Содержание

Перечень сокращений

API – Application Programming Interface (Интерфейс Программирования Приложений);

DPI – Dots per Inch (количество точек на дюйм);

GUI – Graphic User Interface (Графический Интерфейс Пользователя);

IDE – Integrated Development Environment (Интегрированная Среда Разработки);

JVM – Java Virtual Machine (Виртуальная Машина Java);

ИМ – Имитационное Моделирование;

СДС – Сложная Дискретная Система.

Терминология

Зуммирование – увеличение/уменьшение размеров отображаемого на экране объекта.

Мьютекс – объект, ограничивающий одновременный доступ потоков к определенному участку кода, используется для защиты общих для нескольких потоков данных.

Парсер – (от англ. parser; «parse» — анализ, разбор) часть программы, преобразующая входные данные в структурированный формат, т. е. выполняет их синтаксический анализ.

Парсить – выполнять синтаксический анализ данных для преобразования их в требуемый формат выходных данных.

Плагин – независимо компилируемый программный модуль, динамически подключаемый к основной программе и предназначенный для расширения и/или использования её возможностей.

Поток – наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.

Процесс – совокупность кода и данных, разделяющих общее виртуальное адресное пространство.

Семафор – объект, ограничивающий число потоков, которые могут войти в заданный участок кода.

Сериализация – процесс перевода какой-либо структуры данных в последовательность битов.

Слушатель – механизм, позволяющий экземпляру какого-либо класса получать оповещения от других объектов об изменении их состояния и задавать алгоритм реакции системы на это изменение.

Трассировка – получение информационных сообщений о работе приложения во время выполнения.

Флаг – логическая переменная, принимающая значение истины или лжи и характеризующая состояние какого-либо объекта или части системы.

1. Введение

Имитационное моделирование (ИМ)[1] на ЭВМ находит широкое применение при исследовании и управлении сложными дискретными системами (СДС) и процессами, в них протекающими. К таким системам можно отнести экономические и производственные объекты, морские порты, аэропорты, комплексы перекачки нефти и газа, ирригационные системы, программное обеспечение сложных систем управления, вычислительные сети и многие другие. Широкое использование ИМ объясняется тем, что размерность решаемых задач и неформализуемость сложных систем не позволяют использовать строгие методы оптимизации. Эти классы задач определяются тем, что при их решении необходимо одновременно учитывать факторы неопределенности, динамическую взаимную обусловленность текущих решений и последующих событий, комплексную взаимозависимость между управляемыми переменными исследуемой системы, а часто и строго дискретную и четко определенную последовательность интервалов времени. Указанные особенности свойственны всем сложным системам.

Проведение имитационного эксперимента позволяет:

1. Сделать выводы о поведении СДС и ее особенностях:
   - Без ее построения, если это проектируемая система;
   - Без вмешательства в ее функционирование, если это действующая система, проведение экспериментов над которой или слишком дорого, или небезопасно;
   - Без ее разрушения, если цель эксперимента состоит в определении пределов воздействия на систему.
2. Синтезировать и исследовать стратегии управления.
3. Прогнозировать и планировать функционирование системы в будущем.
4. Обучать и тренировать управленческий персонал и т.д.

ИМ является эффективным, но и не лишенным недостатков, методом. Трудности использования ИМ, связаны с обеспечением адекватности описания системы, интерпретацией результатов, обеспечением стохастической сходимости процесса моделирования, решением проблемы размерности и т.п. К проблемам применения ИМ следует отнести также и большую трудоемкость данного метода.

Интеллектуальное ИМ, характеризующиеся возможностью использования методов искусственного интеллекта и прежде всего знаний, при принятии решений в процессе имитации, при управлении имитационным экспериментом, при реализации интерфейса пользователя, создании информационных банков ИМ, использовании нечетких данных, снимает часть проблем использования ИМ.

Разработка интеллектуальной среды имитационного моделирования РДО выполнена в Московском государственном техническом университете (МГТУ им. Н. Э. Баумана) на кафедре "Компьютерные системы автоматизации производства". Причинами ее проведения и создания РДО явились требования универсальности ИМ относительно классов моделируемых систем и процессов, легкости модификации моделей,

моделирования сложных систем управления совместно с управляемым объектом (включая использование ИМ в управлении в реальном масштабе времени) и ряд других, сформировавшихся у разработчиков при выполнении работ, связанных с системным анализом и организационным управлением сложными системами различной природы.

2. Предпроектное исследование

2.1. Основные положения языка РДО

Основные положения системы РДО могут быть сформулированы следующим образом[3]:

- Все элементы СДС представлены как ресурсы, описываемые некоторыми параметрами. Ресурсы могут быть разбиты на несколько типов; каждый ресурс определенного типа описывается одними и теми же параметрами.
- Состояние ресурса определяется вектором значений всех его параметров; состояние СДС - значением всех параметров всех ресурсов.
- Процесс, протекающий в СДС, описывается как последовательность целенаправленных действий и нерегулярных событий, изменяющих определенным образом состояние ресурсов; действия ограничены во времени двумя событиями: событиями начала и событиями конца.
- Нерегулярные события описывают изменения состояния СДС, непредсказуемые в рамках продукционной модели системы (влияние внешних по отношению к СДС факторов либо факторов, внутренних по отношению к ресурсам СДС). Моменты наступления нерегулярных событий случайны.
- Действия описываются операциями, которые представляют собой модифицированные продукционные правила, учитывающие временные связи. Операция описывает предусловия, которым должно удовлетворять состояние участвующих в операции ресурсов, и правила изменения состояния ресурсов в начале и в конце соответствующего действия.
- Множество ресурсов R и множество операций O образуют модель СДС.

2.2. Поиск решения на графе пространства состояний

В языке имитационного моделирования РДО помимо возможности использования для описания законов управления формализмов продукционных правил введены так называемые точки принятия решений, позволяющие осуществлять оптимальное управление[2].

Механизм точек принятия решений в языке имитационного моделирования РДО позволяет гибко сочетать имитацию с оптимизацией. Для этого используется поиск на графе состояний.

Примерами задач, которые решаются с использованием точек принятия решений, могут служить:

- Различные транспортные задачи (например, выбор последовательности объезда пунктов транспортным средством при минимуме пройденного пути, времени или стоимости).
- Задачи укладки грузов при минимизации занимаемого ими объема (в более общем случае – задачи размещения).

- Нахождение решения логических задач за минимальное число ходов (например, расстановка фишек в игре «Пятнашки»).
- Задачи теории расписаний (например, задачи определения последовательности обработки различных деталей на станках при минимизации времени обработки, либо обслуживания клиентов с минимумом отклонений от запланированного времени изготовления заказов и т. д.).

Граф — основной объект изучения математической теории графов, совокупность непустого множества вершин и наборов пар вершин (связей между вершинами).

Граф $G = (S, E)$ задается двумя множествами:

- $S = \{s_i\}$ – множество вершин графа. Каждой вершине s_i ставится в соответствие состояние системы (база данных);
- $E = \{e_{ij}(s_i, s_j : s_i \in S, s_j \in S)\}$ – множество дуг. Каждой дуге e_{ij}, принадлежащей E и соединяющей пару вершин, ставится в соответствие правило преобразования (продукционное правило). Если дуга направлена от вершины s_i к вершине s_j, то s_i в данном случае будет являться вершиной-родителем, а s_j – вершиной-потомком (преемником).

Маршрутом в графе называют конечную последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей в последовательности вершиной ребром.

Цепью называется маршрут без повторяющихся рёбер.

Циклом называют цепь, в которой первая и последняя вершины совпадают.

Граф называется связным, если для любых вершин u, v существует путь из u в v.

Граф называется деревом, если он связный и не содержит нетривиальных циклов.

В графах, представляющих интерес для поиска, у каждой вершины должно быть конечное число вершин-преемников. С дугой может быть связана некоторая величина c_{ij} – стоимость дуги, она отражает затраты (в смысле заданного критерия оптимизации) применения соответствующего правила.

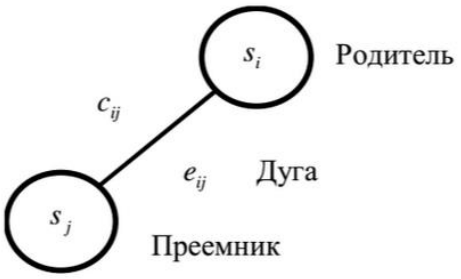

Рис. 1. Простейший граф состояний

В графе имеются две особые вершины:

- s_0 – начальная вершина, или другими словами, вершина, представляющая собой исходную базу данных;
- s_t – целевая вершина, иначе – вершина, представляющая собой базу данных, удовлетворяющую терминальному условию поиска. Таких вершин может быть не одна, а множество, и такое множество будет называться целевым множеством.

Путь в графе – последовательность вершин, в которой каждая последующая является преемником:

$$m = \{s_n, s_{n+1}, s_{n+2}, \dots, s_{n+k}\},$$

где $k + 1$ – длина пути.

Каждому пути m ставится в соответствие его стоимость, которая равна сумме стоимостей применения правил по всему пути на графе G[2].

2.3. Программный комплекс RAO-XT

Программный комплекс RAO-XT предназначен для разработки и отладки имитационных моделей на языке РДО. Основные цели данного комплекса - обеспечение пользователя легким в обращении, но достаточно мощным средством разработки текстов моделей на языке РДО, обладающим большинством функций по работе с текстами программ, характерных для сред программирования, а также средствами проведения и обработки результатов имитационных экспериментов.

Система имитационного моделирования RAO-XT представляет собой плагин для Eclipse IDE – свободной интегрированной среды разработки модульных кроссплатформенных приложений. Система написана на языке Java и состоит из трех основных компонентов:

- rdo – компонент, производящий преобразование кода на языке РДО в код на языке Java.
- rdo.lib – библиотека системы. Этот компонент реализует ядро системы имитационного моделирования.
- rdo.ui – компонент, реализующий графический интерфейс системы с помощью библиотеки SWT.

На момент начала выполнения курсового проекта, в систему RAO-XT был интегрирован модуль визуализации поиска решения на графе пространства состояний, были проведены его тесты на этапе опытной эксплуатации.

2.4. Постановка задачи

Проектирование любой системы начинается с выявления проблемы, для которой она создается. Под проблемой понимается несовпадение характеристик состояния систем, существующей и желаемой.

В результате предпроектного исследования на курсовом проектировании в прошлом семестре было выявлено отсутствие в программном комплексе RAO-XT подсистемы графической визуализации поиска решения на графе пространства состояний.

Несмотря на то, что в результате выполнения проекта подсистема была внедрена успешно, в рамках исследования этого семестра были выявлены следующие ее недостатки:

- Результаты работы подсистемы доступны для пользователя только по окончании моделирования, в течение самого процесса пользователь не получает никакой информации о работе системы;
- Представленный интерфейс и набор функций не предлагает пользователю гибкой настройки режимов и параметров отображения графа на экране;
- Геометрические размеры элементов интерфейса зависят от конкретной реализации устройства вывода, что представляет проблему при переносе ее на другие платформы;
- Вызов интерфейса подсистемы не всегда интуитивен и основывается на результатах работы трассировочного модуля.

Задачей этого семестра стало расширение функционала подсистемы и устранение перечисленных выше недостатков.

3. Формирование ТЗ

3.1. Общие сведения

Основание для разработки: задание на курсовой проект.

Заказчик: Кафедра «Компьютерные системы автоматизации производства» МГТУ им. Н. Э. Баумана

Разработчик: студент кафедры «Компьютерные системы автоматизации производства» Стрижов К. А.

Наименование темы разработки: «Подсистема визуализации поиска решения на графе состояний в РДО»

3.2. Назначение разработки

Разработать подсистему визуализации поиска решений на графе состояний и добиться ее интеграции в систему моделирования RAO-XT.

3.3. Требование к программе или программному изделию

3.3.1.Требования к функциональным характеристикам

Модуль визуализации должен реализовывать следующий функционал:

- Построение графа в процессе моделирования;
- Масштабирование;
- Работа в относительной системы координат;
- Построение графа из дерева трассируемых объектов.

3.3.2.Требования к надежности

Основное требование к надежности направлено на поддержание в исправном и работоспособном состоянии ЭВМ, на которой происходит использование программного комплекса RAO-XT.

Сохранение работоспособности системы при отказе по любым причинам подсистемы или ее части.

3.3.3.Условия эксплуатации

Эксплуатация должна производиться на оборудовании, отвечающем требованиями к составу и параметрам технических средств, и с применением программных средств, отвечающим требованиям к программной совместимости.

Аппаратные средства должны эксплуатироваться в помещениях с выделенной розеточной электросетью 220В ±10%, 50 Гц с защитным заземлением.

3.3.4.Требование к составу и параметрам технических средств

Программный продукт должен работать на компьютерах со следующими характеристиками:

- Объем ОЗУ не менее 2 Гб;
- Объем жесткого диска не менее 50 Гб;
- Микропроцессор с тактовой частотой не менее 1ГГц;
- Монитор с разрешением от 1280*1024 и выше.

3.3.5.Требование к информационной и программной совместимости

Система должна работать под управлением следующих ОС:

- Windows 7, 8;
- Ubuntu 14.10.

3.3.6.Требование к маркировке и упаковке

Требования к маркировке и упаковке не предъявляются.

3.3.7.Требование к транспортированию и хранению

Требования к транспортированию и хранению не предъявляются.

3.4. Требования к программной документации

Требования к программной документации не предъявляются.

3.5. Стадии и этапы разработки

Разработка должна быть проведена в три стадии:

- Техническое задание;
- Технический и рабочий проекты;
- Внедрение.

На стадии «Техническое задание» должен быть выполнен этап разработки и согласования настоящего технического задания.

На стадии «Технический и рабочий проект» должны быть выполнены перечисленные ниже этапы работ:

- Разработка программы;
- Разработка методики тестирования;
- Испытания программы.

На стадии «Внедрение» должен быть выполнен этап разработки «Подготовка и передача программы».

3.6. Порядок контроля и приемки

Контроль и приемка работоспособности системы осуществляются с помощью следующих методов:

- Опрос экспертов. Используется для оценки дизайна, качества и удобства использования новой системы графического вывода;
- Многократное ручное тестирование с помощью имитационных моделей, написанных на языке РДО.

4. Концептуальный этап проектирования системы

На концептуальном этапе проектирования требовалось:

- Описать и дополнить общую структуру подсистемы визуализации и схему её взаимодействия с другими компонентами системы RAO-XT;
- Разработать основной функционал зуммирования для модуля визуализации;
- Описать принципы работы модуля в системе относительных координат;
- Определить общую схему вызова интерфейса подсистемы визуализации в системе RAO-XT;
- Определить критерии, по которым необходимо провести оценку возможностей подсистемы на этапе рабочего проектирования.

4.1. Структура подсистемы визуализации

На этапе концептуального проектирования ключевой задачей является разработка правильной схемы взаимодействия подсистемы визуализации с остальными частями системы RAO-XT.

На этапе формирования ТЗ к подсистеме было предъявлено требование реализовать отображение графа в режиме реального времени в процессе моделирования (пункт 3.3.1). При работе в данном режиме необходимо обновлять графическую информацию, представленную на экране пользователю, по мере выполнения модели. Для этого необходимо выполнять оповещение модуля системой RAO-XT, причем скорость обновления информации должна быть основана на конечном масштабе времени, устанавливаем пользователем. Также модуль визуализации должен реагировать на возможные изменения пользователем этого масштаба по ходу моделирования.

Ниже приведены основные требования к модулю визуализации как к компоненту системы:

- Сохранение информации о состоянии системы в каждый момент модельного времени производится компонентом *Database* в бинарном формате. Таким образом компонент *Database* осуществляет сериализацию данных, т.е. сохранение их в компактном, но недоступном для восприятия пользователем формате. Модуль визуализации должен уметь интерпретировать эти данные и формировать на выходе древовидную структуру, по которой можно будет восстановить и построить граф пространства состояний и затем вывести его на экран пользователя.
- Модуль визуализации должен предоставлять выходные данные исключительно для пользователя. Никакие другие компоненты системы не должны основывать свою работу на выходных данных модуля визуализации.
- Модуль визуализации должен состоять из двух частей:
    - Библиотечная часть;
    - Графическая часть;
- Библиотечная часть модуля визуализации должна заниматься исключительно преобразованием бинарных данных в древовидную структуру. Далее с полученной

структурой работает графическая часть модуля, расположенная в пакете *UI*. Эта часть отвечает за отрисовку графа на экране и за графический интерфейс пользователя. При этом данные, которые отображает графическая часть модуля, должны полностью соответствовать тем данным, которые на данный момент преобразовала библиотечная часть.

- Разрабатываемый модуль должен быть активен сразу после старта прогона модели. Модуль выводит результат на экран, основываясь на данных, сформированных во время прогона, и должен быть доступен в течение всего процесса работы, реагировать на паузы и остановки процесса моделирования, коррелировать с конечным масштабом времени системы.

4.2. Функционал зуммирования

4.2.1. Описание общей концепции зуммирования

Зуммирование графа является необходимой пользователю функцией, в общем случае на экране одновременно может быть представлено достаточно большое количество информации, отображение которого возможно либо в очень маленьком масштабе, либо в более читаемом, но по частям. Важно предоставить пользователю гибкий инструмент настройки размера изображения.

На этапе концептуального проектирования необходимо определить перечень функций зуммирования графа. Было принято решение реализовать следующие варианты:

- Вписывание графа по ширине в любой момент времени отрисовки;
- Увеличение изображения графа на конечный коэффициент при нажатии клавиши или прокрутке колеса мыши;
- Аналогичное предыдущему пункту уменьшение изображения графа.

Должны быть обеспечены варианты вызова перечисленного функционала как с клавиатуры устройства, так и при помощи колеса мыши.

4.2.2. Характерные размеры вершин

В последней реализации подсистемы каждая вершина графа отображается на экране вместе со своим порядковым номером, соответствующим очередности поступления информации о ней в базе данных системы. Однако при наличии функции масштабирования возможно увеличить количество информации, отображаемой внутри вершины, исходя из размеров последней.

Предлагается ввести три характерных размера вершины, по достижении которого каждый раз при увеличении или уменьшении размеров графа информация, отображаемая внутри вершины, будет изменяться:

- Минимальный;
- Промежуточный;
- Максимальный.

При достижении минимального размера вершиной информация, отображаемая внутри нее, как и раньше должна ограничиваться порядковым номером этой вершины.

В промежуточном варианте было решено помимо порядкового номера выводить значение функции оценки стоимости пути f до целевой вершины через данную в следующем формате:

< порядковый_номер_вершины >

$< f > = < g > + < h >$

Для максимального размера вершины было принято решение помимо информации промежуточного варианта отображать имя примененного правила при переходе из предыдущего состояния в данное и стоимость его применения. Имена релевантных ресурсов решено не выводить по причине того, что их может быть достаточно много в общем случае.

Предлагаемый формат для вершины максимального размера:

< порядковый_номер_вершины >

$< f > = < g > + < h >$

< имя_правила > = < стоимость_правила >

Также для корректной работы функционала масштабирования и для читабельности информации, отображенной на экране, необходимо ввести понятие наименьшего размера вершины, при котором дальнейшее уменьшение масштаба не имеет смысла ввиду неразличимости вершин для глаза пользователя. Этот размер важен для функции вписывания графа по ширине, при достижении этого размера вершинам изображение графа более не уменьшается, дабы не сделать картинку неразличимой для глаза пользователя. Граф располагается в окне интерфейса как есть, и дальнейшее перемещение по нему должно осуществляться при помощи бегунков прокрутки.

4.3. Работа в системе относительных координат

4.3.1. Общие положения

Графическая библиотека, интерфейсы которой используются модулем для отображения информации на экране пользователя, изначально использует систему абсолютных

координат, ведущую отсчет по осям в пикселях экрана пользователя. Расположение осей и начало координат представлено на рисунке 2.

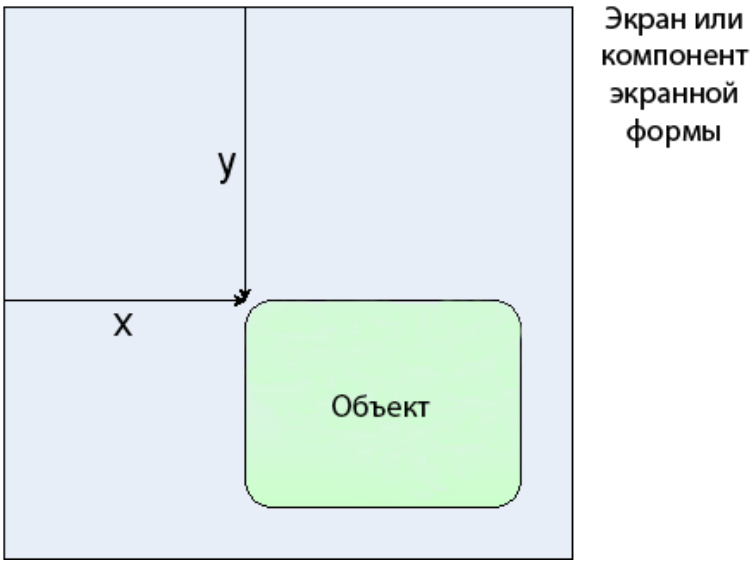

Рис.2. Расположение осей для системы абсолютных координат

Такая система координат имеет один существенный недостаток – для различных устройств вывода, характеристика DPI которых в общем случае неодинакова, изображение на экране будет выглядеть по-разному. Чтобы уйти от привязки разработчика к конкретной модели устройства, на котором будет выводиться граф, необходимо перейти на систему относительных координат, вид которой представлен на рисунке 3.

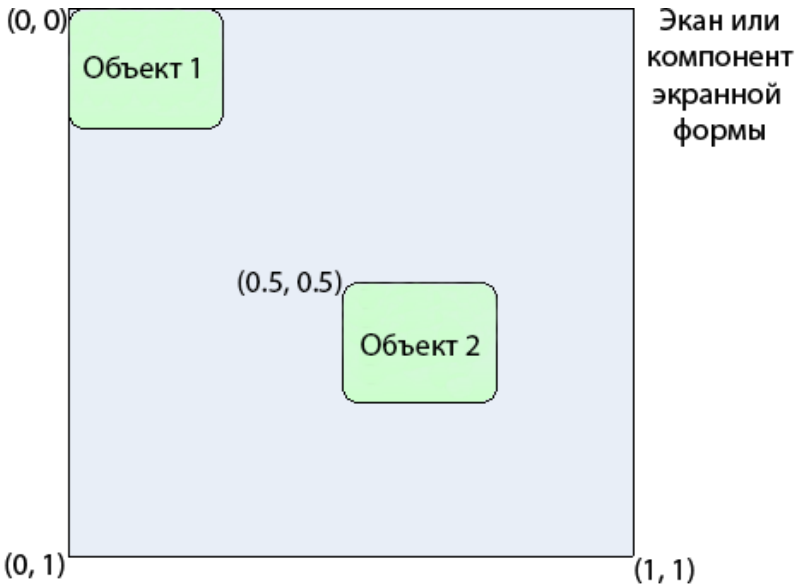

Рис. 3. Пример отсчета позиций элементов изображения в системе относительных координат

Такая система координат лишена подобного недостатка. Единственный нюанс, который необходимо учесть при проектировании – прямоугольная форма практически всех дисплеев, представленных на современном рынке. При наличии экрана прямоугольной формы (не квадратной) для сохранения пропорций по высоте и ширине, необходимо ввести коэффициент пропорциональности по одной из осей, равный отношению этих величин.

4.3.2. Свойства выбранной системы координат

Направления осей можно выбрать произвольными, для простоты было принято оставить их по умолчанию.

Переход на систему относительных координат должен затронуть только графическую часть подсистемы визуализации. Размер экранной формы должен задаваться в отношении к устройству вывода пользователя, а размеры графа – в отношении к размерам экранной формы.

4.4. Вызов окна интерфейса подсистемы

4.4.1. Существующая реализация вызова

На этапе опытной эксплуатации основным и единственным способом вызова окна графического интерфейса модуля визуализации был осуществлен через обращение к модулю трассировки системы RAO-XT. Пользователь двойным нажатием клавиши мыши по строке в полотне трассировки мог вызвать изображение графа по интересующей его точке принятия решений.

При разработке трассировочного модуля системы RAO-XT к нему было предъявлено требование о том, что результаты вывода данного модуля не должны являться входной информацией для других компонентов системы RAO-XT и использоваться где бы то ни было еще, кроме интерфейса модуля. Поэтому было принято решение организовать независимый вызов окна интерфейса подсистемы визуализации, не опирающийся на результат работы модуля трассировки.

4.4.2. Независимый вызов подсистемы визуализации

В последней версии системы RAO-XT появился интерфейс, отображающий для пользователя все объекты, которые должны быть сериализованы в базу данных системы. Данный интерфейс имеет древовидную структуру и отображает все типы объектов, которые могут находиться в базе данных. К числу этих объектов относятся и точки принятия решений типа search, представляющие интерес для разрабатываемого модуля.

На данном этапе было принято решение реализовать независимый вызов подсистемы через элементы дерева описанной выше структуры.

После опроса экспертов выяснилось, что на этапе опытной эксплуатации системы старый способ вызова показался достаточно быстрым и удобным. Ввиду этого факта было решено от него не отказываться и оставить реализацию вызова через строку полотна трассировки

в системе. В зависимости от ситуации, пользователю будет дана возможность выбора, что не добавит избыточного нагромождения в системе и вместе с тем предоставит гибкость использования ее функционала.

4.5. Критерии оценки возможностей подсистемы визуализации

На этапе апробирования системы следует разработать методику тестирования подсистемы визуализации для ее исследования на предмет следующих показателей:

1. Время построения графа для конечного количества вершин;
2. Точка переполнения памяти.

Значения первого показателя, собранные для графа с различным количеством вершин, позволит оценить скорость работы подсистемы и среднее время построения ожидания отклика пользователем в режим моделирования «без анимации».

Значение второго показателя позволят оценить ресурс подсистемы и ее возможности по обработке больших массивов данных.

5. Технический этап проектирования

5.1. Схема работы в процессе моделирования

Предыдущая версия системы имитационного моделирования RAO-XT позволяла отображать граф пространства состояний только в одном режиме: в конце моделирования на основе информации, содержащейся в базе данных системы, формировалась древовидная структура, элементы которой представляли собой отдельное состояние модели, и на основе этой структуры подсистема визуализации отображала граф на экране пользователя.

Для реализации новых требований необходимо решить проблему формирования древовидной структуры на этапе прогона модели и постоянного обновления графа при наличии новой информации.

Работа в таком режиме может быть осуществлена при помощи системы оповещений библиотеки RAO-XT. Для этого класс, которому необходимо получать оповещения о том, что произошли некоторые события, должен реализовывать некий интерфейс, имеющий метод, который будет вызываться системой в момент времени, когда случится событие подобного типа.

Общую схему работы можно разделить на две части, соответствующие частям подсистемы визуализации (согласно пункту 4.1).

5.1.1. Схема работы библиотечной части

В системе RAO-XT в компоненте rdo.lib существует система подписки и оповещения, пользуясь которой необходимо разработать требуемый в ТЗ (п.3.3.1) функционал. Для описания схемы работы необходимо описать структуру системы подписки:

- *Notifier* – интерфейс, содержащий внутренний класс подписки *Subscription* и методы получения списка всех доступных подписок или определенной подписки по категории (например для события остановки моделирования *ExecutionAborted*);
- *Subscription* – внутренний класс, содержащий список подписчиков, методы добавления и удаления новых подписчиков;
- *NotificationManager* – класс, реализующий интерфейс *Notifier*, содержит в себе список подписок (объекты класса *Subscription*), имеет метод *notifySubscribers(String category)*, который оповещает всех подписчиков, находящихся в подписке на событие *category.* В конструктор класса передается массив строк с названиями событий, подписки на которые необходимо хранить в списке.

Компонент *Database* содержит в себе экземпляр класса *NotificationManager,* при создании которого в конструктор класса передается единственное событие – событие появления новой записи в базе данных. Список подписок данного экземпляра будет состоять из одного элемента – списка всех подписчиков на событие пополнения базы данных.

В этот список необходимо добавить класс библиотечной части *TreeBuilder*, для чего он должен реализовать интерфейс *Subsriber*, имеющий следующий вид:

```
public interface Subscriber
{
	public void fireChange();
}
```

В переопределенном методе, который будет вызываться всякий раз при оповещении от базы данных, должен быть реализован следующие действия:

- Считать и распарсить из базы данных последнюю запись;
- Оповестить о наличии нового элемента в формируемой древовидной структуре библиотечную часть.

Для оповещения библиотечной части необходимо реализовать подписку последней на изменения в древовидной структуре класса *TreeBuilder*.

5.1.2. Схема работы графической части

Подписка графической части может быть реализована через анонимный класс, реализующий интерфейс *Subscriber*. Графическая часть должна содержать некий флаг, означающий наличие новой информации для отображения на экране, который будет принимать значение истины по подписке при вызове метода *fireChange()* этого анонимного класса.

Для отрисовки новых вершин в процессе моделирования необходимо соответствующий метод. Алгоритм его работы таков:

- Получить у класса *TreeBuilder* номер последнего объекта древовидной структуры;
- Сравнить его с номером последней отрисованной вершины: если номер последнего объекта больше номера последней вершины, это означает, что есть вершины, еще не отображенные на экране. Разность этих значений покажет количество таких вершин;
- В цикле перебрать все не отображённые вершины и вывести их на экран.

Скорость отрисовки новых вершин будет зависеть от скорости построения древовидной структуры в библиотечной части, которая в свою очередь зависит от скорости заполнения базы данных новой информацией. Настройка этой скорости осуществляется пользователем путем выбора режима моделирования и конечного масштаба времени моделирования. При выборе режима «без анимации» граф можно будет построить сразу целиком после окончания прогона модели.

5.1.3. Реализация многопоточности

Непрерывное обновление изображения графа на экране пользователя есть независимый процесс, который должен выполняться одновременно с процессом моделирования, выполняемым системой RAO-XT. Инструкции, связанные с прогоном модели, должны выполняться параллельно с инструкциями по отрисовке новых вершин графа.

Следовательно, необходимо обеспечить независимость работы модуля визуализации и системы моделирования и разграничить их действие в один и тот же момент времени. Сделать это можно, разделив выполнение кода каждой из систем на разные потоки.

Существует несколько способов запуска отдельного потока:

- Создать объекта класса *Thread* и передать в его конструктор объект, реализующий интерфейс *Runnable*;
- Отнаследоваться от класса *Thread* и переопределит его метод *run()*.

Особенностью реализации работы модуля визуализации по ходу процесса моделирования заключается тот факт, что необходимо постоянно (с маленькой периодичностью) проверять состояние флага наличия новой информации в системе, и однократного выполнения набора инструкций, описанного в методе *run()* соответствующего объекта, очевидно, будет недостаточно. Интерфейс программирования приложений (API) языка Java предоставляет разработчикам набор средств для проектирования многопоточных приложений, и решение задачи периодического выполнения набора инструкций в параллельном потоке возможно при помощи следующих классов:

- *Timer* – класс, методы которого позволяют запланировать событие для однократного или периодического исполнения. В качестве аргумента для таких методов используется экземпляр класса *TimerTask*;
- *TimerTask* – класс, реализующий интерфейс Runnable. Реализуемая сущность – задание для объекта-планировщика класса Timer. Набор инструкций для выполнения приводится в реализуемом методе *run().*

Заведя при старте прогона модели объект класса *Timer* и передав в его метод *scheduleAtFixedRate(TimerTask timerTask, long delay, long period)* объект *TimerTask* с соответствующими инструкциями, можно запланировать событие обновления графа с задержкой delay (следует указать значение 0, чтобы поток запустился вместе с началом процесса моделирования) и периодичностью period (следует задать небольшое значение, чтобы работа системы не зациклилась на бесконечном обновлении картинки).

5.1.4. Синхронизация доступа к общим данным

При проектировании многопоточных приложений всегда следует быть осторожным с организацией доступа к данным, которые могут использоваться сразу несколькими потоками. Может возникнуть ситуация, когда инструкции одного потока изменяют некоторые данные, а другой поток в это же время пытается их считать.

Чтобы операции чтения и записи не перекрывали друг друга, и не возникало ситуации чтения неверных данных или обращения к области памяти, в которой данных вообще не существует, необходимо ввести ограничения на одновременный доступ к ним для каждого из потоков.

Средства языка Java предоставляют различные возможности по синхронизации потоков, как-то:

- Использование модификатора synchronized;
- Использование средств пакета языка java.util.concurrent.

Первый вариант предлагает возможность запрета одновременного доступа несколькими потоками к данным, защищенным объектом-монитором, использующим это ключевое слово. Управление доступом потоков осуществляется средствами виртуальной машины Java (JVM).

Второй подход – вариант реализации объектов-аналогов технологии мьютексов (одноместных аналогов семафора), который также запрещает одновременный доступ. При захвате мьютекса поток получает право доступа к данным на определенных условиях (только чтение, чтение/запись), и другие потоки в порядке очереди ждут, пока мьютекс не будет освобожден. В отличие от первого подхода, ответственность за правильную очередность открытия и закрытия доступа к данным для разных потоков лежит на разработчике, однако взамен разработчик получает более гибкий инструмент для проектирования.

После рассмотрения плюсов и минусов перечисленных подходов, было решено воспользоваться средствами пакета java.util.concurrent с ручным управлением доступом к данным.

Общей для всей подсистемы является древовидная структура, формирование которой осуществляется в главном потоке системы, а чтение из нее должно происходить в параллельном потоке при запуске интерфейса модуля визуализации. Так же следует защищать от одновременного доступа флаг конца моделирования.

В качестве аналога мьютекса был выбран экземпляр класса *ReentrantReadWriteLock*. В документации языка Java приводится следующая конструкция:

```
rwLock.writeLock().lock();
try {

      //доступ к общим даннным

} finally {
      rwLock.writeLock().unlock();
}
```

При каждом обращении к данным, перечисленным выше, следует использовать подобный блок кода.

5.2. Масштабирование

5.2.1. Общие положения

Суть масштабирования – изменение размеров графа, изображенного на экране. В текущей реализации подсистемы, в которой используется графическая библиотека JGraphx, существует два подхода к изменению размеров изображенной на экране информации:

- Использование методов зуммирования компонента окна, представленного классом *mxGraphComponent*;
- Использование методов изменения размеров вершин и расстояний между ними самого графа, представленного экземпляром класса *mxGraph*.

Предполагаемый функционал зуммирования требует, чтобы изменялось содержание информации, представленной в вершинах, изменение размеров является не конечной целью, но сопутствующей. Следовательно, использование второго подхода лучшим образом подходит для решения поставленных задач.

Для реализации такого подхода требуется в классе графической части модуля визуализации содержать информацию о размерах характерных размеров вершин (п. 4.2.2), методы обновления этих размеров и проверки текущего общего для всех вершин размера на соответствие им.

Также должен быть задан наименьший размер вершины, по достижении которого должна быть активна только функция увеличения изображения графа. Данный размер вершины может быть таковым, что даже номер вершины в качестве минимальной информации не будет помещаться в пределах ее границ. Данная ситуация возможна при наличии очень большого числа вершин и желании пользователя увидеть граф целиком. В таком случае не следует отображать никакого текста вообще.

5.2.2. Вписывание графа по ширине

Для того чтобы вписать граф по ширине в окно определенных размеров, прежде всего необходимо понимать, как определяется ширина самого графа, чтобы впоследствии привести ее к размеру окна по ширине.

Можно сказать, что граф состоит из уровней, номер которого определяется расстоянием от вершины графа на текущем уровне до корневой вершины. Следовательно, из предположения о том, что ширина графа складывается из суммы размеров вершины на уровне с максимальным количеством вершин и расстояний между ними, можно предложить такой алгоритм работы:

- Определить максимальное количество вершин в графе по ширине;
- Подсчитать ширину графа, зная ширину вершины и величину расстояния между вершинами;
- Сравнить ширину графа и ширину окна;

- В случае, если размеры графа больше, разделить ширину окна на максимальное количество вершин графа по ширине и узнать количество пикселей, которое можно отвести на одну вершину;
- Изменить текущие величины размера вершины и расстояния между ними;
- Перестроить граф.

При отрисовке графа используется функция автоматической расстановки вершин в форме дерева при помощи функционала, предоставленного классом *mxCompactTreeLayout*, предоставленного библиотекой JGraphx. Особенностью способа расстановки вершин этого класса является то, что ширина графа не обязательно складывается из суммы размеров вершин и расстояний между ними для уровня, имеющего наибольшее число вершин. Однако вершины смежных уровней располагаются друг под другом, что позволяет сделать следующий вывод: зная размер вершины и установленное между ними расстояние, рассчитав размер рамки, в которую можно вписать весь граф, можно однозначно определить количество вершин по ширине.

Размеры рамки, ограничивающей граф, непосредственно зависят от количества вершин по высоте и ширине, однако средства библиотеки JGraphx позволяют определить их, не зная этих величин. Класс *mxGraph* предоставляет метод *getBounds(Object[] cells)*, рассчитывающий размеры рамки, ограничивающей группу вершин экземпляра этого класса, переданную в этот метод в качестве параметра. Существование такого функционала значительно упрощает решаемую задачу и позволяет реализовать приведенный выше алгоритм.

5.2.3.Увеличение изображения

Для изменения размера изображения в классе *GraphFrame* графической части модуля визуализации необходимо создать поле, содержащее значение конечно масштаба. Новые геометрические размеры всех элементов графа можно получить путем умножения или деления на эту величину.

Общий алгоритм изменения размеров элементов графа следующий:

- Задать новые размеры вершины графа;
- Поменять размеры всех вершин;
- Проверить размеры вершин на соответствие одному из характерных размеров;
- Задать для каждой вершины текст, объем информации которого зависит от соответствия определенному характерному размеру.

Для проверки на соответствие какому-либо из характерных размеров удобно описать возможные варианты в виде констант перечисления enum и описать метод, который будет возвращать значение типа данного перечисления. Тогда в блоке switch для каждой из констант можно будет описать те действия, которые необходимо выполнить для формирования текста с информацией того объема, который соответствует каждому из характерных размеров.

5.2.4.Уменьшение изображения

Общий алгоритм работы функции уменьшения аналогичен приведенному выше (п. 5.2.3), за исключением того факта, что размеры графа не должны изменяться при вызове функции по достижении вершинами наименьшего их размера.

При уменьшении размеров вершин может возникнуть ситуация, когда даже минимальный объем информации (содержащий только ее номер) не помещается внутри вершины. Для такого варианта было принято решение отображать только сами вершины без какой-либо информации вообще.

Отдельно следует рассмотреть случай, когда наименьший размер достигнут, и затем следует вызов функции увеличения изображения. При небольших размерах вершин и относительно небольшом масштабе увеличения скорее всего возникнет ситуация, когда после умножения на масштаб новое число пикселей, приходящихся на размер вершины, округлится до своего текущего значения, и изменений размеров графа не произойдет.

Для разрешения проблемы предложено временно увеличивать масштаб до момента, когда количество пикселей на размер вершины будет достаточно большим для того, чтобы граф был чувствителен к увеличению своих размеров с шагом (масштабом) по умолчанию.

5.3. Относительные координаты

Для перехода на новую систему координат первым шагом были найдены места в системе, где используются абсолютные значения в пикселях в качестве размеров или координат для элементов графа или интерфейса самого модуля:

- Размеры экранной формы в методе вызова конструктора класса *GraphFrame*;
- Размеры вершин в методах отрисовки вершин;
- Координаты вершин.

Метод вызова конструктора экземпляра класса новой экранной формы описан в классе *GraphControl*, тогда как определение размеров и координат вершин происходит в классе *GraphFrame*. Окно графа должно создаваться в системе координат, связанной с экраном пользователя, тогда как сам граф и элементы интерфейса должны быть описаны в системе координат, связанной с экранной формой. Следовательно, удобно будет описать отдельные методы преобразования для каждой из систем координат и содержать их там, где они будут использоваться.

5.3.1.Методы преобразования координат

Каждый из двух перечисленных классов должен реализовать приватные методы *setWidth()* и *setHeight()* – методы преобразования относительной величины в количество соответствующих ей пикселей из расчета, что на значение 1 приходится вся ширина (высота) экрана или экранной формы. Эти методы будут вызываться при необходимости задать геометрический размер объекта в системе относительных координат.

Помимо этого, класс *GraphFrame* должен дополнительно реализовать два метода получения абсолютной координаты из относительной, так как все методы отрисовки библиотеки JGraphx принимают в качестве параметров значения в пикселях.

5.3.2. Коэффициент соотношения сторон

Коэффициент пропорциональности, на который необходимо домножать значения по одной из осей (согласно в п.4.3) – коэффициент соотношения сторон экрана или экранной формы (с англ. – *aspect ratio*), обычно вычисляется как отношение ширины объекта к его высоте.

Согласно разработанной на предыдущем этапе концепции перехода на исчисление величин геометрических размеров объектов в относительных координатах, было положено завести две различные системы координат:

- Связанную с дисплеем пользователя;
- Связанную с экранной формой интерфейса подсистемы.

Так как эти системы связаны с объектами, соотношения сторон которых в общем случае могут отличаться, необходимо завести отдельные поля в классах *GraphControl* и *GraphFrame* для каждой из них.

5.4. Независимый вызов интерфейса подсистемы

На этапе технического проектирования было принято решение реализовать вызов интерфейса модуля визуализации при помощи контекстного всплывающего меню, появляющегося при клике пользователя правой кнопкой мыши на элементе дерева сериализуемых объектов, относящемуся к интересующей его точке принятия решений типа search.

5.4.1. Реализация контекстного меню

В классе, реализующем интерфейс дерева сериализуемых объектов, необходимо описать экземпляры объектов следующих классов:

- *Menu* – класс, реализующий сущность контекстного меню;
- *MenuItem* – класс, представляющий элемент контекстного меню, который будет появляться при вызове объекта класса *Menu.*

Для данных объектов можно настраивать имена элементов контекстного меню, их порядок, условия, при которых элемент меню активен или неактивен, а также действия, выполняемые при его выборе.

5.4.2. Индексация элементов дерева сериализуемых объектов

При выделении любого элемента дерева можно получить объект класса *CollectedDataNode*, который проиндексирован объектом класса, являющегося дочерним по отношению к классу *Index* (например, для ресурсов это класс *ResourceIndex*), реализующего интерфейс

AbstractIndex. Для элемента каждого типа класс свой, для подсистемы визуализации интерес представляют элементы, индексированные объектами класса *SearchIndex*.

Чтобы исключить ситуацию вызова активного пункта меню построения графа для элемента дерева, не относящегося к поиску решения на графе, следует делать проверку типа возвращаемого индексирующего объекта на его соответствие классу *SearchIndex*. Если проверка дает положительный результат, при щелчке мышью на объект дерева пункт меню, вызывающий интерфейс модуля визуализации, будет активен.

Кроме того, у полученного экземпляра класса *SearchIndex* можно получить номер точки принятия решения и ее имя. Эти данные должны передаваться непосредственно в конструктор класса *GraphFrame*, реализующего интерфейс модуля визуализации, и вызывать экранную форму с интересующим пользователя графом.

6. Рабочий этап проектирования

На рабочем этапе проектирования были реализованы разработанные на предыдущих этапах концепции и решения.

6.1. Реализация подписки

В данном разделе будут описаны конструкции языка Java, использованные для подключения модуля визуализации к системе оповещения библиотеки RAO-XT.

6.1.1. Библиотечная часть подсистемы

Для оповещения библиотечной части модуля визуализации необходимо, чтобы класс *TreeBuilder* реализовал интерфейс подписчика *Subscriber* и был добавлен в подписку компонента *Database*.

Реализация интерфейса подписчика библиотечной частью подсистемы визуализации выглядит следующим образом:

```
public class TreeBuilder implements Subscriber {

	@Override
	public void fireChange() {
		final Database.Entry entry = Simulator.getDatabase().getAllEntries()
				.get(entryNumber);
		boolean isSearchEntry = parseEntry(entry);
		entryNumber++;
		if (isSearchEntry)
			notifyGUIPart();
	}

	private Subscriber GUISubscriber = null;

	public final void notifyGUIPart() {
		GUISubscriber.fireChange();
	}

	public final void setGUISubscriber(Subscriber subscriber) {
		this.GUISubscriber = subscriber;
	}

	...
}
```

Ниже приведены инструкции добавления библиотечной части в подписку базы данных в классе *ExecutionHandler*:

```
Notifier databaseNotifier = Simulator.getDatabase().getNotifier();

databaseNotifier.getSubscription("EntryAdded")
		.addSubscriber(Simulator.getTreeBuilder());
```

6.1.2. Графическая часть подсистемы

Графическая часть подсистемы визуализации подписана на обновления древовидной структуры библиотечной части, однако класс *GraphFrame* непосредственно не реализует интерфейс *Subscriber*. Вместо этого использован внутренний анонимный класс, вызов метода fireChange() которого устанавливает флаг наличия новых вершин и вызывается библиотечной частью:

```
private static volatile boolean haveNewRealTimeData = false;

public static final Subscriber realTimeUpdater = new Subscriber() {
	@Override
	public void fireChange() {
		haveNewRealTimeData = true;
	}
};
```

Сброс флага происходит каждый раз после того, как отрисованы все новые вершины в текущем цикле работы потока обновления графической информации. После этого графическая часть переводится в режим ожидания до следующей установки флага наличия новых вершин.

6.2. Обновление графической информации

Основной задачей при описании функционала обновления графа по ходу моделирования в рамках рабочего проекта являлась грамотная реализация многопоточности в пределах всей системы RAO-XT.

Также необходимо было соблюдать осторожность при работе с общими для библиотечной и графической части данными, коими являлась древовидная структура библиотечной части. Необходимо было исключить конфликтные ситуации при попытке одновременного чтения и записи данных в контейнер вершин.

6.2.1. Выделение отдельного потока

В рамках рабочего проектирования были внесены коррективы в предыдущие концепции, и было предложено выделять не один, а два независимых потока на каждый граф. Это было вызвано тем, что помимо проверки наличия новых данных для отображения, необходимо осуществлять параллельную проверку статуса графа. Иначе подсистема не имела бы информации о том, все ли вершины графа отрисованы и есть ли у библиотечной части информация о решении.

Задание для таймера, проверяющего наличие новых вершин, выглядит следующим образом:

```
graphFrameUpdateTimerTask = new TimerTask() {
	@Override
	public void run() {
		if (!haveNewRealTimeData)
			return;
		haveNewRealTimeData = false;
		graph.getModel().beginUpdate();
```

```
                try {
                        drawNewVertex(graph, nodeList, dptNum);
                        Dimension frameDimension = getSize();
                        Object[] cells = vertexMap.values().toArray();
                        mxRectangle graphBounds = graph.getBoundsForCells(cells,
                                        false, false, false);
                        if (graphBounds.getWidth() > rameDimension.getWidth()) {
                                layout.setMoveTree(true);
                                zoomToFit(graph, layout, graphComponent);
                                graph.refresh();
                        } else
                                layout.execute(graph.getDefaultParent());
                } finally {
                        graph.getModel().endUpdate();
                }
                boolean isFinished;
                Simulator.getTreeBuilder().rwLock.readLock().lock();
                try {
                        isFinished = Simulator.getTreeBuilder().dptSimulationInfoMap
                                                        .get(dptNum).isFinished;
                } finally {
                        Simulator.getTreeBuilder().rwLock.readLock().unlock();
                }
                if (isFinished)
                        cancel();
        }
};
```

Задание для таймера проверки состояния графа выглядит так:

```
graphFrameFinTimerTask = new TimerTask() {
        @Override
        public void run() {
                Simulator.getTreeBuilder().rwLock.readLock().lock();
                try {
                        isFinished = Simulator.getTreeBuilder().dptSimulationInfoMap
                                                        .get(dptNum).isFinished;
                        if (isFinished) {
                                graph.getModel().beginUpdate();
                                try {
                                        GraphInfo info = Simulator.getTreeBuilder().infoMap
                                                                        .get(dptNum);
                                        ArrayList<Node> solution = Simulator
                                                        .getTreeBuilder().solutionMap.get(dptNum);
                                        colorNodes(vertexMap, nodeList, solution);
                                        insertInfo(graph, info);
                                } finally {
                                        graph.getModel().endUpdate();
                                }
                                graph.refresh();
                                cancel();
                        }
                } finally {
                        Simulator.getTreeBuilder().rwLock.readLock().unlock();
                }
        }
};
```

Оба таймера (каждый из которых суть независимый поток) начинает свою работу в случае, если был вызван интерфейс модуля визуализации, были отрисованы все вершины,

соответствующие текущим записям в базе данных, и флаг состояния графа отрицателен, т.е. граф еще не достроен:

```
if (!isFinished) {
	TimerTask graphUpdateTask = graphFrame
		.getGraphFrameUpdateTimerTask();
	Timer graphUpdateTimer = new Timer();
	graphUpdateTimer.scheduleAtFixedRate(graphUpdateTask, 0, 10);

	TimerTask graphFinTask = graphFrame.getGraphFrameFinTimerTask();
	Timer graphFinTimer = new Timer();
	graphFinTimer.scheduleAtFixedRate(graphFinTask, 0, 10);
}
```

Потоки завершают свою работу по окончании моделирования или при закрытии окна графа пользователем.

6.2.2. Функция отрисовки новых вершин

Алгоритм отрисовки новых вершин, разработанный на этапе технического проектирования (п.5.1.2), был реализован в следующем методе класса *GraphFrame*:

```
private void drawNewVertex(mxGraph graph, ArrayList<Node> nodeList,int dptNum) {
	Simulator.getTreeBuilder().rwLock.readLock().lock();
	try {
		int lastAddedNodeIndex = Simulator.getTreeBuilder()
					.getLastAddedNodeIndexMap().get(dptNum);
	for (int i = lastAddedVertexIndex; i <= lastAddedNodeIndex; i++) {
		Node node = nodeList.get(i);
		if (!vertexMap.containsKey(node)) {
			mxCell vertex = (mxCell) graph.insertVertex(graph
					.getDefaultParent(), null, node,getAbsoluteX(0.5),
					getAbsoluteY(0.05), nodeWidth, nodeHeight,
					fontColor + strokeColor);
			vertexMap.put(node, vertex);
			lastAddedVertexIndex = node.index;
			updateTypicalDimensions(node);
			if (node.parent != null)
				graph.insertEdge(graph.getDefaultParent(), null, null,
					vertexMap.get(node.parent),
					vertexMap.get(node), strokeColor);
			}
		}
	} finally {
		Simulator.getTreeBuilder().rwLock.readLock().unlock();
	}
}
```

6.3. Общая структура методов масштабирования

Для унификации описания всех методов масштабирования были приняты следующие проектные решения:

- Все методы масштабирования в своей основе содержат метод изменения размеров вершин *resizeCells()* библиотеки JGraphx, поэтому был описан один общий метод изменения графа *resizeGraph(),* содержащий в себе инструкции по изменению размеров вершин и детализации информации в них;

- Для детализации информации в вершинах после использования любой из функций зуммирования был описан общий метод *setTextForVertexes()*, возвращающий значение перечисления enum;
- Все размеры вершин содержатся в приватных глобальных переменных класса *GraphFrame*.

6.3.1. Описание метода изменения размеров графа

Метод изменения размеров графа *resizeGraph()* реализован следующим образом:

```
private void resizeGraph(mxGraph graph, mxCompactTreeLayout layout) {
	final mxRectangle bound = new mxRectangle(0, 0, nodeWidth, nodeWidth);
	final mxRectangle[] bounds = new mxRectangle[vertexMap.size()];
	for (int i = 0; i < vertexMap.size(); i++) {
		bounds[i] = bound;
	}
	final Object[] cells = vertexMap.values().toArray();
	graph.getModel().beginUpdate();
	try {
		graph.resizeCells(cells, bounds);
		setTextForVertexes();
		layout.setNodeDistance(nodeDistance);
		layout.execute(graph.getDefaultParent());
	} finally {
		graph.getModel().endUpdate();
	}
	graph.refresh();
}
```

Метод детализации информации по вершинам описан при помощи перечисления *SizeType*:

```
enum SizeType {
	NOTEXT, BRIEF, NORMAL, FULL
}

private void setTextForVertexes() {
	SizeType type = checkSize();
	switch (type) {
	...
	case FULL:
		for (mxCell cell : vertexMap.values()) {
			Node node = (Node) cell.getValue();
			final String number = Integer.toString(node.index);
			final String costFunction = Double.toString(node.g + node.h)
					+ " = " + Double.toString(node.g) + " + "
					+ Double.toString(node.h);
			final String rule = node.ruleName + " = "
					+ Double.toString(node.ruleCost);
			final String shortRule = rule.replaceAll("\\(.*\\)", "");
			final String text = number + "\n"
					+ costFunction + "\n" + hortRule;
			node.label = text;
			cell.setValue(node);
		}
		break;
	}
}
```

6.3.2.Описание функции масштабирования по ширине

Для примера работы функционала зуммирования был взят пример наиболее сложной в реализации функции масштабирования по ширине:

```
private void zoomToFit(mxGraph graph, mxCompactTreeLayout layout, mxGraphComponent
graphComponent) {
	Dimension frameDimension = this.getSize();

	Object[] cells = vertexMap.values().toArray();
	mxRectangle graphBounds = graph.getBoundsForCells(cells, false, false, false);

	maxNumOfNodesInWidth = (int) ((graphBounds.getWidth() +
			2 * nodeDistance) / (nodeWidth + 2 * nodeDistance));

	double period = frameDimension.getWidth() / maxNumOfNodesInWidth;

	if (nodeWidth < minNodeWidth) {
		nodeWidth = minNodeWidth;
		nodeHeight = nodeWidth;
		nodeDistance = nodeWidth;
	} else {
		nodeWidth = (int) (0.8 * period);
		nodeHeight = nodeWidth;
		nodeDistance = (int) (0.1 * period);
	}

	resizeGraph(graph, layout);
	setScrollsToCenter(graphComponent);
	graph.refresh();
}
```

6.4. Методы преобразования координат

Для перехода на систему относительных координат были описаны методы преобразования и введены коэффициенты соотношения сторон для дисплея пользователя и экранной формы интерфейса. Пример описания подобных методов класса GraphControl приведен ниже:

```
final static Rectangle monitorBounds = PlatformUI.getWorkbench()
			.getDisplay().getBounds();

public static final double monitorAspectRatio = (double) monitorBounds.width
		/ monitorBounds.height;

public static int setWidth(Rectangle r, double relativeWidth) {
	return (int) (r.width * relativeWidth / GraphControl.monitorAspectRatio);
}

public static int setHeight(Rectangle r, double relativeHeight) {
	return (int) (r.height * relativeHeight);
}
```

6.5. Создание контекстного меню вызова подсистемы

Контекстное меню должно вызываться по клику мыши на вершину дерева сериализуемых объектов, соответствующую точке принятия решений. Если вершина дерева представляет другой объект, опция вызова должна быть затенена и не интерактивна.

Создание контекстного меню и пункта вызова графа описывается следующими инструкциями:

```
final Menu popupMenu = new Menu(serializedObjectsTreeViewer.getTree());
final MenuItem graphMenuItem = new MenuItem(popupMenu, SWT.CASCADE);
graphMenuItem.setText("Build graph");
serializedObjectsTree.setMenu(popupMenu);
```

Проверка на соответствие вершины требуемому типу:

```
popupMenu.addListener(SWT.Show, new Listener() {

	@Override
	public void handleEvent(Event arg0) {
		boolean enabled = false;

		final CollectedDataNode node = (CollectedDataNode)
			serializedObjectsTreeViewer
			.getTree().getSelection()[0].getData();
		IndexType type = node.getIndex().getType();

		if (type == IndexType.SEARCH) {
			enabled = true;
		}

		graphMenuItem.setEnabled(enabled);
	}
});
```

Реализация вызова метода создания экранной формы из интерфейса дерева трассируемых объектов:

```
graphMenuItem.addSelectionListener(new SelectionListener() {

	@Override
	public void widgetSelected(SelectionEvent event) {
		final CollectedDataNode node = (CollectedDataNode)
				serializedObjectsTreeViewer
				.getTree().getSelection()[0].getData();

		int dptNum = node.getIndex().getNumber();
		String frameName = node.getName();
		FrameInfo frameInfo = new FrameInfo(dptNum, frameName);

		GraphControl.openFrameWindow(frameInfo);
	}
	...
});
```

7. Апробирование разработанной подсистемы в модельных условиях

Для проведения тестов в исходный код системы были внесены изменения, позволяющие замерять времена начала и конца процесса построения графа и отображения интерфейса подсистемы на экране пользователя (в наносекундах) и выводить разницу этих значений в системную консоль среды Eclipse.

7.1. Методика тестирования

Для проведения тестов и отладки кода была разработана методика тестирования подсистемы:

1. Создать в среде RAO-XT новый проект;
2. Создать тестовую модель, используя исходный код модели, приведенный в приложении А;
3. Настроить в дереве отслеживаемых объектов отображение информации по точкам принятия решений;
4. По результатам вывода модуля трассировки наблюдать за количеством вершин, сформированных в древовидной структуре библиотечной части;
5. Остановить прогон по достижении контрольного значения количества вершин;
6. Вызвать интерфейс графа и записать значение времени построения графа, выведенное в системную консоль среды Eclipse.

Серию тестов провести для ряда контрольных значений 500, 5 000, 10 000, 20 000, 30 000 … вершин.

Продолжать тесты, двигаясь с установленным шагом контрольных значений, пока не найдется такое количество вершин в графе, при котором произойдет переполнение физической памяти тестового ПК.

Провести тесты для платформ, указанных в требованиях к информационной и программной совместимости (п.3.3.5).

7.2. Результаты тестирования

Персональный компьютер, на котором проводились тесты, обладает следующими характеристиками:

- Процессор: Intel Core i5-4210U 1.7GHz;
- ОЗУ: 4,0 ГБ;
- ПЗУ: 400 ГБ

Тесты проводились на платформах Windows 8.1 и Ubuntu 15.04. Для каждого контрольного значения количества вершин было сделано по 5 замеров, после чего было посчитано средние значения времен построения графа. Результаты приведены в таблице 1.

Таблица 1.

Количество вершин	Среднее значение времени, с	
	Windows 8.1	Ubuntu 15.04
500	0.58	0.90
5 000	0.69	0.77
10 000	1.60	2.85
20 000	5.568	5.67
30 000	21.30	12.30
40 000	38.29	33.45

После контрольного значения в 40 000 вершин появляется нестабильность работы системы, выражающаяся в периодических отказах по причине переполнения памяти при попытках построить граф. Из двух платформ, Windows 8.1 показала себя как более надежная, практически всегда отображающая граф на экране с более высокой скоростью по сравнению с другой тестовой ОС, вплоть до значения в 50 000 вершин.

8. Заключение

В рамках данного курсового проекта были получены следующие результаты:

- Проведено предпроектное исследование системы имитационного моделирования RAO-XT и ее компонентов на предмет необходимости и возможности расширения функционала подсистемы визуализации поиска решения на графе пространства состояний;
- Определены основные функции и требования к подсистеме, на основе которых было составлено техническое задание;
- На этапах концептуального и технического проектирования была разработана основная структура подсистемы, схема ее взаимодействия с другими компонентами системы RAO-XT, рассмотрены аспекты проектирования многопоточных приложений, нюансы перехода на новую систему координат, описаны тонкости технической реализации;
- На этапе рабочего проектирования был разработан и реализован модуль визуализации поиска на графе пространства состояний для системы имитационного моделирования RAO-XT. Модуль реализован таким образом, что другим компонентам системы не требуется использовать его вывод для своего функционирования;
- Была разработана и использована методика ручного тестирования производительности модуля визуализации. Результаты могут быть использованы при оптимизации и реорганизации исходного кода модуля визуализации в дальнейшем.

Список использованных источников

1. **Емельянов, В. В.** Введение в интеллектуальное имитационное моделирование сложных дискретных систем и процессов. Язык РДО. / В. В. Емельянов, С. И. Ясиновский - М.: "Анвик", 1998. - 427 с., ил. 136.
2. **Емельянов, В. В.** Принятие оптимальных решений в интеллектуальных имитационных системах: Учебное пособие по курсам «Методы системного анализа и синтеза» и «Моделирование технологических и производственных процессов» / В.В. Емельянов, В.И. Майорова, Ю.В. Разумцова и др. – М.: Изд-во МГТУ им. Н.Э. Баумана. – 2002. – 60 с.: ил.
3. Документация по языку РДО [http://www.rdostudio.com/help/index.html]
4. Java™ Platform, Standard Edition 7. API Specification. [http://docs.oracle.com/javase/7/docs/api/]
5. JGraphX (JGraph 6) User Manual [https://jgraph.github.io/mxgraph/docs/manual_javavis.html]
6. JGraphx API Specification [https://jgraph.github.io/mxgraph/java/docs/index.html]

Список использованного программного обеспечения

7. Autodesk AutoCAD2012
8. Eclipse IDE for Java Developers Luna Service Release 1 (4.4.1)
9. Inkscape 0.48.4
10. Microsoft Office Word 2010
11. openjdk version "1.8.0_40-internal"
12. Visual Paradigm for UML 8.0

Приложение А. Исходный код модели, использованной для тестирования модуля

```
/* Game 5 */

$Resource_type Фишка : permanent
$Parameters
  Номер: integer
  Местоположение: integer
$End

$Resource_type Дырка_t : permanent
$Parameters
  Место: integer
$End

$Resources
  Фишка1 = Фишка(1, 2);
  Фишка2 = Фишка(2, 3);
  Фишка3 = Фишка(3, 6);
  Фишка4 = Фишка(4, 4);
  Фишка5 = Фишка(5, 5);
  Фишка6 = Фишка(6, 9);
  Фишка7 = Фишка(7, 7);
  Фишка8 = Фишка(8, 8);
  Дырка = Дырка_t(1);
$End


$Pattern Перемещение_фишки : rule
$Parameters
  Куда_перемещать: such_as Место_дырки
  На_сколько_перемещать: integer
$Relevant_resources
  _Фишка: Фишка   Keep
  _Дырка: Дырка_t Keep
$Body
  _Фишка:
    Choice from Где_дырка(_Фишка.Местоположение) == Куда_перемещать
    first
      Convert_rule    Местоположение    =    _Фишка.Местоположение    +
На_сколько_перемещать;
  _Дырка:
    Choice NoCheck
    first
      Convert_rule Место = _Дырка.Место - На_сколько_перемещать;
$End

$Decision_point Расстановка_фишек : search
$Condition Exist(Фишка: Фишка.Номер <> Фишка.Местоположение)
$Term_condition
  For_All(Фишка: Фишка.Номер == Фишка.Местоположение)
$Evaluate_by IDS()
$Compare_tops = YES
$Activities
  Перемещение_вправо: Перемещение_фишки(справа, 1) value after 1;
  Перемещение_влево : Перемещение_фишки(слева, -1) value after 1;
  Перемещение_вверх : Перемещение_фишки(сверху, -3) value after 1;
```

```
  Перемещение_вниз  : Перемещение_фишки(снизу, 3) value after 1;
$End

$Constant
    Место_дырки: (справа, слева, сверху, снизу, дырки_рядом_нет) =
дырки_рядом_нет
    Длина_поля : integer = 3
$End

$Function IDS: integer
$Type = algorithmic
$Parameters
$Body
    return 0;
$End

$Function Ряд: integer
$Type = algorithmic
$Parameters
    Местоположение: integer
$Body
    return (Местоположение - 1)/Длина_поля + 1;
$End

$Function Остаток_от_деления : integer
$Type = algorithmic
$Parameters
    Делимое   : integer
    Делитель  : integer
$Body
    integer Целая_часть  = Делимое/Делитель;
    integer Макс_делимое = Делитель * Целая_часть;
    return Делимое -  Макс_делимое;
$End

$Function Столбец: integer
$Type = algorithmic
$Parameters
    Местоположение: integer
$Body
    return Остаток_от_деления(Местоположение - 1,Длина_поля) + 1;
$End

$Function Где_дырка : such_as Место_дырки
$Type = algorithmic
$Parameters
    _Место: such_as Фишка.Местоположение
$Body
    if (Столбец(_Место) == Столбец(Дырка.Место) and Ряд(_Место) ==
Ряд(Дырка.Место)+ 1) return сверху;
    if (Столбец(_Место) == Столбец(Дырка.Место) and Ряд(_Место) ==
Ряд(Дырка.Место)- 1) return снизу;
    if (Ряд(_Место) == Ряд(Дырка.Место) and Столбец(_Место) ==
Столбец(Дырка.Место)- 1) return справа;
    if (Ряд(_Место) == Ряд(Дырка.Место) and Столбец(_Место) ==
Столбец(Дырка.Место)+ 1) return слева;
    return дырки_рядом_нет;
```

```
$End

$Function Фишка_на_месте : integer
$Type = algorithmic
$Parameters
    _Номер: such_as Фишка.Номер
    _Место: such_as Фишка.Местоположение
$Body
    if (_Номер == _Место) return 1;
    else                  return 0;
$End

$Function Кол_во_фишек_не_на_месте : integer
$Type = algorithmic
$Parameters
$Body
    return 5 - (Фишка_на_месте(Фишка1.Номер, Фишка1.Местоположение)+
                Фишка_на_месте(Фишка2.Номер, Фишка2.Местоположение)+
                Фишка_на_месте(Фишка3.Номер, Фишка3.Местоположение)+
                Фишка_на_месте(Фишка4.Номер, Фишка4.Местоположение)+
                Фишка_на_месте(Фишка5.Номер, Фишка5.Местоположение));
$End

$Function Расстояние_фишки_до_места : integer
$Type = algorithmic
$Parameters
    Откуда: integer
    Куда  : integer
$Body
    return Math.abs(Ряд(Откуда)-Ряд(Куда)) + Math.abs(Столбец(Откуда)-
Столбец(Куда));
$End

$Function Расстояния_фишек_до_мест : integer
$Type = algorithmic
$Parameters
$Body
    return                        Расстояние_фишки_до_места(Фишка1.Номер,
Фишка1.Местоположение)+
           Расстояние_фишки_до_места(Фишка2.Номер,
Фишка2.Местоположение)+
           Расстояние_фишки_до_места(Фишка3.Номер,
Фишка3.Местоположение)+
           Расстояние_фишки_до_места(Фишка4.Номер,
Фишка4.Местоположение)+
           Расстояние_фишки_до_места(Фишка5.Номер,
Фишка5.Местоположение);
$End

$Results
    f1 : get_value Фишка1.Местоположение
    f2 : get_value Фишка2.Местоположение
    f6 : get_value Дырка.Место

    ft1 : get_value 1
    ft2 : get_value 5
$End
```