СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Во время выполнения выпускной квалификационной работы был разработан комбинированный алгоритм продвижения модельного времени.

Задача продвижения модельного времени является одной из важнейших задач имитационного моделирования [1], находящего применение во множестве областей в связи с дорогими или невозможными исследованиями реальных систем.

1 Формальная постановка задачи

IDEF0-диаграмма для комбинированного алгоритма продвижения модельного времени представлена на рисунках 1.1 1.2.

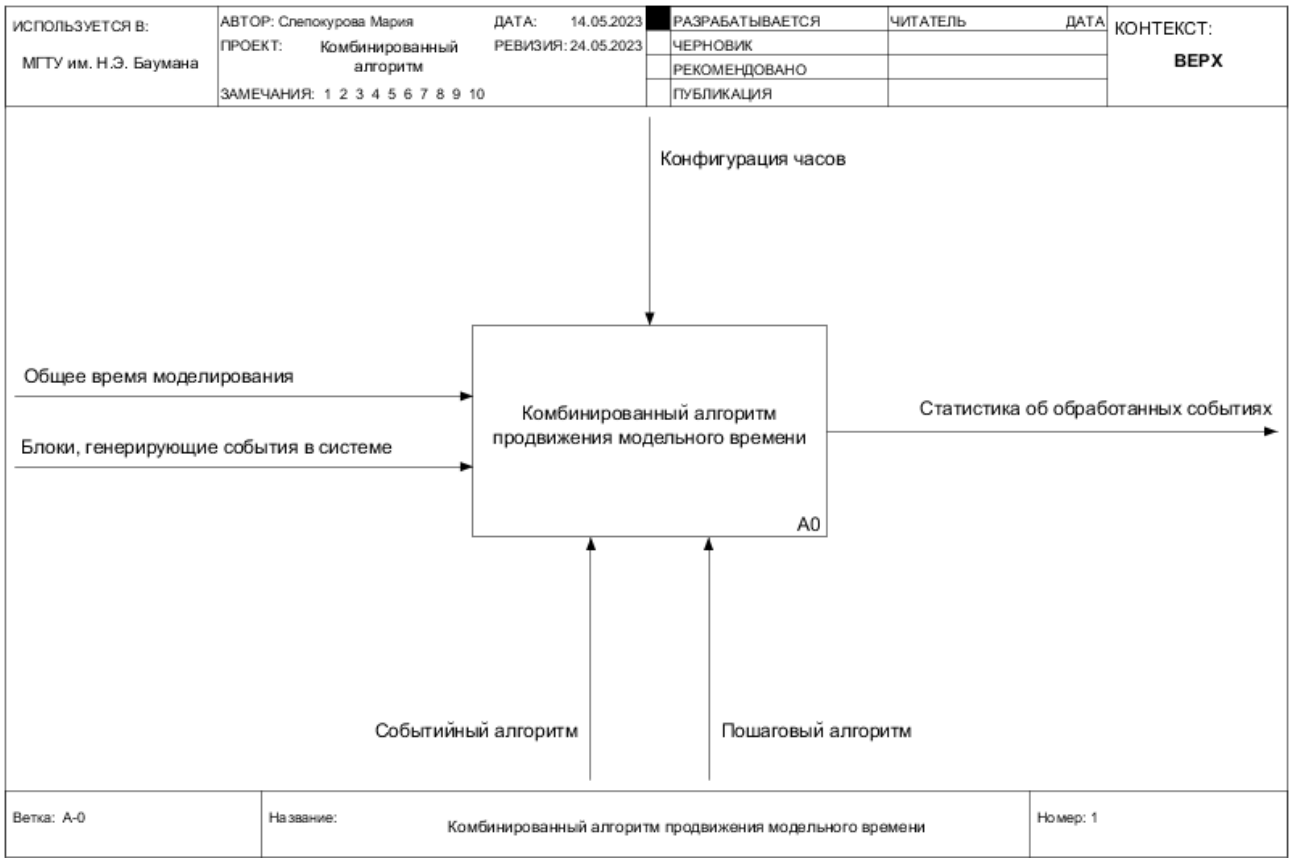

Рисунок 1.1 – IDEF0 нулевого уровня

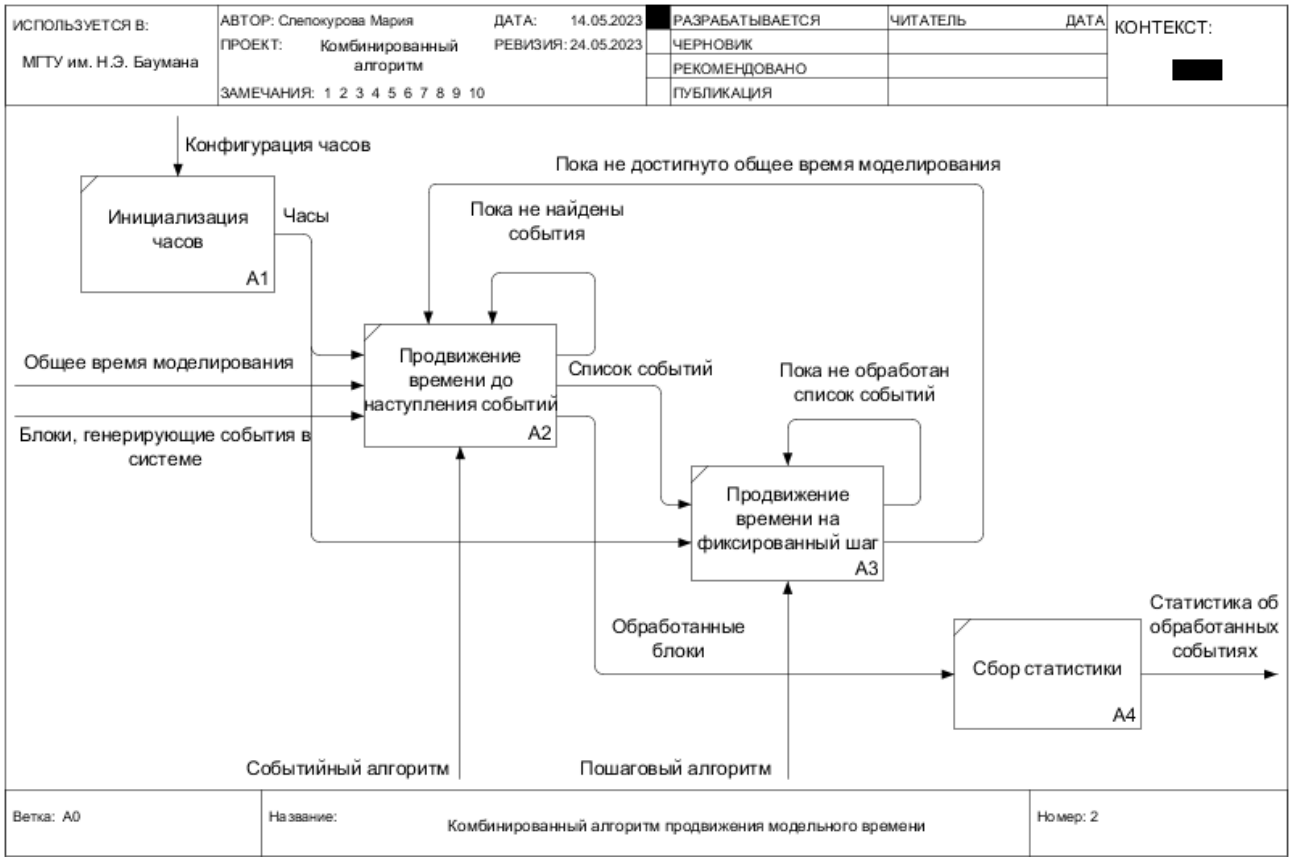

Рисунок 1.2 – IDEF0 первого уровня

2 Разработанный алгоритм

2.1 Схема алгоритма

На рисунке 2.1 представлена схема комбинированного алгоритма продвижения модельного времени.

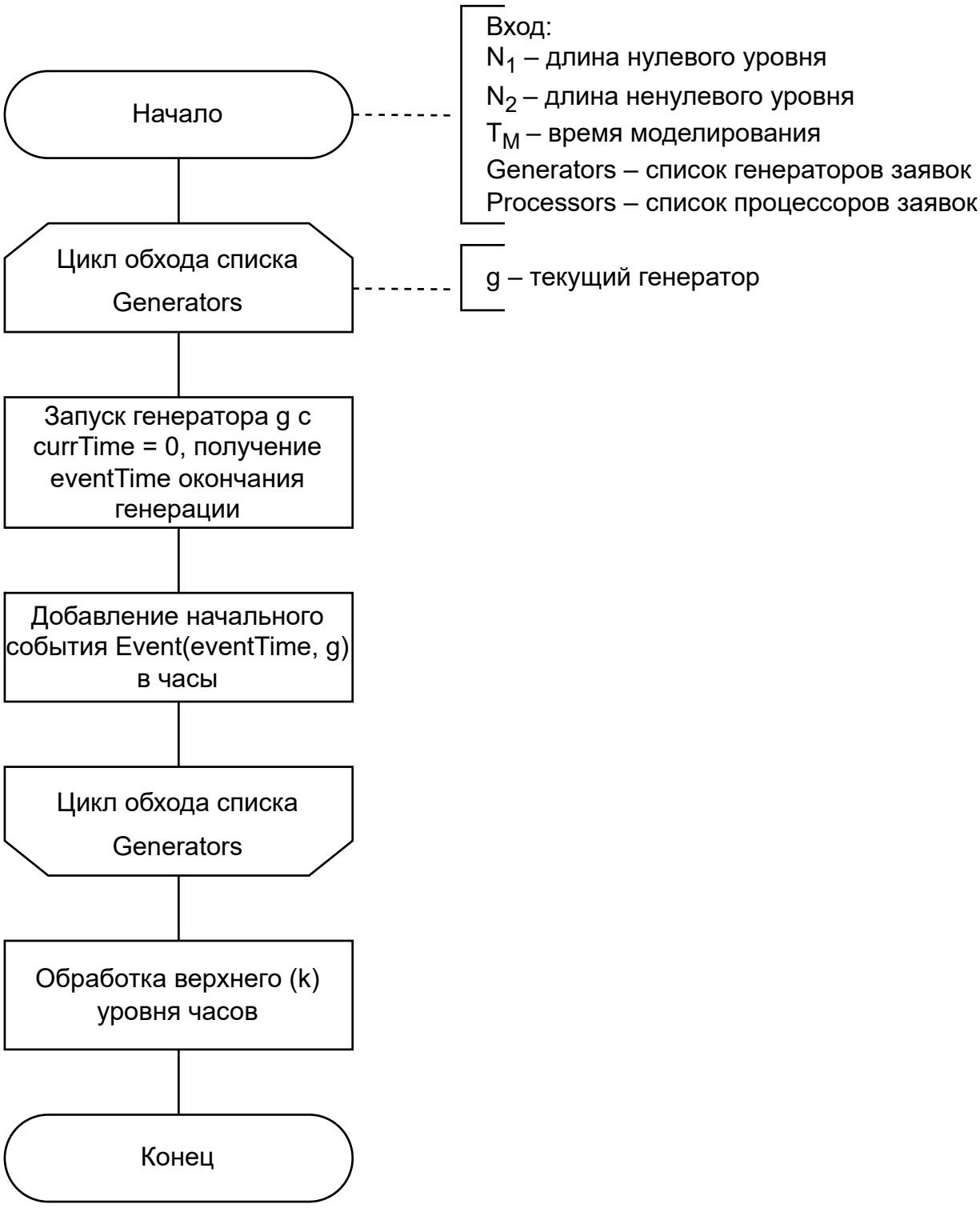

Рисунок 2.1 – Схема комбинированного алгоритма

На рисунке 2.2 представлена схема алгоритма обработки одного уровня часов.

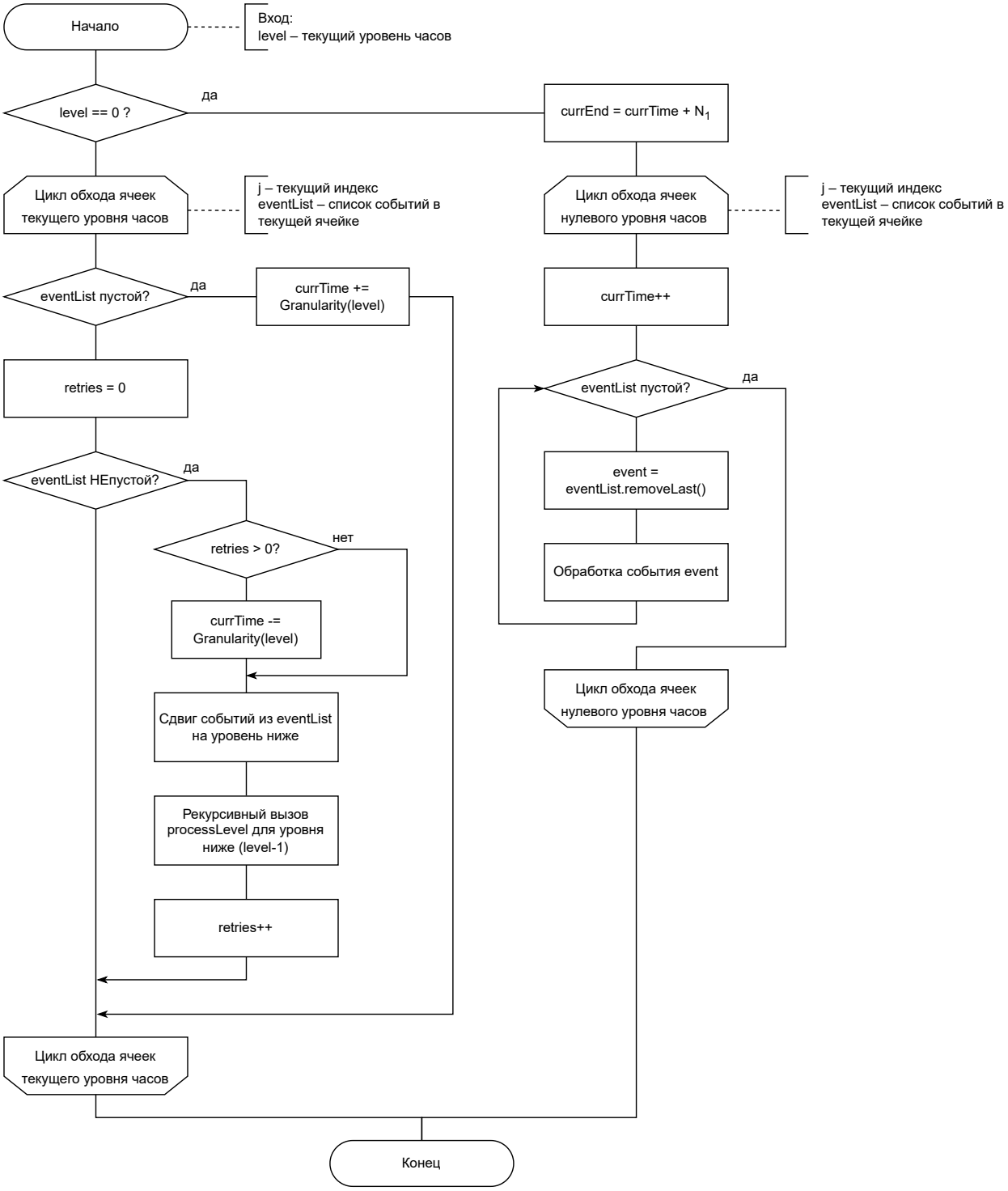

Рисунок 2.2 – Схема алгоритма обработки одного уровня часовой структуры

На рисунке 2.3 представлена схема алгоритма перемещения событий из ячейки часовой структуры на уровень ниже.

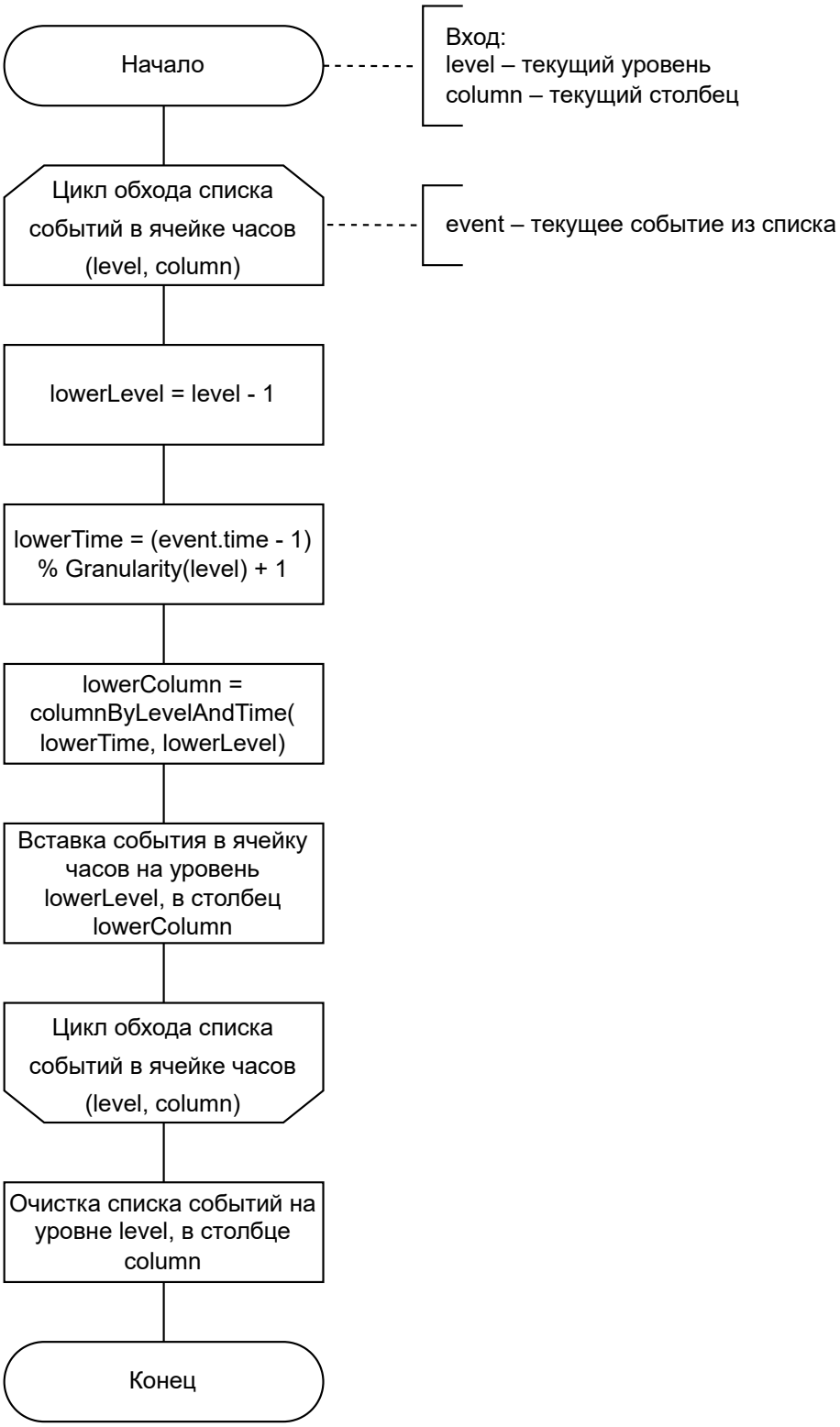

Рисунок 2.3 – Схема алгоритма перемещения событий из ячейки часовой структуры на уровень ниже

На рисунке 2.4 представлена схема алгоритма обработки одного события.

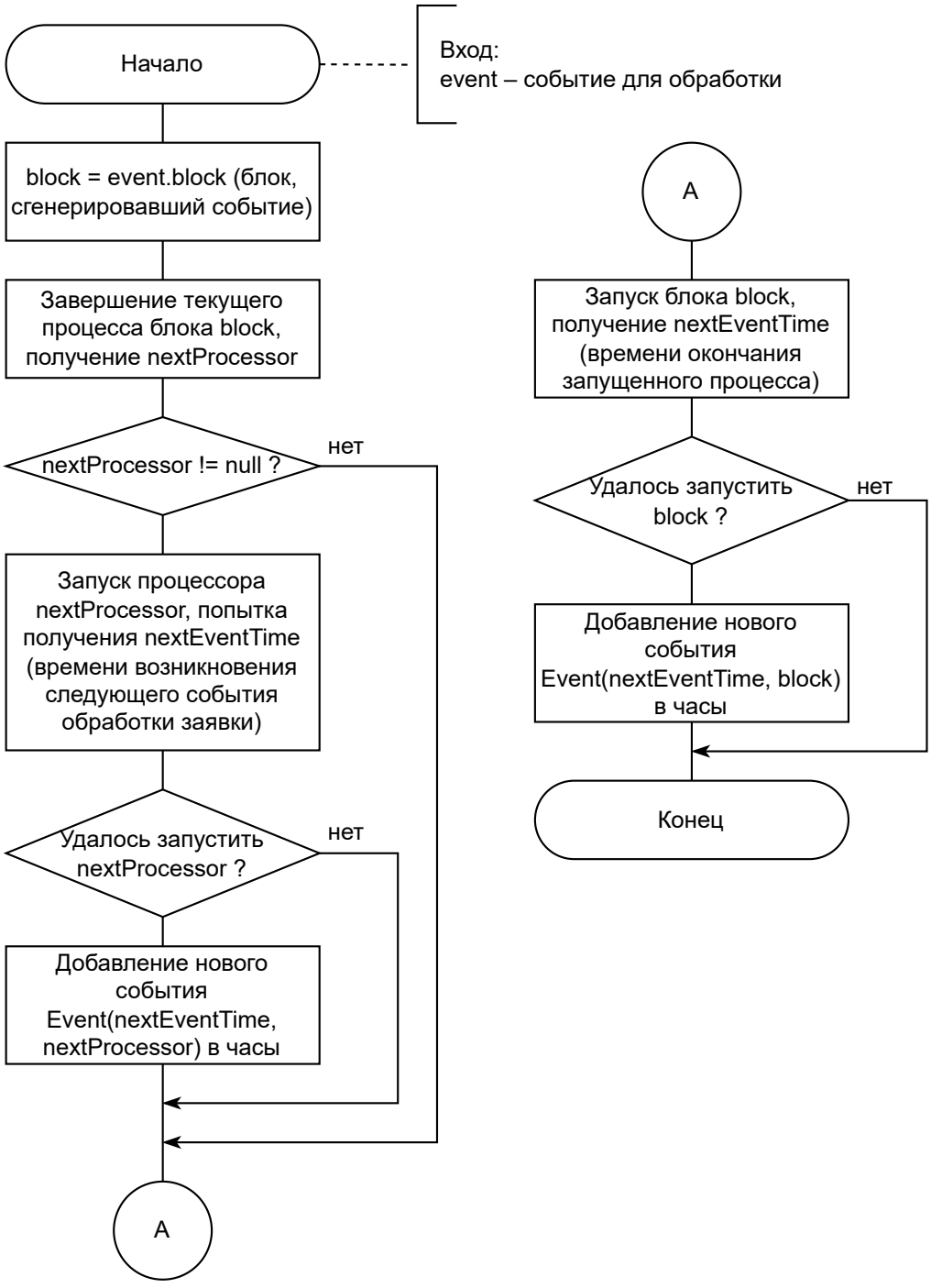

Рисунок 2.4 – Схема алгоритма обработки одного события

2.2 Описание реализации

В листинге 2.1 представлен интерфейс **DurationGenerator**, который реализуют все классы-генераторы продолжительности, например, класс **UniformDurationGenerator**, представленный в листинге 2.2. Метод

generate класса **UniformDurationGenerator** используется для генерации продолжительности 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 2.1 – time/DurationGenerator.kt

```kotlin
package time

typealias Time = Int

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

Листинг 2.2 – 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 2.3 представлен интерфейс **Block**, который реализуют классы

Processor и **Generator**. Метод *cleanupState* используется для очищения всей статистики и текущего 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 2.3 – 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 2.4 представлен класс **Event**, описывающий события в моделируемой системе, где *time* — время наступления события, *block* — блок системы, сгенерировавший событие.

Листинг 2.4 – 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 2.5 представлен класс **Request**, описывающий заявки в моделируемой системе, где *timeIn* — время поступления заявки в с систему, *timeOut* — время выхода заявки из системы.

Листинг 2.5 – 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 2.6 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 2.6 – 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 2.7 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 2.7 – 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 2.8 представлен интерфейс **Simulator**, который реализуют классы-симуляторы для пошагового, событийного и комбинированного алгоритмов. В интерфейсе представлен вложенный класс **Statistics**, описы-

вающий собираемую в процессе моделирования статистику. Класс содержит статистику, собранную со всех процессоров и генераторов, а также общее время выполнения симуляции.

Листинг 2.8 – simulator/Simulator.kt

```
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
}
```

Листинг класса-симулятора **HybridSimulator**, реализующего комбинированный алгоритм продвижения модельного времени, приведен в приложении — А.1

ЗАКЛЮЧЕНИЕ

Было разработано программное обеспечение, демонстрирующее практическую осуществимость разработанного в ходе выполнения выпускной квалификационной работы комбинированного алгоритма продвижения модельного времени.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Кельтон В.Д.*, *Лоу А.М.* Имитационное моделирование. Классика CS. — 3-е изд. — СПб : Питер, Издательская группа BHV, 2004. — Гл. 1.
2. *Вьюненко Л.Ф.*, *Михайлов М.В.*, *ПервозванскаяТ.Н.* Имитационное моделирование. — Москва : Юрайт, 2017. — Гл. 8.
3. *Avor J.* An adaptive time advancement algorithm for discrete simulation // Information processing letters. — 1977. — Т. 6, № 6. — С. 83—86.

ПРИЛОЖЕНИЕ А

Листинг А.1 – simulator/HybridSimulator.kt

```kotlin
package simulator

import kotlin.math.*
import kotlin.system.measureTimeMillis
import time.Time

typealias EventList = MutableList<Event>

class HybridSimulator(
    private val generators: List<Generator>,
    private val processors: List<Processor>,
    private val arraySize: Int,
    private val tableWidth: Int
) : Simulator {

    override fun simulate(time: Time): Simulator.Statistics {
        val clock = Clock(arraySize, tableWidth, time)
        generators.forEach { it.cleanupState() }
        processors.forEach { it.cleanupState() }
        generateInitialEvents(clock)

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

    private fun generateInitialEvents(clock: Clock) {
        generators.forEach{
            val eventTime = it.start(1)
            if (eventTime != null) {
                val event = Event(eventTime, it)
                clock.addInitialEvent(event)
            }
        }
    }
```

```
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 {
132                 var retries = 0
133                 while (events.isNotEmpty()) {
134                     if (retries > 0)
135                         currentTime -= granularity(level)
136                     moveEventsToLowerLevel(level, column=j+1)
137                     processLevel(level - 1, processEvent)
138                     retries++
139                 }
140             }
141         }
142     }
143 
144     private fun addEventToLevel(event: Event, level: Int) {
145         val column = columnByTimeAndLevel(event.time, level)
146         if (level == 0) array[column-1].add(event)
147         else table[level-1][column-1].add(event)
148     }
149 
150     private fun granularity(level: Int): Int =
151         granularityCache[level]
152 
153     private fun levelByTime(time: Time): Int =
```

```
154         ceil(log(time.toDouble() / arraySize, tableWidth.toDouble())
                ↪ ).toInt()
155
156     private fun columnByTimeAndLevel(time: Time, level: Int): Int {
157         if (level == 0) return (time - 1) % arraySize + 1
158         return ceil(time.toDouble() / granularity(level)).toInt()
159     }
160
161     private fun moveEventsToLowerLevel(level: Int, column: Int) {
162         require(level > 0)
163
164         val events = table[level-1][column-1]
165         events.forEach {
166             val lowerLevel = level - 1
167             val lowerTime = (it.time - 1) % granularity(level) + 1
168             val lowerColumn = columnByTimeAndLevel(lowerTime,
                    ↪ lowerLevel)
169
170             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 }
```