1.Введение

Любая ИМ (имитационная модель) – это программа для ЭВМ, поэтому при ее создании этап программирования является одним из основных. Языки имитационного моделирования, за счет снижения гибкости и универсальности, позволяют создать ИМ на несколько порядков быстрее, чем языки алгоритмического программирования и не требуют работы системных программистов. Они обладают двумя наиболее важными достоинствами: удобством программирования и концептуальной выразительностью. Последнее достоинство позволяет четко и ясно описывать различные понятия, что наиболее важно на стадии моделирования и для определения общего подхода к изучению исследуемой системы.

В системе моделирования РДО существует два концептуально важных понятия: база данных и база знаний. В каждый момент времени состояние системы описывается ресурсами системы, которые заводит в РДО разработчик модели (робот, станок, детали и т.д.). Каждый ресурс имеет свои параметры (атрибуты). Ресурсы системы хранятся в базе данных. Можно сказать, что текущее содержание базы данных однозначно определяет состояние моделируемой системы в данный момент времени моделирования. Это – «фотография модели» в данный момент времени. База знаний – это все действия, которые могут происходить в системе и менять ее состояние, изменяя параметры ресурсов. Знания закладываются в базу знаний в виде модифицированных продукционных правил. В результате мы получаем систему ИМ, которая позволяет при создании моделей использовать знания эксперта в виде продукционных правил.

Однако, в любом случае, увеличение гибкости системы ведет к увеличению сложности. Здесь сложность проявляется в процессе написания моделей. Язык РДО – своеобразный язык программирования, достаточно сложный в освоении. Например, для решения задач моделирования систем массового обслуживания (СМО), существует свой процессно-ориентированный подход. Такие задачи решаются довольно просто с использованием языков этого подхода. Решение можно представить и на РДО, но оно будет гораздо сложнее, поскольку это язык моделирования низкого уровня. Скорее всего, никто не станет решать подобные задачи на РДО. А задачи моделирования СМО составляют 90-95% всех задач решаемых в имитационном моделировании.

Интеграции разных подходов моделирования в рамках одной системы интересна и перспективна. Основная цель упростить создание моделей на РДО.

1. Предпроектное исследование

1.1 Основные подходы к построению ИМ

Системы имитационного моделирования СДС в зависимости от способов представления процессов, происходящих в моделируемом объекте, могут быть дискретными и непрерывными, пошаговыми и событийными, детерминированными и статистическими, стационарными и нестационарными. Этапы проведения имитационного эксперимента представлены на следующем рисунке[1, стр.29]:

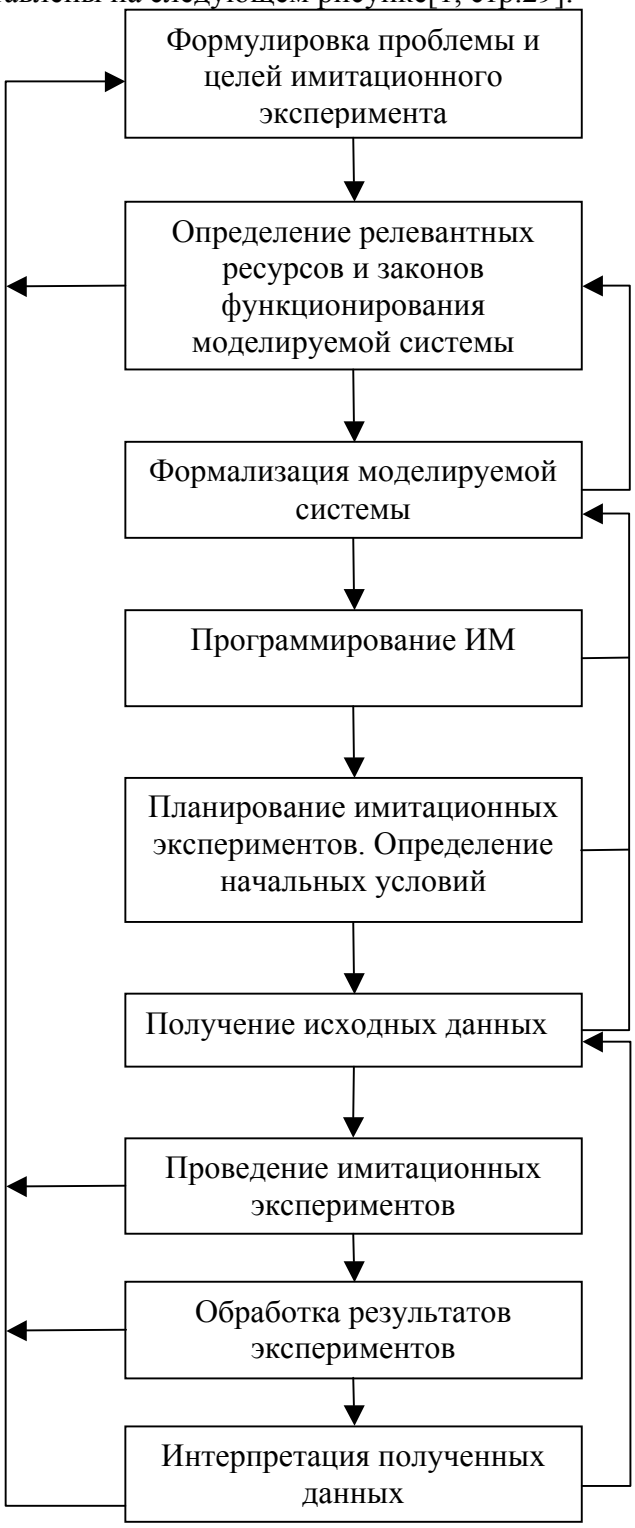


Рис. 1. Этапы имитационного моделирования

Рассмотрим основные моменты этапа создания ИМ. Чтобы описать функционирование СДС надо описать интересующие нас события и действия, после чего создать алфавит, то есть дать каждому из них уникальное имя. Этот алфавит определяется как природой рассматриваемой СДС, так и целями ее анализа. Следовательно, выбор алфавита событий СДС приводит к ее упрощению – не рассматриваются многие ее свойства и действия не представляющие интерес для исследователя.

Событие СДС происходит мгновенно, то есть это некоторое действие с нулевой длительностью. Действие, требующее для своей реализации определенного времени, имеет собственное имя и связано с двумя событиями – начала и окончания. Длительность действия зависит от многих причин, среди которых время его начала, используемые ресурсы СДС, характеристики управления, влияние случайных факторов и т.д. В течение времени протекания действия в СДС могут возникнуть события, приводящие к преждевременному завершению действия. Последовательность действий образует процесс в СДС (Рис. 2.).

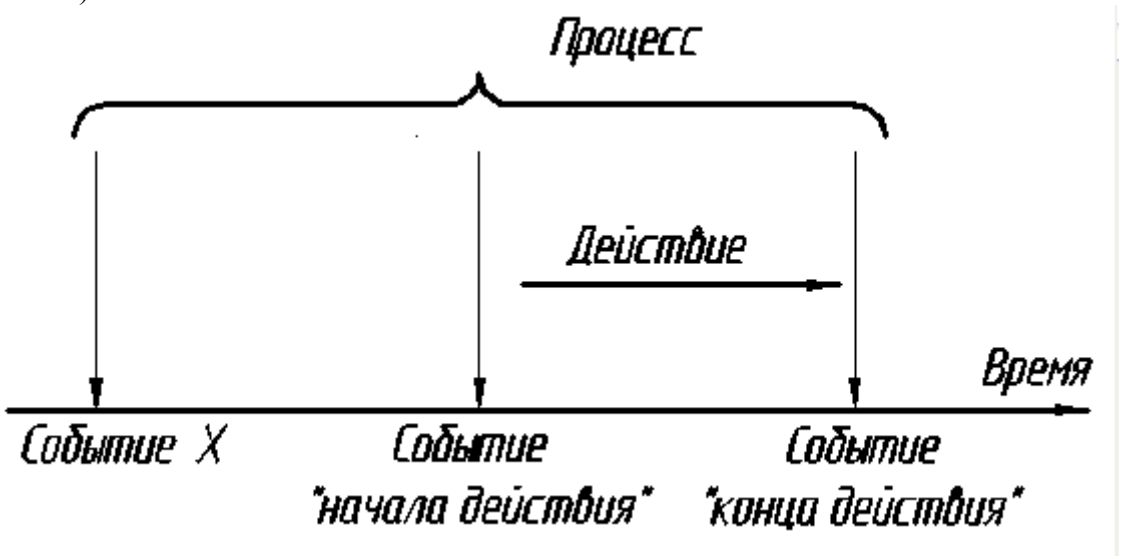


Рис. 2. Взаимосвязь между событиями, действием и процессом

В соответствии с этим выделяют три альтернативных методологических подхода к построению ИМ: событийный, подход сканирования активностей и процессно-ориентированный.

1.2. Событийный подход к построению ИМ

При событийном подходе исследователь описывает события, которые могут изменять состояние системы, и определяет логические взаимосвязи между ними. Начальное состояние устанавливается путем задания значений переменных модели и параметров генераторам случайных числе. Имитация происходит путем выбора из списка будущих событий ближайшего по времени и его выполнении. Выполнение события приводит к изменению состояния системы и генерации будущих событий, логически связанных с выполняемым. Эти события заносятся в список будущих событий и упорядочиваются в нем по времени наступления. Например, событие начала заправки автомобиля на бензозаправочной станции приводит к появлению в списке будущих событий события окончания обслуживания, которое должно наступить в момент времени равный текущему времени плюс время, требуемое на заливку бензина в автомобиль. В событийных системах модельное время фиксируется только в моменты изменения состояний[1, стр. 32.].

1.3. Подход сканирования активностей

При использовании подхода сканирования активностей разработчик описывает все действия, в которых принимают участие элементы системы, и задает условия, определяющие начало и завершение действий. После каждого продвижения имитационного времени условия всех возможных действий проверяются и если условие выполняется, то происходит имитация соответствующего действия. Выполнение действия приводит к изменению состояния системы и возможности выполнения новых действий. Например, для начала действия заправки автомобиля бензином необходимо наличие свободной бензоколонки, наличие прибывшего автомобиля и наличие свободного рабочего заправщика. Если хотя бы одно из этих действий не выполнено, действие не начинается[1,стр.38.].

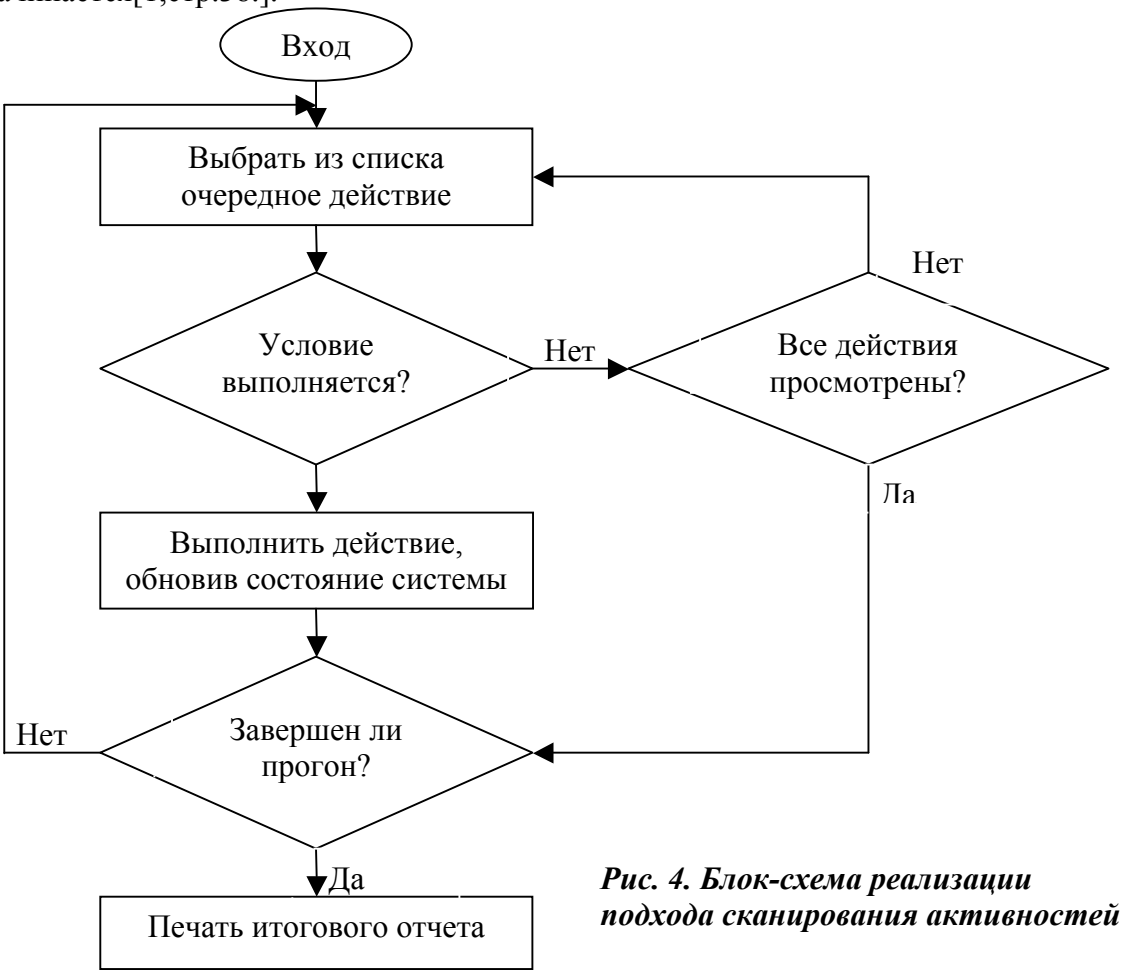


Рис. 4. Блок-схема реализации подхода сканирования активностей

1.4. Процессно-ориентированный подход

В процессно-ориентированном подходе исследователь описывает последовательность компонентов модели, которые возникают по определенной схеме. Логика возникновения определенных компонентов задается одним оператором языка. Имитатор моделирует процесс продвижения активных элементов через систему, который сопровождается соответствующей последовательностью событий. Процессно-ориентированный подход сочетает в себе элементы событийного подхода и подхода сканирования активностей. Например, оператор генерации прибытия автомобилей, оператор занятия автомобилем бензоколонки, либо оператор организации очереди на заправку. Таким образом, пользователь должен описать процесс, с которым сталкиваются в системе некоторые компоненты.

Ниже приведен простейший пример модели системы обслуживания с одним ресурсом и очередью, написанной на процессно-ориентированном языке GPSS:

```
GENERATE    20,5        ;Приход в систему заявок на обслуживание
SEIZE       resourse    ;Занятие ресурса обслуживания
ADVANSE     15,4        ;Обслуживание заявки
RELIESE     resourse    ;Освобождение ресурса обслуживания
TERMINATE   1           ;Удаление заявки из системы обслуживания
```

Здесь приход транзактов (заявок на обслуживание) определяет блок GENERATE. 20 – средний интервал времени между приходами, 5 – половина поля допуска равномерно распределенного интервала времени. Блоком SEIZE определяется занятие ресурса во время прихода в него транзакта. Блок ADVANSE осуществляет задержку транзакта, что тем самым моделирует обработку заявки ресурсом. 15 – задержка на время обслуживания, 4 – половина поля допуска равномерно распределенного интервала времени задержки.

Оператор RELIESE освобождает ресурс после окончания обслуживания заявки. TERMINATE – удаление транзактов (заявок на обслуживание).

2 Концептуальное проектирование

2.1 Проблема интеграции подходов моделирования в РДО

Оба подхода предназначены для решения своего класса задач.

Из всего класса СДС можно выделить подкласс систем массового обслуживания (СМО)(рис. 5) .

Рис. 5 Сложные дискретные системы

Задачи моделирования СМО составляют 90 – 95% задач моделирования и решаются преимущественно на основе процессно-ориентированного подхода. На языке РДО можно писать модели СМО, но это не оправдывает себя. Нет необходимости на столь низком уровне описывать ресурсы и операции моделируемой системы.

Любая СМО характеризуется:

1) входящим потоком требований или заявок, которые поступают на обслуживание;
2) дисциплиной постановки в очередь, и выбором из нее;
3) правилом, по которому осуществляется обслуживание;
4) выходящим потоком требований;
5) режимом работы.

Язык РДО – язык имитационного моделирования низкого уровня, поскольку система работает со знаниями, описываемыми в виде продукций. Эта особенность языка позволяет моделировать управление сложными дискретными системами, что сложно сделать, используя языки процессно-ориентированного подхода.

Любую имитационную модель СДС можно представить в виде трех взаимодействующих моделей: модель управляемого объекта, модель системы управления и модель внутренних случайных возмущений (Рис.6.) Модель управляемого объекта в свою очередь, всегда можно представить в виде СМО. Не будем рассматривать внешние возмущения, поскольку в РДО они моделируются просто, как нерегулярные события. Тогда решая задачу средствами РДО, сталкиваемся с проблемой моделирования СМО. С другой стороны РДО корректно решает задачу создания модели системы управления. И наоборот.

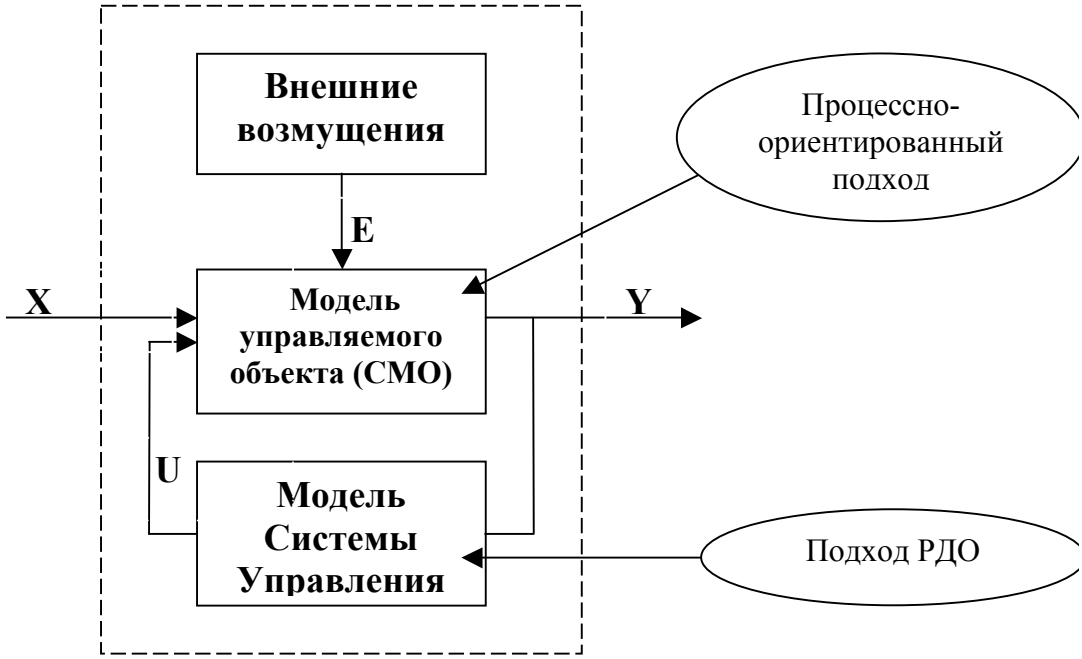


Рис.6 Состав имитационной модели сложной системы

В системе моделирования РДО существует два концептуально важных понятия: база данных и база знаний. В каждый момент времени состояние системы описывается ресурсами системы, которые заводит в РДО разработчик модели (робот, станок, детали и т.д.). Каждый ресурс имеет свои параметры (атрибуты). Ресурсы системы хранятся в базе данных. Можно сказать, что текущее содержание базы данных однозначно определяет состояние моделируемой системы в данный момент времени моделирования. Это – «фотография модели» в данный момент времени. База знаний – это все действия, события, которые могут происходить в системе и менять ее состояние, изменяя параметры ресурсов.

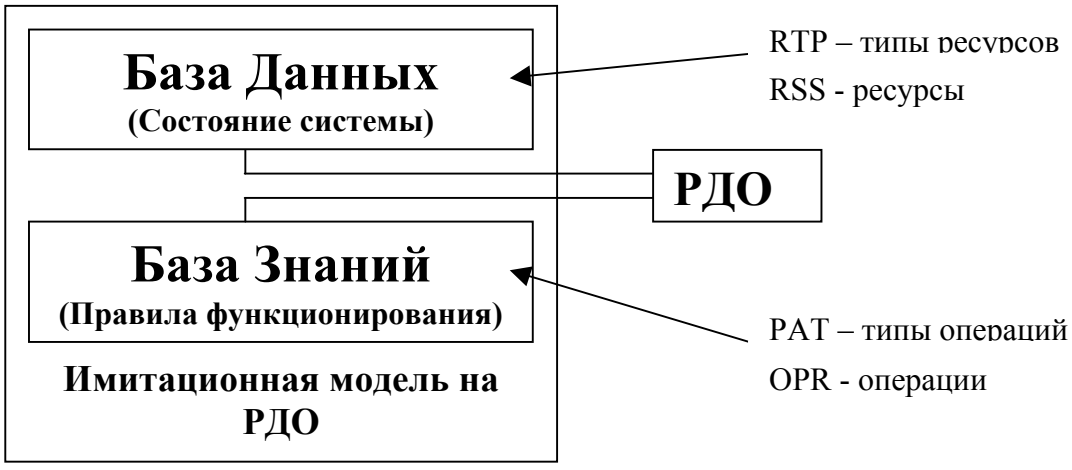


Рис. 7. Структура интеллектуальной системы имитационного моделирования – РДО

Тип ресурса – это некоторый шаблон определенного класса ресурсов, однозначно определяющий набор параметров ресурсов этого класса. Тип ресурса определяет структуру базы знаний. Для описания состояния системы в модельном времени, необходимо заполнить базу данных, которая определяет состояние системы в текущий

момент времени и базу знаний, которая изменяет это состояние с течением модельного времени в зависимости от заложенных в нее знаний. Заполнить базу данных – значит определить ее структуру (типы всех ресурсов моделируемой системы) и завести все ресурсы этой системы (задать параметры ресурсов в момент начала моделирования). Ресурсы, принадлежащие к одному типу, наследуют общие свойства этого типа. Отношение наследования может использоваться как для отражения общности ресурсов, так и для идентификации ассоциативных связей.

Рис. 8. Понятие ресурса и типа ресурса.

Что касается базы знаний, она подобно базе данных имеет свою структуру, которая определяется набором знаний заложенных в нее. Знания представляют собой образцы операций (действий). Они представляются в РДО в виде модифицированных правил продукции: ЕСЛИ(условие), ТО1(событие 1) ЖДАТЬ (время t) ТО2(событие 2). Это введение в РДО-методе позволяет устранить недостатки продукционных систем, связанные с их статичностью, сохраняя в то же время известные преимущества продукционных систем: универсальность (возможность описания широкого класса СДС применительно к различным задачам исследования); независимость формата продукционного правила и механизма поиска решений от физического смысла представляемых знаний; модульность (продукционные правила независимы друг от друга, что позволяет вводить или удалять правило из базы знаний, не затрагивая остальные); соответствие процедурного характера описания знаний в продукционных системах дискретным процессам, имеющим место в СДС, что позволяет естественно использовать их для построения последовательности некоторых действий.

Цели использования методов ИИ в имитационном моделировании – 1) выйти за рамки алгоритмического (жесткого) подхода в процессе принятия решений при моделировании, чтобы можно было автоматизировать ту часть процесса СДС, где используются знания человека;2)сделать процесс моделирования максимально гибким по способам представления информации о СДС.

Чтобы показать преимущество процессно-ориентированного подхода в решении задач моделирования СМО, решим задачу моделирования простейшей СМО разными подходами и сравним два решения. Критерием сравнения является объем исходного кода модели. В качестве процессно-ориентированного языка воспользуемся средствами GPSS.

Задача формулируется следующим образом: Интервалы прихода клиентов в парикмахерскую с одним креслом распределены равномерно на интервале 18±6мин. Время стрижки также распределено равномерно на интервале 16±4мин. Клиенты приходят в парикмахерскую, стригутся в порядке очереди: «первым пришел – первым обслужился». Необходимо построить модель парикмахерской, которая должна обеспечить сбор статистических данных об очереди. Промоделировать работу обслуживания 30 клиентов.

Построение модели на процессно-ориентированном языке GPSS.

Порядок блоков в модели соответствует порядку фаз, в которых клиент оказывается при движении в реальной системе:

1) клиент приходит;
2) если необходимо, ждет своей очереди;
3) садится в кресло парикмахера;
4) парикмахер обслуживает клиента;
5) клиент уходит из парикмахерской.

Единица модельного времени – 1 минута.
Исходный текст модели:

```
;Первый сегмент модели:
        GENERATE 18,6          ;Приход клиентов
        QUEUE    BARBERQ       ;Присоединение к очереди
        SEIZE    BARBER        ;Переход в кресло парикмахера
        DEPART   BARBERQ       ;Выход из очереди
        ADVANCE  16,4          ;Обслуживание у парикмахера
        RELEASE  BARBER        ;Освобождение парикмахера

;Второй сегмент модели:
        TERMINATE 1            ;Уменьшить значение счетчика завершения на 1
        START    30            ;Значение счетчика завершения в момент начала
                               ;моделирования.
```

Построение модели на языке РДО.

Для построения модели такой системы на языке РДО необходимо:
1) Завести один тип ресурса в закладке RTP:

Исходный код модели:

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

2) Завести ресурс с типом **Парикмахерские** в закладке RSS:

Исходный код модели:

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

3) Завести два паттерна в закладке PAT:

Исходный код модели:

```
$Pattern Образец_прихода_клиента : irregular_event        trace
$Relevant_resources
        _Парикмахерская: Парикмахерская Keep
$Time = Интервал_прихода( 18, 6 )
$Body
_Парикмахерская
        Convert_event
```

```
            количество_в_очереди set _Парикмахерская.количество_в_очереди + 1
$End

$Pattern Образец_обслуживания_клиента : operation          trace
$Relevant_resources
        _Парикмахерская: Парикмахерская Keep Keep
$Time = Длительность_обслуживания( 16, 4 )
$Body
_Парикмахерская
        Choice from _Парикмахерская.состояние_парикмахера = Свободен and
_Парикмахерская.количество_в_очереди > 0
        first
        Convert_begin
                количество_в_очереди  set _Парикмахерская.количество_в_очереди - 1
                состояние_парикмахера set Занят
        Convert_end
                состояние_парикмахера  set Свободен
                количество_обслуженных set
_Парикмахерская.количество_обслуженных + 1
$End
```

4) Завести две операции в закладке OPR:

Исходный код модели:

```
$Operations
        Обслуживание_клиента   :Образец_прихода_клиента
        Приход_клиента                  :Образец_обслуживания_клиента
$End
```

5) Определить два равномерных закона распределения в закладке FUN.

Исходный код модели:

```
$Sequence Интервал_прихода : real
$Type = uniform 12345
$End

$Sequence Длительность_обслуживания : real
$Type = uniform 67891
$End
```

6) Объект прогона

Исходный код модели:

```
Model_name     = mymodel
Resource_file  = mymodel
Statistic_file = mymodel
Results_file   = mymodel
Trace_file     = mymodel
Show_mode      = Animation
Terminate_if Парикмахерская.количество_обслуженных >= 30
```

7)Собираемые показатели

Исходный код модели:

```
$Results
```

Занятость_парикмахера : trace watch_state
Парикмахерская.состояние_парикмахера = Занят
Длина_очереди : trace watch_par Парикмахерская.количество_в_очереди
Всего_обслужено : get_value Парикмахерская.количество_обслуженных
Длительность_работы : get_value Time_now / 60

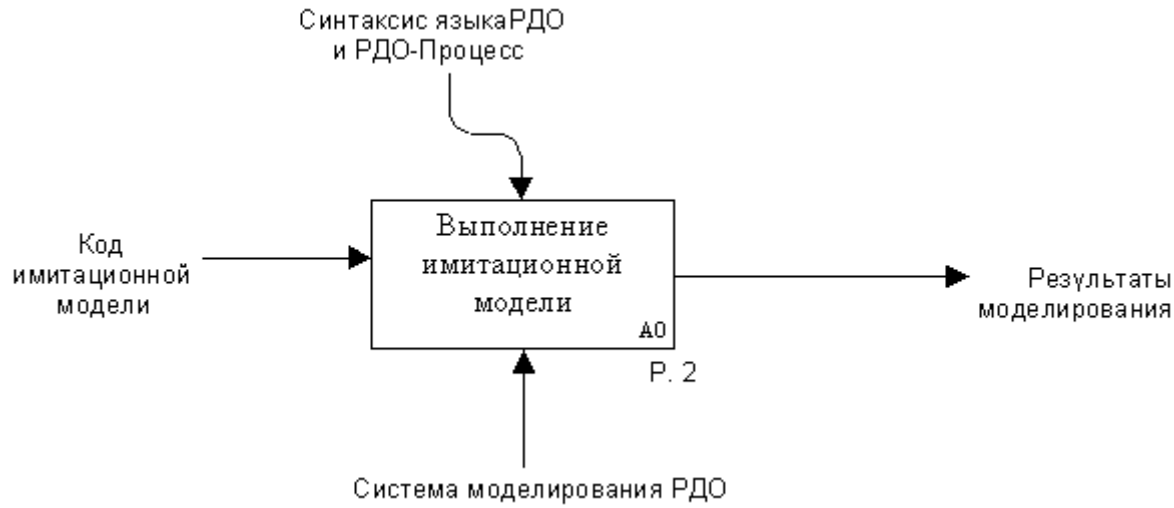
$End

Результат: первая модель – 89 символов кода, вторая модель – 1563 символа кода. Можно делать соответствующие выводы.

2.2 Разработка функциональной модели работы системы

Основная идея курсового проекта – интеграция двух подходов моделирования, РДО (Ресурс Действие Операция) и процессно-ориентированного подхода. На этапе концептуального проектирования необходимо выделить проектируемую систему из общей среды разработки РДО. Для этого необходимо разработать функциональную модель системы РДО.

Кроме этого, функциональная модель необходима в качестве исходных данных для разработки программного обеспечения, т.к. по ней можно проследить алгоритмы функционирования системы, преобразование данных и объектов в процессе ее работы и т.д.



Основная функция РДО как системы – выполнение имитационной модели СДС. Декомпозиция основной функции на первом уровне определяет следующие функции:

1) *Перевод кода модели в объекты, создание списка СДП (событий, действий, процессов).* Эта функция реализуется компилятором РДО, который разбирает текст модели в соответствии с синтаксисом языка. В данном проекте эта функция расширилась добавлением возможности определять список ресурсов у операторов SEIZE и RELEASE языка РДО-Процесс. Подробнее это будет рассмотрено на этапе технического проектирования.
2) *Выбор следующего элемента из списка событий, действий, процессов для ТМВ (текущего модельного времени).*

   В результате функции 1) заполняется список событий, действий и процессов, который меняется в результате выполнения функций 3), 4), 5).
3) *Выполнение события.*
4) *Выполнение действия.*

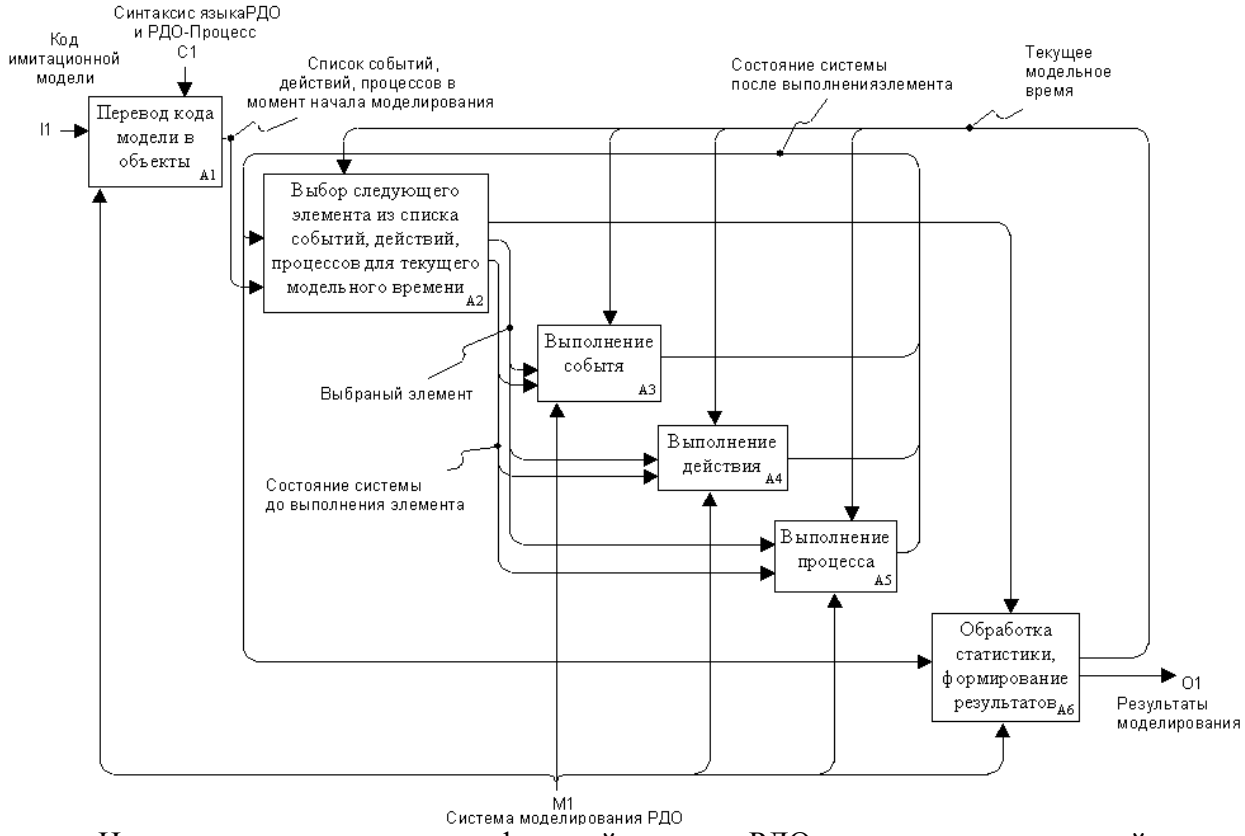
   Третья и четвертая функция относятся к событийному подходу и подходу сканирования активностей.
5) *Выполнение процесса.*

   Эта функция определяет процессный подход в РДО. Она будет декомпозирована далее.

   Функции 2), 3), 4) изменяют состояние моделируемой системы, выполняя соответствующие события, действия и процессы в системе, тем самым, изменяя состояние ресурсов модели.
6) *Обработка статистики, формирование результатов.*

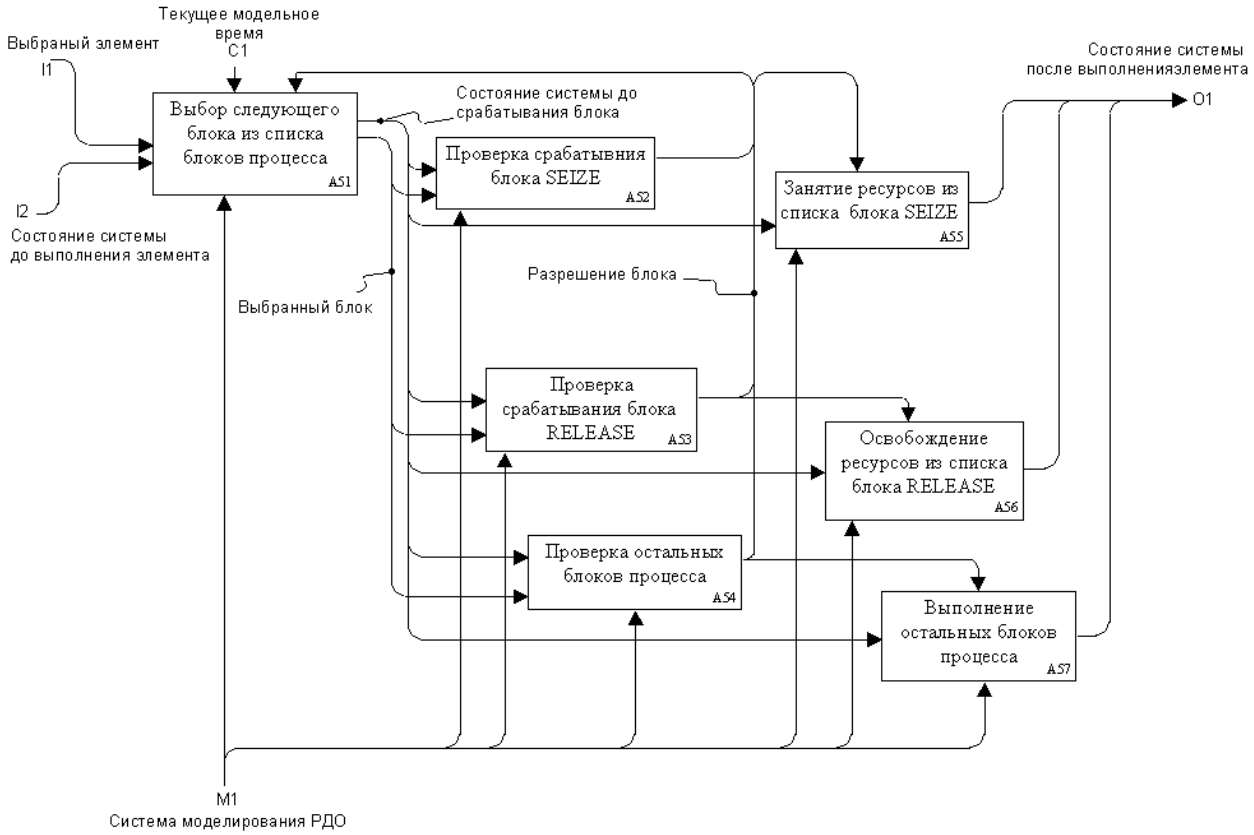
   Разрешение на перевод модельного времени получается исходя результата выполнения предыдущих функций, когда не осталось событий, действий и

процессов, которые могут быть выполнены для текущего модельного времени. Тогда же формируются результаты для текущему моменту модельного времени.



На этом уровне из основных функций системы РДО выделяется процессный подход. Это четвертая функция – выполнение процесса. Она декомпозирована, чтобы очертить конкретные функции, которые необходимо разработать в этом проекте.

1) *Выбор следующего блока из списка блоков процесса.*
   В соответствии с тем, какой блок выбран, выполняется его проверка – *onСруслCondition()* в соответствии с функциями 2), 3), 4) и выполнение – *onDoOperation()* в соответствии с функциями 5), 6), 7). Функции 2), 3), 5), 6) непосредственно разрабатываются в проекте.
2) *Проверка срабатывания блока SEIZE.*
3) *Проверка срабатывания блока RELEASE.*
4) *Проверка срабатывания остальных блоков(GENERATE, SEIZE, ADVANCE, RELEASE, TERMINATE.*
5) *Занятие ресурсов из списка блока SEIZE.*
6) *Освобождение ресурсов из списка блока RELEASE.*
7) *Выполнение остальных блоков процесса (GENERATE, ADVANCE, TERMINATE)*



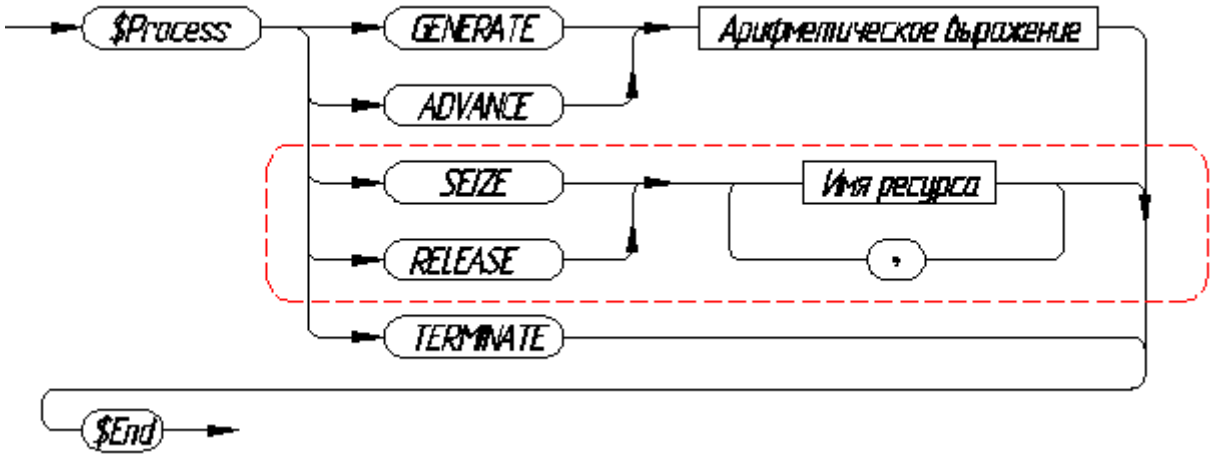
Результатом выполнения приведенных функций является состояние системы после выполнения конкретного процесса, описанного разработчиком на языке РДО-Процесс.

3 Постановка задачи

Задача данного проекта – разработка новых операторов занятия и освобождения ресурсов, которые могли бы работать со списком ресурсов.

Первое, что необходимо сделать — провести необходимые изменения в компиляторе языка РДО-процесс таким образом, чтобы операторы SEIZE или RELEASE могли работать не с одним ресурсом, а со списком ресурсов, определяемым после соответствующих лексем. Эти ресурсы должны создаваться в системе с параметром – «Состояние».

Список решено определять через запятую, сразу после имени оператора. Далее представлена синтаксическая диаграмма разрабатываемого языка РДО-Процесс с вносимыми изменениями.



Второе - это логика работы операторов. Как должен работать блок со списком ресурсов? Когда транзакт может занять ресурс из списка? Как определить, что оператор сработал? Как избежать простоя ресурсов из-за разных причин? Этот ряд вопросов решается на этапах технического и рабочего проектирования, основываясь на функциональном анализе системы, проведенном в этапе концептуального проектирования.

4.Техническое задание

4.1.Общие сведения

В системе РДО разрабатывается процессно-ориентированный язык «РДО - Процесс». Начал работу по этой теме дипломник кафедры РК-9 Манжай Игорь Сергеевич. Основной разработчик РДО – кафедра РК-9, МГТУ им. Баумана.

4.2.Назначение и цели развития системы

4.2.1.Назначение разрабатываемой системы

Процессно-ориентированный подход в РДО разрабатывается для решения отдельного класса задач – моделирования СМО (систем массового обслуживания).

4.2.2.Цели развития системы РДО

Основная цель интеграции подходов моделирования – предоставить разработчику моделей на РДО возможность использовать процессно-ориентированный подход в рамках создания модели на РДО.

На данном этапе проектирования целью является разработка новых операторов занятия и освобождения ресурсов SEIZE и RELEASE, языка РДО-Процесс. Новые операторы работают со списками ресурсов.

4.3.Характеристики объекта автоматизации

РДО – язык имитационного моделирования, основанный на событийном подходе и подходе сканирования активностей. Знания в системе представляются в виде модифицированных продукционных правил [1].

РДО-Процесс — разрабатываемый процессно-ориентированный язык имитационного моделирования, в основу синтаксиса которого положен синтаксис языка GPSS [2].

4.4.Требования к системе

Создавая модель на языке процессно-ориентированного подхода, исследователь описывает последовательность компонентов модели, которые возникают по определенной схеме. Логика возникновения определенных компонентов задается одним оператором языка. Имитатор моделирует процесс продвижения активных элементов через систему, который сопровождается соответствующей последовательностью событий.

Задачей прошлого семестра было исправление логики работы оператора SEIZE с одним и тем же ресурсом. Логика нарушалась, когда в модели, написанной на РДО-Процесс, один ресурс используется несколько раз блоком SEIZE.

Например, с такой ошибкой неправильно работает следующая модель:

```
$Process
      GENERATE      6
      SEIZE         Resource
      ADVANCE       4
      RELEASE       Resource

      SEIZE         Resource
      ADVANCE       5
      RELEASE       Resource

      TERMINATE
$End
```

Поскольку приоритет – порядок расположения блоков в тексте модели, верхний блок SEIZE всегда первым занимает ресурс.

Требование к приоритету: приоритет – время прихода транзакта в очередь блока SEIZE. Первый пришел – первый занимает ресурс.

Задача этого семестра – разработать новые операторы занятия и освобождения ресурсов SEIZE и RELEASE, которые могли бы работать сразу с несколькими ресурсами. Итогом проведенной работы должны быть два новых оператора, которые расширят возможности языка РДО-Процесс. Например, в итоге должна работать правильно такая модель:

```
$Process
    GENERATE  6

    SEIZE       Res1
    ADVANCE   2
    RELEASE   Res1

    SEIZE       Res1, Res2, Res3
    ADVANCE   5
    RELEASE   Res1, Res2, Res3

    TERMINATE

    GENERATE  4

    SEIZE       Res2, Res4, Res5
    ADVANCE   6
    RELEASE   Res2, Res4, Res5

    TERMINATE

    GENERATE  3

    SEIZE       Res3, Res6, Res7
    ADVANCE   6
    RELEASE   Res3, Res6, Res7

    TERMINATE
$End
```

Причем ресурс Res2, например, не должен простаивать в такой модели, когда второй блок SEIZE (**Res1, Res2, Res3**), не может сработать из-за занятого в первом блоке SEIZE (**Res1**) ресурса **Res1**. В таком случае необходимо определить поведение системы таким образом, чтобы третий блок мог занимать ресурс **Res,** когда пришла его очередь. Или резервировать ресурсы **Res2, Res3** для второго блока, если это необходимо. Т.е. возможно определение третьего состояния ресурса: «Зарезервирован».

5 Технический этап проектирования системы

5.1 Общие сведения

Для продолжения работы в заданном направлении возникает необходимость исследовать уже существующую систему, и обозначить ту работу, которая была проведена в этой области. Для этого я исследовал техническую сторону проблемы.

В основу синтаксиса языка РДО-Процесс положен синтаксис языка GPSS. GPSS – язык декларативного типа, построенный по принципу объектно-ориентированного языка. Основными элементами этого языка являются транзакты и блоки, которые отображают соответственно динамические и статические объекты моделируемой системы. Предназначение объектов системы различено. Выбор объектов в конкретной модели зависит от характеристик моделируемой системы.

На момент окончания аспирантом кафедры РК-9 Манжаем И. С. работы над языком, описаны пять блоков языка SEIZE, GENERATE, ADVANCE, RELEASE, TERMINATE. Концепция передачи управления от блока к блоку имеет специфические особенности. Последовательность блоков модели показывает направления, в которых перемещаются элементы. Каждый такой элемент называется транзактом. Транзакты – динамические элементы модели

Содержательное значение транзактов определяет разработчик модели. Именно он устанавливает аналогию между транзактами и реальными динамическими элементами моделируемой системы.

GENERATE (ГЕНЕРИРОВАТЬ) – блок, через который транзакты входят в модель.

ADVANCE (ЗАДЕРЖАТЬ) – блок, который осуществляет задержку тарнзакта на определенное время.

SEIZE (ЗАНЯТЬ) – блок занимает ресурс, если он свободен и в блок пришел транзакт.

RELEASE (ОСВОБОДИТЬ) – блок освобождает ресурс при попадании в него транзакта.

TERMINATE (УДАЛИТЬ) – блок удаляет транзакт из модели.

В приложении 1 представлены диаграммы классов C++ языка РДО, которые были добавлены, изменены или использованы каким-то образом для решения поставленной задачи. Далее идет описание этих классов.

5.2 Классы библиотеки rdodpt.h в парсере

5.2.1 Класс RDOPROCSeize библиотеки rdodpt.h

Класс наследник RDOOperation, является классом блока SEIZE в парсере, объекты которого создаются в момент компиляции модели на РДО-Процесс.

Атрибуты класса

rdoRuntime::RDOPROCSeize* **runtime** – указатель на объект блока SEIZE в рантайме, котрый непосредственно будет занимать ресурсы из его списка.

std::list< std::string > **Resources** – список содержащий имена ресурсов блока.

std::vector< rdoRuntime::parser_for_Seize > **parser_for_runtime** – список объектов структуры **parser_for_Seize**, в которой будут храниться идентификатор ресурса и соответствующего параметра «Состояние» при передачи данных из прасера в рантайм. Определена в рантайме.

```
struct parser_for_Seize
{
int Id_res;
int Id_param;
};
```

Операции класса

static void **makeSeizeType()** – статический метод класса, который используется еще при компиляции в файле грамматики Bison **rdoproc_rtp.y**. Метод «занимается» проверкой и созданием новых типов соответстующих ресурсам блока. Вызывается для каждого ресурса из списка анализируемого блока SEIZE. В качестве параметров передаются указатель на парсер (содержит всю информации о состоянии компилируемой модели на данный момент времени), имя ресурса, и информация, которая указывает на ресурс в тексте модели.

static void **makeSeizeResourse()** – метод аналогичный методу **makeSeizeType**, используется в файле грамматики - **rdoproc_rss.y**. Проверяет и созает нове ресурсы по списку блока. Параметры те же, что и у **makeSeizeType**

void **add_Seize_Resourse()** - добавляет ресурсы в список блока SEIZE.

void **create_runtime_Seize()** – метод, который создает блок SEIZE в рантайме.

5.2.2 Класс RDOPROCRelease библиотеки rdodpt.h

Класс наследник RDOOperation, является классом блока RELEASE в парсере, объекты которого создаются в момент компиляции модели на РДО-Процесс.

Атрибуты класса

rdoRuntime::RDOPROCRelease* **runtime** – указатель на объект блока RELEASE в рантайме, котрый будет освооождать ресурсы из его списка.

std::list< std::string > **Resourses,** std::vector < rdoRuntime::parser_for_Seize > **parser_for_runtime** – те же атрибуты, что и класса RDOPROCSeize.

Операции класса

static void **checkReleaseResourse()** – статический метод класса, который используется еще при компиляции, в файле грамматики Bison - **rdoproc_rss.y**. Метод проверяет наличие освобождаемого ресурса. Вызывается для каждого ресурса из списка анализируемого блока SEIZE. В качестве параметров передаются указатель на парсер

(содержит всю информации о состоянии компилируемой модели на данный момент времени), имя ресурса, и информация, которая указывает на ресурс в тексте модели. В случае отсутствия ресурса или его типа среда РДО выдает ошибку.

void **add_Release_Resourse()** - добавляет ресурсы в список блока RELEASE.

void **create_runtime_Release()** – метод, который создает блок RELEASE в рантайме.

5.3 Классы библиотеки rdoprocess.h в рантайме

5.3.1 Класс RDOPROCResource

RDOPROCResource **–** класс ресурсов, которые на данный момент разработки создаются в системе на этапе комплияции модели, написанной на языке РДО- процесс. В будущем планируется также работать с ресурсами системы созданными в модели на языке РДО. Для этого нужно определить является ли ресурс, ранее созданный в модели на РДО и используемый в модели на РДО-процесс пригодным по типу (есть параметр «Состояние» с необходимыми полями «Занят», «Свободен»). Если да, нужно подхватить этот ресурс и сделать объектом класса ***RDOPROCResource.*** Иначе создавать тип ресурса по имени ресурса с нужным параметром «Состояние», что делается сейчас вне зависимости от того, есть ли ресурс пригодного типа в базе данных модели на РДО.

Атрибуты класса

bool **turnOn** – атрибут, который показывает каким образом занят ресурс. Если атрибут равен true, ресурс занят по своей очереди транзактов, иначе - вне очереди.

RDOPROCBlock* **block** – указатель на блок, в котором занят ресурс.

Операции класса

forResource **AreYouReady()** - метод, который используется в блоке SEIZE, для определения готов ли ресурс быть занят первым транзактом в очереди перед блоком. Тип возвращаемого значения – перечисление:

```
enum  forResource{
turn_On = 1, //ресурс может быть занят транзактом по своей очереди
turn_Off = 2, //ресурс может быть занят транзактом не по своей очереди
not_Seize = 3 //ресурс не может быть занят транзактом
};
```

void **youIs() -** метод задает значение атрибута turnOn, который фиксирует способ занятия ресурса транзактом.

bool **whoAreYou()** - метод, который используется, чтобы узнать как занят ресурс. По своей очереди транзактов или вне этой очереди. Возвращает значение атрибута **turnOn**.

void **setBlock()** - метод записывает в атрибут block, укзатель на блок, в котром занят ресурс. Если ресурс освобождается, запсывает NULL.

RDOPROCBlock* **getBlock()** - метод возвращает указатель на блок, в котором занят ресурс.

5.3.2 Класс RDOPROCTransact

RDOPROCTransact - класс, объекты которого являются динамическими элементами модели в процессно-ориентированном подходе. С помощью этих объектов осуществляется передача управления от блока к блоку.

Атрибуты класса

RDOPROCBlock* **block** - содержит в себе адрес того блока, с которым связан объект RDOPROCTransact, в настоящий момент модельного времени.

Операции класса

void next() – функция, цель которой продвинуть объект класса RDOPROCTransact, у которого она вызвана, в следующий блок. Для этого она, через атрибут класса RDOPROCTransact - block, «добирается» к процессу, в котором «работает» транзакт. И уже у процесса вызывает функцию продвижения транзактов - RDOPROCProcess::next(), передавая в качестве параметра указатель на себя:

```
void RDOPROCTransact::next()
{
        block->process->next( this );
}
```

RDOPROCBlock* **getBlock()** – метод возвращает указатель на блок, в очереди которого "ждет" транзакт.

5.3.3 Класс RDOPROCBlock

Является базовым классом всех блоков модели процессного подхода.

Атрибуты класса

RDOPROCProcess* **process** - атрибут показывает, в каком процессе находится блок.

RDOPROCTransact* **transacts** - очередь транзактов, перед блоком.

Операции класса

virtual void **TransactGoIn()** - виртуальный метод TransactGoIn() класса RDOPROCBlock, определяет поведение каждого блока, при поступлении в него транзакта из предыдущего блока списка блоков. У класса RDOPROCBlock этот метод добавляет в очередь транзактов блока переходящий к нему транзакт. Он передается в качестве параметра.

virtual void **TransactGoOut()** - виртуальный метод класса RDOPROCBlock, определяет поведение каждого блока, по отношению к покидающему его транзакту. Этот транзакт должен попасть в следующий по списку блок. У класса RDOPROCBlock этот метод удаляет из очереди транзактов блока, уходящий из него транзакт, который передается в функцию в качестве параметра RDOPROCTransact* _transact.

5.3.4 Класс RDOPROCBlockForSeize

Класс является родителем для блоков SEIZE и RELEASE.

Атрибуты класса

std::vector < parser_for_Seize > **from_par** - список элемнетов структуры

struct **parser_for_Seize**
{
int Id_res;
int Id_param;
};

который передается из блока SEIZE парсера в конструктор блока SEIZE рантайма. Каждый элемент списка содержит информацию о ресурсе и параметре "Состояние" этого ресурса, из списка блока SEIZE определенного в тексте модели.

Id_res – уникальный номер ресурса.
Id_param – идентификатор параметра «Состояние» у ресурса.

std::vector < runtime_for_Seize > **from_run** - список элементов структуры

struct runtime_for_Seize{
RDOPROCResource* rss;
RDOValue enum_free;
RDOValue enum_buzy;
};

res - указатель, на ресурс используемый блоком SEIZE или RELEASE.
enum_free - значение параметра "состояние ресурса" - свободен.
enum_buzy - значение параметра "состояние ресурса" - занят.

который заполняется после вызова конструктора блока SEIZE рантайма, и переводит информацию о списке ресурсов блока, полученную из парсера в виде списка from_par.

Операции класса

virtual void **onStart()** – метод переводит информацию полученную блоком SEIZE или RELEASE о ресурсах в парсере в виде списка **from_par**, в информацию с которой может работать блок в рантайме в виде списка **from_run**.

5.3.5 Класс RDOPROCSeize

Атрибуты класса

int **Busy_Res** – число занятых в блоке ресурсов в текущий момент времени.

std::map < int, forResource > **for_turn** – ассоциативный массив, каждый элемент которого содержит номер ресурса из списка ресурсов блока, ссылающийся на переменную типа **forResource**, в которой содержиться ответ на вопрос: может ли ресурс быть занят этим блоком, если да, то как (по очереди, вне очереди, не может).

Операции класса

virtual bool **onCheckCondition()** – метод, который занимает ресурсы из списка блока SEIZE и возвращает true, если блок сработал. Считается, что блок SEIZE сработал, когда в нем заняты все ресурсы из его списка.
virtual BOResult **onDoOperation()** – метод, который заканчивает операцию блока SEIZE, продвигает транзакт в следующий блок.

bool **BuzyInAnotherBlockTurnOn ()** – проверяет есть ли среди списка блок SEIZE ресурсы занятые в другом блоке по своей очереди.

bool **AllBuzyInThisBlockn ()** – проверяет все ли ресурсы из списка заняты в этом блоке.

virtual void **TransactGoIn() -** этот метод определяет поведение конкретно блока SEIZE, при поступлении в него транзакта из предыдущего блока. Поскольку блок работает со списком ресурсов, его функция TransactGoIn() включает в себя функцию определяющую стандартное поведение блоков при поступлении в них транзактов - RDOPROCBlockForSeize::TransactGoIn() и свою дополнительную логику. Она состоит в добавлении транзакта в очередь каждого ресурса из списка.

virtual void **TransactGoOut() -** этот метод определяет поведение конкретно блока SEIZE, по отношению к покидающему его транзакту. Этот транзакт должен попасть в следующий по списку блок. Для этого вызывается функция RDOPROCBlockForSeize::TransactGoOut(). Поскольку блок работает со списком ресурсов, его функция TransactGoOut(), кроме того, удаляет транзакт из очереди каждого ресурса списка.

5.3.6 Класс RDOPROCRelease

Операции класса

virtual bool **onCheckCondition()** – метод, который освобождает ресурсы из списка блока RELEASE и возвращает true, если блок сработал.

virtual BOResult **onDoOperation()** – метод, который заканчивает операцию блока RELEASE, продвигает транзакт в следующий блок.

5.4 Компилятор языка РДО

Компилятор языка РДО, т.е. то, что преобразует входной текст модели на РДО в код на языке C++, написан на языке лексических анализаторов Bison. Bison обратно совместим с Yacc: все правильные грамматики Yacc должны без изменений работать с Bison. Любой человек, хорошо знающий Yacc, не должен иметь больших проблем при использовании Bison. Bison написан, в основном, Робертом Корбеттом (Robert Corbett). Ричард Столлмен (Richard Stallman) сделал его совместимым с Yacc. Вильфред Хансен (Wilfred Hansen) из Carnegie Mellon Univerisity добавил поддержку многосимвольных литералов и другие возможности.

Для того, чтобы Bison мог разобрать программу на каком-то языке, этот язык должен быть описан *контекстно-свободной грамматикой*. Это означает, что вы определяете одну или более *синтаксических групп* и задаёте правила их сборки из составных частей. Например, в языке C одна из групп называется `выражение'. Правило для составления выражения может выглядеть так: "Выражение может состоять из знака `минус' и другого выражения". Другое правило: "Выражением может быть целое число". Как вы может видеть, правила часто бывают рекурсивными, но должно быть по крайней мере одно правило, выводящее из рекурсии.

В правилах формальной грамматики языка каждый вид синтаксических единиц или групп называется *символом*. Те из них, которые формируются группировкой меньших конструкций в соответствии с правилами грамматики, называются *нетерминальными символами*, а те, что не могут разбиты -- *терминальными символами* или *типами лексем*. Мы называем часть входного текста, соответствующую одному терминальному символу *лексемой*, а соответствующую нетерминальному символу -- *группой*.

Для примера терминальных и нетерминальных символов можно использовать язык C. Лексемами C являются идентификаторы, константы (числовые и строковые), и различные ключевые слова, знаки арифметических операций и пунктуации. Таким образом, терминальные символы грамматики C это: `идентификатор', `число', `строка' и по одному символу на каждое ключевое слово, знак операции или пунктуации: `if', 'return', `const', `static', `int', `char', `знак плюс', `открывающая скобка', `закрывающая скобка', `запятая' и многие другие (эти лексемы могут быть разбиты на литеры, но это уже вопрос составления словарей, а не грамматики).

Синтаксические группы C это: выражение, оператор, объявление и определение функции. Они представлены в грамматике C нетерминальными символами `выражение', `оператор', `объявление' и `определение функции'. Полная грамматика, для того, чтобы выразить смысл этих четырёх, использует десятки дополнительных языковых конструкций, каждой из которых соответствует свой нетерминальный символ. Пример выше является определением функции, он содержит одно объявление и один оператор. В операторе каждое `x', так же, как и `x * x' являются выражениями.

Каждому нетерминальному символу должны быть сопоставлены правила грамматики, показывающие, как он собирается из более простых конструкций. Например, одним из операторов C является оператор `return`, это может быть описано правилом грамматики, неформально читающимся так:

`Оператор' может состоять из ключевого слова `return', `выражения' и `точки с запятой'.

Должно существовать множество других правил для `оператор', по одному на каждый вид оператора C.

Анализатор Bison читает на входе последовательность лексем и группирует их, используя правила грамматики. Если вход правилен, конечным результатом будет свёртка всей последовательности лексем в одну группу, которой соответствует начальный символ грамматики. Если мы используем грамматику C, весь входной текст в целом должен быть `последовательностью определений и объявлений'. Если это не так, анализатор сообщит о синтаксической ошибке.

5.5 Правила грамматики для блоков SEIZE и RELEASE

Bison принимает на вход спецификацию контекстно-свободной грамматики и создаёт функцию на языке C, которая распознаёт правильные предложения этой грамматики.

Имя входного файла грамматики Bison по соглашению заканчивается на `.y'. В данном проекте изменениям подверглись три файла грамматики проекта РДО: **rdoproc_rtp.y**, **rdoproc_rss.y** и **rdoproc_opr.y**. В этом же порядке они используются в Bison для компляции текста модели, написанного на языке РДО-процесс, в закладке DPT.

Грамматика для блоков SEIZE и RELEASE в данном проекте основана на рекурсивных правилах, поскольку нужно работать с несколькими ресурсами, число которых заранее не известно. Решено было представить список ресурсов после названия блока через запятую. Например: SEIZE ресурс1, ресурс2, ресурс3. Аналогично для блока RELEASE. Причем, работая в блоке SEIZE, придется создавать ресурсы, которых на момент компиляции нет в базе данных системы. А в блоке RELEASE необходимо проверять, существуют ресурсы из его списка. И если нет, то необходимо выводить сообщение об ошибке. Далее будет рассматриваться грамматика для блока SEIZE.

В файле **rdoproc_rtp.y**, который работает первым, для того чтобы определить список из произвольного числа ресурсов, используется следующее рекурсивное правило:

```
dpt_seize_param:
        // empty{
        PARSER->error(rdo::format("ожидается имя ресурса"));
        }

        | RDO_IDENTIF {
        // Имя ресурса
        std::string res_name= *reinterpret_cast<std::string*>($1);
        const RDOParserSrcInfo& info    = @1;
        RDOPROCSeize::makeSeizeType( PARSER, res_name, info );
        }
        | dpt_seize_param ',' RDO_IDENTIF {
        // Имя ресурса
        std::string res_name      = *reinterpret_cast<std::string*>($3);
        const RDOParserSrcInfo& info    = @3;
        RDOPROCSeize::makeSeizeType( PARSER, res_name, info );
        };
```

Правило называется *рекурсивным*, если нетерминал его *результата* появляется также в его правой части. Почти все грамматики Bison должны использовать рекурсию, потому что это единственный способ определить последовательность из произвольного числа элементов.

В результате, на этапе компиляции модели, имеется информация об имени каждого ресурса из списка после ключевого слова SEIZE. И для каждого ресурса, в этом файле выполняется проверка и создание новых типов ресурсов в виде функции **makeSeizeType()** о которой речь пойдет дальше.

Аналогичное правило используется для определения последовательности из произвольного числа ресурсов в файле **rdoproc_rss.y**. Но действие, которое выполняется в результате разбора этого правила, вызывает функцию **makeSeizeResourse()** для кждого ресурса, в результате создаются новые ресурсы из списка после ключевого слова SEIZE.

```
dpt_seize_param:
        // empty {
        PARSER->error(rdo::format("ожидается имя ресурса"));
        }

        | RDO_IDENTIF {
        // Имя ресурса
        std::string res_name      = *(std::string*)$1;
        const RDOParserSrcInfo& info    = @1;
        RDOPROCSeize::makeSeizeResource( PARSER, res_name, info );
        }

        | dpt_seize_param ',' RDO_IDENTIF {
        // Имя ресурса
        std::string res_name      = *(std::string*)$3;
        const RDOParserSrcInfo& info    = @3;
        RDOPROCSeize::makeSeizeResource( PARSER, res_name, info );
        };
```

Последний файл грамматики, который используется при компиляции модели на языке РДО-процесс - **rdoproc_opr.y**. Здесь для списка ресурсов блока SEIZE было также организовано рекурсивное правило:

```
        dpt_seize_param:
        // empty {

        }
        |       RDO_IDENTIF {
        std::string res_name      = *reinterpret_cast<std::string*>($1);
        TRACE( "%s _good\n", res_name.c_str());
        RDOPROCSeize* seize = new RDOPROCSeize( PARSER->getLastPROCProcess(),
"SEIZE");
        seize->add_Seize_Resource(res_name);
        $$ = (int)seize;
        }
        |  dpt_seize_param ',' RDO_IDENTIF {
        RDOPROCSeize* seize  = reinterpret_cast<RDOPROCSeize*>($1);
        std::string res_name = *reinterpret_cast<std::string*>($3);
        TRACE( "%s _good\n", res_name.c_str(), res_name);
        seize->add_Seize_Resource(res_name);
        $$ = $1;
        };
```

В этом файле грамматики создаются все блоки парсера, которые создадут рабочие блоки рантайма. В частности в действии для первого ресурса из списка сначала создается объект класса **RDOPROCSeize**, который и будет являться блоком для этого списка ресурсов. Далее каждый ресурс добавляется в список этого блока методом **add_Seize_Resource()**. Последним действием, после того как список «собран», является создание блока SEIZE со своим списком в рантайме:

seize->create_runtime_Seize(PARSER);

6.Рабочий этап проектирования. Операторы SEIZE и RELEASE

Как было сказано в техническом задании по проекту. В системе РДО разрабатывается процессно-ориентированный язык «РДО - Процесс».
Задача данного проекта – разработка новых операторов занятия и освобождения ресурсов, которые могли бы работать со списком ресурсов. Все, что было сделано в компиляторе языка РДО-Процесс, для определения новых операторов описано в техническом этапе проектирования.

Далее речь идет о логике работы операторов в рантаме, поскольку в ходе решения задачи возникло несколько интересных проблем. Для наглядности рассмотрим следующую модель:

```
$Process
        GENERATE 10

        SEIZE  A
        ADVANCE  30
        RELEASE  A

        SEIZE  A, B, C
        ADVANCE  20
        RELEASE  A, B, C

        TERMINATE

        GENERATE 20

        SEIZE  B, Q, K, P
        ADVANCE  50
        RELEASE  B, Q, K, P

        TERMINATE
$End
```

Основная проблема – простой ресурсов. Как выяснилось, она может возникать из-за нескольких причин. Рассмотрим эти ситуации на примерах в выше приведенной модели:

1) Простой ресурса из-за «жадности» блока SEIZE с меньшим числом критических ресурсов. Например, эта ситуация возникает тогда, когда:
- все ресурсы третьего блок SEIZE свободны
- транзакт, первый в очереди второго блока SEIZE, пришел к нему, раньше чем транзакт, первый в очереди третьего блока SEIZE.
- ресурс А занят в первом блоке SEIZE, что не позволяет сработать второму блоку SEIZE.

В этой ситуации второй блок SEIZE, держит свободный ресурс В, из-за чего не может сработать третий блок SEIZE. Вывод – блок SEIZE должен уметь «делиться» ресурсами.

2) Решить проблему простоя ресурса в предыдущей ситуации можно, дав возможность третьему блоку SEIZE занять свободный ресурс В, пока тот дожидается освобождения ресурса А. Назовем это – «Стратегия-1».

Но возможна следующая ситуация. Позволив третьему блоку обогнать второй, будет занят ресурс В, который в свою очередь нужен во втором блоке SEIZE. И если вдруг этот ресурс останется занятым в третьем блоке, когда освободиться ресурс А, получается

ситуация приведенная в 1) но уже для первого блока. Тогда в соответствии со стратегией 1) ситуация разрешиться тем, что первый блок SEIZE не дожидаясь своей очереди займет ресурс А. И все потому, что у второго блока больше критических ресурсов и когда-то один из них был отдан другому блоку. Поэтому Стратегию-1 можно охарактеризовать тезисом: «чем меньше критических ресурсов, тем «сильнее» блок в борьбе за них». В такой ситуации, второй блок SEIZE может отрабатывать последним, не смотря на то, что транзакты в его очередь могли прийти раньше, чем транзакты в очереди других блоков.

Если же следовать правилу FIFO (первым работает тот блок, в очередь которого транзакт пришел первым) и при этом не допускать простой ресурсов, необходимо исключить простой второго блока, описанный в примере выше. Для этого необходимо ввести третье состояние ресурса – «Зарезервирован», и когда необходимо, переводить в это состояние ресурсы блока SEIZE. Назовем это - «Стратегия-2».

6.2 Реализация предложенного решения

Итак есть две стратегии работы блока SEIZE со списком ресурсов. Стратегия-1 характеризуется тезисом: «чем меньше критических ресурсов, тем «сильнее» блок в борьбе за них». Стратегия-2 – «все блоки равны, вне зависимости от количества критических ресурсов в каждом». Реализовав обе стратегии можно исследовать поведение системы для каждой на пропускную способность, загрузку ресурсов и сделать соответствующие выводы. Далее рассматривается вопрос реализации предложенного решения.

Для решения второй стратегии, нужно фиксировать, как занят ресурс в блоке. Появляются два варианта: ресурс занят блоком по своей очереди, ресурс занят блоком вне очереди (обогнав другой блок). В итоге, если в списке ресурсов блока есть ресурс, занятый в другом блоке по своей очереди, то блок обязан ждать своей очереди и поэтому, может «делиться» остальными свободными ресурсами из его списка. Если же в его списке ресурсов есть только свободные ресурсы или ресурсы, занятые в других блоках, но не по своей очереди, то блок не должен «делиться» ресурсами. Пришла очередь его работы.

Далее представлен исходный код на C++ основных методов, которые были разработаны в рантайме, для реализации указанных выше стратегий. Их назначение коротко описано в этапе технического проектирования. Для переключения между стратегиями до компиляции, использовались директивы препроцессора. Если определен макрос COML, то выполняется вторая стратегия. Дальше приведен исходный код, когда не определен макрос COML, т.е. активирована первая стратегия.

6.2.1 Класс RDOPROCResource

```
class RDOPROCResource: public RDOResource
{
friend class RDOPROCSeize;
protected:
      bool turnOn;            //Указывает как занят ресурс транзактом - по
своей очереди или вне очереди
      RDOPROCBlock* block;
      std::list<RDOPROCTransact*> transacts;

public:
      forResource AreYouReady(RDOPROCTransact* transact);
      bool whoAreYou();
      void youIs(forResource _this);
      RDOPROCBlock* getBlock();
      void setBlock(RDOPROCBlock* _block);
      RDOPROCResource( RDORuntime* _runtime, int _number, unsigned int type,
bool _trace );
};
```

Описание основных методов

```
RDOPROCResource::RDOPROCResource( RDORuntime* _runtime, int _number, unsigned
int type, bool _trace ):
	RDOResource( _runtime, _number, type, _trace )
{
	turnOn = true;
	block = NULL;
}

forResource RDOPROCResource::AreYouReady(RDOPROCTransact* transact)
{
std::list< RDOPROCTransact* >::iterator it = this->transacts.begin();
bool flagOutput = false;
	//Идем по всем транзактам в списке ресурса
	while(it!=transacts.end())
	{
		#ifndef COML
			//Если у блока сиз в котором находится транзакт "*it" нет
ресурсов занятых другим блоком по своей очереди, идем дальше
			if(! (static_cast<RDOPROCSeize*>((*it)->getBlock())-
>BuzyInAnotherBlockTurnOn()))
			{
			//Номер транзакта, который просится в блок
			int aaa = transact->getTraceID();
			//Номер транзакта, который войдет в блок следующим
			int bbb = (*it)->getBlock()->transacts.front()-
>getTraceID();
				//Если они равны, ресурс готов принять транзакт
				if(transact->getTraceID() == (*it)->getBlock()-
>transacts.front()->getTraceID())
				{
					//Если этот транзакт не первый в очереди к
ресурсу, он идет "внеочереди"
					return flagOutput == false ? turn_On : turn_Off;
				}
				else
				{
				return not_Seize;
				}
			}
			else
			flagOutput = true;
		#else
			if(! (static_cast<RDOPROCSeize*>((*it)->getBlock())
->BuzyInAnotherBlock()))
			{
			//Номер транзакта, который просится в блок
			int aaa = transact->getTraceID();
			//Номер транзакта, который войдет в блок следующим
			int bbb = (*it)->getBlock()->transacts.front()
->getTraceID();
				//Если они равны, ресурс готов принять транзакт
				if(transact->getTraceID() == (*it)->getBlock()
->transacts.front()->getTraceID())
				{
					//Если этот транзакт не первый в очереди к
ресурсу, он идет "внеочереди"
					return flagOutput == false ? turn_On : turn_Off;
				}
				else
				{
				return not_Seize;
```

```
                        }
                    }
                    else
                    flagOutput = true;
                #endif
        it++;
        }
return not_Seize;}

bool RDOPROCResource::whoAreYou()
{
        return this->turnOn;
}

void RDOPROCResource::youIs(forResource _this)
{
        switch (_this){
        case turn_On:
                turnOn = true;
                break;
        case turn_Off:
                turnOn = false;
                break;
        }
}

RDOPROCBlock* RDOPROCResource::getBlock()
{
return block;
}

void RDOPROCResource::setBlock(RDOPROCBlock* _block)
{
block = _block;
}
```

6.2.2 Класс RDOPROCTransact

```
class RDOPROCTransact: public RDOResource
{
friend class RDOPROCProcess;

protected:
        RDOPROCBlock* block;
        std::list< RDOPROCResource* > resources;

public:
        RDOPROCBlock* getBlock();
        static int typeID;
        void addRes (RDOPROCResource* res);
        void removeRes (RDOPROCResource* res);
        bool findRes (RDOPROCResource* res);
        RDOPROCTransact( RDOSimulator* sim, RDOPROCBlock* _block );
        void next();
};
```

Описание основных методов

```
void RDOPROCTransact::next()
{
	block->process->next( this );
}

void RDOPROCTransact::addRes (RDOPROCResource* res){
resources.push_back( res );
}

void RDOPROCTransact::removeRes (RDOPROCResource* res){
resources.remove( res );
}

bool RDOPROCTransact::findRes(RDOPROCResource* res)
{
//Ищем ресурс в списке освобождаемых ресурсов транзакта
std::list< RDOPROCResource* >::iterator it_res = std::find(
this->resources.begin(), this->resources.end(), res );
return it_res != this->resources.end() ? true : false;
}

RDOPROCBlock* RDOPROCTransact::getBlock()
{
return this->block;
}
```

6.2.3 Класс RDOPROCBlock

```
class RDOPROCBlock: public RDOBaseOperation
{
friend class RDOPROCTransact;
friend class RDOPROCProcess;
friend class RDOPROCResource;

protected:
	RDOPROCProcess* process;
	std::list< RDOPROCTransact* > transacts;

	RDOPROCBlock( RDOPROCProcess* _process );
	virtual ~RDOPROCBlock() {}

public:
	virtual void TransactGoIn( RDOPROCTransact* _transact );
	virtual void TransactGoOut( RDOPROCTransact* _transact );
};
```

Описание основных методов

```
RDOPROCBlock::RDOPROCBlock( RDOPROCProcess* _process ):
	RDOBaseOperation( _process ),
	process( _process )
{
	process->append( this );
}


void RDOPROCBlock::TransactGoIn( RDOPROCTransact* _transact )
{
	transacts.push_back( _transact );
}
```

```
void RDOPROCBlock::TransactGoOut( RDOPROCTransact* _transact )
{
	transacts.remove( _transact );
}
```

6.2.4 Класс RDOPROCBlockForSeize

```
struct runtime_for_Seize{
RDOPROCResource* rss;
RDOValue	enum_free;
RDOValue	enum_buzy;
RDOValue	enum_reserve;
};

struct parser_for_Seize
{
int Id_res;
int Id_param;
};

class RDOPROCBlockForSeize: public RDOPROCBlock
{
protected:
std::vector < parser_for_Seize > from_par;
std::vector < runtime_for_Seize > from_run;
virtual void onStart( RDOSimulator* sim );

public:
	RDOPROCBlockForSeize( RDOPROCProcess* _process, std::vector <
parser_for_Seize > From_Par  );

	static std::string getStateParamName()		{ return "Состояние"; }
	static std::string getStateEnumFree()		{ return "Свободен";  }
	static std::string getStateEnumBuzy()		{ return "Занят";     }
	static std::string getStateEnumReserve()  { return "Зарезервирован";
}
};
```

Описание основных методов

```
RDOPROCBlockForSeize::RDOPROCBlockForSeize( RDOPROCProcess* _process,
std::vector < parser_for_Seize > From_Par ):
	RDOPROCBlock( _process ),
	from_par( From_Par )
{
}

void RDOPROCBlockForSeize::onStart( RDOSimulator* sim )
{
int size = from_par.size();
std::vector < parser_for_Seize >::iterator it1 = from_par.begin();
	while ( it1 != from_par.end() ) {
		int Id_res = (*it1).Id_res;
		int Id_param = (*it1).Id_param;
		RDOResource* res = static_cast<RDORuntime*>(sim)
->getResourceByID( Id_res );
		runtime_for_Seize elem;
		elem.rss = static_cast<RDOPROCResource*>(res);
		elem.enum_free = RDOValue( elem.rss->getParam(Id_param).getEnum(),
RDOPROCBlockForSeize::getStateEnumFree() );
		elem.enum_buzy = RDOValue( elem.rss->getParam(Id_param).getEnum(),
RDOPROCBlockForSeize::getStateEnumBuzy() );
		elem.enum_reserve = RDOValue( elem.rss-
>getParam(Id_param).getEnum(), RDOPROCBlockForSeize::getStateEnumReserve() );
		from_run.push_back(elem);
		it1++;}}
```

6.2.5 Класс RDOPROCSeize

```
class RDOPROCSeize: public RDOPROCBlockForSeize
{
private:
//Список, каждый элемент которого содержит порядковый номер ресурса в списке
//ресурсов SEIZE, который возможно будет зданиматься в этом блоке и связанный
//с ним элемент перечислимого типа forResource, который указывает каким
//образом будет занят ресурс этим блоком (по очереди или вне очереди)
      std::map < int, forResource > for_turn;
      virtual bool     onCheckCondition( RDOSimulator* sim );
      virtual BOResult onDoOperation   ( RDOSimulator* sim );
public:
      bool BuzyInAnotherBlockTurnOn ();
      bool BuzyInAnotherBlockTurnOff ();
      bool BuzyInAnotherBlock ();
      bool AllBuzyInThisBlockn ();
      RDOPROCSeize( RDOPROCProcess* _process,
std::vector < parser_for_Seize > From_Par ): RDOPROCBlockForSeize( _process,
From_Par ) {}
      virtual void TransactGoIn( RDOPROCTransact* _transact );
      virtual void TransactGoOut( RDOPROCTransact* _transact );

};
```

Описание основных методов

```
//Функция возвращает true, когда находит ресурс, занятый в другом блоке но по
//своей очереди
bool RDOPROCSeize::BuzyInAnotherBlockTurnOn ()
{
int Size_Seize = from_run.size();
      for (int i=0;i<Size_Seize;i++)
      {
            //Если ресурс занят
            if ( (from_run[i].rss->getParam(from_par[i].Id_param) ==
from_run[i].enum_buzy) ){
                  //Если ресурс занят не в этом блоке, но по своей очереди
                  if((from_run[i].rss->getBlock() != this) &&
from_run[i].rss->whoAreYou() == true )
                  {
                        return true;
                  }
            }
      }
return false;
}

bool RDOPROCSeize::BuzyInAnotherBlockTurnOff ()
{
int Size_Seize = from_run.size();
      for (int i=0;i<Size_Seize;i++)
      {
            //Если ресурс занят
            if ( (from_run[i].rss->getParam(from_par[i].Id_param) ==
from_run[i].enum_buzy) ){
                  //Если ресурс занят не в этом блоке, не по своей очереди
                  if((from_run[i].rss->getBlock() != this) && from_run[i].rss-
>whoAreYou() == false )
                  {
                        return true;
                  }
            }
      }
return false;
}
```

```
bool RDOPROCSeize::BuzyInAnotherBlock ()
{
int Size_Seize = from_run.size();
	for (int i=0;i<Size_Seize;i++)
	{
		//Если ресурс занят
		if ( (from_run[i].rss->getParam(from_par[i].Id_param) ==
from_run[i].enum_buzy) ){
			//Если ресурс занят не в этом блоке, не по своей очереди
			if((from_run[i].rss->getBlock() != this))
			{
				return true;
			}
		}
	}
return false;
}

bool RDOPROCSeize::AllBuzyInThisBlockn ()
{
int Size_Seize = from_run.size();
	for (int i=0;i<Size_Seize;i++)
	{
		if ( (from_run[i].rss->getParam(from_par[i].Id_param) !=
from_run[i].enum_buzy) )
		return false;
	}
return true;
}



bool RDOPROCSeize::onCheckCondition( RDOSimulator* sim )
{
	if ( !transacts.empty() ) {
	int Size_Seize = from_run.size();
		#ifndef COML
		if( !BuzyInAnotherBlockTurnOn () && !AllBuzyInThisBlockn ()){
		#else
		if( !BuzyInAnotherBlock() && !AllBuzyInThisBlockn()){
		#endif
			for_turn.clear();
				for(int i=0; i<Size_Seize;i++)
				{
					//Не учитываем ресурс, уже зарезервированный в
//этом сизе
					for_turn[i] = from_run[i].rss-
>AreYouReady(transacts.front());
					if( for_turn[i] == not_Seize )
					return false;
				}
				std::map < int, forResource >::iterator it =
for_turn.begin();
				#ifdef COML
					while( it!= for_turn.end())
					{
					int i = it->first;
						// Запоминаем все, что свободно.
						if ( from_run[i].rss
->getParam(from_par[i].Id_param) == from_run[i].enum_free ){
						from_run[i].rss->setBlock(this);
						from_run[i].rss->youIs(for_turn[i]);
						}
```

```
                                    else
                                    {
                                    }
                              it++;
                              }
                              return true;
                        #endif
                        if( BuzyInAnotherBlockTurnOff() )
                        {
                              while( it!= for_turn.end())
                              {
                              int i = it->first;
                                    // Резервируем все, что свободно.
                                    if ( from_run[i].rss
->getParam(from_par[i].Id_param) == from_run[i].enum_free ){
                                    from_run[i].rss
->setParam(from_par[i].Id_param, from_run[i].enum_reserve);
                                    from_run[i].rss->setBlock(this);
                                    from_run[i].rss->youIs(for_turn[i]);
                                    }
                                    else
                                    {
                                    }
                              it++;
                              }
                        return false;
                        }
                        else
                        {
                              while( it!= for_turn.end())
                              {
                              int i = it->first;
                                    // Запоминаем все, что свободно.
                                    if ( from_run[i].rss
->getParam(from_par[i].Id_param) != from_run[i].enum_buzy ){
                                    from_run[i].rss->setBlock(this);
                                    from_run[i].rss->youIs(for_turn[i]);
                                    }
                                    else
                                    {
                                    }
                              it++;
                              }
                              return true;
                        }
                  }
                  else
                  {
                  return false;
                  }
            }
            else
            {
            //Список транзактов перед блоком пуст
            }
return false;
}
```

```
RDOBaseOperation::BOResult RDOPROCSeize::onDoOperation( RDOSimulator* sim )
{
int Size_Seize = from_run.size();
TRACE( "%7.1f SEIZE-%d\n", sim->getCurrentTime(), Size_Seize );
      for(int i=0;i<Size_Seize;i++){
            if ( from_run[i].rss->getParam(from_par[i].Id_param) !=
from_run[i].enum_buzy ){
            from_run[i].rss->setParam(from_par[i].Id_param,
from_run[i].enum_buzy);
            transacts.front()->addRes(from_run[i].rss);
            }
            else
            {
            }
      }

transacts.front()->next();
return RDOBaseOperation::BOR_done;
}

void RDOPROCSeize::TransactGoIn( RDOPROCTransact* _transact )
{
int Size_Seize = from_run.size();
      for(int i=0;i<Size_Seize; i++){
      from_run[i].rss->transacts.push_back( _transact );
      }
      RDOPROCBlockForSeize::TransactGoIn( _transact );
}

void RDOPROCSeize::TransactGoOut( RDOPROCTransact* _transact )
{
int Size_Seize = from_run.size();
      for(int i=0;i<Size_Seize; i++){
      from_run[i].rss->transacts.remove( _transact );
      }
      RDOPROCBlockForSeize::TransactGoOut( _transact );
}
```

6.2.6 Класс RDOPROCRelease

```
class RDOPROCRelease: public RDOPROCBlockForSeize
{
private:
      virtual bool     onCheckCondition( RDOSimulator* sim );
      virtual BOResult onDoOperation    ( RDOSimulator* sim );

public:
      RDOPROCRelease( RDOPROCProcess* _process, std::vector < parser_for_Seize
> From_Par ): RDOPROCBlockForSeize( _process, From_Par ) {}
};
```

Описание основных методов

```
bool RDOPROCRelease::onCheckCondition( RDOSimulator* sim )
{
      if ( !transacts.empty() ) {
      return true;
      }
return false;
}
```

```
RDOBaseOperation::BOResult RDOPROCRelease::onDoOperation( RDOSimulator* sim )
{
TRACE( "%7.1f RELEASE\n", sim->getCurrentTime() );
int Size_Seize = from_run.size();
      for(int i=0; i<Size_Seize; i++)
      {
            // Занят
            if ( from_run[i].rss->getParam(from_par[i].Id_param) ==
from_run[i].enum_buzy )
            {
            from_run[i].rss->setParam(from_par[i].Id_param,
from_run[i].enum_free);
            from_run[i].rss->setBlock(NULL);
            //from_run[i].rss->youIs(free);
            transacts.front()->removeRes(from_run[i].rss);
            }
      }
transacts.front()->next();
return RDOBaseOperation::BOR_done;
}
```

7 Исследовательская часть проекта

Когда реализованы две стратегии, и есть возможность переключаться между ними до компиляции системы, остается проверить результаты обеих стратегий на реальном примере. Сравнивая результаты, можно делать соответствующие выводы. Рассмотрим модель эксперимента:
Ресурс A задействован в двух последовательных операциях. В первой необходим только ресурс A, во второй задействованы еще два ресурса B и C. Заявки на выполнение приходят через каждые g1 минут. Длительность первой операции t1 минут, второй – t2 минут. Также параллельно выполняется третья операция, в которой задействованы ресурсы B, Q, K, P. Заявки нее приходят через каждые g2 минут. Длительность обслуживания – t3 минут.

```
$Process
        generate        g1
        SEIZE           A
        ADVANCE         t1
        RELEASE         A

        SEIZE           A, B, C
        ADVANCE         t2
        RELEASE         A, B, C
        TERMINATE

        generate        g2

        SEIZE           B, Q, K, P
        ADVANCE         t3
        RELEASE         B, Q, K, P
        TERMINATE
$End
```

Для проведения эксперимента необходимо адекватно задать начальные условия. Это означает, что два процесса должны работать примерно одинаковое время и все этоо время быть загруженными. Поскольку в модели имеется два блока generate, необходимо пропустить через оба блока определенное число транзактов так, чтобы длительности работ над ними в процессах были примерно одинаковыми.

Пусть модель работает определенное время T. За время T через первый блок generate будет сгенерировано примерно T/g1 заявок, каждая из которых будет проходить через две операции в среднем за (t1+t2) минут. Получаем время обработки всех заявок, без учета простоев, равно (t1+t2)*(T/g1). Для второго блока generate аналогично получаем время обработки всех заявок за время T, без учета простоев, равно t3*(T/g2). Тогда приравнивая эти значения получаем условие, соблюдая которое, можно определять примеры моделей для проведения адекватных экспериментов: (t1+t2)*g2=t3*g1.

Далее будем рассматривать модели, для которых выполняется это условие. Условием окончания моделирования считается 100 транзактов, прошедших через модель. Увеличивая время между приходами заявок от эксперимента к эксперименту, тем самым изменяя поток заявок и загрузку ресурсов. После текста каждой модели приведена таблица с результатами.

Модель_1:

```
$Process
        generate Интервал_прихода( 5, 2)
        SEIZE  A
        ADVANCE  Длительность_операции( 30, 3)
        RELEASE  A

        SEIZE  A, B, C
        ADVANCE  Длительность_операции( 10, 3)
        RELEASE  A, B, C
        TERMINATE

        generate Интервал_прихода( 5, 2)
        SEIZE  B, Q, K, P
        ADVANCE  Длительность_операции( 40, 5)
        RELEASE  B, Q, K, P
        TERMINATE
$End
```

Стратегия_1			Стратегия_2		
_A	FALSE 200	0.79911	_A	FALSE 200	0.782345
_B	FALSE 200	0.998766	_B	FALSE 200	0.957968
_C	FALSE 100	0.200803	_C	FALSE 100	0.193778
_Q	FALSE 100	0.797963	_Q	FALSE 100	0.76419
_K	FALSE 100	0.797963	_K	FALSE 100	0.76419
_P	FALSE 100	0.797963	_P	FALSE 100	0.76419
Пропускная способность – 1,22 Максимальная очередь первого блока SEIZE – 82 Максимальная очередь второго блока SEIZE – 100 Максимальная очередь третьего блока SEIZE – 87			Пропускная способность – 1,18 Максимальная очередь первого блока SEIZE – 83 Максимальная очередь второго блока SEIZE – 88 Максимальная очередь третьего блока SEIZE – 88		

Модель_2:

```
$Process
        generate Интервал_прихода( 10, 2)
        SEIZE  A
        ADVANCE  Длительность_операции( 30, 3)
        RELEASE  A

        SEIZE  A, B, C
        ADVANCE  Длительность_операции( 10, 3)
        RELEASE  A, B, C
        TERMINATE

        generate Интервал_прихода( 10, 2)
        SEIZE  B, Q, K, P
        ADVANCE  Длительность_операции( 40, 5)
        RELEASE  B, Q, K, P
        TERMINATE
$End
```

Если проверить, она удовлетворяет условию адекватности проведения экспериментов, которое было определено выше. Далее приводятся результаты работы этой модели для разных стратегий. Условием окончания работы является 100 транзактов сгенерированных блоком generate.

Стратегия_1			Стратегия_2		
_A	FALSE 200	0.792149	_A	FALSE 200	0.752204
_B	FALSE 200	0.99776	_B	FALSE 200	0.942656
_C	FALSE 100	0.196047	_C	FALSE 100	0.187959
_Q	FALSE 100	0.801713	_Q	FALSE 100	0.754697
_K	FALSE 100	0.801713	_K	FALSE 100	0.754697
_P	FALSE 100	0.801713	_P	FALSE 100	0.754697
Пропускная способность – 1,21			Пропускная способность – 1,15		
Максимальная очередь первого блока SEIZE – 65			Максимальная очередь первого блока SEIZE – 68		
Максимальная очередь второго блока SEIZE – 100			Максимальная очередь второго блока SEIZE – 79		
Максимальная очередь третьего блока SEIZE – 74			Максимальная очередь третьего блока SEIZE – 78		

Модель_3:

```
$Process
        generate Интервал_прихода( 20, 2)
        SEIZE  A
        ADVANCE  Длительность_операции( 30, 3)
        RELEASE  A

        SEIZE  A, B, C
        ADVANCE  Длительность_операции( 10, 3)
        RELEASE  A, B, C
        TERMINATE

        generate Интервал_прихода( 20, 2)
        SEIZE  B, Q, K, P
        ADVANCE  Длительность_операции( 40, 5)
        RELEASE  B, Q, K, P
        TERMINATE
$End
```

Стратегия_1			Стратегия_2		
_A	FALSE 200	0.795851	_A	FALSE 200	0.706151
_B	FALSE 200	0.99574	_B	FALSE 200	0.889189
_C	FALSE 100	0.199574	_C	FALSE 100	0.170571
_Q	FALSE 100	0.796165	_Q	FALSE 100	0.718618
_K	FALSE 100	0.796165	_K	FALSE 100	0.718618
_P	FALSE 100	0.796165	_P	FALSE 100	0.718618
Пропускная способность – 1,21			Пропускная способность – 1,09		
Максимальная очередь первого блока SEIZE – 32			Максимальная очередь первого блока SEIZE – 40		
Максимальная очередь второго блока SEIZE – 100			Максимальная очередь второго блока SEIZE – 63		
Максимальная очередь третьего блока SEIZE – 48			Максимальная очередь третьего блока SEIZE – 63		

Модель_4:

```
$Process
        generate Интервал_прихода( 30, 2)
        SEIZE  A
        ADVANCE  Длительность_операции( 30, 3)
        RELEASE  A

        SEIZE  A, B, C
        ADVANCE  Длительность_операции( 10, 3)
        RELEASE  A, B, C
        TERMINATE

        generate Интервал_прихода( 30, 2)

        SEIZE  B, Q, K, P
        ADVANCE  Длительность_операции( 40, 5)
        RELEASE  B, Q, K, P
        TERMINATE
$End
```

Стратегия_1			Стратегия_2		
_A	FALSE 200	0.808991	_A	FALSE 200	0.682125
_B	FALSE 200	0.993662	_B	FALSE 200	0.841854
_C	FALSE 100	0.199931	_C	FALSE 100	0.16971
_Q	FALSE 100	0.793731	_Q	FALSE 100	0.672144
_K	FALSE 100	0.793731	_K	FALSE 100	0.672144
_P	FALSE 100	0.793731	_P	FALSE 100	0.672144
Пропускная способность – 1,22			Пропускная способность – 1,03		
Максимальная очередь первого блока SEIZE – 14			Максимальная очередь первого блока SEIZE – 13		
Максимальная очередь второго блока SEIZE – 86			Максимальная очередь второго блока SEIZE – 53		
Максимальная очередь третьего блока SEIZE – 34			Максимальная очередь третьего блока SEIZE – 52		

Модель_5:

```
$Process
        generate Интервал_прихода( 40, 2)
        SEIZE  A
        ADVANCE  Длительность_операции( 30, 3)
        RELEASE  A

        SEIZE  A, B, C
        ADVANCE  Длительность_операции( 10, 3)
        RELEASE  A, B, C
        TERMINATE

        generate Интервал_прихода( 40, 2)

        SEIZE  B, Q, K, P
        ADVANCE  Длительность_операции( 40, 5)
        RELEASE  B, Q, K, P
        TERMINATE
$End
```

Стратегия_1			Стратегия_2		
_A	FALSE 200	0.779359	_A	FALSE 200	0.659644
_B	FALSE 200	0.991708	_B	FALSE 200	0.83755
_C	FALSE 100	0.189834	_C	FALSE 100	0.159745
_Q	FALSE 100	0.801874	_Q	FALSE 100	0.677805
_K	FALSE 100	0.801874	_K	FALSE 100	0.677805
_P	FALSE 100	0.801874	_P	FALSE 100	0.677805
Пропускная способность – 1,2			Пропускная способность – 1,01		
Максимальная очередь первого блока SEIZE – 9			Максимальная очередь первого блока SEIZE – 1		
Максимальная очередь второго блока SEIZE – 53			Максимальная очередь второго блока SEIZE – 39		
Максимальная очередь третьего блока SEIZE – 11			Максимальная очередь третьего блока SEIZE – 38		

Модель_6:

```
$Process
        generate Интервал_прихода( 50, 2)
        SEIZE  A
        ADVANCE  Длительность_операции( 30, 3)
        RELEASE  A

        SEIZE  A, B, C
        ADVANCE  Длительность_операции( 10, 3)
        RELEASE  A, B, C
        TERMINATE

        generate Интервал_прихода( 50, 2)

        SEIZE  B, Q, K, P
        ADVANCE  Длительность_операции( 40, 5)
        RELEASE  B, Q, K, P
        TERMINATE
$End
```

Стратегия_1			Стратегия_2		
_A	FALSE 200	0.75734	_A	FALSE 200	0.763922
_B	FALSE 200	0.946938	_B	FALSE 200	0.933641
_C	FALSE 100	0.18906	_C	FALSE 100	0.180263
_Q	FALSE 100	0.757878	_Q	FALSE 100	0.753378
_K	FALSE 100	0.757878	_K	FALSE 100	0.753378
_P	FALSE 100	0.757878	_P	FALSE 100	0.753378
			Длительность_работы	86.4168	
Пропускная способность – 1,15			Пропускная способность – 1,16		
Максимальная очередь первого блока SEIZE – 3			Максимальная очередь первого блока SEIZE – 1		
Максимальная очередь второго блока SEIZE – 14			Максимальная очередь второго блока SEIZE – 4		
Максимальная очередь третьего блока SEIZE – 4			Максимальная очередь третьего блока SEIZE – 2		

Модель_7:

```
$Process
        generate Интервал_прихода( 80, 2)
        SEIZE  A
        ADVANCE  Длительность_операции( 30, 3)
        RELEASE  A

        SEIZE  A, B, C
        ADVANCE  Длительность_операции( 10, 3)
        RELEASE  A, B, C
        TERMINATE

        generate Интервал_прихода( 80, 2)

        SEIZE  B, Q, K, P
        ADVANCE  Длительность_операции( 40, 5)
        RELEASE  B, Q, K, P
        TERMINATE
$End
```

Стратегия_1			Стратегия_2		
_A	FALSE 200	0.487991	_A	FALSE 200	0.487991
_B	FALSE 200	0.602916	_B	FALSE 200	0.602916
_C	FALSE 100	0.121816	_C	FALSE 100	0.121816
_Q	FALSE 100	0.4811	_Q	FALSE 100	0.4811
_K	FALSE 100	0.4811	_K	FALSE 100	0.4811
_P	FALSE 100	0.4811	_P	FALSE 100	0.4811
Пропускная способность – 0,74 Максимальная очередь первого блока SEIZE – 1 Максимальная очередь второго блока SEIZE – 2 Максимальная очередь третьего блока SEIZE – 1			Пропускная способность – 0,74 Максимальная очередь первого блока SEIZE – 1 Максимальная очередь второго блока SEIZE – 2 Максимальная очередь третьего блока SEIZE – 1		

Как и следовало ожидать, первая стратегия - «чем меньше критических ресурсов, тем «сильнее» блок в борьбе за них», может приводить к максимальной очереди транзактов перед вторым блоком SEIZE, когда он отрабатывает самым последним. Это объясняется тем, что второй блок SEIZE содержит в своем списке оба критических ресурса A и B, за которые борются остальные два блока, и по первой стратегии имеют приоритет выше, чем второй блок. Вторая же стратегия - «все блоки равны, вне зависимости от количества критических ресурсов в каждом», продолжает поддерживать приоритет – время прихода транзакта в очередь блока SEIZE. И второй блок SEIZE продолжает работать в соответствии со временем прихода транзакта в его очередь. Вторая стратегия имеет возможность резервировать ресурсы, этим объясняется то, что пропускная способность у первой стратегии, как правило, больше чем у второй, хотя и незначительно.

Стратегия2 сложна, и проигрывает в пропускной способности Стратегии1, которая, в свою очередь, может оставить второй блок отрабатывать последним. Но скорее всего полезное свойство Стратегии2 не пригодится разработчику моделей на РДО. Врятли кому-то будет необходимо описывать частный случай поведения по этой стратегии. Недостаток Стратегии1 можно устранить введением понятия приоритета транзакта. Что в свою очередь расширит возможности языка РДО-Процесс. Над этим следует задуматься при проведении дальнейшей работы.

8 Заключение

В данном курсовом проекте проведена работа по развитию процессного подхода в продукционной системе имитационного моделирования РДО. В частности разработаны два новых оператора занятия и освобождения ресурсов SEIZE и RELEASE, которые могут работать со списком ресурсов. При этом проведена работа с компилятором языка РДО-Процесс, где определен синтаксис новых операторов. А также продумана и реализована логика работы операторов со списками ресурсов. На этапе рабочего проектирования было реализовано две стратегии поведения оператора SEIZE по занятию ресурсов из его списка, которые были проанализированы в исследовательской части. Первая стратегия остается в текущей версии языка РДО-Процесс.