingValueModified).add(blockingModified);
            }
        }
    }

    /**
```

```java
     * creates per interfering link a frequency dependent blocking mask
     *
     * @param vlrFrequencyDistributionType
     *
     * @param vlrFrequencyDistribution
     * @param iltFrequencyDistributionTypes2
     * @param iltFrequencyDistributions
     * @param iltBW
     * @param vlrBandwidth
     */
    private void generateMasks(String vlrFrequencyDistributionType, Distribution vlrFrequencyDistribution, List<String> iltFrequencyDistributionTypes,
            List<Distribution> iltFrequencyDistributions, List<Double> iltBW, double vlrBandwidth) {

        double vfMin, vfMax, ifMin, ifMax = 0;
        double lowestOffset = 0, highestOffset = 0;
        double minOffset = 0;

        /* define min and max frequencies of VL */
        if (vlrFrequencyDistributionType.contains("Constant")) {
            vfMax = vlrFrequencyDistribution.getConstant();
            vfMin = vfMax;
        } else {
            vfMax = vlrFrequencyDistribution.getMax();
            vfMin = vlrFrequencyDistribution.getMin();
        }

        /* do calculation for each interfering link */
        for (int i = 0; i < iltFrequencyDistributions.size(); i++) {
            modified = new DiscreteFunction();
            if (iltFrequencyDistributionTypes.get(i).contains("Constant")) {
                ifMax = iltFrequencyDistributions.get(i).getConstant();
                ifMin = ifMax;
            } else {
                ifMax = iltFrequencyDistributions.get(i).getMax();
                ifMin = iltFrequencyDistributions.get(i).getMin();
            }

            /* define range and step width of the masks */
            if (ifMin > vfMax) {
                lowestOffset = ifMin - vfMax;
                highestOffset = ifMax - vfMin;
            } else {
                lowestOffset = ifMax - vfMin;
                highestOffset = ifMin - vfMax;
            }

            if (lowestOffset > highestOffset) { // changes the borders of the range
                double rOff = lowestOffset;
                lowestOffset = highestOffset;
                highestOffset = rOff;
            }

            minOffset = iltBW.get(i) / 2 + vlrBandwidth / 2; // start point of the integral in MHz; otherwise it would be co-channel

            double checkCoChannel = lowestOffset;
            while (Math.abs(checkCoChannel - minOffset) < 0) { // to avoid misleading blocking values in case the scenario is co-channel
                Point2D rValue = new Point2D(checkCoChannel, 200);
                blockMask.add(rValue);
                checkCoChannel += vlrBandwidth / 2;
            }

            /* generate the mask values by integrating the power over the ILT bandwidth */
            for (double offset = checkCoChannel; offset <= highestOffset + vlrBandwidth; offset += vlrBandwidth) {
                double sum = 0, rAtt = 0, att = 0, refBW = vlrBandwidth / iltBW.get(i) / 2;
                for (double channel = offset - iltBW.get(i) / 2 + vlrBandwidth / 2; channel <= offset + iltBW.get(i) / 2 - vlrBandwidth / 2; channel += vlrBandwidth / 2) {
                    try {
                        rAtt = blockingAttenuationOrigin.evaluate(channel);
                        sum += Math.pow(10, -rAtt/10) * refBW;
                    } catch (Exception e) {
                        e.printStackTrace();
                    }

                }
                att = -10*Math.log10(sum);
                modified.addPoint(offset, att);
            }

            blockingMasksModified.add(modified);
        }

    }

    /**
     * converts the blocking mask of the VLR to user defined attenuation values
     * methods are inspired by the class BlockingInterference.java
     *
     * @param scenario
     * @param results
     * @param blockingMask
     * @return
     */
    private Function2D getBlockingAttenuation(Scenario scenario, Iterable<EventResult> results, Function2D blockingMask) {
        double offset = 0;
        if (((GenericReceiver) scenario.getVictimSystem().getReceiver()).getBlockingAttenuationMode() == GenericReceiver.BlockingAttenuationMode.PROTECTION_RATIO) {
            offset = vrAttenuationProcRatio(scenario);
        } else if (((GenericReceiver) scenario.getVictimSystem().getReceiver()).getBlockingAttenuationMode() == GenericReceiver.BlockingAttenuationMode.MODE_SENSITIVITY) {
            offset = vrAttenuationSens(scenario);
        }

        Function2D offsetFunction = blockingMask.offset(offset);
        return offsetFunction;
    }

    /**
     * offset if sensitivity is selected
     *
     * @param scenario
     * @return
     */
    private double vrAttenuationSens(Scenario scenario) {
        double rOffset, rCNI, rSens, rIN;
        rCNI = ((GenericReceiver) scenario.getVictimSystem().getReceiver()).getExtendedProtectionRatio();
        rSens = ((GenericReceiver) scenario.getVictimSystem().getReceiver()).getSensitivity();
        rIN = ((GenericReceiver) scenario.getVictimSystem().getReceiver()).getInterferenceToNoiseRatio();
        rOffset = rCNI - rSens - rIN;
        return rOffset;
    }

    /**
     * offset if protection ratio is selected
     *
     * @param scenario
     * @return
     */
    private double vrAttenuationProcRatio(Scenario scenario) {
        double rOffset, rCNI, rNIN, rIN;
        rCNI = ((GenericReceiver) scenario.getVictimSystem().getReceiver()).getExtendedProtectionRatio();
        rNIN = ((GenericReceiver) scenario.getVictimSystem().getReceiver()).getExtendedProtectionRatio();
        rIN = ((GenericReceiver) scenario.getVictimSystem().getReceiver()).getExtendedProtectionRatio();
        rOffset = rNIN + rCNI - rIN;
        return rOffset;
    }

    /**
     * personally preferred to configure the plugin
     *
     * @param scenario
     * @param params
     */
    private void initPlugin(Scenario scenario, List<Parameter> params) {
        /* declaration of parameters to be used globally */
    }

    @Override
    public String toString() {
        return "BlockingIntegral5"; //
    }

    /* taken from genericSystemParameters example */
    private void addResultVector(List<VectorResultType> vectors, String namePrefix, String unit, Map<String, List<Double>> positionILR_Y) {
        for (Map.Entry<String, List<Double>> entry : positionILR_Y.entrySet()) {
            vectors.add(new VectorResultType(entry.getKey() + namePrefix, unit, entry.getValue()));
        }
    }

    private List<Double> ensureName(String name, Map<String, List<Double>> container) {
        if (!container.containsKey(name)) {
            container.put(name, new ArrayList<Double>());
        }
        return container.get(name);
    }

}
```