Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика, искусственный интеллект и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

«Комбинированный алгоритм продвижения модельного времени при моделировании системы массового обслуживания»

Студент ИУ7-86Б _____ М. Ф. Слепокурова
(Группа) (Подпись, дата) (И. О. Фамилия)

Руководитель НИР _____ И. В. Рудаков
(Подпись, дата) (И. О. Фамилия)

2023 г.

РЕФЕРАТ

Расчетно-пояснительная записка 67 с., 19 рис., 0 табл., 11 источн., 2 прил.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Имитационное моделирование — это универсальный способ цифрового представления реальной системы при помощи средств компьютерной техники, вычислительных алгоритмов и технологий программирования. Потребность в имитационном моделировании возникает в связи с дорогими и/или невозможными исследованиями реальных систем.

На сегодняшний день выделяют три вида имитационного моделирования: агентное моделирование, дискретно-событийное моделирование и системную динамику. Наиболее развитым подходом среди перечисленных является дискретно-событийное моделирование [1]. При таком подходе не учитывается непрерывная природа событий, и рассматриваются только основные изменения состояния моделируемой системы, а именно события, вызывающие эти изменения.

Дискретно-событийное моделирование применяется во многих областях, где необходимо анализировать и оптимизировать производительность системы. К областям применения можно отнести производственные системы, логистику, финансовую аналитику, бизнес-процессы, транспортные системы и другие. Дискретно-событийное моделирование может помочь оптимизировать использование ресурсов, повысить эффективность производства, определить оптимальные стратегии управления запасами, маршрутизации транспорта, инвестирования и управления рисками, оптимизировать бизнес-процессы, использование дорожных сетей и многое другое.

Таким образом, имитационное, а в частности дискретно-событийное, моделирование — мощный инструмент для анализа и оптимизации производительности систем, имеющий применение во множестве различных областей.

1 Аналитический раздел

В данном разделе будет проведен анализ предметной области с целью выявления особенностей дискретно-событийного моделирования и распределения событий в реальных системах массового обслуживания. Будут проанализированы существующие алгоритмы продвижения модельного времени, а также формализована постановка задачи в виде IDEF0-диаграммы.

1.1 Анализ предметной области

Многие задачи дискретно-событийного моделирования тесно связаны с теорией массового обслуживания [2]. Объектом изучения теории массового обслуживания (ТМО) являются системы массового обслуживания (СМО). В настоящее время разработано множество математических моделей, позволяющих изучать совершенно различные реально протекающие процессы обслуживания в сфере транспорта, в промышленном производстве, образовании, медицине, военном деле, торговле, телефонии, компьютерных сетях и т.д [3]. Модель представляет собой абстрактное описание системы, уровень детализации которой зависит от цели моделирования и возможности получения исходных данных с необходимой точностью [4]. Конечная цель развиваемых в ТМО методов состоит в поиске рациональной структуры обслуживающей системы и ее параметров, а также организации обслуживания, обеспечивающей необходимое его качество.

Под системой массового обслуживания понимают системы, производящие обслуживание поступающих в нее запросов на выполнение каких-либо услуг. Каждая СМО состоит из некоторого числа обслуживающих устройств — каналов (приборов) обслуживания. На вход СМО поступает один или несколько потоков запросов (заявок), требующих однотипного обслуживания. К основным элементам СМО относят:

– входящий поток заявок;

– очередь;

– каналы обслуживания;

– выходящий поток обслуженных заявок.

Для описания системы массового обслуживания необходимо задать вероятностные законы, определяющие последовательность моментов поступления заявок в систему и продолжительность обслуживания, а также принцип, в соответствии с которым поступающие в систему заявки выбираются из очереди на обслуживание (FIFO, LIFO, случайный отбор и т.д.).

Системы массового обслуживания можно классифицировать по следующим признакам:

1. По числу фаз обслуживания — *однофазные* и *многофазные*.

2. По числу каналов — *одноканальные* и *многоканальные*. Многоканальные СМО, в свою очередь, могут подразделяться на *полнодоступные*, подразумевающие однородные каналы (с одинаковыми характеристиками), и *неполнодоступные*, соответственно, с различными характеристиками каналов.

3. По вероятностным характеристикам времени обслуживания — *со случайным временем обслуживания* и *с фиксированным постоянным временем обслуживания*.

4. По характеру случайного процесса, происходящего в СМО — *марковские* и *немарковские*. Случайный процесс называется марковским, если для каждого момента времени вероятность любого состояния системы в будущем зависит только от ее состояния в настоящем и не зависит от того, когда и каким образом система пришла в это состояние, т.е. не зависит от того, как процесс развивался в прошлом. Для марковской СМО легко описать математическую модель, однако в реальной жизни таких систем не бывает. Немарковские системы требуют применения статистического моделирования с использованием ЭВМ.

5. По наличию возможности ожидания обслуживания — *с отказами* и *с ожиданием*. В системах с отказами заявка, поступившая в момент, когда все каналы заняты, получает отказ и покидает систему. В системах с ожиданием при занятости всех каналов обслуживания заявка становится в очередь и ждет освобождения одного из каналов. СМО с ожиданием, в свою очередь, можно разделить на системы *с ограниченным* (по

длине очереди или по времени ожидания в очереди) и *неограниченным* ожиданием.

6. По наличию приоритетов обслуживания — *без приоритетов* и *с приоритетами.*

7. По наличию ограничений, накладываемых на поток заявок, — *замкнутые*, подразумевающие ограниченный поток заявок, которые после обслуживания могут возвращаться в СМО, и *открытые.*

Как уже было сказано ранее, системы массового обслуживания оперируют понятием заявки. Такие события, как поступление новой заявки, начало ее обслуживания и т.д. вызывают изменение состояния системы и образуют *поток событий* — последовательность однородных событий, следующих одно за другим через некоторые интервалы времени. Такой поток обладает свойством дискретности и носит стохастический характер функционирования ввиду того, что момент попадания новой заявки и время ее обслуживания случайны.

Для имитацию параллельных событий, происходящих в реальной системе, вводят некоторую глобальную переменную, обеспечивающую синхронизацию всех событий в системе, которую называют модельным (или системным) временем. Для того, чтобы вести отсчет модельного времени и обеспечить правильную хронологическую последовательность наступления основных событий в имитационной модели, используется так называемый таймер модельного времени, который представляет собой переменную для хранения текущего значения модельного времени.

В процессе моделирования системы таймер модельного времени должен постоянно корректироваться в соответствии с основными событиями, возникающими в реальной системе, поэтому одной из важнейших задач имитационного моделирования является **продвижение модельного времени**.

При рассмотрении реальных СМО можно заметить, что распределение событий в них отнюдь не однородно: события группируются вдоль временной оси [5]. Начало возникновения групп связано с наступлением некоторого события или их множества. Такой момент времени называется *синхронным моментом*, а наступившее в этот момент событие — *событием синхронизации.* Интервал, следующий за синхронным моментом времени, обладает плотностью событий, значительно выше средней, и называется *пиковым интервалом.*

Причиной появления пиковых интервалов является наличие в системе особых событий, возникновение которых порождает другие события. В качестве примера можно привести синхронизирующие сигналы, переключающие множество триггеров в цифровых сетях.

Рассмотрим временную ось, представленную на рисунке 1.1. Интервалы вида $[t_i, t_i + t_p]$ $(i = 1, 2, ...)$ являются пиковыми, а точки t_i $(i = 1, 2, ...)$ представляют собой синхронные моменты времени.

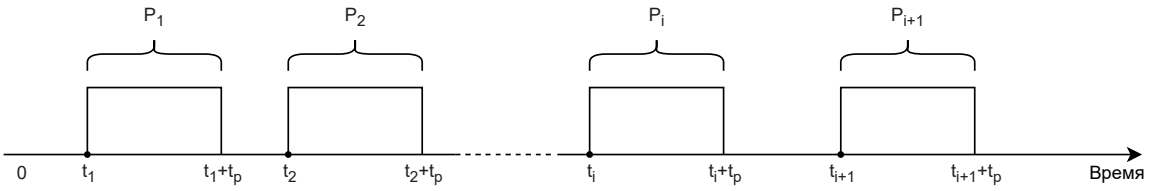

Рисунок 1.1 – Квазисинхронное распределение событий

Квазисинхронным (пиковым) распределением событий называется распределение, при котором значение p вероятности возникновения события значительно больше в интервалах времени $[t_i, t_i + t_p]$, чем в промежуточных интервалах $[t_i + t_p, t_{i+1}]$ $(i = 1, 2, ...)$, при $t_{i+1} - t_i > t_p$.

В рассматриваемом случае допускается одновременное возникновение нескольких событий. Также не исключается возможность того, $t_{i+1} - t_i < t_p$ при $t_i \neq i \cdot t_1$.

Алгоритмы продвижения модельного времени должны быть адаптированы под моделирование систем, распределение событий в которых является пиковым.

1.2 Алгоритмы продвижения модельного времени

1.2.1 Пошаговый алгоритм

Данный алгоритм подразумевает продвижение модельного времени на фиксированное δt единиц. После каждого обновления часов проверяется, произошли ли какие-либо события в течение предыдущего интервала времени δt. Если на этот интервал запланировано одно или несколько событий, считается, что данные события происходят в конце интервала, после чего состояние системы и счетчики обновляются соответствующим образом.

Продвижение времени посредством пошагового алгоритма представлено на рисунке 1.2.

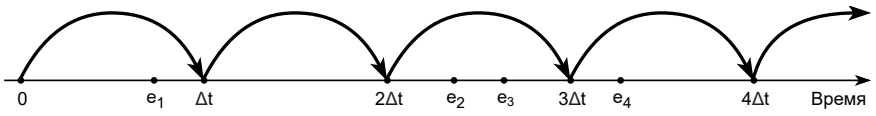

Рисунок 1.2 – Пример продвижения времени посредством пошагового алгоритма

Изогнутыми стрелками обозначено продвижение часов модельного времени, а e_i $(i = 1, 2, ...)$ — это реальное время возникновения события i любого типа.

Рассмотрим несколько представленных интервалов:

– в интервале $[0, \delta t)$ событие происходит в момент времени δt;

– в интервале $[\delta t, 2\delta t)$ не происходит ни одного события, однако проверка на наличие событий все равно выполняется;

– в интервале $[2\delta t, 3\delta t)$ события происходят в моменты времени e_2, e_3, однако считается, что они произошли в момент времени $3\delta t$.

Заметим, что возникает сложность обработки событий, произошедших в один и тот же временной интервал. Данную проблему можно решить уменьшением размера интервала, однако в таком случае возрастает число проверок возникновения событий, что приводит к значительному увеличению времени выполнения программы.

Ввиду этой особенности продвижение модельного времени с помощью пошагового алгоритма не используется в тех случаях, когда интервалы времени между последовательными событиями слишком отличны по своей продолжительности [6].

Таким образом, продвижение времени посредством постоянного шага имеет два недостатка:

– сложность обработки событий, рассматриваемых как одновременные, однако в действительности произошедших в разное время;

– значительные затраты вычислительных ресурсов.

В основном этот алгоритм используется для моделирования систем, в которых можно допустить, что все события в действительности происходят в один из моментов n времени δt $(n = 0, 1, 2, ...)$ для соответственно выбранного δt [6].

1.2.2 Событийный алгоритм

При использовании данного алгоритма продвижение модельного времени происходит в моменты возникновения событий в системе. Предварительно определяется время возникновения будущих событий, после чего таймер модельного времени принимает значение, равное времени возникновения ближайшего события, и в этот момент обновляется состояние системы с учетом произошедшего события, а также сведения о времени возникновения будущих событий. Затем значение таймера модельного времени продвигается ко времени возникновения нового ближайшего события, аналогично обновляется состояние системы и определяется время будущих событий, и т. д.

Процесс продвижения модельного времени от времени возникновения одного события ко времени возникновения другого продолжается до тех пор, пока не будет выполнено какое-либо заранее установленное условие останова. Поскольку в дискретно-событийной имитационной модели все изменения происходят только во время возникновения событий, периоды бездействия системы просто пропускаются, и часы модельного времени переводятся со времени возникновения одного события на время возникновения другого. При таком подходе длительность интервала продвижения модельного времени от одного события к другому может быть различной.

На рисунке 1.3 представлен механизм продвижения модельного времени в системе массового обслуживания с одним устройством обслуживания.

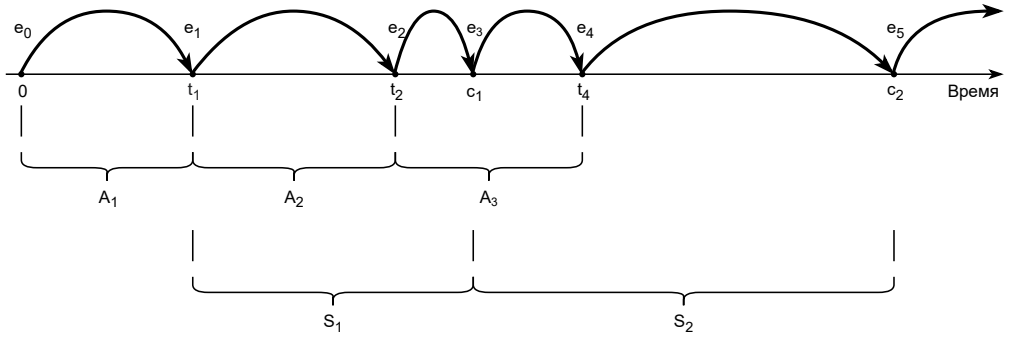

Рисунок 1.3 – Пример продвижения времени посредством событийного алгоритма

Рассмотрим следующие обозначения:

– t_i — время поступления i-й заявки ($t_0 = 0$);

– $A_i = t_i - t_{i-1}$ — время между поступлением заявок $i - 1$ и i;

– S_i — время, потраченное устройством на обслуживание требования i (без учета времени задержки заявки в очереди);

– D_i — время задержки i-й заявки в очереди;

– $c_i = t_i + D_i + S_i$ — время ухода i-й заявки по окончании обслуживания;

– e_i — время возникновения события i любого типа.

Допустим, известны законы распределения вероятностей для времени между поступлением заявок A_i и A_{i+1} и для времени обслуживания S_i, заданные функциями распределения F_A и F_S соответственно. Тогда значения A_i будут генерироваться на основе F_A, а S_i — на основе F_S. В момент времени $e_0 = 0$ обработчик заявок незанят. После генерации времени поступления первой заявки A_1 таймер модельного времени принимает значение времени возникновения следующего (первого) события $e_1 = t_1$. Ввиду того, что в момент поступления в систему первой заявки обработчик заявок незанят, он немедленно начинает обслуживание с задержкой заявки в очереди $D_1 = 0$, изменяя свое состояние на занятое. Время окончания обслуживания заявки c_1 вычисляется путем прибавления сгенерированного S_1 к t_1.

Время поступления в систему второй заявки t_2 определяется по формуле $t_2 = t_1 + A_2$. Если $t_2 < c_1$, как показано на рисунке 1.3, таймер модельного времени принимает значение времени возникновения следующего события $e_2 = t_2$, иначе — значение окончания обслуживания первой заявки c_1. Так как в момент времени поступления новой заявки t_2 обработчик заявок находится в состоянии обслуживания предыдущей заявки (занят), число заявок в очереди увеличивается с 0 до 1. В таком случае сохраняется время поступления заявки, однако не генерируется время обслуживания S_2.

Время поступления в систему третьей заявки вычисляется по формуле $t_3 = t_2 + A_3$. Если $_1 < t_3$, таймер модельного времени принимает значение времени возникновения следующего события $e_3 = c_1$. Когда заявка, обслуживание которой завершено, покидает систему, начинается обслуживание заявки в очереди, при этом длина очереди уменьшается на 1. Время задержки взятой из очереди заявки вычисляется по формуле $D_2 = c_1 - t_2$, а время окончания обслуживания обработчиком — $c_2 = c_1 + S_2$. Аналогично если $t_3 < c_2$, таймер модельного времени принимает значение времени возникновения следующего события $e_4 = t_3$ и т.д.

Моделирование может быть завершено, например, по окончании обслуживания заданного числа заявок или по истечении заданного максимального времени моделирования.

Данный алгоритм целесообразно использовать при моделировании систем, в которых события распределены во времени неравномерно или в тех случаях, когда необходимо точное определение или квазипараллельная обработка одновременных событий [6].

1.3 Оценка сложности алгоритмов

Сложность рассмотренных алгоритмов в общем зависит от вида распределения событий в моделируемой системе, от их количества, а также от общего времени моделирования. Заметим, что сложность пошагового алгоритма также зависит от количества блоков, генерирующих события в системе, т.к. при каждом тике времени происходит полный обход всех блоков системы для проверки наличия нового события.

1.3.1 Пошаговый алгоритм

Приняв минимальный тик таймера модельного за времени за единицу, будем считать, что пошаговый алгоритм, используя это значение в качестве Δt, исключает возможность пропуска событий.

Оценка алгоритмической сложности пошагового алгоритма продвижения модельного времени зависит от количества операций, которые необходимо выполнить при каждом тике таймера модельного времени. Для системы с множеством блоков сложность может быть оценена как $O(N * M)$, где N — общее время моделирования, выраженное в тиках таймера, а M — количество операций, необходимых для обработки каждого тика (пропорционально количеству блоков, генерирующих события в моделируемой системе).

1.3.2 Событийный алгоритм

Событийный алгоритм представляет собой обход сортированного списка (очереди с приоритетом) с последующим добавлением в этот же список (очередь) новых событий. Приоритет событий в очереди определяется временем возникновения события в системе. В худшем случае при обработке одного события в очередь будут добавлены два новых события — событие об оконча-

нии генерации новой заявки и событие об окончании обработки заявки. Для оценки сложности алгоритма необходимо учесть сложность операций вставки и получения элемента в очередь с приоритетом.

Алгоритмическая сложность вставки элемента в очередь с приоритетом зависит от реализации очереди. Наиболее распространенная реализация очереди с приоритетом — куча (heap) [7]. Сложность операции вставки элемента в такую очередь с приоритетом составляет $O(logN)$, где N - количество элементов в очереди [8]. При использовании других реализаций (например, на основе бинарного или сбалансированного дерева), сложность операции вставки элемента может быть другой, однако не хуже приведенной выше [7].

Сложность получения элемента из очереди с приоритетом имеет алгоритмическую сложность $O(1)$, ввиду того, что корень кучи всегда содержит элемент с наивысшим приоритетом [7]. В других реализациях очереди с приоритетом сложность операции получения элемента может быть выше, например, $O(\log N)$, где N — количество элементов в очереди [7]. Это связано с тем, что при поиске элемента с наивысшим приоритетом может потребоваться пройти по дереву от корня до листа, что занимает логарифмическое время. В данной работе рассматривается событийный алгоритм, использующий для списка будущих событий очередь с приоритетом, основанную на куче, ввиду чего сложность операций вставки и получения события составляет соответственно $O(\log N)$ и $O(1)$.

Тогда оценка алгоритмической сложности событийного алгоритма продвижения модельного времени зависит от общего количества обрабатываемых событий и, с учетом рассмотренной выше реализации, может быть оценена как $O(N \log N)$, где N — количество событий, обрабатываемых за указанное время моделирования.

1.3.3 Выводы

На основе приведенной оценки сложности алгоритмов можно сделать вывод о том, что временная эффективность алгоритмов стремительно падает в случае пикового распределения событий в моделируемой системе. Падение эффективности тем больше, чем большее количество блоков, генерирующих события, содержит система. Пошаговый алгоритм становится тем неэффективнее, чем больше длина временных промежутков между пиковыми интервалами,

ввиду невозможности пропустить временные промежутки, не содержащие событий. Событийный же алгоритм стремительно теряет эффективность из-за высокой плотности событий на пиковых интервалах, ввиду роста сложности вставки событий в список будущих событий.

1.4 Постановка задачи

Таким образом, необходимо разработать алгоритм, рассчитанный на моделирование сложных систем (состоящих из большого числа блоков), распределение событий в которых является пиковым. Алгоритм, комбинирующий в себе особенности пошагового и событийного алгоритма, должен решить проблемы падения временной эффективности по причинам, приведенным в предыдущем пункте.

IDEF0-диаграмма для комбинированного алгоритма продвижения модельного времени представлена на рисунках 1.4 1.5.

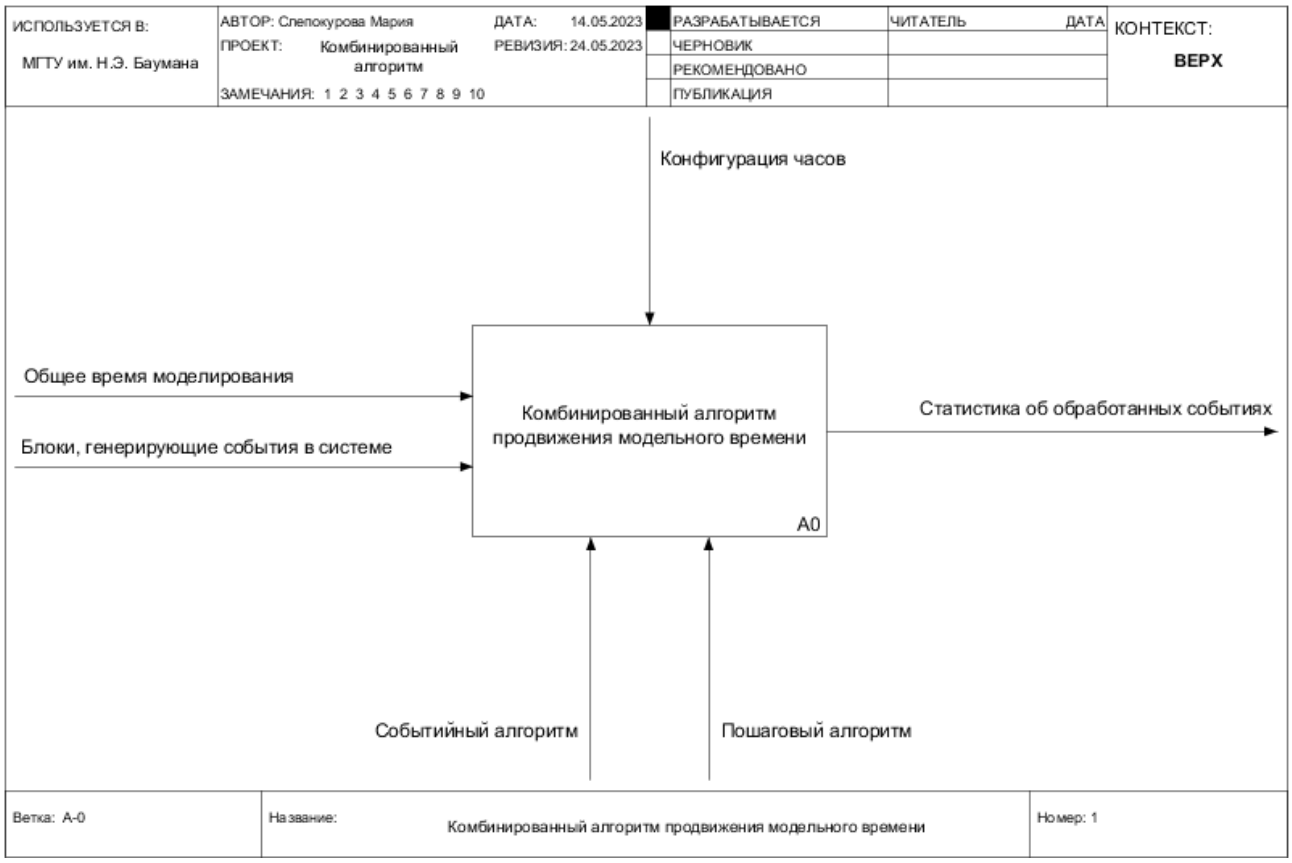

Рисунок 1.4 – IDEF0 нулевого уровня

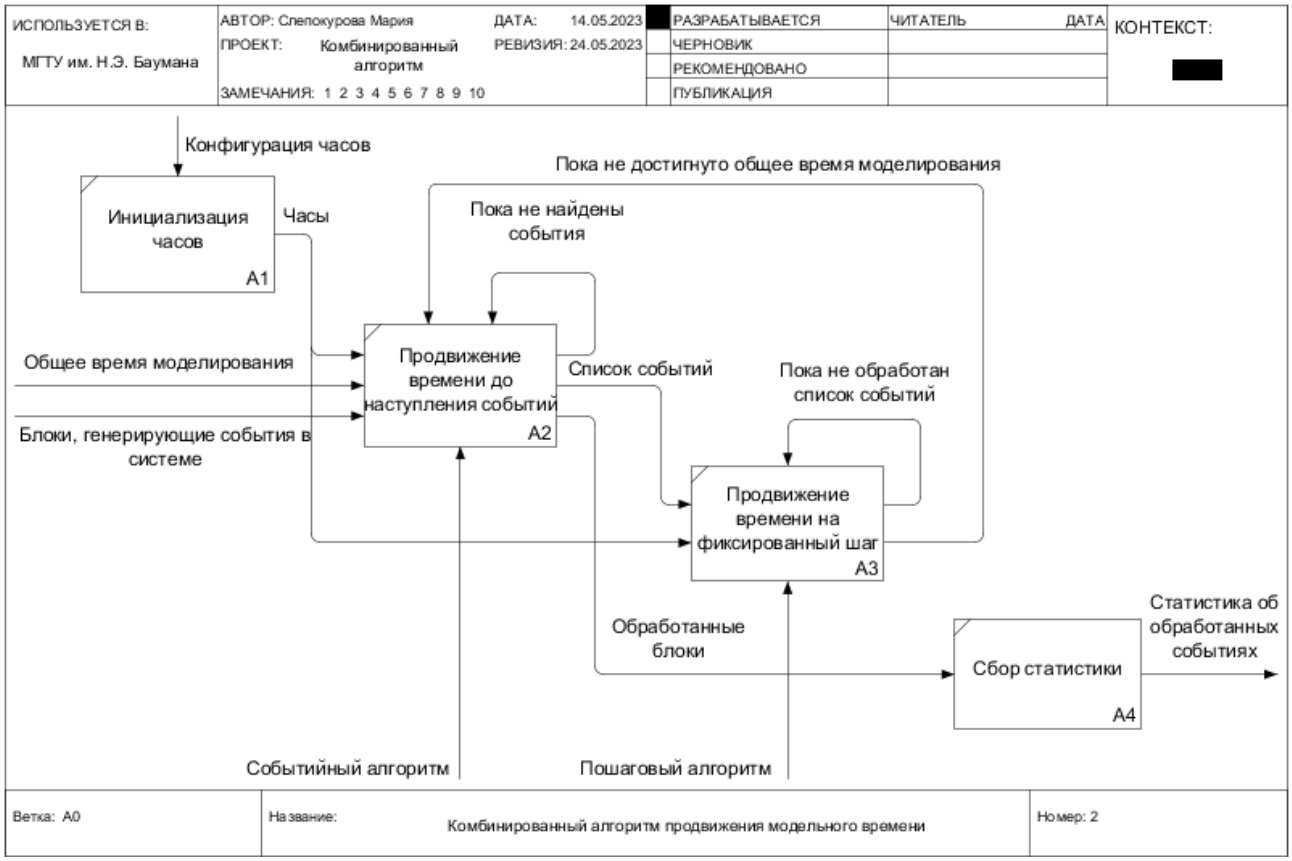

Рисунок 1.5 – IDEF0 первого уровня

1.5 Выводы

В рамках данного раздела были рассмотрены существующие алгоритмы продвижения модельного времени. Была проведена оценка сложности рассмотренных алгоритмов, а также выявлены их недостатки в контексте проанализированной предметной области. На основе полученных заключений была сформулирована цель данной работы, а также формализована постановка задачи в виде IDEF0-диаграммы.

2 Конструкторский раздел

В данном разделе будут сформулированы и описаны в виде схем основные шаги и особенности разрабатываемого комбинированного алгоритма. Также будут приведены схемы реализаций пошагового и событийного алгоритмов, которые в дальнейшем будут использованы для исследования разработанного алгоритма.

2.1 Комбинированный алгоритм

С учетом выявленных недостатков пошагового и событийного алгоритмов предлагается реализовать комбинированный алгоритм таким образом, чтобы на пиковых интервалах он приближался к пошаговому алгоритму, а в промежуточных — к событийному с большим шагом. Таким образом, проблема неэффективности пошагового алгоритма решается использованием подхода событийного алгоритма на промежуточных интервалах. Однако для решения проблем событийного алгоритма, а именно — растущей сложности операций вставки и получения событий в список будущих событий, необходимо реализовать структуру для хранения событий более эффективным способом.

2.2 Часовая структура данных

Предлагается реализовать структуру данных для хранения событий, сложность операций вставки и получения элементов для которой будет составлять $O(1)$.

На рисунке 2.1 представлена разработанная для этой задачи структура данных. В основу положена идея об иерархической системе контроля времени, управляемой событиями (HECTAS) [9].

Часовая структура представлена многоуровневой системой списков событий. Каждая ячейка системы — связный список событий.

Нулевой уровень представлен массивом таких списков и имеет длину N_1. События, попадая на нулевой уровень, обрабатываются аналогично пошаговому алгоритму: временной шаг между соседними ячейками равен минимальному тику таймера модельного времени (единице). Для оптимального моделирования системы с пиковым распределением рекомендуется подобрать длину нулевого уровня таким образом, чтобы она покрывала максимальную

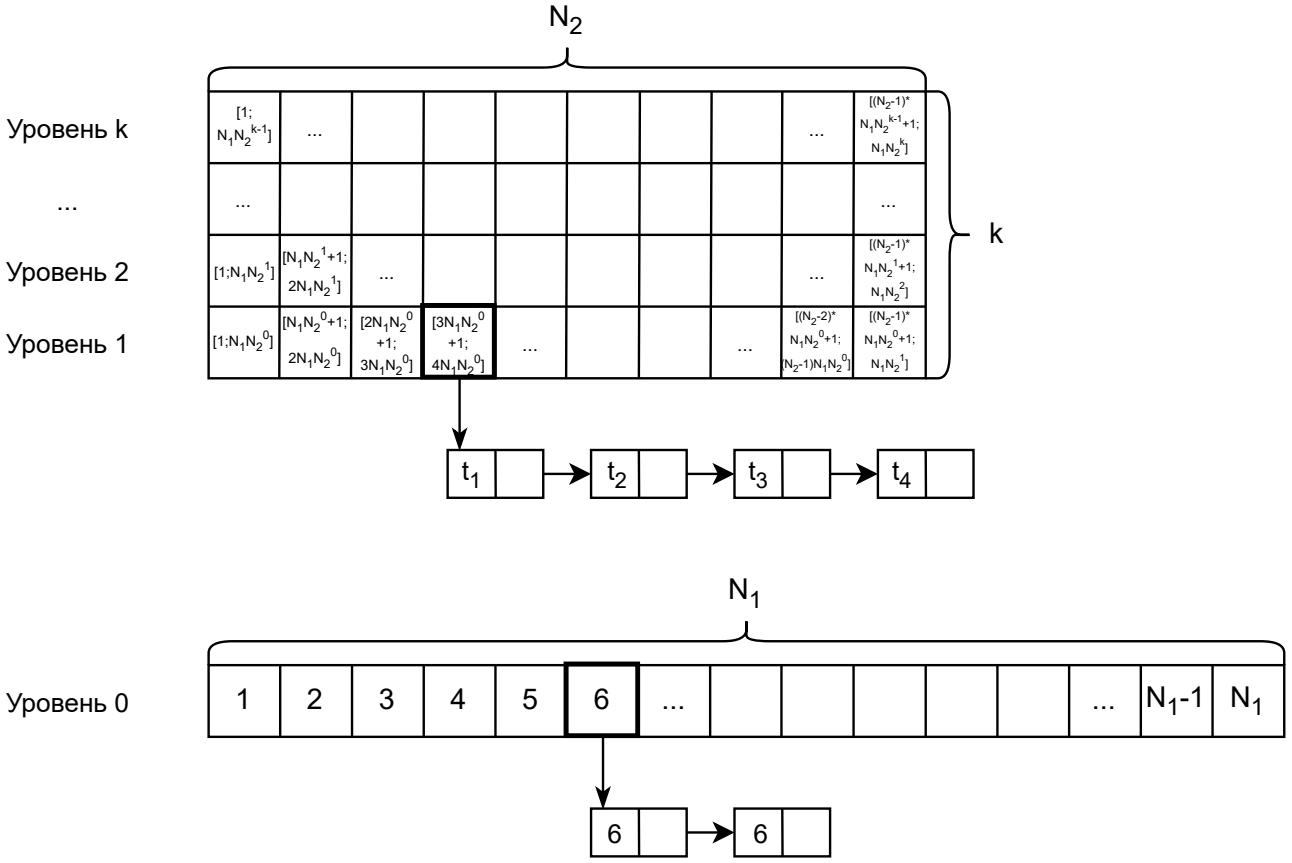

Рисунок 2.1 – Часовая структура данных для хранения событий

длину пиковых интервалов.

Ненулевые уровни системы объединены в массив, каждый уровень, аналогично нуелвому, представлен массивом списков событий с фиксированной длиной N_2, отличной от длины нулевого уровня (совпадение размерностей допустимо). Для удобства массив уровней будет называться далее таблицей событий. Первая ячейка каждого уровня таблицы описывает временной диапазон от 1 до максимального значения предыдущего уровня. Временной шаг на каждом уровне (гранулярность уровня) совпадает с вместимостью предыдущего уровня.

Гранулярность k-го уровня можно рассчитать по формуле:

$$G(k) = \begin{cases} 1 & , k = 0 \\ N_1 N_2^{k-1} & \text{иначе,} \end{cases} \tag{2.1}$$

где:

N_1 — размер нулевого уровня часовой структуры,

N_2 — размер табличного уровня часовой структуры.

Необходимое количество уровней в зависимости от требуемого времени моделирования может быть рассчитано по формуле:

$$k(maxTime) = \frac{maxTime}{N_1}, \tag{2.2}$$

где:
$maxTime$ — общее время моделирования системы.

Для вставки события в часовую структуру необходимо рассчитать столбец и уровень нужной ячейки. Нумерация столбцов начинается с 1, как и нумерация уровней в таблице.

Для определения уровня, в который нужно поместить новое событие, необходимо воспользоваться следующей формулой:

$$Level(time) = \lceil \log_{N_2} \frac{time}{N_1} \rceil, \tag{2.3}$$

где:
$time$ — время возникновения события.

Для определения столбца, в который нужно поместить новое событие, необходимо предварительно определить уровень, после чего воспользоваться следующей формулой:

$$Column(time, level) = \begin{cases} (time - 1) \mod N_1 + 1 & , level = 0 \\ \lceil \frac{time}{G(level)} \rceil & \text{иначе.} \end{cases} \tag{2.4}$$

где:
N_1 — размер нулевого уровня часовой структуры,
$G(level)$ — гранулярность уровня $level$.

Подбор значений N_1 и N_2 зависит от требований, предъявляемых к моделируемой системе. Как уже говорилось ранее, в случае пикового распределения в системе значение N_1 следует подобрать таким образом, чтобы оно покрывало максимальный размер пикового интервала. Для N_2 же наиболее оптимальным значением будет 10: ввиду множества операций взятия логарифма по основанию N_2 возможно использование оптимизированных реализаций взятия десятичного алгоритма. Очевидно, при увеличении обоих величин масштабирование времени в часовой структуре будет больше, что приведет к уменьшению количества уровней в таблице и, как следствие, к уменьшению операций, связанных с перемещением списков событий с одного уровня на другой.

Для наглядности на рисунке 2.2 приведен пример заполненной часовой структуры с размеров нулевого уровня $N_1 = 15$, размером табличного уровня $N_2 = 10$ и количеством уровней таблицы $k = 3$.

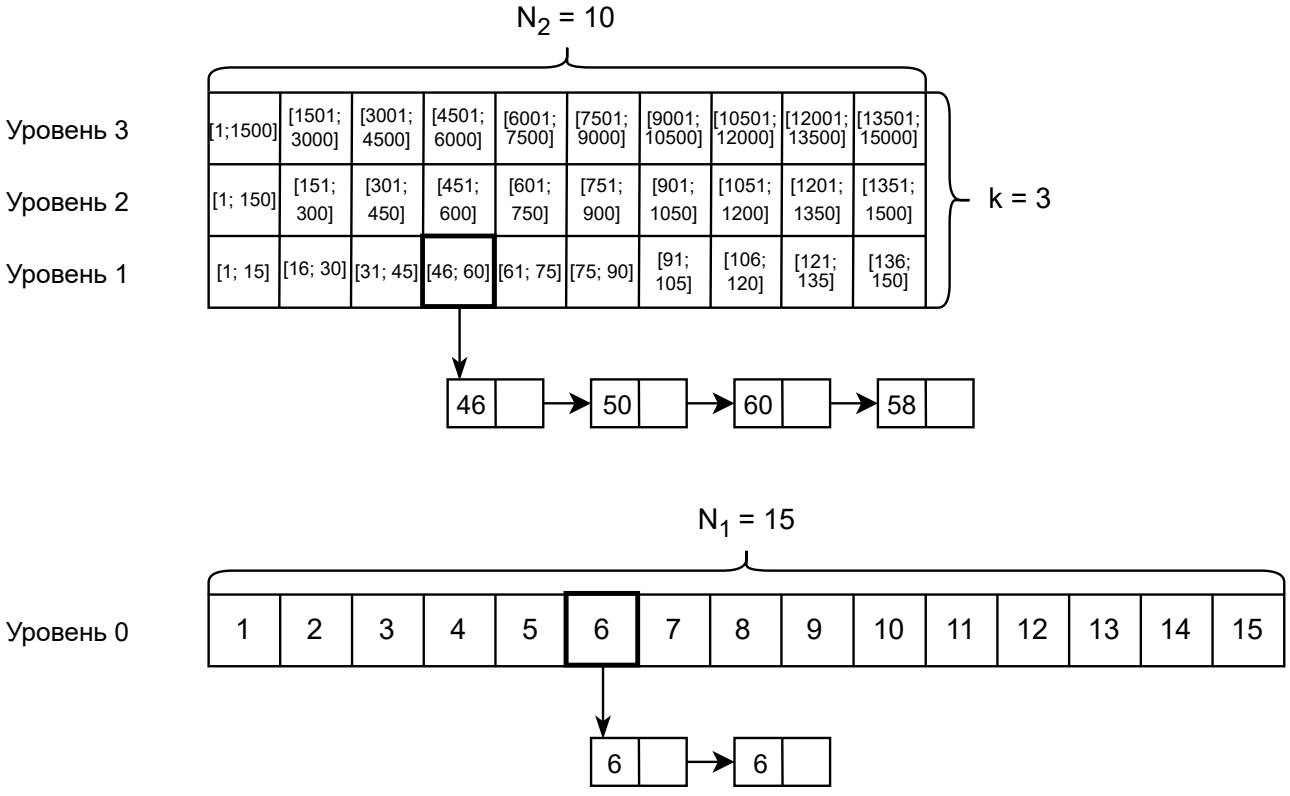

Рисунок 2.2 – Часовая структура данных для хранения событий

Видно, насколько сильно масштабируется время на верхних уровнях даже при такой небольшой размерности N_1. Если при тех же k и N_2 взять N_1 равным, например, 100, то максимальное время моделирования, описываемое такой структурой, будет достигать 1 млн тиков таймера модельного времени.

2.3 Описание алгоритма

Прежде всего необходимо инициализировать часовую структуру в соответствии с требованиями, предъявляемыми к моделируемой системе. Оптимальный подбор значений описан в предыдущем пункте.

При дальнейшем описании алгоритма будет называть генератором блок системы, который генерирует заявки, а процессором — блок, который обрабатывает заявки.

Перед началом процесса моделирования необходимо «запустить» все генераторы системы, т.е. начать генерацию заявок и поместить в часовую структуру события, связанные с окончанием генерации этих заявок. На этом этапе (инициализации) события помещаются в ячейки верхнего уровня таблицы событий, однако все последующие события, сгенерированные блоками моделируемой системы размещаются либо в ячейке нулевого уровня, либо в таблице событий, в зависимости от текущего времени и времени возникновения события.

Процесс моделирования представляет собой рекурсивный обход уровней часовой структуры и начинается с верхнего уровня. Текущая ячейка проверяется на наличие событий: если события не найдены, текущее значение таймера увеличивается на гранулярность текущего уровня, иначе начинается обработка списка событий.

Обработка списка событий заключается в последовательном «спуске» всех событий на уровень ниже до тех пор, пока не будет достигнут нулевой уровень. Для каждого события из рассматриваемого списка происходит пересчет столбца для уровня ниже. Для этого необходимо пересчитать время события в рамках уровня, на который происходит перемещение. Сделать это можно, использовав следующую формулу:

$$time'(level) = time \mod G(level) + 1, \tag{2.5}$$

где:
$time$ — время возникновения события,
$G(level)$ — гранулярность уровня, с которого происходит спуск события.

Тогда пересчет столбца для уровня ниже может быть осуществлен с помощью формулы 2.4 как $Column(time', level - 1)$.

После спуска списка событий по уровню, на который были перемещены события, осуществляется такой же проход, как и по верхнему: в случае нахождения не пустой ячейки происходит последовательный спуск на уровень ниже, а иначе — продвижение времени на гранулярность текущего уровня.

Обработка каждого события происходит непосредственно на нулевом уровне часовой структуры. Перед тем, как начать обход нулевого уровня, необходимо определить его текущее окончание временного диапазона. Сделать это можно по формуле:

$$currentEnd = currentTime + N_1, \tag{2.6}$$

где:
$currentTime$ — текущее значение таймера модельного времени,
N_1 — размер нулевого уровня часовой структуры.

Гранулярность нулевого уровня всегда равна минимальному значению тика модельного времени, т.е. 1, так что при обходе нулевого уровня значение таймера модельного времени инкрементируется, аналогично пошаговому алгоритму. Каждая ячейка нулевого уровня проверяется на наличие событий: если событие найдено, происходит его обработка, иначе — переход к следующей ячейке с увеличением значения таймера.

При обработке каждого события (окончание генерации заявки и окончание обработки заявки) происходит создание и добавление в часовую структуру новых событий: о следующем окончании генерации и о следующем окончании обработки заявки. При добавлении нового события необходимо прежде всего проверить, есть ли возможность поместить его на нулевом уровне, сравнив значение времени возникновения события со значением текущего окончания временного диапазона, описываемого нулевым уровнем ($currentEnd$). В случае, если время нового события не превышает это значение, новое событие помещается на нулевой уровень в столбец $Column(time, 0)$, иначе для нового события предварительно вычисляется номер уровня, после чего уже рассчитывается номер столбца ячейки таблицы. События, размещенные таким

образом на нулевом уровне будут обработаны во время текущего «пошагового» прохода.

По завершении обхода нулевого уровня происходит рекурсивный возврат на предыдущие по следованию уровни, после чего аналогично продолжается их обход. В итоге произойдет возврат в уже пустую ячейку верхнего уровня.

Может возникнуть ситуация, при которой по возвращении в ячейку таблицы список событий, содержащийся в этой ячейке все еще будет непустым. Это происходит ввиду генерации новых событий во время обработки уже существующих. Для обработки таких кейсов введена переменная, содержащая количество повторных обработок текущего уровня — $retries$, значение которой при первичной обработке ячейки равно 0.

При обработке непустой ячейки таблицы происходит проверка значения переменной $retries$: если оно равно 0 — ячейка обрабатывается в первый раз, т.е. происходит обычная, описанная выше, обработка списка событий, иначе — ячейка обрабатывается повторно и прежде, чем приступать к обработке списка событий, необходимо уменьшить текущее значение таймера модельного времени на величину гранулярности обрабатываемого уровня. Это необходимо сделать из-за того, что в таком случае произойдет повторный обход пустых ячеек уже обработанных уровней и, соответственно, ложное увеличение модельного времени.

Процесс моделирования завершается по наступлении максимального времени моделирования: при добавлении события в часовую структуру время события сравнивается с максимальным временем, и, в случае превышения лимита, добавление события не происходит. Таким образом, при наступлении максимального времени моделирования произойдет возврат на верхний уровень таблицы событий и обход пустых ячеек, оставшихся на нем.

Результатом работы такого симулятора является собранная со всех блоков системы статистика: информация о среднем времени генерации/обработки заявки, общее количество сгенерированных/обработанных заявок и среднее время ожидания завки для каждого процессора.

На рисунке 2.3 представлена схема комбинированного алгоритма продвижения модельного времени. Для удобства часовая структура в схемах называется часами.

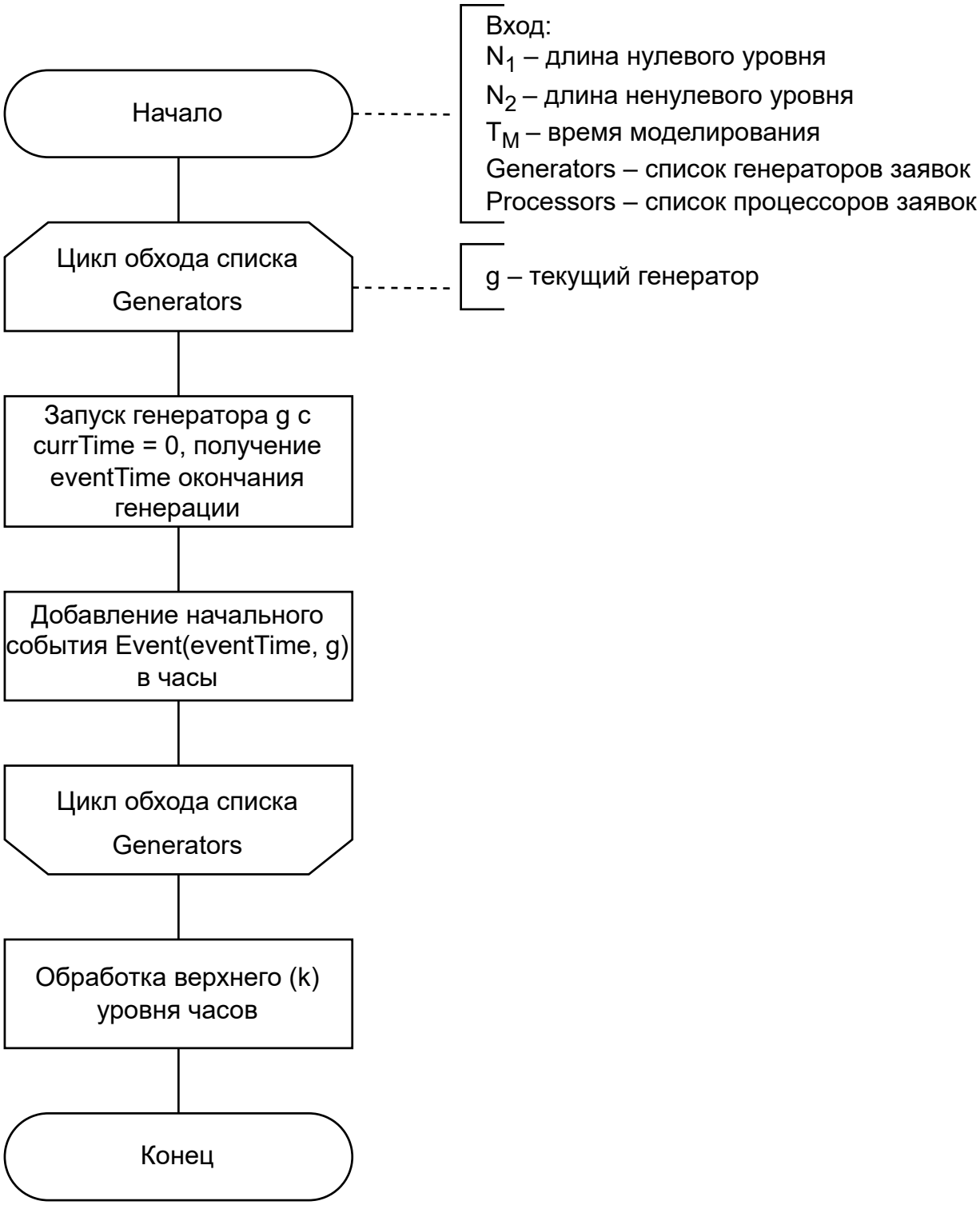

Рисунок 2.3 – Схема комбинированного алгоритма

На рисунке 2.4 представлена схема алгоритма обработки одного уровня часовой структуры.

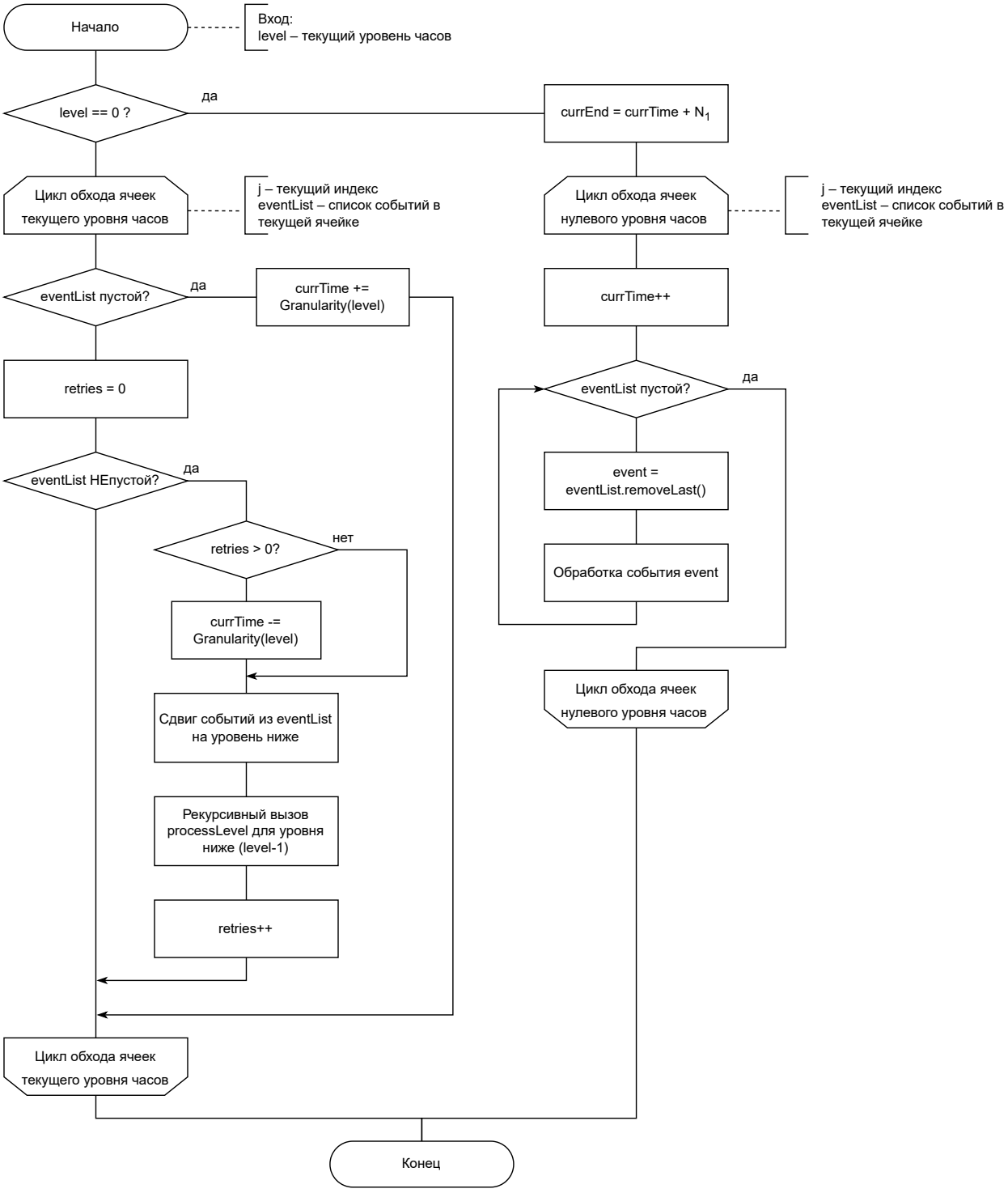

Рисунок 2.4 – Схема алгоритма обработки одного уровня часовой структуры

На рисунке 2.5 представлена схема алгоритма перемещения событий из ячейки часовой структуры на уровень ниже.

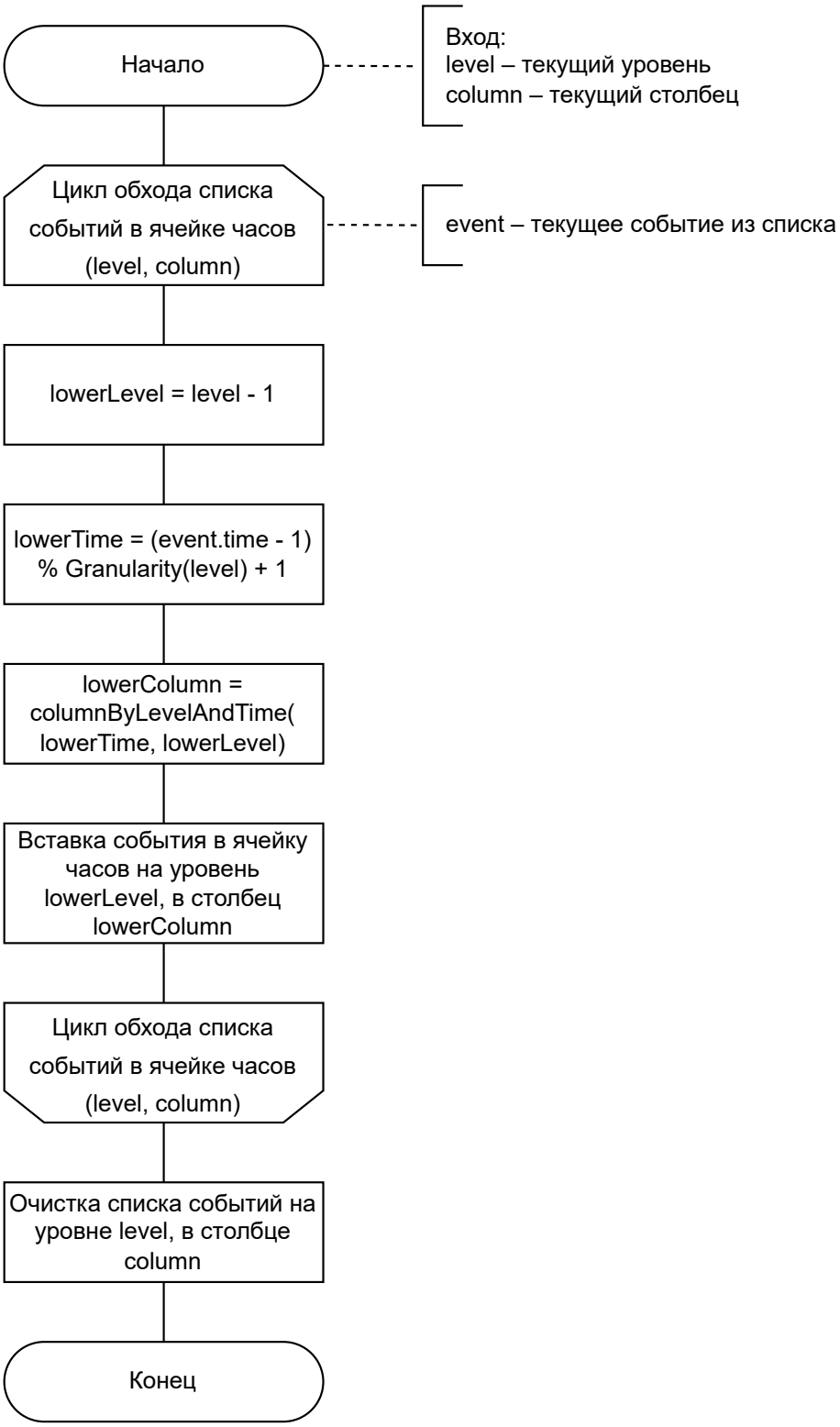

Рисунок 2.5 – Схема алгоритма перемещения событий из ячейки часовой структуры на уровень ниже

На рисунке 2.6 представлена схема алгоритма обработки одного события.

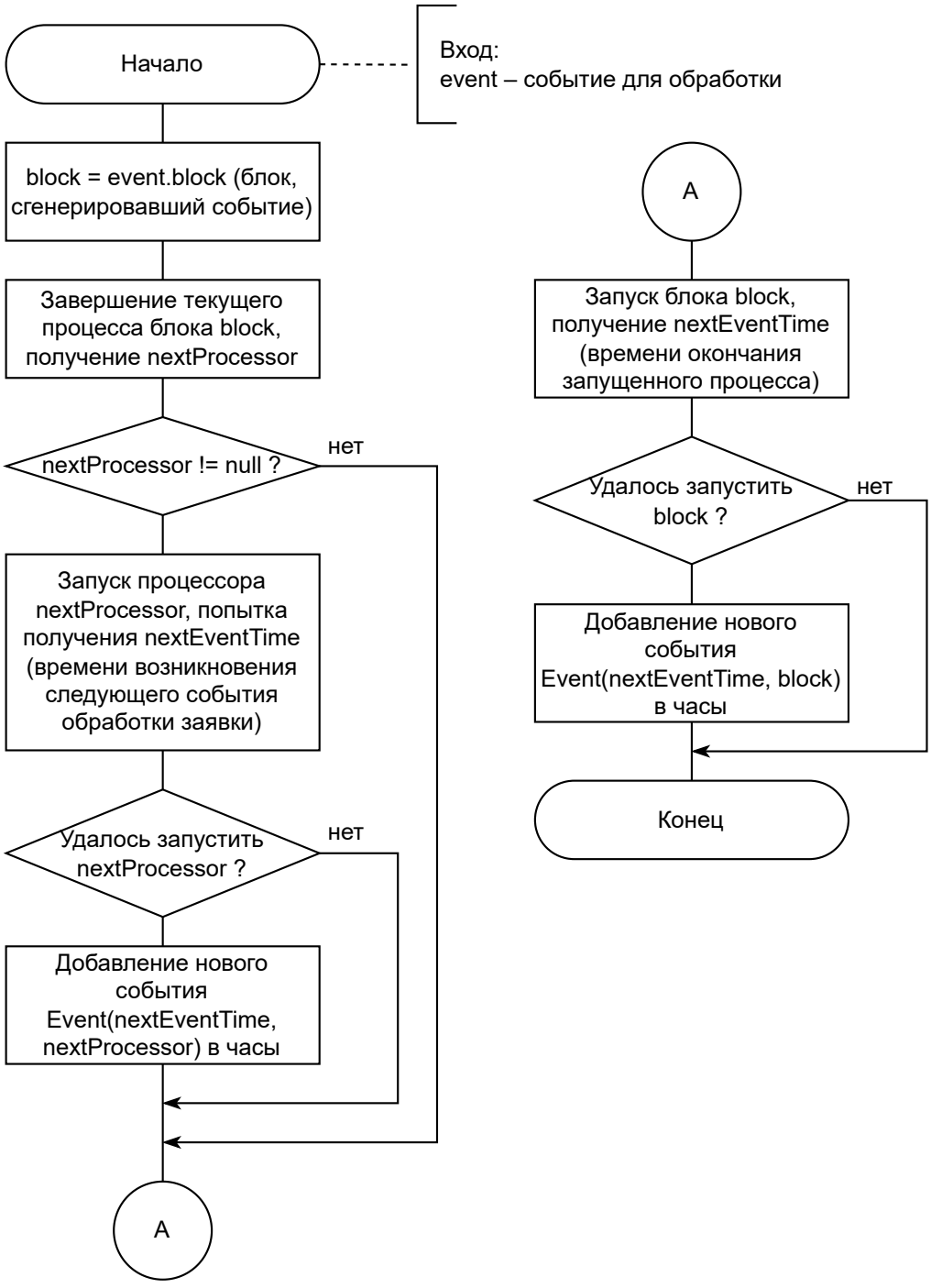

Рисунок 2.6 – Схема алгоритма обработки одного события

2.4 Пошаговый алгоритм

Пошаговый алгоритм подробно описан в соответствующем пункте аналитического раздела.

Схема пошагового алгоритма представлена на рисунке 2.7.

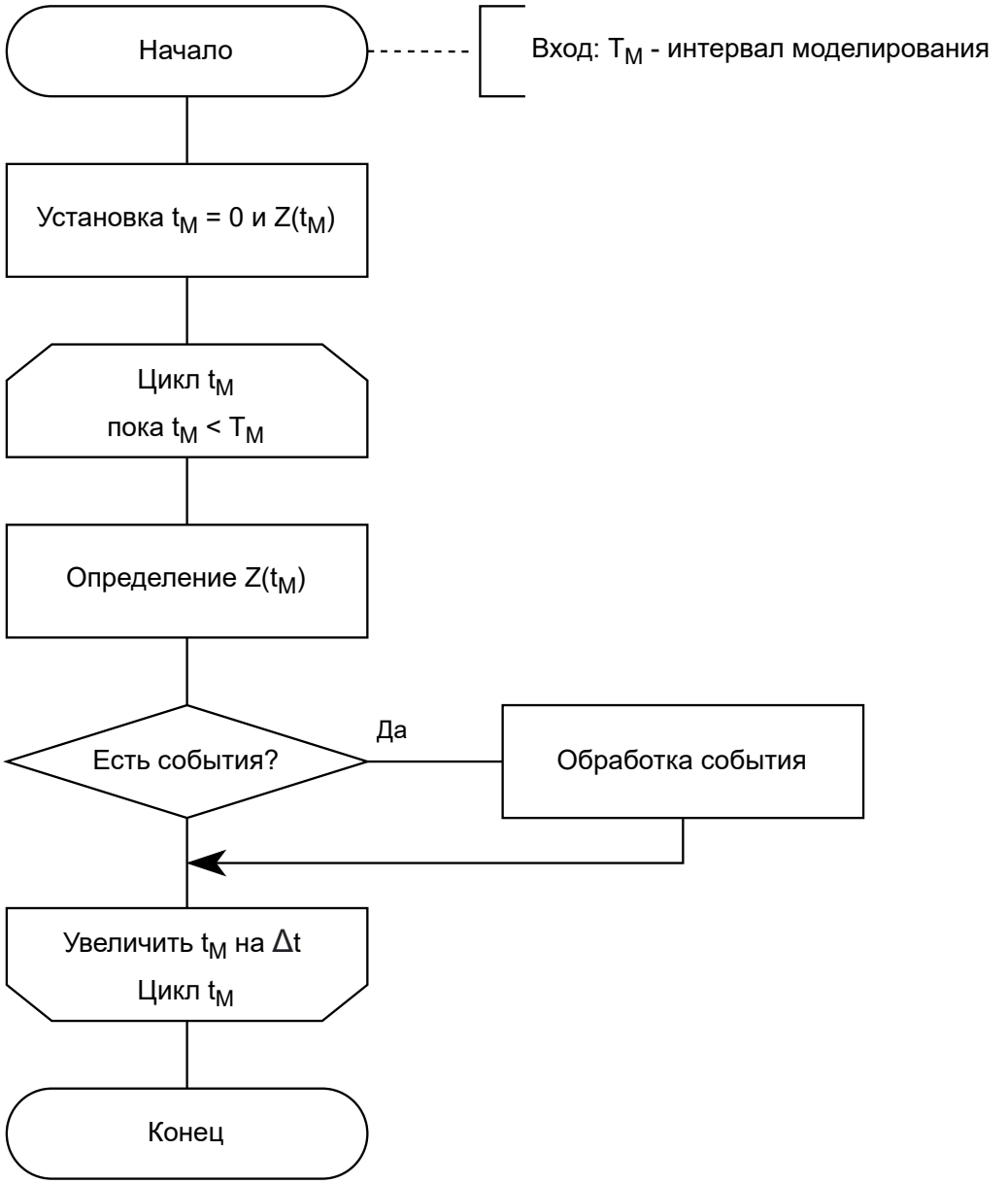

Рисунок 2.7 – Схема пошагового алгоритма

Используемые обозначения:

– t_M — текущее значение модельного времени;

– $Z(t_M)$ — состояние системы в текущий момент времени;

– T_M — интервал моделирования.

2.5 Событийный алгоритм

Событийный алгоритм подробно описан в соответствующем пункте аналитического раздела.

Схема событийного алгоритма представлена на рисунке 2.8.

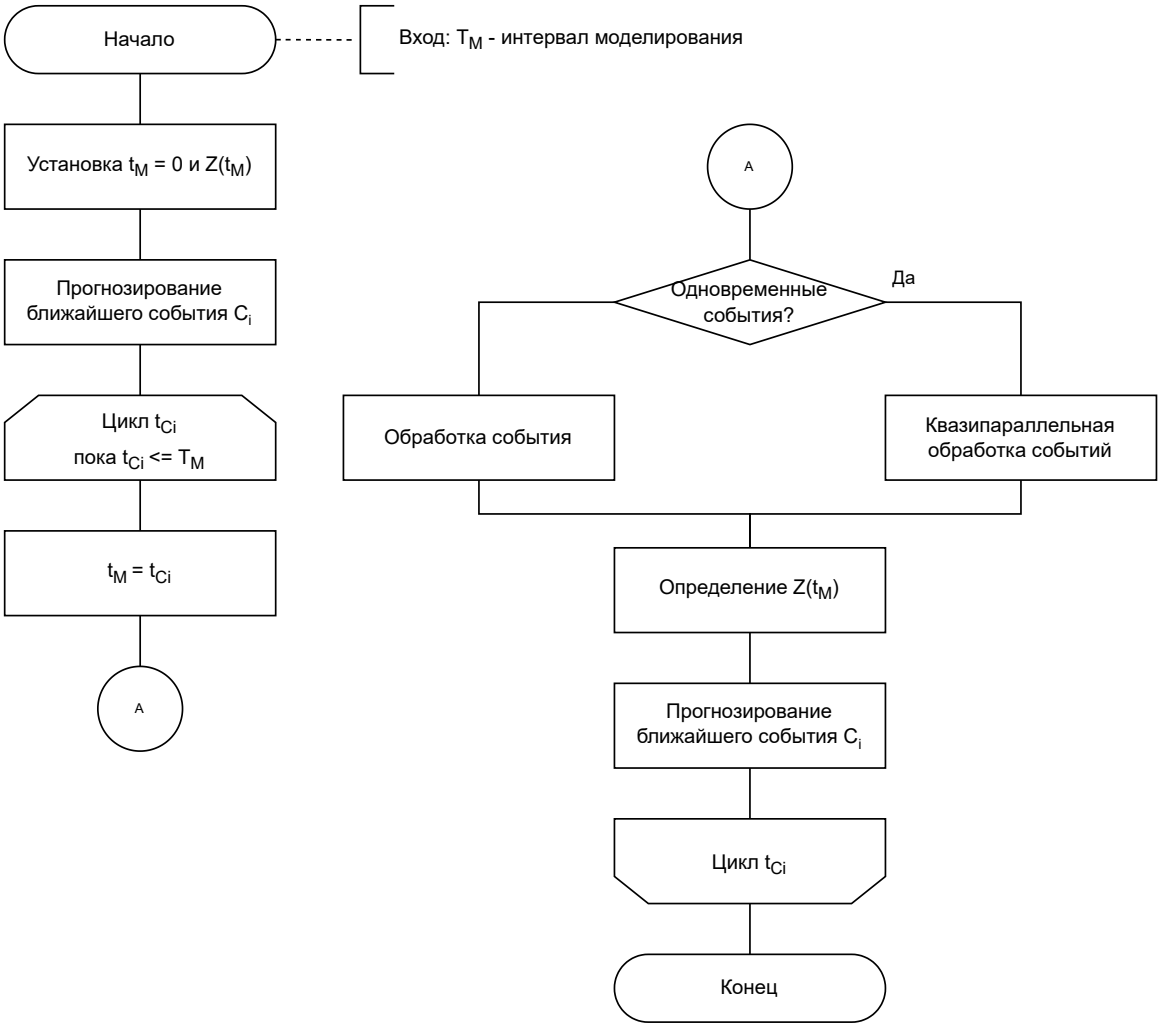

Рисунок 2.8 – Схема событийного алгоритма

Используемые обозначения:

– t_M — текущее значение модельного времени;

– $Z(t_M)$ — состояние системы в текущий момент времени;

– T_M — интервал моделирования;

– t_i — время возникновения i-го события.

2.6 Моделируемая СМО

В качестве моделируемой системы предлагается рассмотреть модель МФЦ.

Согласно информации, предоставленной официальным сайтом мэра Москвы [10], наименее загруженными днями для МФЦ считаются выходные, а максимальная загрузка приходится на вторник. В будние дни наименьшая загрузка наблюдается с 08.00 до 11.00 и с 14.00 до 20.00. Время работы МФЦ — с 08.00 до 20.00 без перерывов и выходных. Среднее время ожидания приема в центрах госуслуг составляет 3 минуты, максимальное — 15 минут.

Согласно статье с сайте Портала Правительства Московской области [11] количество окон обслуживания в одном из подмосковных МФЦ было увеличено до 30.

Тогда моделирумая система будет состоять из:

1. 2-х автоматов с талонами. Время подбора и получения талона клиентом на каждом автомате составляет 15 ± 60 секунд.

2. 30-ти окон обслуживания. Время обслуживания одного клиента в окне составляет от 10 до 30 минут.

3. 10-ти работников архива. После обслуживания клиента в одном из окон, информация о предоставленной услуге вместе с данными о нужных документах заносится в систему одним из работников архива, Длительность этого процесса зависит от оказываемой услуги и количества документов, необходимых для подготовки, ввиду чего время обслуживания обладает большим разбросом и может занимать от 5 до 20 минут.

Необходимо смоделировать работу такого МФЦ в течение одной недели.

За минимальный шаг времени принимается 1 секунда. Тогда общее время моделирования для 7 12-часовых рабочих дней эквивалентно 302400 секундам. Пиковые интервалы приходятся на время с 11.00 до 14.00, т.е. их

длина составляет 10800 секунд (3 часа), а непиковые интервалы приходятся на оставшиеся 9 часов рабочего дня с общей длительностью в 32400 секунды. Ввиду отсутствия в открытом виде статистической выборки по посещаемости МФЦ среднее время появление нового посетителя будем считать 30 секунд для пикового интервала и 420 секунд (7 минут) для непикового. Появление посетителей подчинено экспоненциальному закону.

На рисунке 2.9 представлено распределение для генератора заявок, созданное на основе предоставленной информации о загруженности мфц.

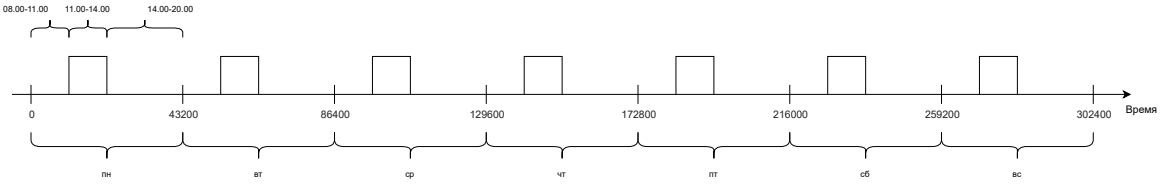

Рисунок 2.9 – Распределение событий поступления заявок в МФЦ

2.7 Выводы

В рамках данного раздела были описаны основные особенности разрабатываемого комбинированного алгоритма продвижения модельного времени. Были изложены ключевые этапы комбинированного, а также пошагового и событийного алгоритмов в виде схем алгоритмов. Также были описаны требования к моделируемой в рамках данной работы системы массового обслуживания.

3 Технологический раздел

В данном разделе будет обоснован выбор средств программной реализации алгоритмов. Будет разработано программное обеспечение, реализующее комбинированный алгоритм продвижения модельного времени, выполнено его тестирование.

3.1 Выбор средств программной реализации

Для реализации комбинированного, а также пошагового и событийного алгоритмов был выбран язык программирования Kotlin и графическая библиотека Swing.

Язык Kotlin был выбран по ряду следующих причин:

1. **Простота и выразительность.** Kotlin предлагает чистый и лаконичный синтаксис, предоставляет множество удобных функций и других инструментов, что значительно упрощает реализацию сложных алгоритмов продвижения модельного времени.

2. **Высокая производительность.** Kotlin компилируется в байт-код JVM, что позволяет достичь высокой производительности. Это важно для алгоритмов модельного времени, которые требуют обработки больших объемов данных и выполнения вычислительно интенсивных операций.

3. **Кросс-платформенность.** Код, написанный на Kotlin может работать на различных платформах, включая JVM, Android, JavaScript и Native, что открывает множество возможностей для интеграции реализованных алгоритмов в различные системы.

4. **Поддержка объектно-ориентированного программирования.** Kotlin предлагает гибкую парадигму программирования, которая позволяет использовать как функциональный, так и объектно-ориентированный стиль программирования. С сущностями поставленной задачи продвижения модельного времени удобнее всего работать как с системой классов.

5. **Богатая экосистема.** Kotlin имеет широкую и активную экосистему инструментов и библиотек, которые могут быть полезны при реали-

зации алгоритмов. К таким библиотекам относится библиотека Swing, предоставляющая инструменты для создания графического интерфейса.

6. **Безопасность типов.** Kotlin обеспечивает безопасность типов и проверку на этапе компиляции, что сокращает время отладки и повышает стабильность программного обеспечения.

В свою очередь, библиотека Swing была выбрана в связи со следующими преимуществами:

1. **Кросс-платформенность.** Как и Kotlin, Swing является кросс-платформенной библиотекой, что означает, что приложения, созданные с использованием Swing, могут работать на различных операционных системах. Это обеспечивает широкий охват пользователей и удобство разработки.

2. **Богатый набор компонентов.** Swing предоставляет обширный набор графических компонентов, что позволяет создавать самые разнообразные интерфейсы в зависимости от требований проекта.

3. **Простота использования.** Swing предоставляет интуитивно понятный и простой в использовании API. Kotlin, в свою очередь, предлагает лаконичный и выразительный синтаксис. Сочетание этих двух инструментов делает разработку GUI более удобной и эффективной.

4. **Хорошая документация и сообщество.** Swing имеет обширную документацию и активное сообщество разработчиков. Это облегчает процесс изучения и получения поддержки, а также предоставляет доступ к множеству ресурсов и примеров кода.

3.2 Разработка программного обеспечения

Полный код разработанного программного обеспечения представлен в приложении.

В рамках данного раздела будут рассмотрены основные классы, необходимые для решения поставленной задачи.

В листинге 3.1 представлен интерфейс **DurationGenerator**, который реализуют все классы-генераторы продолжительности, например,

класс **UniformDurationGenerator**, представленный в листинге 3.2. Метод *generate* класса **UniformDurationGenerator** используется для генерирации продолжительность согласно равномерному распределению на основе значений *min* и *max*, переданных в конструктор. Интерфейс **DurationGenerator** используется всеми сущностями, реализующими блоки системы, — процессорами и генераторами, для генерации времени выполнения той или иной операции.

Листинг 3.1 – time/DurationGenerator.kt

```kotlin
package time

typealias Time = Int

interface DurationGenerator {
    fun generate(): Time
}
```

Листинг 3.2 – time/UniformDurationGenerator.kt

```kotlin
package time

import kotlin.random.Random

class UniformDurationGenerator(
    private val min: Time,
    private val max: Time
) : DurationGenerator {
    private val random = Random(System.currentTimeMillis())

    override fun generate(): Time {
        return random.nextInt(min, max + 1)
    }

    fun getMin(): Time {
        return min
    }

    fun getMax(): Time {
        return max
    }
}
```

В листинге 3.3 представлен интерфейс **Block**, который реализуют классы **Processor** и **Generator**. Метод *cleanupState* используется для очищение всю статистику и текущее состояние блока. Метод *currentFinishTime* — метод-геттер, используемый для получения времени окончания текущего запущенного процесса. Метод *start* используется для запуска процесса блока: для генератора — процесса генерации заявки, для процессора — процесса обработки заявки.

Листинг 3.3 – simulator/Block.kt

```
1  package simulator
2
3  import time.Time
4
5  interface Block {
6      fun cleanupState()
7      fun currentFinishTime(): Time
8      fun start(currentTime: Time): Time?
9      fun finish(currentTime: Time): Processor?
10 }
```

В листинге 3.4 представлен класс **Event**, описывающий события в моделируемой системе, где *time* — время наступления события, *block* — блок системы, сгенерировавший событие.

Листинг 3.4 – simulator/Event.kt

```
1  package simulator
2
3  import time.Time
4
5  data class Event(
6      val time: Time,
7      val block: Block
8  )
```

В листинге 3.5 представлен класс **Request**, описывающий заявки в моделируемой системе, где *timeIn* — время поступления заявки в с систему, *timeOut* — время выхода заявки из системы.

Листинг 3.5 – simulator/Request.kt

```
1  package simulator
2
3  import time.Time
```

```
4
5  class Request(val timeIn: Time) {
6      var timeOut: Time = 0
7  }
```

В листинге 3.6 представлен класс **Processor**, описывающий блок процессора в моделируемой системе. Как уже было сказано выше, он реализует интерфейс **Block**. Рассмотрим подробнее поля класса:

- *durationGenerator* — генератор продолжительности обработки заявки;
- *receivers* — список процессоров-получателей обработанной заявки;
- *queue* — очередь заявок процессора;
- *currentRequest* — текущая обрабатываемая заявка;
- *currentStartTime* — время начала обработки текущей заявки;
- *currentFinishTime* — время окончания обработки текущей заявки;
- *totalRequests* — общее количество обработанных заявок;
- *totalProcessingTime* — общее время работы процессора;
- *totalWaitingTime* — общее время ожидание заявок в очереди.

Помимо этого, класс **Processor** содержит вложенный класс **Statistics**, описывающий собираемую по процессору статистику — общее количество обработанных заявок, среднее время обработки заявки и среднее время ожидания заявки в очереди. Статистика вычисляется с помощью метода *statistics* на основе полученных за время моделирования значений *totalRequests*, *totalProcessingTime*, *totalWaitingTime*.

Листинг 3.6 – simulator/Processor.kt

```
1  package simulator
2
3  import time.Time
4  import time.DurationGenerator
5  import mathutils.average
6
7  class Processor(
```

```
8       var durationGenerator: DurationGenerator,
9       var receivers: List<Processor>?
10  ) : Block {
11      data class Statistics(
12          val totalRequests: Int,
13          val averageProcessingTime: Time,
14          val averageWaitingTime: Time
15      )
16
17      private val queue: MutableList<Request> = mutableListOf()
18      private var currentRequest: Request? = null
19      private var currentStartTime: Time = 0
20      private var currentFinishTime: Time = 0
21      private var totalRequests: Int = 0
22      private var totalProcessingTime: Time = 0
23      private var totalWaitingTime: Time = 0
24
25      fun statistics(): Statistics = Statistics(
26          totalRequests = totalRequests,
27          averageProcessingTime = average(totalProcessingTime,
                ↪ totalRequests),
28          averageWaitingTime =  average(totalWaitingTime,
                ↪ totalRequests),
29      )
30
31      fun enqueue(request: Request) {
32  //          println(this)
33          queue.add(request)
34      }
35
36      fun queueSize(): Int = queue.size
37
38      override fun cleanupState() {
39          queue.clear()
40          currentRequest = null
41          currentStartTime = 0
42          currentFinishTime = 0
43          totalRequests = 0
44          totalProcessingTime = 0
45          totalWaitingTime = 0
46      }
```

```
47
48      override fun currentFinishTime(): Time = currentFinishTime
49
50      override fun start(currentTime: Time): Time? {
51          if (currentRequest != null || queue.isEmpty())
52              return null;
53
54          val finishTime = currentTime + durationGenerator.generate()
55          currentRequest = queue.removeFirst()
56          currentStartTime = currentTime
57          currentFinishTime = finishTime
58          totalWaitingTime += currentTime - currentRequest!!.timeIn
59
60          return currentFinishTime
61      }
62
63      override fun finish(currentTime: Time): Processor? {
64          if (currentRequest == null) return null
65
66          totalRequests += 1
67          totalProcessingTime += currentFinishTime - currentStartTime
68          currentRequest!!.timeOut = currentTime
69
70          val receiver = receivers?.minByOrNull { it.queueSize() }
71          receiver?.enqueue(currentRequest!!)
72
73          currentRequest = null
74          currentStartTime = 0
75          currentFinishTime = 0
76
77          return receiver
78      }
79  }
```

В листинге 3.7 представлен класс **Generator**, описывающий блок генератора в моделируемой системе. Как уже было сказано выше, он реализует интерфейс **Block**. Рассмотрим подробнее поля класса:

– *durationGenerator* — генератор продолжительности генерации заявки;

– *receivers* — список процессоров-получателей сгенерированной заявки;

– *currentRequest* — текущая генерируемая заявка;

– *currentStartTime* — время начала генерации текущей заявки;

– *currentFinishTime* — время окончания генерации текущей заявки;

– *totalRequests* — общее количество сгенерированных заявок;

– *totalGenerationTime* — общее время работы генератора.

Помимо этого, класс **Generator** содержит вложенный класс **Statistics**, описывающий собираемую по генератору статистику — общее количество обработанных заявок и среднее время генерации заявки. Статистика вычисляется с помощью метода *statistics* на основе полученных за время моделирования значений *totalRequests*, *totalGenerationTime*.

Листинг 3.7 – simulator/Generator.kt

```
1  package simulator
2
3  import time.Time
4  import time.DurationGenerator
5  import mathutils.average
6
7  class Generator(
8      var durationGenerator: DurationGenerator,
9      var receivers: List<Processor>?
10 ) : Block {
11     data class Statistics(
12         val totalRequests: Int,
13         val averageGenerationTime: Time
14     )
15
16     private var currentRequest: Request? = null
17     private var currentStartTime: Time = 0
18     private var currentFinishTime: Time = 0
19     private var totalRequests: Int = 0
20     private var totalGenerationTime: Time = 0
21
22     fun statistics(): Statistics = Statistics(
23         totalRequests = totalRequests,
24         averageGenerationTime = average(totalGenerationTime,
           ↪ totalRequests)
```

```
25      )
26
27      override fun cleanupState() {
28          currentRequest = null
29          currentStartTime = 0
30          currentFinishTime = 0
31          totalRequests = 0
32          totalGenerationTime = 0
33      }
34
35      override fun currentFinishTime(): Time = currentFinishTime
36
37      override fun start(currentTime: Time): Time? {
38          if (currentRequest != null) return null
39
40          val finishTime = currentTime + durationGenerator.generate()
41          currentRequest = Request(finishTime)
42          currentStartTime = currentTime
43          currentFinishTime = finishTime
44
45          return currentFinishTime
46      }
47
48      override fun finish(currentTime: Time): Processor? {
49          if (currentRequest == null) return null
50
51          totalRequests += 1
52          totalGenerationTime += currentFinishTime - currentStartTime
53
54          val receiver = receivers?.minByOrNull { it.queueSize() }
55          receiver?.enqueue(currentRequest!!)
56
57          currentRequest = null
58          currentStartTime = 0
59          currentFinishTime = 0
60
61          return receiver
62      }
63  }
```

В листинге 3.8 представлен интерфейс **Simulator**, который реализуют классы-симуляторы для пошагового, событийного и комбинированного

алгоритмов. В интерфейсе представлен вложенный класс **Statistics**, описывающий собираемую в процессе моделирования статистику. Класс содержит статистику, собранную со всех процессоров и генераторов, а также общее время выполнения симуляции.

Листинг 3.8 – simulator/Simulator.kt

```kotlin
package simulator

import time.Time

interface Simulator {
    data class Statistics (
        val elapsed: Long,
        val generators: List<Generator.Statistics>,
        val processors: List<Processor.Statistics>
    )

    fun simulate(time: Time): Statistics
}
```

Листинги классов-симуляторов **HybridSimulator**, **TimeBasedSimulator** и **EventBasedSimulator** приведены в приложении — А.11, А.13 и А.12 соответственно — и реализуют описанные алгоритмы продвижения модельного времени.

3.2.1 Пример использования разработанного ПО

3.2.2 Моделирование МФЦ

3.2.3 Выводы

В рамках данного раздела был обоснован выбор программных средств реализации алгоритма. Было разработано программное обеспечение, реализующее комбинированный алгоритм продвижения модельного времени, и описаны ключевые особенности его реализации. Также был разработан пользовательский интерфейс и продемонстрирован пример работы программного обеспечения.

4 Исследовательский раздел

В данном разделе проводится оценка временных затрат для разработанного комбинированного алгоритма в сравнении с пошаговым и событийным алгоритмами. Также анализируются временные затраты на моделирование рассматриваемой в рамках данной работы системы массового обслуживания (МФЦ).

Ключевым недостатком пошагового алгоритма в сравнении с комбинированным является невозможность пропустить интервалы времени, не содержащие никаких событий. Как следствие, пошаговый алгоритм плохо адаптирован под квазисинхронное распределение событий, рассмотренное в аналитическом разделе. Событийный алгоритм, в свою очередь, теряет свою временную эффективность при росте очереди событий, а также при увеличении числа обращений к ней ввиду не константной сложности вставки событий.

Для измерения времени выполнения программного обеспечения используется устройство MacBook Air M1, 8-core GPU, 16 ГБ.

Для повышения точности результатов (ввиду наличия фактора случайности в процессе работы программного обеспечения) для каждого набора данных производится 100 прогонов алгоритмов с получением в итоге среднего значения.

4.1 Сравнение алгоритмов

Квазисинхронное распределение событий

Предлагается сравнить временные затраты пошагового, событийного и комбинированного алгоритмов на системе, состоящей из одного генератора и одного процессора. Продолжительность генерации заявки генерируется согласно равномерному квазисинхронному распределению с длиной пикового интервала в 100 тиков, минимальным и максимальным значением в 1 и 2 тика соответственно. Продолжительность обработки заявки генерируется в соответствии с равномерным распределением с минимальным и максимальным значением в 1 и 2 тика соответственно. Моделируется работа системы в течение 100000 тиков таймера модельного времени. При этом для пикового распределения изменяется длина непикового интервала времени (интервала между пиками). Для комбинированного алгоритма размер нулевого уровня

выбран в соответствии с длиной пикового интервала (100 тиков), а размер ненулевого уровня выбран равным 10. Шаг таймера модельного времени для пошагового алгоритма составляет 1 тик.

На рисунке 4.1 представлен график сравнения временных затрат алгоритмов в зависимости от длины непиковых интервалов.

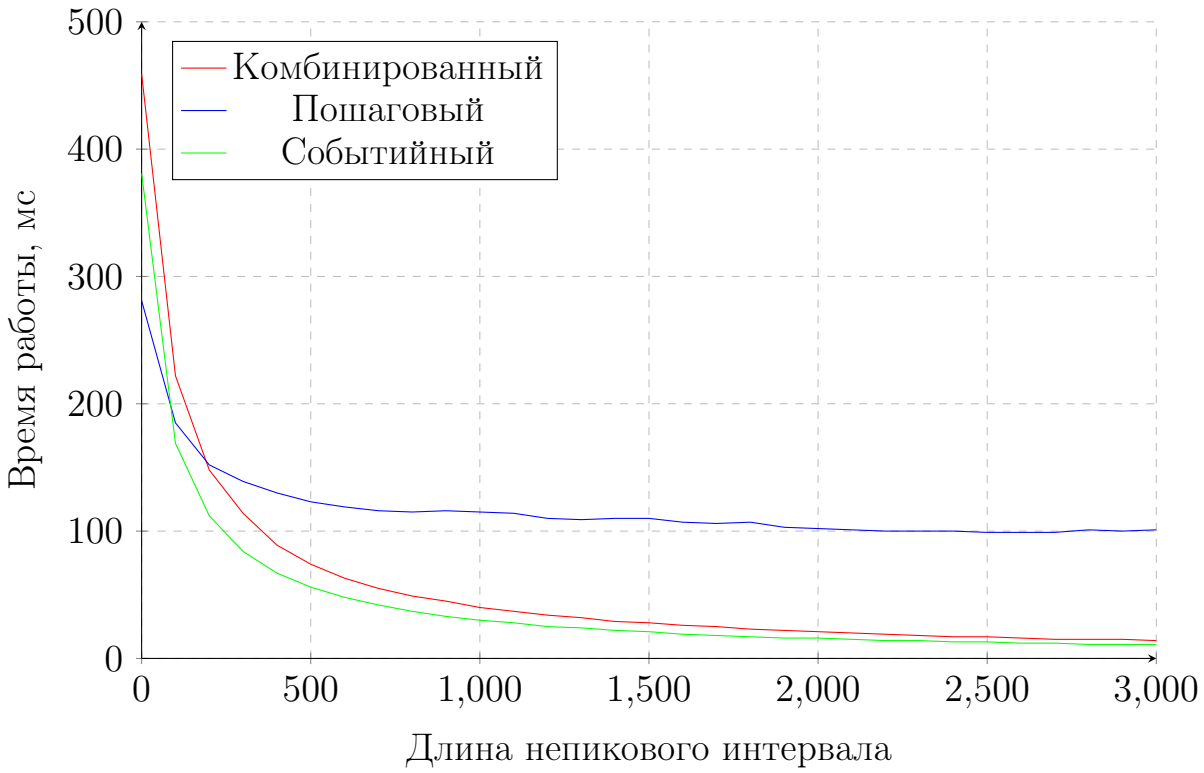

Рисунок 4.1 – Время работы комбинированного и пошагового алгоритмов в зависимости от длины непиковых интервалов

Проанализируем полученный результат. При малых (0—200 тиков) размерах непиковых интервалов временные затраты комбинированного алгоритма превышают временные затраты пошагового и событийного алгоритмов. Это связано с тем, что при данных величинах непикового интервала распределение событий стремится к равномерному, т.е. не квазисинхронному. Чем меньше длина непикового интервала, тем меньше промежутков, не насыщенных событиями, сооответственно, тем меньше теряет свою эффективность пошаговый алгоритм. Однако с увеличением длины непикового интервала на временной оси появляется все больше промежутков, не содержащих (или содержащих значительно меньше событий, чем пиковые интервалы), вследствие чего пошаговый алгоритм совершает «холостые» обходы блоков системы. Комбинированный алгоритм же ввиду особенностей часовой структуры имеет

возможность пропускать интервалы, не содержащие событий, что сокращает общее число операций. Подбор длины нулевого уровня в соответствии с длиной пикового интервала позволяет за один обход нулевого уровня обработать весь пиковый интервал целиком без необходимости повторных возвратов. Таким образом, временные затраты комбинированного алгоритма в сравнении с пошаговым будут меньше при увеличении длины непиковых интервалов. Событийный алгоритм однако показывает лучшие результаты в связи с тем, что рассмотренная система не нагружена событиями из-за малого количества блоков (1 генератор и 1 процессор).

Количество блоков в системе

Предлагается сравнить временные затраты пошагового, событийного и комбинированного алгоритмов на системе, состоящей из одного генератора и изменяющегося числа процессоров. Продолжительность генерации заявки генерируется согласно равномерному распределению с минимальным и максимальным значением в 1 и 2 тика соответственно. Продолжительность обработки заявки генерируется в соответствии с равномерным распределением с минимальным и максимальным значением в 15 и 16 тиков соответственно. Соответственно, интенсивность генерации зявок превышает интенсивность их обработки. Моделируется работа системы в течение 100000 тиков таймера модельного времени. При этом изменяется количество процессоров в системе для увеличения количества событий в системе. Для комбинированного алгоритма размер нулевого уровня выбран равным 100 тиков, а размер ненулевого уровня выбран равным 10. Шаг таймера модельного времени для пошагового алгоритма составляет 1 тик.

На рисунке 4.2 представлен график сравнения временных затрат алгоритмов в зависимости от количества процессоров в системе.

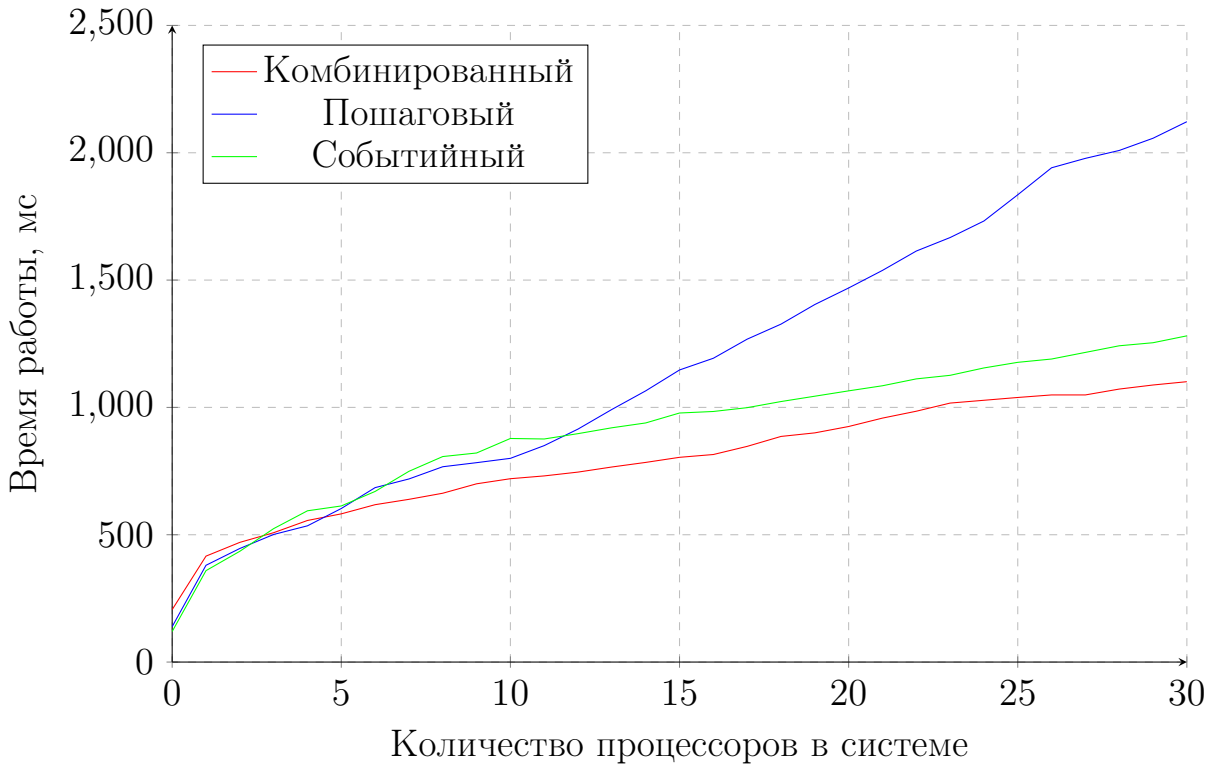

Рисунок 4.2 – Время работы алгоритмов в зависимости от количества процессоров в системе

Проанализируем полученные результаты. При малом (0—3) количестве процессоров в моделируемой системе комбинированный алгоритм уступает пошаговому и событийному по временным затратам. Это связано с тем, что очередь событийного алгоритма при таких данных не заполнена событиями. А с учетом рассматриваемого распределения для генерации заявок (равномерное от 1 до 2 тиков) пошаговый алгоритм практически на каждом тике находит событие и может быть даже эффективнее событийного. Однако с ростом количества процессоров для событийного алгоритма увеличивается количество одновременных событий в очереди и обращений к очереди в целом, а с учетом нелинейной сложности операций вставки в очередь временные затраты событийного алгоритма выше, чем временные затраты комбинированного. Ввиду обхода всего списка блоков системы на каждой итерации временные затраты пошагового алгоритма также становятся выше временных затрат комбинированного алгоритма.

4.2 Моделирование МФЦ

Проведем моделирование МФЦ, рассмотренной в конструкторском разделе. Длина нулевого уровня комбинированного алгоритма подобрана в соответствии с заявленной длиной пикового интервала (10800 тиков). Рассмотрим временные затраты алгоритмов в зависимости от числа окон обслуживания в МФЦ.

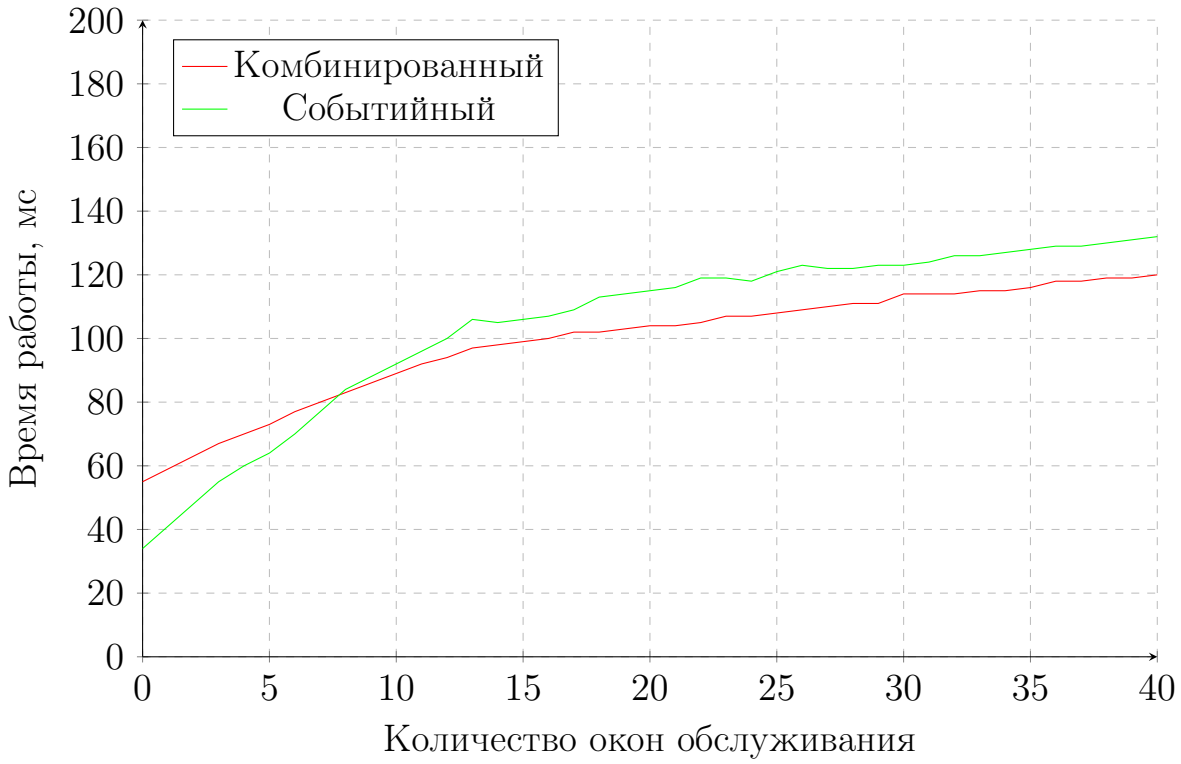

Рисунок 4.3 – Время работы комбинированного и событийного алгоритмов в зависимости от количества окон обслуживания

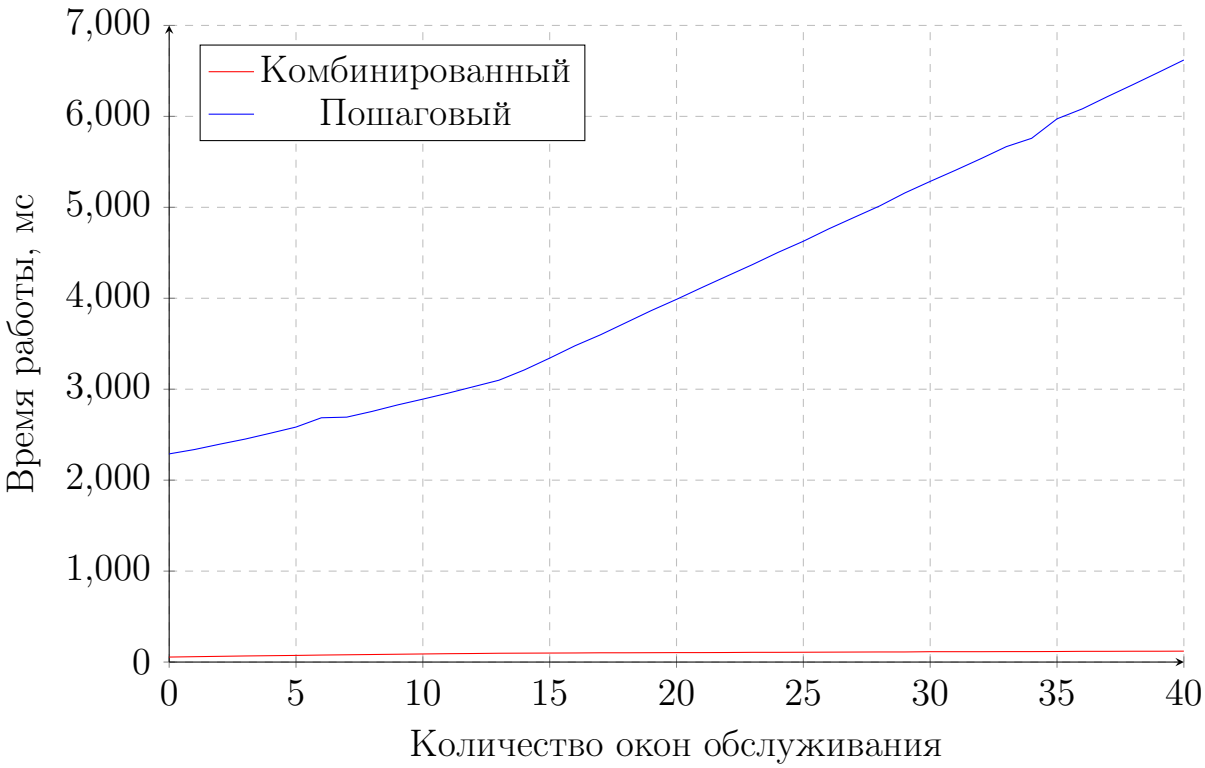

Рисунок 4.4 – Время работы комбинированного и пошагового алгоритмов в зависимости от количества окон обслуживания

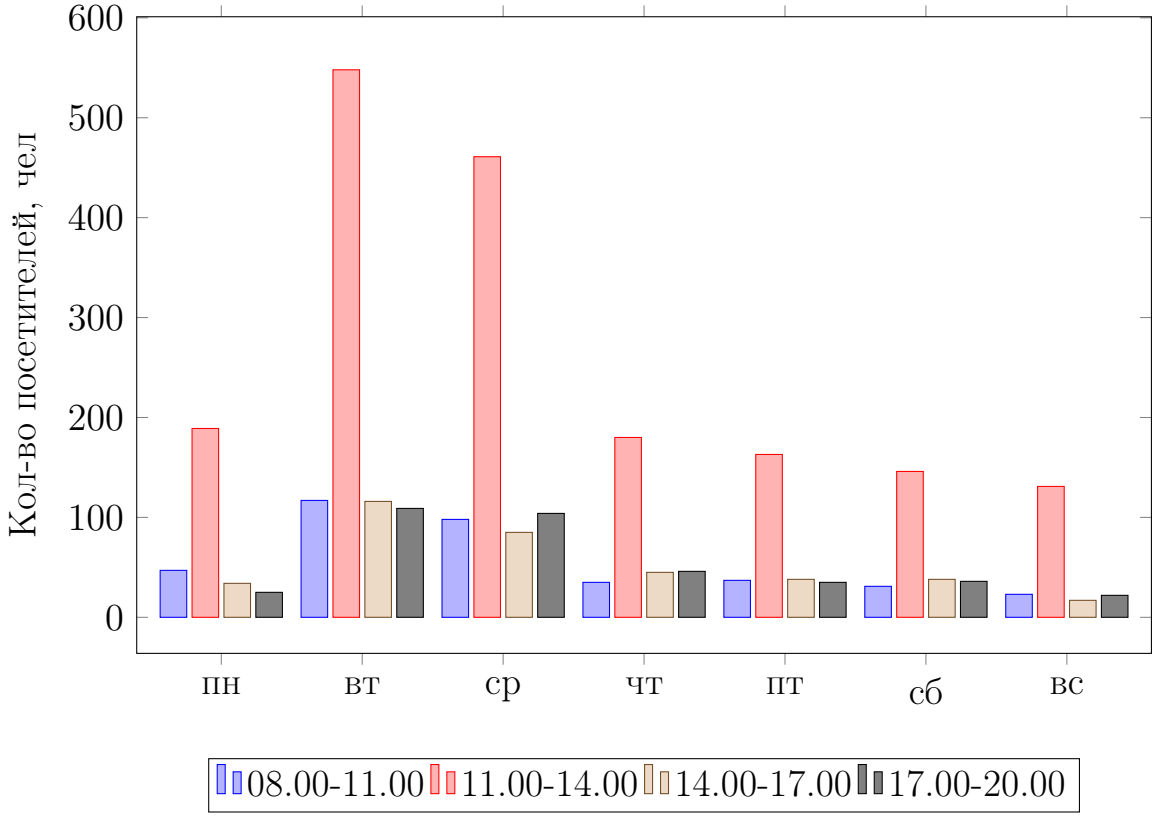

Рисунок 4.5 – Время работы комбинированного и пошагового алгоритмов в зависимости от количества окон обслуживания

Временные затраты событийного алгоритма оказались в среднем в 1.2 раза выше временных затрат комбинированного алгоритма при количестве окон > 7. Временные затраты пошагового алгоритма оказались в среднем в 35 раз выше временных затрат комбинированного алгоритма.

4.3 Выводы

В рамках данного раздела было проведено исследование характеристик разработанного комбинированного алгоритма продвижения модельного времени. Также был проведен сравнительный анализ комбинированного алгоритма с пошаговым и событийным.

ЗАКЛЮЧЕНИЕ

В результате выполнения данной работы были выполнены следующие задачи:

– рассмотрены существующие подходы к решению задачи протяжки модельного времени: пошаговый алгорим, событийный алгоритм и алгоритм Дельфт;

– сформулированы критерии сравнения рассмотренных методов такие, как простота реализации, универсальность, точность и эффективность;

– проведен сравнительный анализ алгоритмов по выделенным критериям, результаты сравнения наглядно оформлены в виде таблицы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Варфоломеев В.И.* Алгоритмическое моделирование элементов экономических систем. — Москва : Финансы и статистика, 2000. — Гл. 1.
2. *Вьюненко Л.Ф.*, *Михайлов М.В.*, *ПервозванскаяТ.Н.* Имитационное моделирование. — Москва : Юрайт, 2017. — Гл. 8.
3. *ТомашевскийВ.Н. Ж.* Имитационное моделирование средствами. — Киев : Питер, Издательская группа BHV, 1998. — Гл. 1.
4. *РоманенкоВ.А.* Системы и сети массового обслуживания. — Самара : Издательство Самарского университета, 2021. — Гл. 1.
5. *Аврамчук Е.Ф.*, *Вавилов А.А.*, *Емельянов С.В.* Технология системного моделирования. — Берлин : Машиностроение, 1988. — Гл. 9.
6. *Кельтон В.Д.*, *Лоу А.М.* Имитационное моделирование. Классика CS. — 3-е изд. — СПб : Питер, Издательская группа BHV, 2004. — Гл. 1.
7. *SedgewickR.*, *WayneK.* Algorithms. — Westford, Massachusetts : Addison-Wesley Professional, 2011. — Гл. 2.
8. *H. C.*, *E. L.*, *L. R.* Introduction to algorithms. — London, England : The MIT Press, 2001. — Гл. 6.
9. *Avor J.* An adaptive time advancement algorithm for discrete simulation // Information processing letters. — 1977. — Т. 6, № 6. — С. 83—86.
10. «Мои документы» обновили графики загруженности всех центров госуслуг столицы [Электронный ресурс]. — Режим доступа: `https://www.mos.ru/news/item/30714073/` (дата обращения: 15.05.2023).
11. Новый МФЦ открыт в Балашихе [Электронный ресурс]. — Режим доступа: `https://mits.mosreg.ru/sobytiya/novosti-ministerstva/28-03-2022-14-47-46-novyy-mfts-otkryt-v-balashikhe` (дата обращения: 15.05.2023).

ПРИЛОЖЕНИЕ А

Листинг А.1 – time/DurationGenerator.kt

```kotlin
package time

typealias Time = Int

interface DurationGenerator {
    fun generate(): Time
}
```

Листинг А.2 – time/UniformDurationGenerator.kt

```kotlin
package time

import kotlin.random.Random

class UniformDurationGenerator(
    private val min: Time,
    private val max: Time
) : DurationGenerator {
    private val random = Random(System.currentTimeMillis())

    override fun generate(): Time {
        return random.nextInt(min, max + 1)
    }

    fun getMin(): Time {
        return min
    }

    fun getMax(): Time {
        return max
    }
}
```

Листинг А.3 – time/PoissonDurationGenerator.kt

```kotlin
package time

import kotlin.math.ln
import kotlin.math.exp
```

```kotlin
import kotlin.random.Random

class PoissonDurationGenerator(
    private val min: Int,
    private val max: Int,
    private val period: Int
) : DurationGenerator {
    private val random = Random(System.currentTimeMillis())
    private var counter: Int = 0

    override fun generate(): Time {
        if (counter == period) {
            counter = 0
            return random.nextInt(500, 1000)
        }
        counter++
        return random.nextInt(min, max + 1)
    }

    fun getMin(): Time {
        return min
    }

    fun getMax(): Time {
        return max
    }

    fun getPeakLength(): Time {
        return period
    }
}
```

Листинг А.4 – mathutils/MathUtils.kt

```kotlin
package mathutils

fun average(sum: Int, count: Int): Int {
    if (count == 0) return 0
    return sum / count
}
```

Листинг А.5 – simulator/Event.kt

```kotlin
package simulator
```

```
2
3  import time.Time
4
5  data class Event(
6      val time: Time,
7      val block: Block
8  )
```

Листинг А.6 – simulator/Request.kt

```
1  package simulator
2
3  import time.Time
4
5  class Request(val timeIn: Time) {
6      var timeOut: Time = 0
7  }
```

Листинг А.7 – simulator/Block.kt

```
1  package simulator
2
3  import time.Time
4
5  interface Block {
6      fun cleanupState()
7      fun currentFinishTime(): Time
8      fun start(currentTime: Time): Time?
9      fun finish(currentTime: Time): Processor?
10 }
```

Листинг А.8 – simulator/Processor.kt

```
1  package simulator
2
3  import time.Time
4  import time.DurationGenerator
5  import mathutils.average
6
7  class Processor(
8      var durationGenerator: DurationGenerator,
9      var receivers: List<Processor>?
10 ) : Block {
11     data class Statistics(
12         val totalRequests: Int,
```

```
13          val averageProcessingTime: Time,
14          val averageWaitingTime: Time
15      )
16
17      private val queue: MutableList<Request> = mutableListOf()
18      private var currentRequest: Request? = null
19      private var currentStartTime: Time = 0
20      private var currentFinishTime: Time = 0
21      private var totalRequests: Int = 0
22      private var totalProcessingTime: Time = 0
23      private var totalWaitingTime: Time = 0
24
25      fun statistics(): Statistics = Statistics(
26          totalRequests = totalRequests,
27          averageProcessingTime = average(totalProcessingTime,
              ↪ totalRequests),
28          averageWaitingTime =  average(totalWaitingTime,
              ↪ totalRequests),
29      )
30
31      fun enqueue(request: Request) {
32  //          println(this)
33          queue.add(request)
34      }
35
36      fun queueSize(): Int = queue.size
37
38      override fun cleanupState() {
39          queue.clear()
40          currentRequest = null
41          currentStartTime = 0
42          currentFinishTime = 0
43          totalRequests = 0
44          totalProcessingTime = 0
45          totalWaitingTime = 0
46      }
47
48      override fun currentFinishTime(): Time = currentFinishTime
49
50      override fun start(currentTime: Time): Time? {
51          if (currentRequest != null || queue.isEmpty())
```

```
52              return null;
53
54          val finishTime = currentTime + durationGenerator.generate()
55          currentRequest = queue.removeFirst()
56          currentStartTime = currentTime
57          currentFinishTime = finishTime
58          totalWaitingTime += currentTime - currentRequest!!.timeIn
59
60          return currentFinishTime
61      }
62
63      override fun finish(currentTime: Time): Processor? {
64          if (currentRequest == null) return null
65
66          totalRequests += 1
67          totalProcessingTime += currentFinishTime - currentStartTime
68          currentRequest!!.timeOut = currentTime
69
70          val receiver = receivers?.minByOrNull { it.queueSize() }
71          receiver?.enqueue(currentRequest!!)
72
73          currentRequest = null
74          currentStartTime = 0
75          currentFinishTime = 0
76
77          return receiver
78      }
79  }
```

Листинг А.9 – simulator/Generator.kt

```
1   package simulator
2
3   import time.Time
4   import time.DurationGenerator
5   import mathutils.average
6
7   class Generator(
8       var durationGenerator: DurationGenerator,
9       var receivers: List<Processor>?
10  ) : Block {
11      data class Statistics(
12          val totalRequests: Int,
```

```
13          val averageGenerationTime: Time
14      )
15
16      private var currentRequest: Request? = null
17      private var currentStartTime: Time = 0
18      private var currentFinishTime: Time = 0
19      private var totalRequests: Int = 0
20      private var totalGenerationTime: Time = 0
21
22      fun statistics(): Statistics = Statistics(
23          totalRequests = totalRequests,
24          averageGenerationTime = average(totalGenerationTime,
                ↪ totalRequests)
25      )
26
27      override fun cleanupState() {
28          currentRequest = null
29          currentStartTime = 0
30          currentFinishTime = 0
31          totalRequests = 0
32          totalGenerationTime = 0
33      }
34
35      override fun currentFinishTime(): Time = currentFinishTime
36
37      override fun start(currentTime: Time): Time? {
38          if (currentRequest != null) return null
39
40          val finishTime = currentTime + durationGenerator.generate()
41          currentRequest = Request(finishTime)
42          currentStartTime = currentTime
43          currentFinishTime = finishTime
44
45          return currentFinishTime
46      }
47
48      override fun finish(currentTime: Time): Processor? {
49          if (currentRequest == null) return null
50
51          totalRequests += 1
52          totalGenerationTime += currentFinishTime - currentStartTime
```

```
53
54          val receiver = receivers?.minByOrNull { it.queueSize() }
55          receiver?.enqueue(currentRequest!!)
56
57          currentRequest = null
58          currentStartTime = 0
59          currentFinishTime = 0
60
61          return receiver
62      }
63  }
```

Листинг А.10 – simulator/Simulator.kt

```
1  package simulator
2
3  import time.Time
4
5  interface Simulator {
6      data class Statistics (
7          val elapsed: Long,
8          val generators: List<Generator.Statistics>,
9          val processors: List<Processor.Statistics>
10     )
11
12     fun simulate(time: Time): Statistics
13 }
```

Листинг А.11 – simulator/HybridSimulator.kt

```
1  package simulator
2
3  import kotlin.math.*
4  import kotlin.system.measureTimeMillis
5  import time.Time
6
7  typealias EventList = MutableList<Event>
8
9  class HybridSimulator(
10     private val generators: List<Generator>,
11     private val processors: List<Processor>,
12     private val arraySize: Int,
13     private val tableWidth: Int
14 ) : Simulator {
```

```
15
16      override fun simulate(time: Time): Simulator.Statistics {
17          val clock = Clock(arraySize, tableWidth, time)
18          generators.forEach { it.cleanupState() }
19          processors.forEach { it.cleanupState() }
20          generateInitialEvents(clock)
21
22          return Simulator.Statistics(
23              elapsed = measureTimeMillis { runSimulate(clock) },
24              generators = generators.map { it.statistics() },
25              processors = processors.map { it.statistics() }
26          )
27      }
28
29      private fun generateInitialEvents(clock: Clock) {
30          generators.forEach{
31              val eventTime = it.start(1)
32              if (eventTime != null) {
33                  val event = Event(eventTime, it)
34                  clock.addInitialEvent(event)
35              }
36          }
37      }
38
39      private fun runSimulate(clock: Clock) {
40          clock.processLevel(clock.tableHeight, ::processEvent)
41      }
42
43      private fun processEvent(clock: Clock, event: Event) {
44          require(event.time == clock.currentTime)
45
46          val block = event.block
47
48          val nextProcessor = block.finish(clock.currentTime)
49          if (nextProcessor != null) {
50              val nextProcessorNextEventTime = nextProcessor.start(
                    ↪ clock.currentTime)
51              if (nextProcessorNextEventTime != null) {
52                  val nextEvent = Event(nextProcessorNextEventTime,
                        ↪ nextProcessor)
53                  clock.addEvent(nextEvent)
```

```
54                  }
55              }
56
57              val nextEventTime = block.start(clock.currentTime)
58              if (nextEventTime != null) {
59                  val nextEvent = Event(nextEventTime, block)
60                  clock.addEvent(nextEvent)
61              }
62          }
63      }
64
65      class Clock {
66          val arraySize: Int
67          val array: Array<EventList>
68
69          val tableHeight: Int
70          val tableWidth: Int
71          val table: Array<Array<EventList>>
72
73          val maxTime: Int
74          var currentTime: Time
75          var currentEnd: Time
76
77          val granularityCache: Array<Int>
78
79          constructor(arraySize: Int, tableWidth: Int, maxTime: Time) {
80              this.arraySize = arraySize
81              this.tableWidth = tableWidth
82
83              this.maxTime = maxTime
84              this.currentTime = 0
85              this.currentEnd = arraySize
86
87              val arraysCount = maxTime.toDouble() / arraySize.toDouble()
88              this.tableHeight = ceil(log(arraysCount, tableWidth.toDouble
                    ↪ ())).toInt()
89
90              array = Array(arraySize) { mutableListOf() }
91              table = Array(tableHeight) { Array(tableWidth) {
                    ↪ mutableListOf() } }
92
```

```
93          granularityCache = Array(tableHeight + 1) {
94              if (it == 0) 1
95              else (arraySize * tableWidth.toDouble().pow(it - 1)).
                    ↪ toInt()
96          }
97      }
98
99      fun addInitialEvent(event: Event) {
100         if (event.time > maxTime) return
101         addEventToLevel(event, tableHeight)
102     }
103
104     fun addEvent(event: Event) {
105         if (event.time > maxTime) return
106         if (event.time <= currentEnd) {
107             addEventToLevel(event, 0)
108         } else {
109             val level = levelByTime(event.time)
110             addEventToLevel(event, level)
111         }
112     }
113
114     fun processLevel(level: Int, processEvent: (Clock, Event) ->
        ↪ Unit) {
115         if (level == 0) {
116             currentEnd = currentTime + arraySize
117             array.forEach { events ->
118                 currentTime++
119                 while (events.isNotEmpty()) {
120                     val event = events.removeLast()
121                     processEvent(this, event)
122                 }
123             }
124             return
125         }
126
127         val row = table[level-1]
128         row.forEachIndexed { j, events ->
129             if (events.isEmpty()) {
130                 currentTime += granularity(level)
131             } else {
```

```
                var retries = 0
                while (events.isNotEmpty()) {
                    if (retries > 0)
                        currentTime -= granularity(level)
                    moveEventsToLowerLevel(level, column=j+1)
                    processLevel(level - 1, processEvent)
                    retries++
                }
            }
        }
    }

    private fun addEventToLevel(event: Event, level: Int) {
        val column = columnByTimeAndLevel(event.time, level)
        if (level == 0) array[column-1].add(event)
        else table[level-1][column-1].add(event)
    }

    private fun granularity(level: Int): Int =
        granularityCache[level]

    private fun levelByTime(time: Time): Int =
        ceil(log(time.toDouble() / arraySize, tableWidth.toDouble())
            ↪ ).toInt()

    private fun columnByTimeAndLevel(time: Time, level: Int): Int {
        if (level == 0) return (time - 1) % arraySize + 1
        return ceil(time.toDouble() / granularity(level)).toInt()
    }

    private fun moveEventsToLowerLevel(level: Int, column: Int) {
        require(level > 0)

        val events = table[level-1][column-1]
        events.forEach {
            val lowerLevel = level - 1
            val lowerTime = (it.time - 1) % granularity(level) + 1
            val lowerColumn = columnByTimeAndLevel(lowerTime,
                ↪ lowerLevel)

            if (lowerLevel == 0) array[lowerColumn-1].add(it)
```

```
171                 else table[lowerLevel-1][lowerColumn-1].add(it)
172             }
173             events.clear()
174         }
175
176         fun print() {
177             table.reversed().forEachIndexed { rowIndex, row ->
178                 val index = table.size - rowIndex
179                 print("${index}) ")
180                 row.indices.forEach { colIndex ->
181                     print("${colIndex+1}{${row[colIndex].joinToString(",
                        ↪ ") { "${it.time}" }}} ")
182                 }
183                 println()
184             }
185             print("0) ")
186             array.indices.forEach { rowIndex ->
187                 print("${rowIndex+1}{${array[rowIndex].joinToString(",")
                    ↪  { "${it.time}" }}} ")
188             }
189             println()
190             println()
191         }
192 }
```

Листинг А.12 – simulator/EventBasedSimulator.kt

```
1  package simulator
2
3  import kotlin.system.measureTimeMillis
4  import java.util.PriorityQueue
5  import time.Time
6
7  class EventBasedSimulator(
8      private val generators: List<Generator>,
9      private val processors: List<Processor>
10 ) : Simulator {
11     private val queue: PriorityQueue<Event> =
12         PriorityQueue { a, b -> a.time - b.time }
13
14     override fun simulate(time: Time): Simulator.Statistics {
15         generators.forEach { it.cleanupState() }
16         processors.forEach { it.cleanupState() }
```

```
17          generateInitialEvents()
18
19          return Simulator.Statistics(
20              elapsed = measureTimeMillis { runSimulate(time) },
21              generators = generators.map { it.statistics() },
22              processors = processors.map { it.statistics() }
23          )
24      }
25
26      private fun generateInitialEvents() {
27          generators.forEach {
28              val eventTime = it.start(0)
29              if (eventTime != null)
30                  queue.add(Event(eventTime, it))
31          }
32      }
33
34      private fun runSimulate(maxTime: Time) {
35          while (queue.isNotEmpty()) {
36  //              println(queue.size)
37              val event = queue.poll()
38              val block = event.block
39              val currentTime = block.currentFinishTime()
40
41              val nextProcessor = block.finish(currentTime)
42              if (nextProcessor != null) {
43                  val nextEventTime = nextProcessor.start(currentTime)
44                  if (nextEventTime != null && nextEventTime < maxTime
                        ↪ ) {
45                      val nextEvent = Event(nextEventTime,
                            ↪ nextProcessor)
46                      queue.add(nextEvent)
47                  }
48              }
49
50              val nextEventTime = block.start(currentTime)
51              if (nextEventTime != null && nextEventTime < maxTime) {
52                  val nextEvent = Event(nextEventTime, block)
53                  queue.add(nextEvent)
54              }
55          }
```

```
    }
}
```

Листинг А.13 – simulator/TimeBasedSimulator.kt

```kotlin
package simulator

import kotlin.system.measureTimeMillis
import time.Time

class TimeBasedSimulator(
    private val generators: List<Generator>,
    private val processors: List<Processor>,
    private val deltaT: Int = 1
) : Simulator {

    override fun simulate(time: Time): Simulator.Statistics {
        generators.forEach { it.cleanupState() }
        processors.forEach { it.cleanupState() }

        return Simulator.Statistics(
            elapsed = measureTimeMillis { runSimulate(time) },
            generators = generators.map { it.statistics() },
            processors = processors.map { it.statistics() }
        )
    }

    private fun runSimulate(maxTime: Time) {
        var currentTime: Time = 0
        while (currentTime < maxTime) {
            generators.forEach { processBlock(currentTime, it) }
            processors.forEach { processBlock(currentTime, it) }
            currentTime += deltaT
        }
    }

    private fun processBlock(currentTime: Time, block: Block) {
        if (block.currentFinishTime() <= currentTime) {
            val nextProcessor = block.finish(currentTime)
            nextProcessor?.start(currentTime)
            block.start(currentTime)
        }
    }
```

```
39 }
```

ПРИЛОЖЕНИЕ Б