Листинг на фрагменти от кода

# Даннов модел

## Стохастични променливи

**public interface StochasticVariable {**

public String getId();

public double sample();

public double[] sample(int sampleSize);

}

@XmlSeeAlso({ExponentialVariable.class, GammaVariable.class,

GaussianVariable.class, LogNormalVariable.class, UniformVariable.class})

@XmlAccessorType(XmlAccessType.FIELD)

**public abstract class AbstractVariable implements StochasticVariable {**

@XmlTransient

protected static final Logger log = LoggerFactory.getLogger(AbstractVariable.class);

@XmlTransient

protected static final AtomicLong idGenerator = new AtomicLong();

@XmlTransient

protected RandomGenerator random;

@XmlAttribute

protected String id;

public AbstractVariable() {

this("" + idGenerator.getAndIncrement());

}

public AbstractVariable(String id) {

this(id, System.currentTimeMillis());

}

public AbstractVariable(String id, long seed) {

this.id = id;

this.random = new JDKRandomGenerator();

random.setSeed(seed);

}

@Override

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public void setSeed(long seed) {

random.setSeed(seed);

}

@Override

public double[] sample(int sampleSize) {

double[] samples = new double[sampleSize];

for (int i = 0; i < sampleSize; i++) {

samples[i] = sample();

}

return samples;

}

@Override

public boolean equals(Object obj) {

if (obj == null) {

return false;

}

if (getClass() != obj.getClass()) {

return false;

}

final AbstractVariable other = (AbstractVariable) obj;

return Objects.equals(this.id, other.id);

}

@Override

public int hashCode() {

int hash = 3;

hash = 67 \* hash + Objects.hashCode(this.id);

return hash;

}

}

@XmlRootElement(name = "gaussian")

**public class GaussianVariable extends AbstractVariable {**

private static final double defaultMean = 0.5; // Default standard deviation set to a third of the mean

private static final double defaultStandardDeviation = 1.0 / 3.0;

private double mean;

private double deviation;

public GaussianVariable() {

super();

mean = defaultMean;

deviation = defaultStandardDeviation;

}

public GaussianVariable(String id) {

super(id);

mean = defaultMean;

deviation = defaultStandardDeviation;

}

public GaussianVariable(String id, long seed) {

this(id, seed, defaultMean, defaultStandardDeviation);

}

public GaussianVariable(String id, long seed, double mean, double deviation) {

super(id, seed);

this.mean = mean;

this.deviation = deviation;

}

public Double getMean() {

return mean;

}

public void setMean(Double mean) {

this.mean = mean;

}

public Double getDeviation() {

return deviation;

}

public void setDeviation(Double deviation) {

this.deviation = deviation;

}

@Override

public double sample() {

return random.nextGaussian() \* deviation + mean;

}

}

## Операционни възли

**public interface Node {**

String getId();

String getRole();

void setRole(String role);

String getDescription();

void setDescription(String description);

double getValue(SimulationContext context);

}

@XmlSeeAlso({AbstractUnaryNode.class, AbstractBinaryNode.class,

AbstractGroupNode.class, ConstantNode.class, VariableNode.class,})

@XmlAccessorType(XmlAccessType.NONE)

**public abstract class AbstractNode implements Node {**

protected static final Logger log = LoggerFactory.getLogger(AbstractNode.class);

private static final AtomicLong idGenerator = new AtomicLong();

@XmlAttribute(name = "nodeId")

private String id;

@XmlAttribute

private String role;

@XmlAttribute

private String description;

@XmlAttribute(name = "index")

private Integer nodeIndex;

public AbstractNode() {

this(null);

}

public AbstractNode(String description) {

id = String.valueOf(idGenerator.getAndIncrement());

this.description = description;

}

@Override

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

@Override

public String getRole() {

return role;

}

@Override

public void setRole(String role) {

this.role = role;

}

@Override

public String getDescription() {

return description;

}

@Override

public void setDescription(String description) {

this.description = description;

}

public Integer getNodeIndex() {

return nodeIndex;

}

public void setNodeIndex(Integer nodeIndex) {

this.nodeIndex = nodeIndex;

}

@Override

public double getValue(SimulationContext context) {

double value = calculate(context);

context.getValueLogger().logValue(this, value, context);

return value;

}

abstract protected double calculate(SimulationContext context);

}

@XmlRootElement(name = "constant")

**public class ConstantNode extends AbstractNode {**

@XmlAttribute

private double value;

public ConstantNode() {

}

public ConstantNode(double value) {

super();

this.value = value;

}

public ConstantNode(double value, String description) {

super(description);

this.value = value;

}

public double getValue() {

return value;

}

public void setValue(double value) {

this.value = value;

}

@Override

protected double calculate(SimulationContext context) {

return value;

}

}

@XmlRootElement(name = "variable")

**public class VariableNode extends AbstractNode {**

@XmlAttribute

private String name;

private int variableIndex;

public VariableNode() {

}

public VariableNode(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getVariableIndex() {

return variableIndex;

}

public void setVariableIndex(int index) {

this.variableIndex = index;

}

@Override

protected double calculate(SimulationContext context) {

return context.getVariableValue(variableIndex);

}

}

@XmlSeeAlso({AbsoluteNode.class, CosineNode.class, CotangentNode.class,

InvertNode.class, SineNode.class, TangentNode.class})

**public abstract class AbstractUnaryNode extends AbstractNode {**

@XmlAnyElement(lax = true)

protected Node argument;

public AbstractUnaryNode() {

}

public AbstractUnaryNode(Node argument) {

super();

this.argument = argument;

}

public Node getArgument() {

return argument;

}

public void setArgument(Node argument) {

this.argument = argument;

}

}

@XmlRootElement(name = "absolute")

**public class AbsoluteNode extends AbstractUnaryNode {**

public AbsoluteNode() {

}

public AbsoluteNode(Node argument) {

super(argument);

}

@Override

protected double calculate(SimulationContext context) {

return Math.abs(argument.getValue(context));

}

}

@XmlSeeAlso({DivisionNode.class, ExponentiationNode.class,

LogarithmNode.class, RootNode.class})

**public abstract class AbstractBinaryNode extends AbstractNode {**

@XmlElementWrapper(name = "arguments")

@XmlAnyElement(lax = true)

protected Node[] arguments;

public AbstractBinaryNode() {

super();

arguments = new Node[2];

}

public Node[] getArguments() {

return arguments;

}

public void setArguments(Node[] arguments) {

this.arguments = arguments;

}

}

@XmlRootElement(name = "logarithm")

**public class LogarithmNode extends AbstractBinaryNode {**

public LogarithmNode() {

super();

}

public LogarithmNode(AbstractNode argument, AbstractNode base) {

super();

arguments[0] = argument;

arguments[0].setRole("argument");

arguments[1] = base;

arguments[1].setRole("base");

}

public Node getArgument() {

return arguments[0];

}

public void setArgument(Node argument) {

arguments[0] = argument;

arguments[0].setRole("argument");

}

public Node getBase() {

return arguments[1];

}

public void setBase(Node base) {

arguments[1] = base;

arguments[1].setRole("base");

}

@Override

protected double calculate(SimulationContext context) {

return Math.log(getArgument().getValue(context)) / Math.log(getBase().getValue(context));

}

}@XmlSeeAlso({MaxNode.class, MinNode.class,

AverageNode.class, ProductNode.class, SumNode.class})

**public abstract class AbstractGroupNode extends AbstractNode implements GroupNode {**

@XmlAnyElement(lax = true)

protected List<Node> children;

public AbstractGroupNode() {

this(new ArrayList<>());

}

public AbstractGroupNode(List<Node> children) {

super();

this.children = children;

}

@Override

public void addChild(Node child) {

children.add(child);

}

public List<Node> getChildren() {

return children;

}

public void setChildren(List<Node> children) {

this.children = children;

}

}

@XmlRootElement(name = "average")

**public class AverageNode extends AbstractGroupNode {**

public AverageNode() {

super();

}

public AverageNode(List<Node> children) {

super(children);

}

@Override

protected double calculate(SimulationContext context) {

double sum = 0;

for (Node child : children) {

sum += child.getValue(context);

}

return sum / children.size();

}

}

## Симулационна конфигурация

@XmlAccessorType(XmlAccessType.FIELD)

**public class SimulationProperties {**

private String title;

private String description;

private Integer simulationRuns;

public SimulationProperties() {

}

public SimulationProperties(String title, String description, Integer simulationRuns) {

this.title = title;

this.description = description;

this.simulationRuns = simulationRuns;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

public Integer getSimulationRuns() {

return simulationRuns;

}

public void setSimulationRuns(Integer simulationRuns) {

this.simulationRuns = simulationRuns;

}

}

@XmlSeeAlso({AbstractVariable.class})

@XmlAccessorType(XmlAccessType.NONE)

**public class StochasticVariableRegistry {**

@XmlAnyElement(lax = true)

private List<StochasticVariable> variables;

public StochasticVariableRegistry() {

variables = new ArrayList<>();

}

public StochasticVariableRegistry(List<StochasticVariable> variables) {

this.variables = variables;

}

public void addVariable(StochasticVariable variable) {

variables.add(variable);

}

public List<StochasticVariable> getVariables() {

return variables;

}

public void setVariables(List<StochasticVariable> variables) {

this.variables = variables;

}

}

@XmlSeeAlso({AbstractNode.class})

@XmlAccessorType(XmlAccessType.FIELD)

**public abstract class SimulationConfiguration {**

// SimulationRequest extends SimulationConfiguration without adding functionality for JAXB purposes

private SimulationProperties properties;

private StochasticVariableRegistry variables;

@XmlAnyElement(lax = true)

@XmlElementWrapper(name = "formula")

private final Node[] formula;

public SimulationConfiguration() {

formula = new Node[1];

}

public SimulationConfiguration(SimulationProperties properties, StochasticVariableRegistry variables, Node formula) {

this.properties = properties;

this.variables = variables;

this.formula = new Node[1];

this.formula[0] = formula;

}

public SimulationProperties getProperties() {

return properties;

}

public void setProperties(SimulationProperties properties) {

this.properties = properties;

}

public void setVariableRegistry(StochasticVariableRegistry variables) {

this.variables = variables;

}

public StochasticVariableRegistry getVariableRegistry() {

return variables;

}

public void setFormula(Node formula) {

this.formula[0] = formula;

}

public Node getFormula() {

return formula[0];

}

}

@XmlRootElement(name = "simulation")

@XmlAccessorType(XmlAccessType.FIELD)

**public class SimulationResponse extends SimulationConfiguration {**

@XmlElementWrapper(name = "values")

@XmlElement(name = "node")

private List<NodeValues> values;

public SimulationResponse() {

super();

}

public SimulationResponse(SimulationRequest request, SimulationManager manager) {

this(

request.getProperties(),

request.getVariableRegistry(),

request.getFormula(),

manager

);

}

public SimulationResponse(SimulationProperties configuration,

StochasticVariableRegistry variables, Node formula, SimulationManager manager) {

super(configuration, variables, formula);

values = new ArrayList<>();

double[][] valueRegistry = manager.getValueRegistry();

for (Map.Entry<Integer, Node> entry : manager.getNodeIndex().entrySet()) {

values.add(

new NodeValues(

entry.getValue().getId(),

valueRegistry[entry.getKey()]

)

);

}

}

public List<NodeValues> getValues() {

return values;

}

public void setValues(List<NodeValues> values) {

this.values = values;

}

}

# Сървърен модул

## Реализация на променливите

**public class SampledVariableRegistry {**

private final Map<Integer, StochasticVariable> variableIndex;

private final double[][] sampleRegistry;

public SampledVariableRegistry(StochasticVariableRegistry variableRegistry, int runs) {

List<StochasticVariable> variables = variableRegistry.getVariables();

variableIndex = new HashMap<>();

sampleRegistry = new double[variables.size()][];

for (int i = 0; i < variables.size(); i++) {

StochasticVariable current = variables.get(i);

variableIndex.put(i, current);

sampleRegistry[i] = current.sample(runs);

}

}

public Map<Integer, StochasticVariable> getVariableIndex() {

return variableIndex;

}

public double getVariableValue(int index, int run) {

return sampleRegistry[index][run];

}

}

## Обхождане на изчислителните дървета

**public interface NodeWalker {**

void walk(Node node);

}

**public interface NodeHandler<AbstractNode extends Node> {**

void handle(AbstractNode node);

}

**public class ReflectiveNodeWalker<Handler extends NodeHandler> implements NodeWalker {**

protected static final Logger log = LoggerFactory.getLogger(ReflectiveNodeWalker.class);

private final Handler handler;

public ReflectiveNodeWalker(Handler handler) {

this.handler = handler;

}

public Handler getHandler() {

return handler;

}

@Override

public void walk(Node node) {

handler.handle(node);

getChildNodes(node).forEach((k, v) -> walk(v));

}

private Map<String, Node> getChildNodes(Node node) {

Map<String, Node> children = new LinkedHashMap<>();

List<Field> fields = getAllFields(node.getClass());

for (Field field : fields) {

try {

field.setAccessible(true);

Object object = field.get(node);

if (object == null) {

continue;

}

if (object instanceof Iterable) {

markCollectionForWalking(children, field.getName(), (Iterable) object);

} else if (object.getClass().isArray()) {

markArrayForWalking(children, field.getName(), (Object[]) object);

} else {

markItemForWalking(children, field.getName(), field.get(node));

}

field.setAccessible(false);

} catch (IllegalArgumentException | IllegalAccessException ex) {

log.error("Exception while marking children of "

+ node.getClass().getSimpleName(), ex);

}

}

return children;

}

private static List<Field> getAllFields(Class<?> type) {

List<Field> fields = new ArrayList<>();

for (Class<?> c = type; c != null; c = c.getSuperclass()) {

fields.addAll(Arrays.asList(c.getDeclaredFields()));

}

return fields;

}

private void markItemForWalking(Map<String, Node> pending,

String name, Object fieldValue) {

if (fieldValue instanceof Node) {

pending.put(name, (Node) fieldValue);

}

}

private void markArrayForWalking(Map<String, Node> pending, String name, Object[] object) {

markCollectionForWalking(pending, name, Arrays.asList(object));

}

private void markCollectionForWalking(Map<String, Node> pending,

String name, Iterable collection) {

int count = 0;

for (Object item : collection) {

if (item instanceof Node) {

pending.put(name + "[" + count + "]", (Node) item);

count++;

}

}

}

}

**public class NodeCountingHandler implements NodeHandler {**

private Integer count;

public NodeCountingHandler() {

count = 0;

}

public Integer getNodeCount() {

return count;

}

@Override

public void handle(Node node) {

count++;

}

}

**public class NodeIndexResolvingHandler implements NodeHandler<AbstractNode> {**

private final Map<Integer, Node> registry;

private int nextIndex;

public NodeIndexResolvingHandler() {

registry = new HashMap<>();

nextIndex = 0;

}

public Map<Integer, Node> getNodeRegistry() {

return registry;

}

@Override

public void handle(AbstractNode node) {

int currentIndex = nextIndex++;

registry.put(currentIndex, node);

node.setNodeIndex(currentIndex);

}

}

**public class VariableIndexResolvingHandler implements NodeHandler<AbstractNode> {**

private static final Logger log = LoggerFactory.getLogger(VariableIndexResolvingHandler.class);

Map<Integer, StochasticVariable> variableIndex;

public VariableIndexResolvingHandler(SampledVariableRegistry registry) {

variableIndex = registry.getVariableIndex();

}

@Override

public void handle(AbstractNode node) {

if (VariableNode.class.isAssignableFrom(node.getClass())) {

VariableNode variableNode = (VariableNode) node;

String variableName = variableNode.getName();

for (Map.Entry<Integer, StochasticVariable> entry : variableIndex.entrySet()) {

if (entry.getValue().getId().equals(variableName)) {

variableNode.setVariableIndex(entry.getKey());

return;

}

}

log.error("Unable to resolve index of variable node");

}

}

}

## Симулационен контекст

**public interface SimulationContext {**

int getRunNumber();

double getVariableValue(int index);

ValueLogger getValueLogger();

}

**public class SimulationContextImpl implements SimulationContext {**

private final SampledVariableRegistry registry;

private final ValueLogger logger;

private int runNumber;

public SimulationContextImpl(SampledVariableRegistry registry, ValueLogger logger) {

this.registry = registry;

this.logger = logger;

}

public SimulationContextImpl(SimulationContextImpl context) {

this.registry = context.getRegistry();

this.logger = context.getValueLogger();

}

public void setRunNumber(int runNumber) {

this.runNumber = runNumber;

}

@Override

public int getRunNumber() {

return runNumber;

}

@Override

public double getVariableValue(int index) {

return registry.getVariableValue(index, runNumber);

}

@Override

public ValueLogger getValueLogger() {

return logger;

}

public SampledVariableRegistry getRegistry() {

return registry;

}

}

## Съхранение на резултати

**public interface ValueLogger<AbstractNode extends Node> {**

void logValue(AbstractNode node, Double value, SimulationContext context);

}

**public class CompositeValueLogger implements ValueLogger {**

private final Map<Class, ValueLogger> loggers;

public CompositeValueLogger() {

loggers = new HashMap<>();

}

public void putValueLogger(ValueLogger logger) {

loggers.put(logger.getClass(), logger);

}

public ValueLogger getValueLogger(Class clazz) {

return loggers.get(clazz);

}

@Override

public void logValue(Node node, Double value, SimulationContext context) {

loggers.forEach((k, v) -> {

v.logValue(node, value, context);

});

}

}

**public class MatrixValueLogger implements ValueLogger<AbstractNode> {**

private final Map<Integer, Node> nodeIndex;

private final double[][] valueRegistry;

public MatrixValueLogger(Node root, int runs) {

nodeIndex = TreeUtilities.resolveNodeIndices(root);

valueRegistry = new double[nodeIndex.size()][runs];

}

public Map<Integer, Node> getNodeIndex() {

return nodeIndex;

}

public double[][] getValueRegistry() {

return valueRegistry;

}

@Override

public void logValue(AbstractNode node, Double value, SimulationContext context) {

valueRegistry[node.getNodeIndex()][context.getRunNumber()] = value;

}

}

## Симулационни мениджъри

**public interface SimulationCompletionListener {**

void notify(SimulationManager manager);

}

**public class CompositeCompletionListener implements SimulationCompletionListener {**

private final Map<Class, SimulationCompletionListener> listeners;

public CompositeCompletionListener() {

listeners = new HashMap<>();

}

public void putCompletionListener(SimulationCompletionListener listener) {

listeners.put(listener.getClass(), listener);

}

public SimulationCompletionListener getCompletionListener(Class clazz) {

return listeners.get(clazz);

}

@Override

public void notify(SimulationManager manager) {

listeners.forEach((k, v) -> {

v.notify(manager);

});

}

}

**public class LatchLoweringCompletionListener implements SimulationCompletionListener {**

private final CountDownLatch latch;

public LatchLoweringCompletionListener(CountDownLatch latch) {

this.latch = latch;

}

@Override

public void notify(SimulationManager manager) {

latch.countDown();

}

}

**public interface SimulationManager extends Runnable {**

public Node getRoot();

public Map<Integer, Node> getNodeIndex();

public double[][] getValueRegistry();

public void await();

}

**public abstract class AbstractSimulationManager implements SimulationManager {**

protected static final Logger log = LoggerFactory.getLogger(AbstractSimulationManager.class);

protected final Node root;

protected final MatrixValueLogger valueRegistry;

protected final SimulationContextImpl context;

protected final Pair<Integer, Integer> runs;

protected SimulationCompletionListener completionListener;

protected CountDownLatch progressLatch;

protected AbstractSimulationManager(Node root, StochasticVariableRegistry variables, int runs) {

this.root = root;

this.valueRegistry = new MatrixValueLogger(root, runs);

SampledVariableRegistry sampledVariableRegistry = new SampledVariableRegistry(variables, runs);

TreeUtilities.resolveVariableNodeIndices(root, sampledVariableRegistry);

this.context = new SimulationContextImpl(sampledVariableRegistry, valueRegistry);

this.runs = new Pair<>(0, runs - 1);

}

protected AbstractSimulationManager(Node root, MatrixValueLogger valueRegistry, SimulationContextImpl context, Pair<Integer, Integer> runs) {

this.root = root;

this.valueRegistry = valueRegistry;

this.context = context;

this.runs = runs;

}

@Override

public Node getRoot() {

return root;

}

@Override

public void await() {

run();

try {

progressLatch.await();

} catch (InterruptedException ex) {

log.error("Simulation interrupted", ex);

}

}

public void setCompletionListener(SimulationCompletionListener listener) {

this.completionListener = listener;

}

@Override

public Map<Integer, Node> getNodeIndex() {

return valueRegistry.getNodeIndex();

}

@Override

public double[][] getValueRegistry() {

return valueRegistry.getValueRegistry();

}

}

**public class SingleThreadSimulationManager extends AbstractSimulationManager {**

public SingleThreadSimulationManager(Node root, StochasticVariableRegistry variables, int runs) {

super(root, variables, runs);

}

public SingleThreadSimulationManager(

Node root,

MatrixValueLogger valueRegistry,

SimulationContextImpl context,

Pair<Integer, Integer> runs) {

super(root, valueRegistry, context, runs);

}

@Override

public void run() {

// Start a simulation only if one has not been started yet

if (progressLatch == null) {

progressLatch = new CountDownLatch(1);

for (int i = runs.getKey(); i <= runs.getValue(); i++) {

context.setRunNumber(i);

root.getValue(context);

}

progressLatch.countDown();

if (completionListener != null) {

completionListener.notify(this);

}

}

}

}

**public class ParallelSimulationManager extends AbstractSimulationManager {**

public static List<Pair<Integer, Integer>> calculateSimulationRanges(int runs, int poolSize) {

List<Pair<Integer, Integer>> ranges = new ArrayList<>();

int defaultSimulationRuns = runs / poolSize;

int hardworkingThreads = runs % poolSize; // Some of the threads need to do one additional simulation

int workerRangeStart = 0;

for (int i = 0; i < poolSize; i++) {

int workerSimulationRuns

= defaultSimulationRuns + (i < hardworkingThreads ? 1 : 0);

ranges.add(new Pair<>(workerRangeStart, workerRangeStart + workerSimulationRuns - 1));

workerRangeStart += workerSimulationRuns;

}

return ranges;

}

public static int calculateThreadPoolSize(int totalWorkload, int threadWorkload) {

int poolSize;

for (poolSize = 1; poolSize \* threadWorkload < totalWorkload; poolSize++) {

// Increase size of thread pool until it can cope with the workload

}

return poolSize;

}

private final int threadLoad;

public ParallelSimulationManager(Node root, StochasticVariableRegistry registry, int runs, int threadLoad) {

super(root, registry, runs);

this.threadLoad = threadLoad;

}

@Override

public void run() {

// Start a simulation only if one has not been started yet

if (progressLatch == null) {

int runCount = runs.getValue() - runs.getKey() + 1;

int treeSize = TreeUtilities.getTreeSize(root);

int totalWorkload = runCount \* treeSize;

int poolSize = calculateThreadPoolSize(totalWorkload, threadLoad);

progressLatch = new CountDownLatch(poolSize);

CompositeCompletionListener compositeCompletionListener

= new CompositeCompletionListener();

compositeCompletionListener

.putCompletionListener(

new LatchLoweringCompletionListener(progressLatch)

);

if (completionListener != null) {

compositeCompletionListener

.putCompletionListener(completionListener);

}

log.info(String.format("Size:%d\tRuns:%d\tThreads:%d", treeSize, runCount, poolSize));

for (Pair<Integer, Integer> simulationRange

: calculateSimulationRanges(runCount, poolSize)) {

SingleThreadSimulationManager worker = new SingleThreadSimulationManager(

root, valueRegistry, new SimulationContextImpl(context), simulationRange);

worker.setCompletionListener(compositeCompletionListener);

Thread thread = new Thread(worker);

thread.start();

}

}

}

}

## Симулационна web услуга

@WebService

@SOAPBinding(style = Style.RPC)

**public interface SimulationService {**

@WebMethod

SimulationResponse simulate(SimulationRequest request);

}

@WebService(endpointInterface = "simulation.service.SimulationService")

**public class SimulationServiceImpl implements SimulationService {**

private static final Logger logger = LoggerFactory.getLogger(SimulationServiceImpl.class);

private static final int threadLoad = 100000 \* 1000 / 4;

@Override

public SimulationResponse simulate(SimulationRequest request) {

logger.info("Simulation request \"" + request.getProperties().getTitle() + "\" received");

long start = System.currentTimeMillis();

Node root = request.getFormula();

StochasticVariableRegistry variables = request.getVariableRegistry();

int runs = request.getProperties().getSimulationRuns();

SimulationManager simulationManager = new ParallelSimulationManager(root, variables, runs, threadLoad);

simulationManager.run();

simulationManager.await();

SimulationResponse response = new SimulationResponse(request, simulationManager);

logger.info("Simulation of \"" + request.getProperties().getTitle()

+ "\" took " + (System.currentTimeMillis() - start) + " ms");

return response;

}

}

**public class SimulationServicePublisher {**

private static final Logger logger = LoggerFactory.getLogger(SimulationServicePublisher.class);

private static final SimulationServicePublisher instance;

public String url;

private final Endpoint endpoint;

public static SimulationServicePublisher getInstance() {

return instance;

}

static {

instance = new SimulationServicePublisher();

instance.setPort(9999);

}

SimulationServicePublisher() {

endpoint = Endpoint.create(new SimulationServiceImpl());

}

public SimulationServicePublisher setPort(int port) {

url = "http://localhost:" + port + "/SimulationService";

return this;

}

public void publish() {

endpoint.stop();

endpoint.publish(url);

logger.info("Simulation service started on address " + url);

}

public void stop() {

endpoint.stop();

logger.info("Simulation service stopped");

}

}

**public class StartService {**

public static void main(String[] args) {

SimulationServicePublisher.getInstance().setPort(9999).publish();

}

}

# Клиентски модул

## Възлова статистика

**public class NodeStatistics {**

private final String id;

private final double[] values;

private final double mean;

private double min;

private double max;

public NodeStatistics(String id, double[] values) {

this.id = id;

this.values = values;

min = values[0];

max = values[0];

double sum = 0;

for (double value : values) {

sum += value;

if (value < min) {

min = value;

}

if (value > max) {

max = value;

}

}

mean = sum / values.length;

}

public String getId() {

return id;

}

public double getMean() {

return mean;

}

public double getMin() {

return min;

}

public double getMax() {

return max;

}

public double[] getValues() {

return values;

}

public double getValue(int run) {

return values[run];

}

}

## Дебъгван възел

**public class DebuggedNode {**

private final Node node;

private boolean valueVisible;

public DebuggedNode(Node node) {

this.node = node;

valueVisible = false;

}

public Node getNode() {

return node;

}

public boolean isValueVisible() {

return valueVisible;

}

public void setValueVisible(boolean valueVisible) {

this.valueVisible = valueVisible;

}

}

## Адаптиране на калкулационните възли

**public class JTreeBuilder {**

private static final JTreeBuilder INSTANCE = new JTreeBuilder();

private static final Logger log = LoggerFactory.getLogger(JTreeBuilder.class);

public static TreeModel buildTreeModel(Node node) {

return new DefaultTreeModel(INSTANCE.walk(node));

}

public DefaultMutableTreeNode walk(Node node) {

DefaultMutableTreeNode treeNode = new DefaultMutableTreeNode(new DebuggedNode(node));

getChildNodes(node).forEach((k, v) -> treeNode.add(walk(v)));

return treeNode;

}

private static List<Field> getAllFields(Class<?> type) {

List<Field> fields = new ArrayList<>();

for (Class<?> c = type; c != null; c = c.getSuperclass()) {

fields.addAll(Arrays.asList(c.getDeclaredFields()));

}

return fields;

}

private Map<String, Node> getChildNodes(Node node) {

Map<String, Node> children = new LinkedHashMap<>();

List<Field> fields = getAllFields(node.getClass());

for (Field field : fields) {

try {

field.setAccessible(true);

Object object = field.get(node);

if (object == null) {

continue;

}

if (object instanceof Iterable) {

markCollectionForWalking(children, field.getName(), (Iterable) object);

} else if (object.getClass().isArray()) {

markArrayForWalking(children, field.getName(), (Object[]) object);

} else {

markItemForWalking(children, field.getName(), field.get(node));

}

field.setAccessible(false);

} catch (IllegalArgumentException | IllegalAccessException ex) {

log.error("Exception while marking children of "

+ node.getClass().getSimpleName(), ex);

}

}

return children;

}

private void markItemForWalking(Map<String, Node> pending,

String name, Object fieldValue) {

if (fieldValue instanceof Node) {

pending.put(name, (Node) fieldValue);

}

}

private void markArrayForWalking(Map<String, Node> pending, String name, Object[] object) {

markCollectionForWalking(pending, name, Arrays.asList(object));

}

private void markCollectionForWalking(Map<String, Node> pending,

String name, Iterable collection) {

int count = 0;

for (Object item : collection) {

if (item instanceof Node) {

pending.put(name + "[" + count + "]", (Node) item);

count++;

}

}

}

}

## Контекст на дебъгване

**public class DebugContext {**

private int currentRun;

private int runCount;

private NodeStatistics selectedNode;

private DefaultMutableTreeNode currentlyDebuggedNode;

private SimulationResponse currentSimulation;

private DefaultMutableTreeNode root;

private final Map<String, NodeStatistics> statistics;

private final Map<String, DefaultMutableTreeNode> nodes;

public DebugContext() {

statistics = new HashMap<>();

nodes = new HashMap<>();

currentRun = 1;

}

public DebugContext(SimulationResponse response) {

this();

setup(response);

}

public final void setup(SimulationResponse response) {

currentSimulation = response;

setupTree(response.getFormula());

setupStatistics(response.getValues());

}

public SimulationResponse getCurrentSimulation() {

return currentSimulation;

}

public DefaultMutableTreeNode getRoot() {

return root;

}

public int getCurrentRun() {

return currentRun;

}

public void setCurrentRun(int currentRun) {

if (!statistics.isEmpty()

&& currentRun > 0

&& currentRun <= runCount) {

this.currentRun = currentRun;

resetDebugging();

}

}

public int getRunCount() {

return runCount;

}

public NodeStatistics getSelectedNodeStatistics() {

return selectedNode;

}

public void setSelectedNode(String nodeId) {

selectedNode = statistics.get(nodeId);

}

public DefaultMutableTreeNode getCurrentlyDebuggedNode() {

return currentlyDebuggedNode;

}

public void setCurrentlyDebuggedNode(DefaultMutableTreeNode currentlyDebugged) {

currentlyDebuggedNode = currentlyDebugged;

}

public Double getValue(String nodeId) {

NodeStatistics nodeStats = statistics.get(nodeId);

return nodeStats == null ? null : nodeStats.getValue(currentRun - 1);

}

public void resetDebugging() {

currentlyDebuggedNode = root;

resetDebugging(root);

}

private void resetDebugging(DefaultMutableTreeNode current) {

DebuggedNode node = (DebuggedNode) current.getUserObject();

node.setValueVisible(false);

Enumeration children = current.children();

while (children.hasMoreElements()) {

resetDebugging((DefaultMutableTreeNode) children.nextElement());

}

}

private void setupStatistics(List<NodeValues> nodeValuesList) {

statistics.clear();

selectedNode = null;

if (nodeValuesList == null || nodeValuesList.isEmpty()) {

statistics.clear();

runCount = 0;

return;

}

for (NodeValues nodeValues : nodeValuesList) {

statistics.put(nodeValues.getNodeId(),

new NodeStatistics(

nodeValues.getNodeId(),

nodeValues.getValues()));

}

currentRun = 1;

NodeValues v = nodeValuesList.get(0);

if (v != null) {

runCount = v.getValues().length;

} else {

runCount = 0;

}

}

private void setupTree(Node formula) {

root = (DefaultMutableTreeNode) JTreeBuilder.buildTreeModel(formula).getRoot();

nodes.clear();

currentlyDebuggedNode = root;

mapNodes(root);

}

private void mapNodes(DefaultMutableTreeNode current) {

Node node = ((DebuggedNode) current.getUserObject()).getNode();

nodes.put(node.getId(), current);

Enumeration children = current.children();

while (children.hasMoreElements()) {

mapNodes((DefaultMutableTreeNode) children.nextElement());

}

}

}

## Изглед за възлова статистика

**public class StatisticsView extends JPanel {**

private static final Dimension preferredSize = new Dimension(400, 600);

private static final Dimension preferredFieldSize = new Dimension(100, 30);

private static final Dimension preferredChartSize = new Dimension(400, 220);

private final DebugContext context;

private final JTextField maxField;

private final JTextField minField;

private final JTextField meanField;

private final XYSeries series;

private final JFreeChart valueChart;

private final JPanel histogramPanel;

public StatisticsView(DebugContext context) {

this.context = context;

this.setLayout(new BorderLayout(30, 30));

this.setBackground(Styles.defaultPanelBackgroundColor);

this.setPreferredSize(preferredSize);

this.setBorder(Styles.padding);

minField = new JTextField();

maxField = new JTextField();

meanField = new JTextField();

JPanel fieldPanel = new JPanel();

fieldPanel.setBackground(Styles.defaultPanelBackgroundColor);

fieldPanel.setLayout(new BoxLayout(fieldPanel, BoxLayout.X\_AXIS));

fieldPanel.add(setupFieldPanel(minField, "Min"));

fieldPanel.add(Box.createHorizontalStrut(Styles.strut));

fieldPanel.add(setupFieldPanel(maxField, "Max"));

fieldPanel.add(Box.createHorizontalStrut(Styles.strut));

fieldPanel.add(setupFieldPanel(meanField, "Mean"));

this.add(packTitleAndFields(fieldPanel), BorderLayout.NORTH);

this.add(Box.createVerticalStrut(Styles.strut));

series = new XYSeries("Values");

valueChart = ChartFactory.createXYLineChart("Values",

null, null, new XYSeriesCollection(series),

PlotOrientation.VERTICAL, false, false, false);

XYPlot plot = (XYPlot) valueChart.getPlot();

plot.setBackgroundPaint(Styles.chartBackgroundColor);

plot.getDomainAxis().setStandardTickUnits(NumberAxis.createIntegerTickUnits());

ChartPanel chartPanel = new ChartPanel(valueChart);

chartPanel.setPreferredSize(preferredChartSize);

this.add(chartPanel, BorderLayout.CENTER);

histogramPanel = new JPanel();

histogramPanel.setPreferredSize(preferredChartSize);

histogramPanel.setBackground(Styles.defaultPanelBackgroundColor);

this.add(histogramPanel, BorderLayout.SOUTH);

update();

}

private JPanel packTitleAndFields(JPanel fieldPanel) {

JPanel titleFieldPanel = new JPanel();

titleFieldPanel.setLayout(new FlowLayout(FlowLayout.LEFT));

titleFieldPanel.setPreferredSize(new Dimension(400, 100));

titleFieldPanel.setBackground(Styles.defaultPanelBackgroundColor);

JLabel title = new JLabel("Selected node statistics");

title.setFont(Styles.titleFont);

this.add(title, BorderLayout.NORTH);

titleFieldPanel.add(title);

titleFieldPanel.add(Box.createVerticalStrut(Styles.strut));

titleFieldPanel.add(fieldPanel);

return titleFieldPanel;

}

private JPanel setupFieldPanel(JTextField field, String labelText) {

JPanel panel = new JPanel();

panel.setBackground(Styles.defaultPanelBackgroundColor);

panel.setLayout(new BoxLayout(panel, BoxLayout.Y\_AXIS));

JLabel label = new JLabel(labelText);

label.setFont(Styles.labelFont);

panel.add(label);

field.setFont(Styles.labelFont);

field.setEditable(false);

field.setBackground(Styles.chartBackgroundColor);

field.setPreferredSize(preferredFieldSize);

panel.add(field);

return panel;

}

public void update() {

NodeStatistics statistics = context.getSelectedNodeStatistics();

if (statistics == null) {

this.setVisible(false);

return;

}

this.setVisible(true);

minField.setText(String.format(Styles.valueFormat, statistics.getMin()));

maxField.setText(String.format(Styles.valueFormat, statistics.getMax()));

meanField.setText(String.format(Styles.valueFormat, statistics.getMean()));

updateNodeValues();

recreateHistogramChartPanel();

}

private void updateNodeValues() {

series.clear();

NodeStatistics stats = context.getSelectedNodeStatistics();

double[] values = stats.getValues();

for (int i = 0; i < values.length; i++) {

series.add(i + 1, values[i]);

}

updateRangeAxis();

}

private void updateRangeAxis() {

XYPlot plot = (XYPlot) valueChart.getPlot();

plot.getDomainAxis().setRange(0, context.getRunCount());

Range dataRange = plot.getDataRange(plot.getRangeAxis());

if (dataRange.getLowerBound() == dataRange.getUpperBound()) {

plot.getRangeAxis().setRange(

dataRange.getLowerBound() - 1,

dataRange.getUpperBound() + 1);

} else {

plot.getRangeAxis().setRange(dataRange);

}

}

private void recreateHistogramChartPanel() {

histogramPanel.removeAll();

NodeStatistics stats = context.getSelectedNodeStatistics();

HistogramDataset dataset = new HistogramDataset();

dataset.addSeries(

"Distribution",

stats == null ? new double[1] : stats.getValues(),

15);

JFreeChart histogram = ChartFactory.createHistogram("Distribution",

null, null, dataset, PlotOrientation.VERTICAL, false, false, false);

histogram.getPlot().setBackgroundPaint(Styles.chartBackgroundColor);

ChartPanel panel = new ChartPanel(histogram);

panel.setPreferredSize(preferredChartSize);

histogramPanel.add(panel);

histogramPanel.revalidate();

}

}

## Изглед за дебъгване

**public class AbstractNodeRenderer implements TreeCellRenderer {**

protected DebugContext context;

private final JLabel valueLabel;

private final JLabel roleLabel;

private final JLabel descriptionLabel;

public AbstractNodeRenderer(DebugContext context) {

this.context = context;

valueLabel = makeLabel("=", Styles.labelFont, Styles.labelColor);

roleLabel = makeLabel("role =", Styles.labelFont, Styles.labelColor);

descriptionLabel = makeLabel("description =", Styles.labelFont, Styles.labelColor);

}

protected final JLabel makeLabel(String text, Font font, Color color) {

JLabel label = new JLabel(text);

if (font != null) {

label.setFont(font);

}

if (color != null) {

label.setForeground(color);

}

return label;

}

private String getNodeName(Node node) {

String className = node.getClass().getSimpleName();

return className.trim().replace("Node", "");

}

private JLabel getName(Node node) {

return makeLabel(getNodeName(node), Styles.valueFont, Styles.valueColor);

}

private JLabel getValue(Node node) {

String id = node.getId();

Double value = context.getValue(id);

if (value == null) {

return null;

}

return makeLabel(String.format(Styles.valueFormat, value), Styles.valueFont, Styles.valueColor);

}

private JLabel getRole(Node node) {

String role = node.getRole();

if (role == null || role.equals("")) {

return null;

}

return makeLabel(role, Styles.valueFont, Styles.valueColor);

}

private JLabel getDescription(Node node) {

String description = node.getDescription();

if (description == null || description.equals("")) {

return null;

}

return makeLabel(description, Styles.valueFont, Styles.valueColor);

}

private void fillPanel(JPanel panel, DebuggedNode debuggedNode) {

Node node = debuggedNode.getNode();

panel.add(getName(node));

if (debuggedNode.isValueVisible() == true) {

JLabel value = getValue(node);

if (value != null) {

panel.add(valueLabel);

panel.add(value);

panel.add(Box.createHorizontalStrut(Styles.strut));

}

}

JLabel role = getRole(node);

if (role != null) {

panel.add(roleLabel);

panel.add(role);

panel.add(Box.createHorizontalStrut(Styles.strut));

}

JLabel description = getDescription(node);

if (description != null) {

panel.add(descriptionLabel);

panel.add(description);

panel.add(Box.createHorizontalStrut(Styles.strut));

}

}

@Override

public JPanel getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus) {

DefaultMutableTreeNode treeNode = (DefaultMutableTreeNode) value;

DebuggedNode node = (DebuggedNode) treeNode.getUserObject();

JPanel panel = new JPanel();

fillPanel(panel, node);

if (selected) {

panel.setBackground(Styles.selectedBackgroundColor);

} else {

panel.setBackground(Styles.defaultNodeBackgroundColor);

}

DefaultMutableTreeNode currentlyDebuggedTreeNode = context.getCurrentlyDebuggedNode();

if (currentlyDebuggedTreeNode != null) {

String currentlyDebuggedNodeId

= ((DebuggedNode) currentlyDebuggedTreeNode.getUserObject())

.getNode()

.getId();

if (node.getNode().getId().equals(currentlyDebuggedNodeId)) {

panel.setBackground(Styles.currentlyDebuggedBackgroundColor);

}

}

return panel;

}

}

**public class VariableNodeRenderer extends AbstractNodeRenderer {**

public VariableNodeRenderer(DebugContext context) {

super(context);

}

@Override

public JPanel getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus) {

JPanel result = super.getTreeCellRendererComponent(tree, value, selected, expanded, leaf, row, hasFocus);

DefaultMutableTreeNode treeNode = (DefaultMutableTreeNode) value;

DebuggedNode node = (DebuggedNode) treeNode.getUserObject();

VariableNode variable = (VariableNode) node.getNode();

result.add(makeLabel(variable.getName(), Styles.valueFont, Styles.valueColor), 1);

return result;

}

}

**public class NodeRendererResolver implements TreeCellRenderer {**

private final Map<Class, AbstractNodeRenderer> renderers;

private final AbstractNodeRenderer abstractNodeRenderer;

private final TreeCellRenderer defaultNodeRenderer;

public NodeRendererResolver(DebugContext context) {

abstractNodeRenderer = new AbstractNodeRenderer(context);

defaultNodeRenderer = new DefaultTreeCellRenderer();

renderers = new HashMap<>();

renderers.put(VariableNode.class, new VariableNodeRenderer(context));

}

private TreeCellRenderer getRenderer(Object object) {

DebuggedNode node = (DebuggedNode) object;

TreeCellRenderer renderer = renderers.get(node.getNode().getClass());

if (renderer != null) {

return renderer;

}

if (DebuggedNode.class.isAssignableFrom(object.getClass())) {

return abstractNodeRenderer;

}

return defaultNodeRenderer;

}

@Override

public Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus) {

return getRenderer(((DefaultMutableTreeNode) value).getUserObject())

.getTreeCellRendererComponent(tree, value, selected, expanded, leaf, row, hasFocus);

}

}

**public abstract class AbstractDebugController implements ActionListener {**

protected final DebugContext context;

protected final DebuggingView panel;

public AbstractDebugController(DebugContext context, DebuggingView panel) {

this.context = context;

this.panel = panel;

}

protected void debugChildren(DefaultMutableTreeNode node) {

((DebuggedNode) (node.getUserObject())).setValueVisible(true);

Enumeration children = node.children();

while (children.hasMoreElements()) {

debugChildren((DefaultMutableTreeNode) children.nextElement());

}

}

abstract void hook();

abstract DefaultMutableTreeNode handle();

@Override

public void actionPerformed(ActionEvent e) {

DefaultMutableTreeNode changed = handle();

if (changed != null) {

after(changed);

}

}

private void after(DefaultMutableTreeNode changedBranch) {

JTree tree = panel.getTree();

List<Integer> expandedRows = new ArrayList<>();

for (int i = 0; i < tree.getRowCount(); i++) {

if (tree.isExpanded(i)) {

expandedRows.add(i);

}

}

TreePath selectionPath = tree.getSelectionPath();

((DefaultTreeModel) tree.getModel()).nodeStructureChanged(changedBranch);

tree.setSelectionPath(selectionPath);

for (int i = 0; i < expandedRows.size(); i++) {

tree.expandRow(expandedRows.get(i));

}

if (context.getCurrentlyDebuggedNode() != null) {

tree.scrollPathToVisible(new TreePath(context.getCurrentlyDebuggedNode().getPath()));

}

}

}

**public class StepIntoDebugController extends AbstractDebugController {**

public StepIntoDebugController(DebugContext context, DebuggingView panel) {

super(context, panel);

hook();

}

@Override

final void hook() {

panel.setStepIntoButtonListener(this);

}

@Override

public DefaultMutableTreeNode handle() {

DefaultMutableTreeNode node = (DefaultMutableTreeNode) context

.getCurrentlyDebuggedNode();

if (node != null && node.getChildCount() > 0) {

panel.getTree().expandPath(new TreePath(node.getPath()));

context.setCurrentlyDebuggedNode(

(DefaultMutableTreeNode) node.getFirstChild());

}

return node;

}

}

**public class StepOverDebugController extends AbstractDebugController {**

public StepOverDebugController(DebugContext context, DebuggingView panel) {

super(context, panel);

hook();

}

@Override

final void hook() {

panel.setStepOverButtonListener(this);

}

@Override

public DefaultMutableTreeNode handle() {

DefaultMutableTreeNode node = context.getCurrentlyDebuggedNode();

if (node != null) {

debugChildren(node);

DefaultMutableTreeNode next = node.getNextSibling();

if (next != null) {

context.setCurrentlyDebuggedNode(next);

} else {

context.setCurrentlyDebuggedNode((DefaultMutableTreeNode) node.getParent());

}

}

return node;

}

}

**public class StepOutDebugController extends AbstractDebugController {**

public StepOutDebugController(DebugContext context, DebuggingView panel) {

super(context, panel);

hook();

}

@Override

final void hook() {

panel.setStepOutButtonListener(this);

}

@Override

public DefaultMutableTreeNode handle() {

DefaultMutableTreeNode node = context.getCurrentlyDebuggedNode();

if (node != null) {

DefaultMutableTreeNode parent = (DefaultMutableTreeNode) node.getParent();

context.setCurrentlyDebuggedNode((DefaultMutableTreeNode) node.getParent());

if (parent == null) {

parent = node;

}

do {

debugChildren(node);

node = node.getNextSibling();

} while (node != null);

return (DefaultMutableTreeNode) parent;

}

return null;

}

}

**public class ResetDebugController extends AbstractDebugController {**

public ResetDebugController(DebugContext context, DebuggingView panel) {

super(context, panel);

hook();

}

@Override

final void hook() {

panel.setResetButtonListener(this);

}

@Override

public DefaultMutableTreeNode handle() {

context.resetDebugging();

return context.getRoot();

}

}

**public class NodeSelectionController implements TreeSelectionListener {**

private final DebugContext context;

private final StatisticsView panel;

public NodeSelectionController(DebugContext context, StatisticsView panel) {

this.context = context;

this.panel = panel;

}

@Override

public void valueChanged(TreeSelectionEvent e) {

JTree tree = (JTree) e.getSource();

DefaultMutableTreeNode node = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();

if (node != null) {

DebuggedNode debuggedNode = (DebuggedNode) node.getUserObject();

if (debuggedNode != null) {

String selectedNodeId = debuggedNode.getNode().getId();

context.setSelectedNode(selectedNodeId);

if (panel != null) {

panel.update();

panel.repaint();

}

}

}

}

}

**public class DebuggingView extends JPanel {**

private static final Dimension preferredSize = new Dimension(600, 600);

private final JLabel title;

private final JButton resetButton;

private final JButton stepOverButton;

private final JButton stepIntoButton;

private final JButton stepOutButton;

private final JTree tree;

public DebuggingView(DebugContext context) {

this.setLayout(new BoxLayout(this, BoxLayout.Y\_AXIS));

this.setBorder(Styles.padding);

this.setBackground(Styles.defaultPanelBackgroundColor);

JPanel heading = new JPanel(new BorderLayout());

heading.setBackground(Styles.defaultPanelBackgroundColor);

title = new JLabel("Debugging run 1");

title.setFont(Styles.titleFont);

heading.add(title, BorderLayout.WEST);

JPanel controlPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));

controlPanel.setBackground(Styles.defaultPanelBackgroundColor);

stepIntoButton = new JButton("Step into");// (F5)

stepIntoButton.setFont(Styles.labelFont);

controlPanel.add(stepIntoButton);

stepOverButton = new JButton("Step over");// (F6)

stepOverButton.setFont(Styles.labelFont);

controlPanel.add(stepOverButton);

stepOutButton = new JButton("Step out");// (F7)

stepOutButton.setFont(Styles.labelFont);

controlPanel.add(stepOutButton);

resetButton = new JButton("Reset");

resetButton.setFont(Styles.labelFont);

controlPanel.add(resetButton);

heading.add(controlPanel, BorderLayout.EAST);

this.add(heading);

tree = new JTree(context.getRoot());

tree.setCellRenderer(new NodeRendererResolver(context));

JTreeUtils.expandAllNodes(tree);

JScrollPane scrollPane = new JScrollPane(tree);

scrollPane.setPreferredSize(preferredSize);

this.add(scrollPane);

}

public void updateTitle(int run) {

title.setText("Debugging run " + run);

}

public JTree getTree() {

return tree;

}

public final void setTreeModel(TreeModel model) {

tree.setModel(model);

JTreeUtils.expandAllNodes(tree);

}

public void setTreeSelectionListener(TreeSelectionListener listener) {

tree.addTreeSelectionListener(listener);

}

public void setResetButtonListener(ActionListener listener) {

clearButtonListeners(resetButton);

resetButton.addActionListener(listener);

}

public void setStepOverButtonListener(ActionListener listener) {

clearButtonListeners(stepOverButton);

stepOverButton.addActionListener(listener);

}

public void setStepIntoButtonListener(ActionListener listener) {

clearButtonListeners(stepIntoButton);

stepIntoButton.addActionListener(listener);

}

public void setStepOutButtonListener(ActionListener listener) {

clearButtonListeners(stepOutButton);

stepOutButton.addActionListener(listener);

}

private void clearButtonListeners(JButton button) {

ActionListener[] actionListeners = button.getActionListeners();

for (ActionListener listener : actionListeners) {

button.removeActionListener(listener);

}

}

public void reset() {

resetButton.doClick();

}

}

## Изглед за избор на симулационен цикъл

**public class RunSelectionController implements ListSelectionListener {**

private final DebugContext context;

private final DebuggingView panel;

public RunSelectionController(DebugContext context, DebuggingView panel) {

this.context = context;

this.panel = panel;

}

@Override

public void valueChanged(ListSelectionEvent e) {

JList list = (JList) e.getSource();

context.setCurrentRun(list.getSelectedIndex() + 1);

context.resetDebugging();

if (panel != null) {

panel.updateTitle(context.getCurrentRun());

panel.reset();

}

}

}

**public class RunSelectionView extends JPanel {**

private static final Dimension preferredSize = new Dimension(150, 600);

private final DebugContext context;

private final JLabel title;

private final JList<Integer> runList;

public RunSelectionView(DebugContext context) {

this.context = context;

this.setLayout(new BorderLayout());

this.setBackground(Styles.defaultPanelBackgroundColor);

this.setPreferredSize(preferredSize);

this.setBorder(Styles.padding);

title = new JLabel("Select run");

title.setFont(Styles.titleFont);

this.add(title, BorderLayout.CENTER);

runList = new JList<>();

JScrollPane scrollPane = new JScrollPane(runList);

scrollPane.setPreferredSize(new Dimension(100, 600));

updateRunList();

this.add(scrollPane, BorderLayout.SOUTH);

}

public final void updateRunList() {

Integer[] runs = new Integer[context.getRunCount()];

if (runs.length == 0) {

this.setVisible(false);

return;

}

this.setVisible(true);

for (int i = 0; i < runs.length; i++) {

runs[i] = i + 1;

}

runList.setListData(runs);

runList.setSelectedIndex(0);

}

public void setListSelectionListener(ListSelectionListener listener) {

runList.addListSelectionListener(listener);

}

}

## Основно меню

**public class JaxbUtils {**

protected static final Logger log = LoggerFactory.getLogger(JaxbUtils.class);

private static final JaxbUtils instance = new JaxbUtils();

public static JaxbUtils getInstance() {

return instance;

}

private Unmarshaller unmarshaller;

private Marshaller marshaller;

JaxbUtils() {

try {

JAXBContext context = JAXBContext.newInstance(SimulationResponse.class);

unmarshaller = context.createUnmarshaller();

marshaller = context.createMarshaller();

marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);

} catch (JAXBException ex) {

log.error("JAXB context creation error", ex);

}

}

public SimulationResponse unmarshal(String url) {

try {

return (SimulationResponse) unmarshaller.unmarshal(new File(url));

} catch (JAXBException ex) {

return null;

}

}

public void marshal(SimulationResponse response, String url) {

try {

marshaller.marshal(response, new File(url));

} catch (JAXBException ex) {

}

}

}

**public class OpenController implements ActionListener {**

private final SimulationFrame frame;

private final JFileChooser fileChooser;

public OpenController(SimulationFrame frame) {

this.frame = frame;

fileChooser = new JFileChooser("src/main/resources/");

}

@Override

public void actionPerformed(ActionEvent e) {

if (fileChooser.showDialog(frame, "Select simulation") == JFileChooser.APPROVE\_OPTION) {

File file = fileChooser.getSelectedFile();

SimulationResponse unboxed

= JaxbUtils

.getInstance()

.unmarshal(file.getPath());

if (unboxed != null) {

frame.getMenuView().setFilename(file.getPath());

DebugContext context = frame.getDebugContext();

context.setup(unboxed);

frame.getRunSelectionView().updateRunList();

frame.getDebuggingView().setTreeModel(new DefaultTreeModel(context.getRoot()));

frame.getStatisticsView().update();

} else {

JOptionPane.showMessageDialog(frame, "Selected file is not a simulation");

}

}

}

}

**public class SaveController implements ActionListener {**

private final SimulationFrame frame;

private final JFileChooser fileChooser;

public SaveController(SimulationFrame frame) {

this.frame = frame;

fileChooser = new JFileChooser("src/main/resources/");

}

@Override

public void actionPerformed(ActionEvent e) {

if (frame.getDebugContext().getCurrentSimulation() != null) {

if (fileChooser.showDialog(frame, "Select file to save the current simulation")

== JFileChooser.APPROVE\_OPTION) {

File file = fileChooser.getSelectedFile();

JaxbUtils

.getInstance()

.marshal(

frame.getDebugContext().getCurrentSimulation(),

file.getPath());

frame.getMenuView().setFilename(file.getPath());

}

}

}

}

**public class SimulationServiceClient {**

private static final Logger logger = LoggerFactory.getLogger(SimulationServiceClient.class);

private static final String defaultHost = "localhost";

private static final String defaultPort = "9999";

private SimulationService simulationService;

private String host;

private String port;

public SimulationServiceClient() {

this(defaultHost, defaultPort);

}

public SimulationServiceClient(String host, String port) {

this.host = host;

this.port = port;

setup();

}

private void setup() {

try {

URL url = new URL("http://" + host + ":" + port + "/SimulationService?wsdl");

QName qname = new QName("http://service.simulation/", "SimulationServiceImplService");

Service service = Service.create(url, qname);

simulationService = service.getPort(SimulationService.class);

} catch (Exception ex) {

logger.info("Unable to connect to service");

simulationService = null;

}

}

public boolean isConnected() {

return simulationService != null;

}

public void reconnect() {

try {

setup();

} catch (Exception ex) {

// Nothing meaningful to do here

}

}

public SimulationResponse simulate(SimulationRequest request) {

if (simulationService != null) {

return simulationService.simulate(request);

}

return null;

}

public SimulationResponse simulate(SimulationResponse response) {

if (simulationService != null) {

SimulationRequest request = new SimulationRequest(

response.getProperties(),

response.getVariableRegistry(),

response.getFormula());

return simulationService.simulate(request);

}

return null;

}

}

**public class SimulateController implements ActionListener {**

private final SimulationServiceClient service;

private final SimulationFrame frame;

public SimulateController(SimulationFrame frame) {

this.frame = frame;

service = new SimulationServiceClient();

}

@Override

public void actionPerformed(ActionEvent e) {

if(!service.isConnected()){

service.reconnect();

}

SimulationResponse currentSimulation = frame.getDebugContext().getCurrentSimulation();

if (currentSimulation != null && service.isConnected()) {

SimulationResponse response = service.simulate(currentSimulation);

frame.getMenuView().setFilename(response.getProperties().getTitle());

DebugContext context = frame.getDebugContext();

context.setup(response);

frame.getRunSelectionView().updateRunList();

frame.getDebuggingView().setTreeModel(new DefaultTreeModel(context.getRoot()));

frame.getStatisticsView().update();

}

}

}

**public class MenuView extends JPanel {**

private static final Dimension preferredSize = new Dimension(1200, 40);

private static final Dimension preferredFilenameSize = new Dimension(770, 30);

private static final int strut = 5;

private final JTextField filename;

private final JButton open;

private final JButton simulate;

private final JButton save;

public MenuView() {

this.setLayout(new FlowLayout(FlowLayout.LEFT));

this.setPreferredSize(preferredSize);

this.setBackground(Styles.defaultPanelBackgroundColor);

open = new JButton("Open");

open.setFont(Styles.labelFont);

this.add(open);

this.add(Box.createHorizontalStrut(strut));

simulate = new JButton("Simulate");

simulate.setFont(Styles.labelFont);

this.add(simulate);

this.add(Box.createHorizontalStrut(strut));

save = new JButton("Save");

save.setFont(Styles.labelFont);

this.add(save);

this.add(Box.createHorizontalStrut(strut));

JLabel label = new JLabel("Loaded configuration:");

label.setFont(Styles.labelFont);

this.add(label);

this.add(Box.createHorizontalStrut(strut));

filename = new JTextField("");

filename.setEditable(false);

filename.setFont(Styles.labelFont);

filename.setPreferredSize(preferredFilenameSize);

this.add(filename);

}

public void setFilename(String filename) {

this.filename.setText(filename);

}

public void setOpenButtonListener(ActionListener listener) {

open.addActionListener(listener);

}

public void setSimulateButtonListener(ActionListener listener) {

simulate.addActionListener(listener);

}

public void setSaveButtonListener(ActionListener listener) {

save.addActionListener(listener);

}

}

## Симулационна рамка

**public class SimulationFrame extends JFrame {**

private final DebugContext context;

private final MenuView menuView;

private final RunSelectionView runSelectionView;

private final DebuggingView debuggingView;

private final StatisticsView statisticsView;

public SimulationFrame(String title, DebugContext context,

MenuView menuView,

RunSelectionView runSelectionView,

DebuggingView debuggingView,

StatisticsView statisticsView) {

this.menuView = menuView;

this.runSelectionView = runSelectionView;

this.debuggingView = debuggingView;

this.statisticsView = statisticsView;

this.setLayout(new BorderLayout(20, 20));

this.add(menuView, BorderLayout.NORTH);

this.add(runSelectionView, BorderLayout.WEST);

this.add(debuggingView, BorderLayout.CENTER);

this.add(statisticsView, BorderLayout.EAST);

this.context = context;

this.setTitle(title);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.pack();

this.setVisible(true);

this.repaint();

this.setResizable(false);

// Center frame on screen

Dimension dimension = Toolkit.getDefaultToolkit().getScreenSize();

int x = (int) ((dimension.getWidth() - this.getWidth()) / 2);

int y = (int) ((dimension.getHeight() - this.getHeight()) / 2);

this.setLocation(x, y);

}

public MenuView getMenuView() {

return menuView;

}

public RunSelectionView getRunSelectionView() {

return runSelectionView;

}

public DebuggingView getDebuggingView() {

return debuggingView;

}

public StatisticsView getStatisticsView() {

return statisticsView;

}

public DebugContext getDebugContext() {

return context;

}

}

**public class ViewFactory {**

private static final ViewFactory instance = new ViewFactory();

private ViewFactory() {

}

public static ViewFactory getInstance() {

return instance;

}

public RunSelectionView makeRunSelectionView(DebugContext context, DebuggingView debuggingView) {

RunSelectionView panel = new RunSelectionView(context);

panel.setListSelectionListener(new RunSelectionController(context, debuggingView));

return panel;

}

public DebuggingView makeDebuggingView(DebugContext context, StatisticsView statisticsPanel) {

DebuggingView panel = new DebuggingView(context);

panel.setTreeSelectionListener(new NodeSelectionController(context, statisticsPanel));

panel.setResetButtonListener(new ResetDebugController(context, panel));

panel.setStepIntoButtonListener(new StepIntoDebugController(context, panel));

panel.setStepOutButtonListener(new StepOutDebugController(context, panel));

panel.setStepOverButtonListener(new StepOverDebugController(context, panel));

return panel;

}

public StatisticsView makeStatisticsView(DebugContext context) {

return new StatisticsView(context);

}

public MenuView makeMenuView(DebugContext context) {

MenuView panel = new MenuView();

return panel;

}

public SimulationFrame makeSimulationFrame(String title, DebugContext context) {

MenuView mainMenu = makeMenuView(context);

StatisticsView nodeStatisticsPanel = makeStatisticsView(context);

DebuggingView debugTreePanel = makeDebuggingView(context, nodeStatisticsPanel);

RunSelectionView runSelectorPanel = makeRunSelectionView(context, debugTreePanel);

SimulationFrame simulationFrame = new SimulationFrame(

title, context, mainMenu, runSelectorPanel, debugTreePanel, nodeStatisticsPanel);

mainMenu.setOpenButtonListener(new OpenController(simulationFrame));

mainMenu.setSimulateButtonListener(new SimulateController(simulationFrame));

mainMenu.setSaveButtonListener(new SaveController(simulationFrame));

return simulationFrame;

}

}