Государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ_____РК_____

КАФЕДРА ___Компьютерные системы автоматизации производства РК-9_____

_____

Р А С Ч Ё Т Н О - П О Я С Н И Т Е Л Ь Н А Я З А П И С К А

к курсовому проекту на тему:

Реализация возможности множества прогонов в системе имитационного моделирования РДО

_____

_____

_____

_____

_____

_____

Студент _____ _____ _Медведев Я.В._

(Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ _____ _Урусов А.В.___

(Подпись, дата) (И.О.Фамилия)

Москва, 2012

Государственное образовательное учреждение высшего профессионального образования
«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой __РК9_____
(Индекс)
_____ _____
(И.О.Фамилия)
« ___ » _____ 20 __ г.

З А Д А Н И Е
на выполнение курсового проекта

по дисциплине__*Реализация возможности множества прогонов в системе имитационного моделирования РДО*_____

_____
(Тема курсового проекта)
_____

Студент__*Медведев Я.В. РК9-99*_____
(Фамилия, инициалы, индекс группы)

График выполнения проекта: 25% к ___ нед., 50% к ___ нед., 75% к __ нед., 100% к ___ нед.

1. Техническое задание
*Реализовать возможность множества прогонов в системе имитационного моделирования РДО*_____
_____
_____
_____

2. Оформление курсового проекта
2.1. Расчетно-пояснительная записка на 31 листе формата А4.
2.2. Перечень графического материала (плакаты, схемы, чертежи и т.п.)_____
*лист 1—А1 – плакат "постановка задачи"*_____
*лист 2 – А2 –диаграмма компонентов*_____
*лист 3—А2 – синтаксическая диаграмма*_____
*лист 4—А1 – блок-схема алгоритма прогона*_____
*лист 5—А1 – диаграмма классов*_____
*лист 6—А1 – плакат "Результаты*_____

Дата выдачи задания « » _____ 2012г.

Руководитель курсового проекта _____ ___*Урусов А.В.*_____
(Подпись, дата) (И.О.Фамилия)

Студент _____ ___*Медведев Я.В.*_____
(Подпись, дата) (И.О.Фамилия)

Оглавление

1. ВВЕДЕНИЕ

1.1. Введение

В рамках имитационного моделирования постоянно приходится сталкиваться со стохастическими системами, в которых внешнее воздействие или внутренние параметры связаны со случайными факторами. Эти стохастические свойства моделируются математически с помощью случайных величин. Распределения случайных величин моделируются на компьютере в том числе с помощью системы имитационного моделирования РДО. Для генерирования одной последовательности псевдослучайных чисел нужно чтото менять, например интервал прихода, или длительность выполнения какого0либо процесса и т.д. Но, так как в задачах чаще всего строго заданы все интервалы времени, то нужно менять значение базы генератора. И если мы промоделировали случайную величину, то в результате мы получаем тоже случайную величину, которая имеет свое среднее и дисперсию равную бесконечности. Брать эту величину в качестве результата на основании одного прогона модели не совсем правильно, ведь это только одно из возможных значений. Поэтому нужно брать значения из серии прогонов, меняя в модели последовательности псевдослучайных чисел через базу генератора. Из графика (рис.1) видно как выглядит распределение Стьюдента от числа степеней свободы, которая в свою очередь зависит от количества проведенных прогонов.

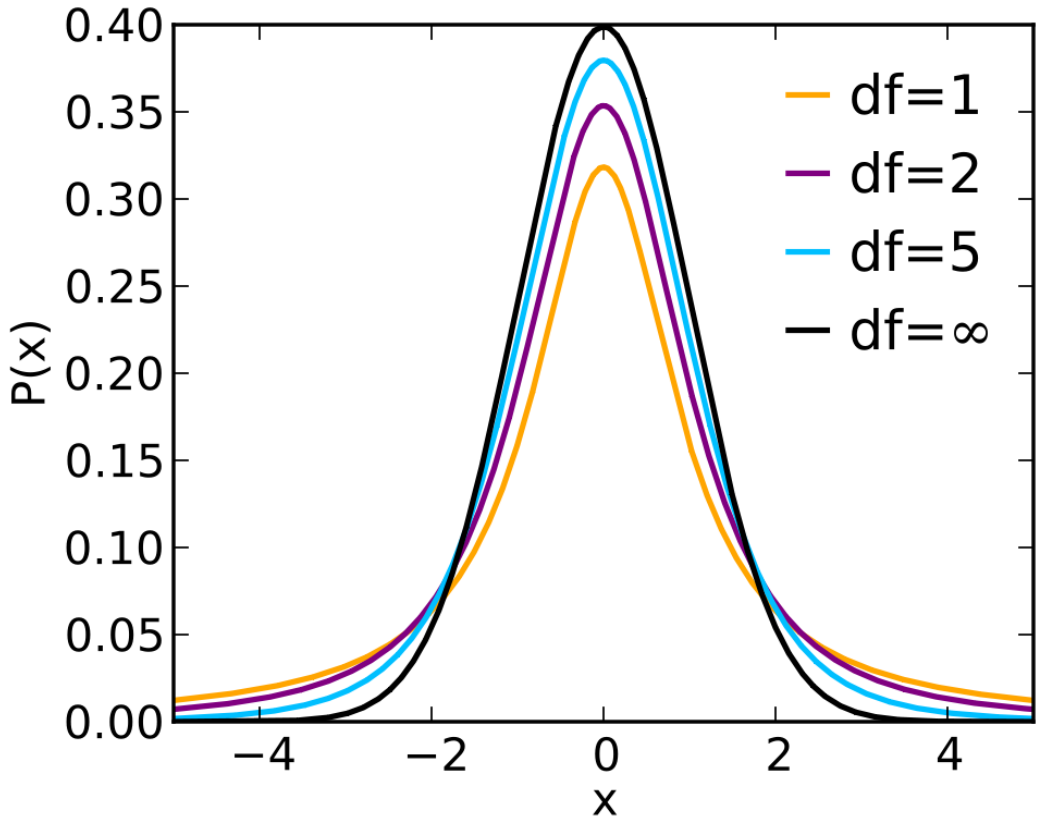


Рис. 1

P(x) – плотность вероятности; x – случайная величина; df=N-1 – число степеней свободы; N- количество проведенных прогонов.

Имея результаты с разных прогонов мы сможем рассчитать доверительный интервал и более точно оценить результаты моделирования.

1.2. Последовательности

В имитационном моделировании постоянно приходится сталкиваться со случайными величинами. Как правило, это величины, связанные с временами (периоды прихода заказов, время обработки заказов), реже – с количествами.

Термины "генератор псевдослучайных чисел" и "датчик псевдослучайных чисел" следует считать синонимами.

Большинство случайных величин подчиняется законам распределения. Система имитационного моделирования РДО поддерживает:

- нормальный
- экспоницальный
- равномерный
- треугольный

1.3. База генератора

Парадокс заключается в том, что надо случайную величину заставить подчиняться определенному закону, когда на самом деле по-настоящему случайной величины нет. Вся текущая работа выполняется на цифровой вычислительной технике. Эта техника подчиняется строгим законам. Можно сказать, она строго детерминирована, иначе она не смогла бы полноценно функционировать. Поэтому случайности тут взяться неоткуда. Она бы постоянно давала сбои, если бы смогла запуститься.

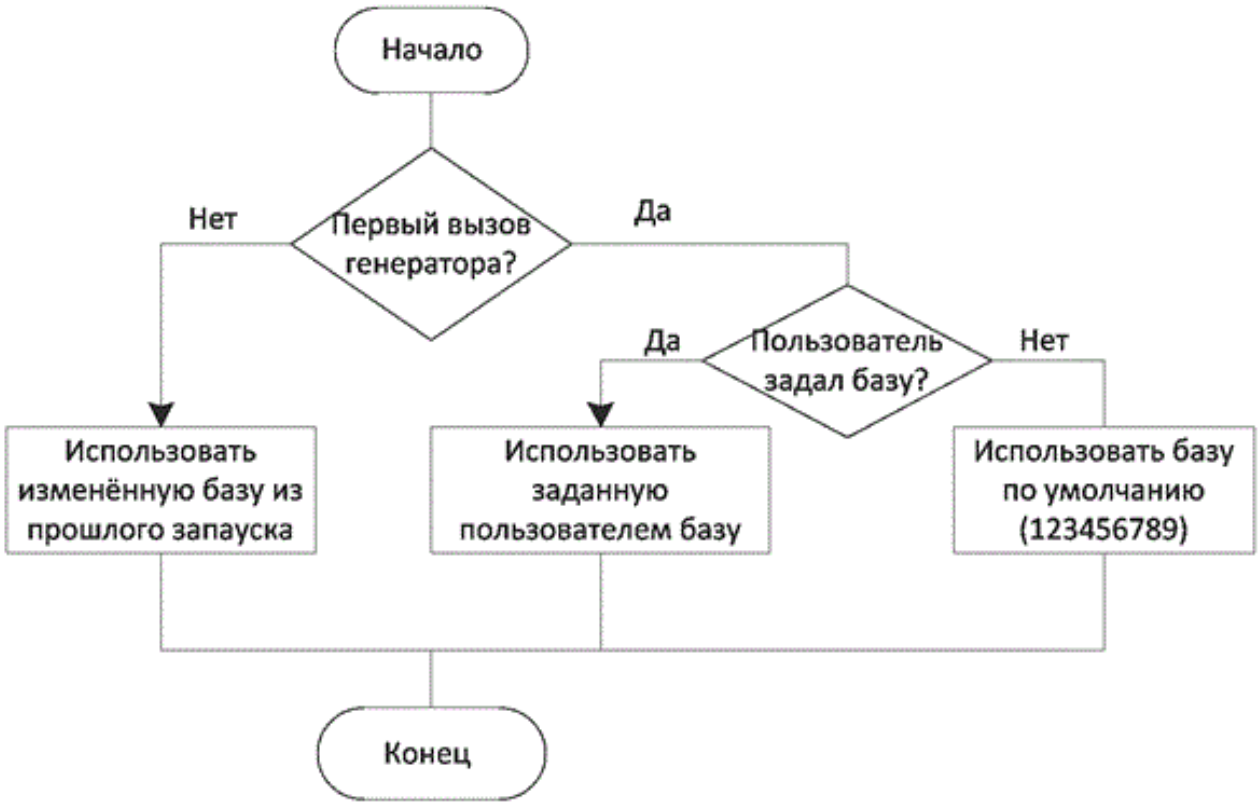
На самом деле в мире техники получить случайное число – большая проблема. Очень часто работа со случайными числами сводятся к двум этапам: найти основу для вычислений (случайность) и преобразовать ее для определенных нужд.

Основа – чаще всего число (база генератора), которое получают из разных источников. Этот этап реализуют по-разному. Часто в качестве базы используют время, т.к. это число постоянно меняется. Но это не идеальное решение: оно меняется медленно. Иногда, когда нужно получить одно-единственное число,

пользователя просят подвигать мышью, чтобы позже на основе тех перемещений курсора получить базу: опыт сложно повторим, поэтому можно считать величину случайной. Одна из компаний, предоставляющих интернет-услуги, которой нужно было ежедневно генерировать множество случайных чисел, использовала погодные данные в качестве баз генератора.

В РДО источником случайности является пользователь, который вводит базу генератора, которую должен использовать генератор псевдослучайных чисел. Пользователь вводит базу для первого числа. Далее РДО получает новые базы на основе введенной пользователем, естественно, по четким правилам. Такой подход позволяет обеспечить повторяемость опытов, а случайность каждого нового эксперимента определяется фантазией пользователя.

Ниже приведен алгоритм определения базы генератора для нового числа из последовательности.



1.4. Основные понятия языка РДО

При выполнении работ, связанных с использованием имитационного моделирования в среде РДО, пользователь оперирует следующими основными понятиями:

Модель - совокупность объектов РДО-языка, описывающих какой-то реальный объект, собираемые в процессе имитации показатели, кадры анимации и графические элементы, используемые при анимации, результаты трассировки.

Прогон - это единая неделимая точка имитационного эксперимента. Он характеризуется совокупностью объектов, представляющих собой исходные данные и результаты, полученные при запуске имитатора с этими исходными данными.

Проект - один или более прогонов, объединенных какой-либо общей целью. Например, это может быть совокупность прогонов, которые направлены на исследование одного конкретного объекта или выполнение одного контракта на имитационные исследования по одному или нескольким объектам.

Объект - совокупность информации, предназначенной для определенных целей и имеющая смысл для имитационной программы.

Объектами исходных данных являются:

- типы ресурсов (с расширением .rtp);
- ресурсы (с расширением .rss);
- образцы операций (с расширением .pat);
- точки принятия решений (с расширением .dpt);
- константы, функции и последовательности (с расширением .fun);
- кадры анимации (с расширением .frm);
- требуемая статистика (с расширением .pmd);
- прогон (с расширением .smr);
- процессы обслуживания (с расширением .prc);
- события (с расширением .evn).

Объекты, создаваемые РДО-имитатором при выполнении прогона:

- результаты (с расширением .pmv);
- трассировка (с расширением .trc).

1.5. Постановка задачи

В настоящее время в РДО чтобы сделать несколько прогонов модели нужно:

1. запустить модель с заданной пользователем базой генератора
2. сохранить результаты прогона в отдельный файл
3. поменять значения базы генератора
4. запустить модель с измененной базой
5. сохранить результаты
6. повторять пункты (3), (4), (5) до нужного количества прогонов

Это неудобно и плохо автоматизируется, поэтому в систему нужно внести возможноть при единственном запуске модели произвести нужное количество прогонов и записать результаты и трассировку с каждого прогона в отдельный файл.

2. ТЕХНИЧЕСКОЕ ЗАДАНИЕ

2.1. Общие сведения

В систему РДО внедряется поддержка множественного прогона модели с разными начальными условиями для каждого из них.

Основной разработчик РДО – кафедра РК-9, МГТУ им. Н.Э. Баумана.

2.2. Назначение и цели развития системы.

Основная цель данного курсового проекта – разработать механизм для возможности описания серии экспериментов имитационной модели с разными начальными условиями для каждого эксперимента.

2.3. Характеристика объекта автоматизации

РДО – система имитационного моделирования, поддерживающая все дискретные подходы имитационного моделирования.

2.4. Требования к системе

Во вкладке симуляции модели (SMR) пользователь может для каждого прогона задать начальные условия, такие как время начала планирования первого события, начальное значение базы генератора, условие остановки модели, новое значение ресурса, новое значение константы и др.

Пример написания кода во вкладке SMR:

```
run_count = 2;

{
        Show_mode    = NoShow
        Show_rate     = 3600.0

        Образец_прихода_клиента.planning( time_now + Интервал_прихода( 30 ) )
        Интервал_прихода.seed = 46783
        Terminate_if Time_now >= 12 * 7 * 60
}

{
        Show_mode    = NoShow
        Show_rate     = 3600.0

        Образец_прихода_клиента.planning( time_now + Интервал_прихода( 30 ) )
        Интервал_прихода.seed = 456
        Terminate_if Time_now >= 12 * 7 * 60
}
```

3. КОНЦЕПТУАЛЬНОЕ ПРОЕКТИРОВАНИЕ

3.1. Компоненты РДО

Система имитационного моделирования РДО является сложной системой. На это указывает сложная иерархическая структура системы со множеством различных связей между компонентами.

Иерархическая структура и модульность системы определяют направление изучения системы сверху вниз. Мы разбиваем нашу систему на подсистемы до тех пор, пока не дойдем до объекта , который не нуждается в дальнейшей деталицации для решения поставленной задачи.

Базовый функционал представленных на листе диаграммы компонентов.

На рис.2 представлена упрощенная диаграмма компонентов.

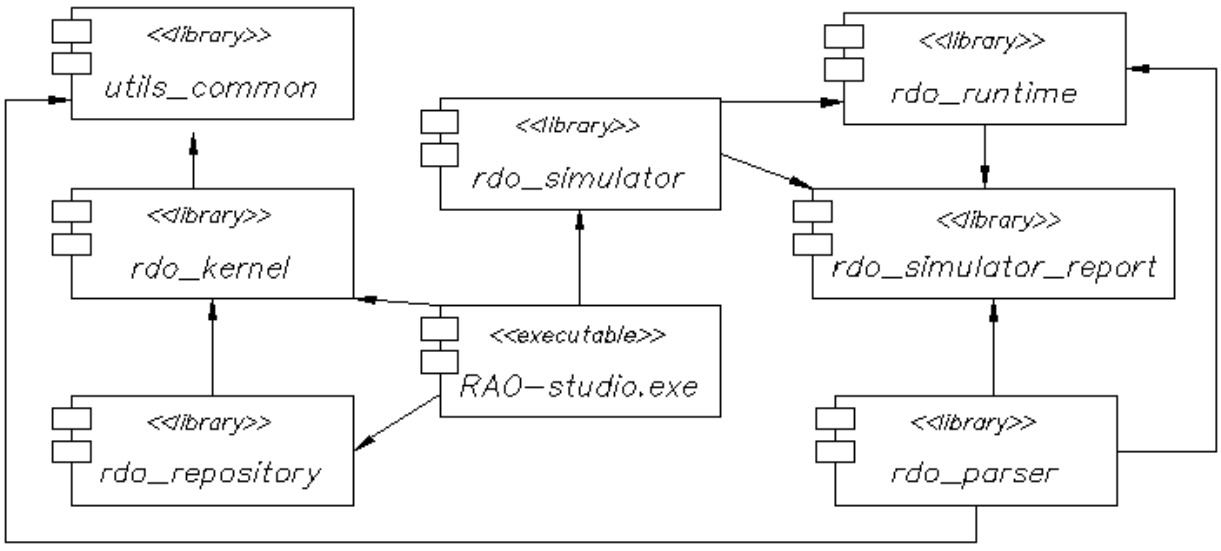


Рис. 2

rdo_kernel реализует функции ядра системы. Не изменяется при разработке системы.

RAO-studio.exe реализует графический интерфейс пользователя. Не изменяется при разработки системы.

rdo_repository реализует управление потоками данных внутри системы и отвечает за хранение и получение информации о модели. Модернизируется при разработке системы.

rdo_simulator управляет процессом моделирования на всех его этапах. Модернизируется при разработке системы.

rdo_parser производит лексический и синтаксический разбор исходных текстов модели, написанной на языке РДО. Модернизируется при разработке системы.

rdo_runtime отвечает за непосредственное выполнение модели, управление базой данных, базой знаний, планирование и выполнение событий, и работу процессов. Не изменяется при разработке системы.

Таким образом, основные изменения должны затронуть модули rdo_parser, rdo_simulator и rdo_repository.

rdo_simulator осуществляет координацию и управление компонентами rdo_runtime и rdo_parser. А объекты компонента rdo_runtime инициализируются при синтаксическом анализе компонентом rdo_parser.

Взаимодействие компонентов rdo_simulator, rdo_repository, rdo_kernel осуществляется с помощью передачи сообщений. Для этого при инициализации атрибутов класса создается контейнер notifies, в который записываются сообщения на которые будет реагировать этот класс. Сообщения передаются с помощью методов broadcastMessage(RDOTreadMessage message, void* param, rbool lock). А обрабатываются сообщения с помощью метода proc(REF(RDOMessageInfo) msg). Т.е. если мы отправляем сообщение, то во все классы , которые подписаны на это сообщение выполнят свой “блок кода”, который реализован внутри метода proc(REF(RDOMessageInfo) msg), для этого сообщения.

3.2. Алгоритм работы цикла

Алгоритм работы цикла серии прогонов представлен на листе блок-схемы.

При прогоне начинается построение модели , которое включает в себя синтаксический анализ и инициализацию числа количества запусков. Если все прошло без ошибок то закрывается компилятор и симулятор, и начинается компиляция i-го прогона, далее модель запускается и прогоняется, если все прошло без ошибок то результаты и трассировка записываются в файлы которые лежат в той же папке что и модель. Они будут иметь формат имени <номер_прогона><име_модели>.pmv и <номер_прогона><име_модели>.trc соответственно. После сохранения результатов у нас закрываются компилятор и симулятор, далее если нужно еще раз прогнать модель, то начинается опять компиляция i-го прогона и так до тех пор пока не выполняться все прогоны.

При компиляции начинается построение модели. Здесь происходит синтаксический анализ модели, если ошибок нет то в окно компиляции

выводяться сообщения что компиляция завершена и предупреждения если они есть, а если есть ошибки то в окно компиляции выводится ошибка с указаниями в какой вкладке и какой строчке.Построение заканчивается и происходит закрытие компилятора и симулятора.

4. ТЕХНИЧЕСКИЙ ЭТАП ПРОЕКТИРОВАНИЯ

4.1. Разработка синтаксиса множества прогонов

В качестве синтаксиса было предложено использовать фигурные скобки : “{” , “}”. Пример:

Инициализация количества запусков

{
[Блок прогонов]
}

Блок множества прогонов может содержать в себе все инструкции для одного прогона, которые задают начальные условия для запуска модели.

Более подробно синтаксис разобран на листе синтаксической диаграммы.

4.2. Изменение компонента rdo_parser

Для возможности обработки новой конструкции в коде модели требуют изменений лексический и синтаксический анализаторы РДО.

В классе RDOParser необходимо добавить новый метод, который будет при каждом i-ом прогоне отправлять в RDOSMR текущее значение прогона и будет компилировать этот прогон.

В классе RDOSMR необходимо добавить новые методы, благодаря которым синтаксический анализатор сможет считывать только скобки нужного нам прогона.

4.3. Изменение компонента rdo_repository

Для возможности записи результатов и трассировки после каждого прогона в классе RDOThreadRepository необходимо добавить счетчик который будет соответствовать номеру текущего прогона и отправлять этот номер в название файла с расширением .pmv и .trc

4.4. Изменение компонента rdo_simulator

При прогоне модели классу RDOThreadSimulator необходимо принять значение количества запусков из RDOParser и реализовать цикл для запуска каждого прогона. При запуске первого прогона должна пройти полная синтаксическая проверка всей модели, после должен остановиться компилятор и симулятор и после запуститься по новой, компилятор должен прочитать значения только для первого прогона, далее прогоняется модель и записываются результаты и трассировка в отдельные файлы, закрывается компилятор, симулятор и запускается следующий прогон.

При компиляции модели должен быть произведен полный синтаксический анализ модели. Если есть ошибки или их нету, то выдать соответсвующее сообщение. После компилятор и симулятор закрываются.

Для реализации задачи в классе RDOThreadSimulator нужно ввести счетчик который будет равен номеру прогона, он нужен для того тчобы при первом запуске компилятор считал только первый блок прогона, а не все блоки. Что инициализировать число прогонов только 1 раз нужно изменить метод parseModel(), добавив в него новый метод getInitialRunCount(), который считывает значение числа прогонов из RDOParser. И нужно создать новый метод parseModel_i() без инициализации количества запусков. Т.е. для синтаксического анализа всей модели применяется parseModel(), а для анализа i-го прогона parseModel_i().

5. РАБОЧИЙ ЭТАП ПРОЕКТИРОВАНИЯ

5.1. Синтаксический анализ

В первую очередь необходимо добавить новые термальные символы в лексический анализатор РДО и нетермальные символы в грамматический анализатор.

В лексическом анализаторе(flex) я добавил новый токен RDO_Run_Count :

```
run_count           return(RDO_Run_Count);
Run_count           return(RDO_Run_Count);
```

Этот токен необходимо также добавить в синтаксический анализатор bison:

```
%token     RDO_Run_Count
```

Далее нужно реализовать чтение этого значения с РДО:

Для этого введем новую переменную в RDOSMR m_runCount и введем новый метод считывающий его:

```
void RDOSMR::setRunCount(ruint value)
{
     m_runCount = value;
}
```

Для чтения с РДО в синтаксическом анализаторе добавим код:

```
smr_cond
     : /* empty */
     | smr_cond smr_multirun
     | smr_cond RDO_Run_Count '=' RDO_INT_CONST ';'
     {
          LPRDOSMR pSMR = PARSER->getSMR();
          ASSERT(pSMR);
          rsint count = PARSER->stack().pop<RDOValue>($4)-
>value().getInt();
          if (count < 0)
          {
               PARSER->error().error(@4, _T("Число прогонов должно
быть больше нуля"));
          }
          pSMR->setRunCount(ruint(count));
     }
     |…
```

Этот код считывает значение m_runCount и сохраняет его в контейнере RDOSMR.

Далее необходимо прочитать все инструкции находящиеся в скобочках от “{” до “}”:

```
smr_multirun
     : '{' smr_multirun_body '}'
     {
          PARSER->getSMR()->setIterator();
     }
     ;

smr_multirun_body
     : /* empty */
     | smr_multirun_body smr_multirun_body_desc
     ;

smr_multirun_body_desc
     : RDO_IDENTIF '.' RDO_Planning '(' arithm_list ')'
     {
          LPRDOSMR pSMR = PARSER->getSMR();
          ASSERT(pSMR);
          if(pSMR->setCheck() )
          {
               tstring    eventName          = PARSER-
>stack().pop<RDOValue>($2)->value().getIdentificator();
```

```
                LPArithmContainer pArithmList = PARSER-
>stack().pop<ArithmContainer>($5);
                LPRDOEvent pEvent             = PARSER-
>findEvent(eventName);
                if (!pEvent)
                {
                     PARSER->error().error(@2, rdo::format(_T("Попытка
запланировать неизвестное событие: %s"), eventName.c_str()));
                }

                ArithmContainer::Container::const_iterator arithmIt =
pArithmList->getContainer().begin();
                if (arithmIt == pArithmList->getContainer().end())
                {
                     PARSER->error().error(@1, rdo::format(_T("Не
указано время планирования события: %s"), eventName.c_str()));
                }

                LPRDOFUNArithm pTimeArithm = *arithmIt;
                ASSERT(pTimeArithm);
                ++arithmIt;

                LPArithmContainer pParamList =
rdo::Factory<ArithmContainer>::create();
                ASSERT(pParamList);

                while (arithmIt != pArithmList->getContainer().end())
                {
                     pParamList->addItem(*arithmIt);
                     ++arithmIt;
                }

                rdo::runtime::LPRDOCalc pCalcTime = pTimeArithm-
>createCalc();
                pCalcTime->setSrcInfo(pTimeArithm->src_info());
                ASSERT(pCalcTime);

                LPIBaseOperation pBaseOperation = pEvent-
>getRuntimeEvent();
                ASSERT(pBaseOperation);

                rdo::runtime::LPRDOCalcEventPlan pEventPlan =
rdo::Factory<rdo::runtime::RDOCalcEventPlan>::create(pCalcTime);
                pEventPlan->setSrcInfo(RDOParserSrcInfo(@1, @6,
rdo::format(_T("Планирование события %s в момент времени %s"),
eventName.c_str(), pCalcTime->srcInfo().src_text().c_str())));
                ASSERT(pEventPlan);
                pEvent->setParamList(pParamList);
                pEventPlan->setEvent(pBaseOperation);
                pEvent->setInitCalc(pEventPlan);
           }
     }
     | RDO_IDENTIF '.' RDO_Seed '=' RDO_INT_CONST
     {
```

```
            LPRDOSMR pSMR = PARSER->getSMR();
            ASSERT(pSMR);
            if(pSMR->setCheck() )
            {
                 pSMR->setSeed(PARSER->stack().pop<RDOValue>($2)-
>src_info(), PARSER->stack().pop<RDOValue>($5)->value().getInt());
            }
      }
      | RDO_IDENTIF '.' RDO_Seed '=' error
      {
            PARSER->error().error(@4, @5, _T("Ожидается база
генератора"));
      }
      | RDO_IDENTIF '.' RDO_Seed error
      {
            PARSER->error().error(@4, _T("Ожидается '='"));
      }
      | error
      {
            PARSER->error().error(@1, _T("Неизвестная ошибка"));
      }
      | RDO_Terminate_if fun_logic
      {
            LPRDOSMR pSMR = PARSER->getSMR();
            ASSERT(pSMR);
            if(pSMR->setCheck() )
            {
                 LPRDOFUNLogic pLogic = PARSER-
>stack().pop<RDOFUNLogic>($2);
                 ASSERT(pLogic);
                 PARSER->getSMR()->setTerminateIf(pLogic);
            }
      }
      | RDO_Terminate_if error
      {
            PARSER->error().error(@1, @2, _T("Ошибка логического
выражения в терминальном условии"));
      }
      ;
```

Чтобы читались определенные скобки для нужного прогона в каждой инструкции есть метод проверки setCheck() находящийся в RDOSMR, который проверяет соответствуют ли читающиеся на данный момент скобки прогону. Чтобы узнай какой прогон идет нужно из RDOThreadSimulator отправить в RDOSMR значение счетчика показывающий номер текущего прогона. Для этого в RDOSMR добавим новый метод setCount(ruint value) и новую переменную

m_run в которую будет записывать номер. Если условие выполняется то выполняется код внутри. Когда анализатор понимает что он прочел все что внутри скобок и дошел до закрывающей скобки происходит инкриминация счетчика. Счетчик показывает какие скобки читаются на данный момент. Чтобы это реализовать в классе RDOSMR нужно добавить новые методы setCheck(), setIterator() и добавить новую переменную m_iterator.

Реализация кода:

```
RDOSMR::RDOSMR()
     : m_frameNumber    (1 )
     , m_showRate       (60)
     , m_runStartTime   (0 )
     ,
m_traceStartTime(rdo::runtime::RDOSimulatorTrace::UNDEFINE_TIME)
     , m_traceEndTime
(rdo::runtime::RDOSimulatorTrace::UNDEFINE_TIME)
     , m_showMode       (rdo::service::simulation::SM_NoShow
)
     , m_run            (0 )
     , m_iterator       (0 )
     , m_runCount       (0 )
{}

void RDOSMR::setCount(ruint value)
{
     m_run = value;
}

rbool RDOSMR::setCheck()
{
     if (m_run == m_iterator)
     {
          return true;
     }
     else
     {
          return false;
     }
}

void RDOSMR::setIterator()
{
     ++m_iterator;
}
```

В заголовочном файле соответственно определить их:

```
ruint                                   m_runCount;
ruint                                   m_run;
ruint                                   m_iterator;


//! число экспериментов (прогонов) в серии
void  setRunCount       (ruint  value);
//! номер выполняемого прогона
void  setCount          (ruint  value);
rbool setCheck          ();
void  setIterator       ();
```

5.2. Изменения класса RDOThreadSimulator

В RDOThreadSimulator вводим новый метод parseModel_i() и getInitialRunCount(). Первый служит для компиляции i-го прогона, который содержит в себе номер прогона. И второй метод служащий для инициализации количества прогонов.

```
rbool RDOThreadSimulator::parseModel_i()
{
     terminateModel();
     closeModel();

     m_pParser =
rdo::Factory<rdo::compiler::parser::RDOParserModel>::create();
     ASSERT(m_pParser);
     m_pParser->init();
     m_pRuntime = m_pParser->runtime();
     ASSERT(m_pRuntime);

     try
     {
          m_exitCode = rdo::simulation::report::EC_OK;
          m_pParser->parse(m_run);
     }
     catch (REF(rdo::compiler::parser::RDOSyntaxException))
     {
          m_exitCode = rdo::simulation::report::EC_ParserError;
          broadcastMessage(RT_SIMULATOR_PARSE_ERROR);
          closeModel();
          return false;
     }
     catch (REF(rdo::runtime::RDORuntimeException) ex)
     {
```

```
            tstring mess = ex.getType() + _T(" : ") + ex.message();
            broadcastMessage(RT_SIMULATOR_PARSE_STRING, &mess);
            m_exitCode = rdo::simulation::report::EC_ParserError;
            broadcastMessage(RT_SIMULATOR_PARSE_ERROR);
            closeModel();
            return false;
        }

        m_showMode = getInitialShowMode();
        m_showRate = getInitialShowRate();

        broadcastMessage(RT_SIMULATOR_PARSE_OK);

        return true;
        }
```

Привыполнении этого кода создается Runtime и Parser, и производится синтаксический разбор. И здесь же отправляется атрибут m_run который равен значению текущего прогона в parser, для дальнейшего его использования там.

Здесь же реализован цикл прогонов модели. Для того чтобы прочитались только первые скобки “{” ”}”: в функции runModel() введено условие, что если это первый прогон то закрываются parser и runtime и комплируется первый прогон и увеличивается на единицу m_run.

```
void RDOThreadSimulator::runModel()
{
    if (m_run == 0)
    {
        closeModel();
        parseModel_i();
        ++m_run;
    }

    ASSERT(m_pParser );
    ASSERT(m_pRuntime);

    m_pParser->error().clear();
    m_exitCode = rdo::simulation::report::EC_OK;
    m_pRuntime->setStudioThread(kernel->studio());
    m_pThreadRuntime =
rdo::Factory<rdo::runtime::RDOThreadRunTime>::create();
}
```

Далее когда уже записались результаты и модель остановлена происходит проверка на условие : m_run < m_runCount? Оно находиться в сообщении RT_THREAD_STOP_AFTER.

```
case RT_THREAD_STOP_AFTER:
        {
            if (msg.from == m_pThreadRuntime.get())
            {
                m_exitCode = m_pRuntime->m_whyStop;
                if (!m_pThreadRuntime->runtimeError())
                {
                    //! Остановились сами нормально
                    broadcastMessage(RT_SIMULATOR_MODEL_STOP_OK);
                    closeModel();
                    if (m_run < m_runCount)
                    {
                        parseModel_i();
                        runModel();
                        ++m_run;
                    }
                    else
                    {
                        m_run = 0;
                    }
                }
                else
                {
                    //! Остановились сами, но не нормально

    broadcastMessage(RT_SIMULATOR_MODEL_STOP_RUNTIME_ERROR);
                    closeModel();
                }
……
```

Если условие выполняется то происходит следущий запуск модели, если нет то итератор номера запуска обнуляется.

5.3. Изменения класса RDOParser

В RDOParser нужно добавить новый метод

```
void parse(ruint count);
```

В него присылается наше значение m_run из RDOThreadSimulator и он отправляет его в RDOSMR с помощью метода setCount(ruint value)

```
void RDOParser::parse()
{
	RDOParserContainer::Iterator it = begin();
	while (it != end())
	{
		m_parser_item = it->second;
		it->second->parse(this);
		m_parser_item = NULL;
		it++;
	}
}

void RDOParser::parse(ruint count)
{
	getSMR()->setCount(count);
	parse();
}
```

5.4. Изменения класса RDOThreadRepository

В класс RDOThreadRepository нужно добавить два счетчика m_count и m_firstStart которые будут соответствовать значению текущего прогона и это значение мы будем отправлять в название файла результатов и трассировки.

Для этого в заголовочном файле определим их:

```
tstring         m_modelName;
tstring         m_modelPath;
rbool           m_hasModel;
rdo::ofstream   m_traceFile;
FileList        m_files;
rbool           m_realOnlyInDlg;
ProjectName     m_projectName;
SystemTime      m_systemTime;
ruint           m_firstStart;
ruint           m_count;
```

В файле rdo_repository.cpp изменим методы beforeModelStart() и stopModel()

beforeModelStart() :

```
void RDOThreadRepository::beforeModelStart()
{
	m_systemTime = boost::posix_time::second_clock::local_time();

	if (m_traceFile.is_open())
	{
		m_traceFile.close();
	}
	if (m_files[rdoModelObjects::TRC].m_described)
	{
		++m_firstStart;
		tstring m_buffer = rdo::format("%i", m_firstStart);
		m_traceFile.open((m_modelPath + m_buffer +
getFileExtName(rdoModelObjects::TRC)).c_str(), std::ios::out | std::ios::binary);
		if (m_traceFile.is_open())
		{
			writeModelFilesInfo(m_traceFile);
			rdo::textstream model_structure;
			sendMessage(kernel->simulator(), RT_SIMULATOR_GET_MODEL_STRUCTURE,
&model_structure);
			m_traceFile << std::endl << model_structure.str() << std::endl;
			m_traceFile << _T("$Tracing") << std::endl;
		}
	}
}
```

stopModel() :

```
void RDOThreadRepository::stopModel()
{
	if (m_traceFile.is_open())
	{
		m_traceFile.close();
	}
	if (m_files[rdoModelObjects::PMV].m_described)
	{
		m_count++;
		//char buffer[100];
		//_itoa (m_count, buffer, 10);
		tstring buffer = rdo::format("%i", m_count);
		rdo::ofstream results_file;
		results_file.open((m_modelPath + buffer +
getFileExtName(rdoModelObjects::PMV)).c_str(), std::ios::out | std::ios::binary );
		if (results_file.is_open())
		{
			writeModelFilesInfo(results_file);
			rdo::textstream stream;
			sendMessage(kernel->simulator(),
RT_SIMULATOR_GET_MODEL_RESULTS_INFO, &stream);
			results_file << std::endl << stream.str() << std::endl;
			stream.str(_T(""));
			stream.clear();
			sendMessage(kernel->simulator(), RT_SIMULATOR_GET_MODEL_RESULTS,
&stream);
			results_file << std::endl << stream.str() << std::endl;
		}
	}
}
```

6. Вывод

В рамках данного курсового проекта были получены следующие результаты:

1. В систему имитационного моделирования добавлена возможность множественного прогона модели с различными начальными условиями и параметрами.

2. На этапе концептуального проектирования с помощью диаграммы классов UML показана часть внутреннего устройства РДО, и выделены те компоненты которые нужно изменить в ходе работы.

3. На этапе технического проектирования в синтаксический анализатор был разработан и добавлен синтаксис для множественного прогона модели, который представлен на синтаксической диаграмме.С помощью блок-схемы разработан алгоритм, реализующий множественный прогон модели.С помощью диаграммы классов разработа архитектура новой системы.

4. На этапе рабочего проектирования написан программный код для реализации поставленной задачи. Изменены такие компоненты как : rdo_parser, rdo_simulator, rdo_repository.

5. При каждом прогоне модели создаются отдельные файлы для результатов и трассировки с расширением .pmv и .trc соответственно

6. Для демонстрации новых возможностей написана модель в системе РДО. Написаны модели автотестов.

Приложение 1. Код модели

Вкладка RTP :

```
$Resource_type Парикмахерские: permanent
$Parameters
   состояние_парикмахера : ( Свободен, Занят )
   количество_в_очереди  : integer
   количество_обслуженных: integer
$End
```

Вкладка RSS :

```
$Resources
    Парикмахерская: Парикмахерские trace Свободен 0 0
$End
```

Вкладка EVN :

```
$Pattern Образец_прихода_клиента : event
$Relevant_resources
    _Парикмахерская: Парикмахерская Keep
$Body
_Парикмахерская
    Convert_event
         Образец_прихода_клиента.planning( time_now +
Интервал_прихода( 30 ) );
         количество_в_очереди++;
$End
```

Вкладка PAT :

```
$Pattern Образец_обслуживания_клиента : operation
$Relevant_resources
    _Парикмахерская: Парикмахерская Keep Keep
$Time = Длительность_обслуживания( 20, 40 )
$Body
_Парикмахерская
    Choice from _Парикмахерская.состояние_парикмахера ==
Свободен and _Парикмахерская.количество_в_очереди > 0
    Convert_begin
         количество_в_очереди--;
         состояние_парикмахера = Занят;
    Convert_end
         состояние_парикмахера  = Свободен;
         количество_обслуженных++;
$End
```

Вкладка DPT :

```
$Decision_point model: some
$Condition NoCheck
$Activities
     Обслуживание_клиента: Образец_обслуживания_клиента
$End
```

Вкладка FUN :

```
$Sequence Интервал_прихода : real
$Type = exponential 123456789
$End

$Sequence Длительность_обслуживания : real
$Type = uniform 123456789
$End
```

Вкладка SMR :

```
Show_mode      = NoShow
Show_rate      = 3600.0

run_count = 2;

{
     Образец_прихода_клиента.planning( time_now +
Интервал_прихода( 30 ) )
     Интервал_прихода.seed = 46783
     Terminate_if Time_now >= 12 * 7 * 60
}

{
     Образец_прихода_клиента.planning( time_now )
     Интервал_прихода.seed = 987654323
     Terminate_if Time_now >= 12 * 7 * 60
}
```

Вкладка PMD :

```
$Results
     Занятость_парикмахера : watch_state
Парикмахерская.состояние_парикмахера == Занят
     Длина_очереди         : watch_par
Парикмахерская.количество_в_очереди
     Всего_обслужено       : get_value
Парикмахерская.количество_обслуженных
     Пропускная_способность: get_value
Парикмахерская.количество_обслуженных / Time_now * 60
     Длительность_работы   : get_value   Time_now / 60
$End
```