МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ИНСТИТУТ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ СИСТЕМ
ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ
КАФЕДРА ЭЛЕКТРОННЫХ ВЫЧИСЛИТЕЛЬНЫХ МАШИН

Направление 09.03.01 - Информатика и вычислительная техника
(код и наименование направления)
Профиль 01 – Вычислительные машины, комплексы, системы и сети

Допускаю к защите
Заведующий кафедрой ЭВМ
_____ /Страбыкин Д.А. /
(подпись) (Ф.И.О.)

Разработка программы проверки операторных схем алгоритмов

Пояснительная записка выпускной квалификационной работы
ТПЖА 090301.014 ПЗ

Разработал: студент гр.ИВТб-4301-01-00 _____ / Баташев П.А./ _____

Руководитель: к.т.н., доцент _____ / Скворцов А.А./ _____

Нормоконтролер: к.т.н., доцент _____ / Скворцов А.А./ _____
(подпись) *(Ф.И.О.)* *(дата)*

Киров 2020

Реферат

Баташев П.А. Разработка программы проверки операторных схем алгоритмов: ТПЖА.090301.014 ПЗ: Выпускная квалификационная работа / ВятГУ, каф. ЭВМ; рук. к.т.н., доцент Скворцов А.А. – Киров, 2020. – Гр.ч 8 л. ф. А1; ПЗ 103 с., 77 рис., 10 источников, 12 прил.

МАТЕМАТИЧЕСКАЯ ЛОГИКА, ТЕОРИЯ АЛГОРИТМОВ, ОПЕРАТОРНЫЕ СХЕМЫ, ОПЕРАТОР, ЛОГИЧЕСКОЕ УСЛОВИЕ, ГРАФИЧЕСКАЯ СХЕМА, МАТРИЧНАЯ СХЕМА, ЛОГИЧЕСКАЯ СХЕМА, ГЕНЕРАЦИЯ АЛГОРИТМА, ОБЪЕКТЫ, ВЕРШИНЫ, ЛИНИИ, РЕКУРСИВНЫЙ ОБХОД ГРАФА, АВТОМАТ МУРА, ЛЕКСЕМЫ, ЛЕКСИЧЕСКИЙ АНАЛИЗ, СИНТАКСИЧЕСКИЙ АНАЛИЗ.

Объект выпускной квалификационной работы – лабораторная работа «Схемы алгоритмов» по дисциплине «Математическая логика и теория алгоритмов».

Целью данной выпускной квалификационной работы является повышение качества обучения студентов за счет разработки лабораторной установки по дисциплине «Математическая логика и теория алгоритмов».

В рамках данной выпускной квалификационной работы было разработано представление операторной схемы алгоритма в виде класса, был разработан генератор задания в виде словесного описания алгоритма, были разработаны интерфейсы построения и редактирования операторных схем алгоритмов, были разработаны алгоритмы проверки построенных операторных схем алгоритмов.

Продолжением данной выпускной квалификационной работы может быть расширение функционала разработанной программы до возможности выполнения лабораторной работы «Объединение операторных схем алгоритмов» по дисциплине «Математическая логика и теория алгоритмов»

Содержание

Введение 4
1. Анализ состояния проблемы 5
1.1. Обзор аналогов и научной литературы 5
1.2. Направление разработки программы 5
2. Постановка задачи 7
2.1. Основание для разработки 7
2.2. Назначение разработки 7
2.3. Требования к программе 7
2.4. Требования к документации 8
3. Разработка модульной структуры программы 10
4. Генерация задания 13
4.1. Вершины алгоритма 13
4.2. Операторный алгоритм 14
4.3. Генерация операторного алгоритма 15
4.4. Программная реализация 25
5. Инструменты построения графической схемы алгоритма 29
5.1. Объекты графической схемы 29
5.2. Программная реализация классов объектов 34
5.3. Интерфейс взаимодействия с объектами 41
6. Проверка графической схемы алгоритма 49
7. Проверка матричной схемы алгоритма 54
7.1. Разбиение строки на лексемы 54
7.2. Преобразование набора лексем в набор переходов 56
7.3. Проверка набора переходов 61
8. Проверка логической схемы алгоритма 62
8.1. Преобразования набора лексем в набор переходов 62
8.2. Проверка наборов переходов 70

9. Разработка пользовательского интерфейса..... 71
10. Построение автоматов для преобразования формул переходов..... 75
10.1. Система формул переходов S1 75
10.2. Система скобочных формул переходов S2..... 80
10.3. Система схемных формул переходов S3..... 85
Заключение..... 90
Приложение А. Автомат преобразования ячейки матрицы..... 92
Приложение Б. Автомат преобразования системы формул переходов..... 93
Приложение В. Автомат преобразования системы скобочных формул..... 94
Приложение Г. Автомат преобразования системы схемных формул..... 95
Приложение Д. Автомат преобразования логической схемы алгоритма..... 96
Приложение Е. Входные сигналы автоматов..... 97
Приложение Ж. Выходные сигналы автомата преобразования ячейки матрицы..... 98
Приложение З. Выходные сигналы автомата преобразования системы формул переходов..... 99
Приложение И. Выходные сигналы автомата преобразования системы скобочных формул..... 100
Приложение Й. Выходные сигналы автомата преобразования системы схемных формул..... 101
Приложение К. Выходные сигналы автомата преобразования логической схемы алгоритма..... 102
Приложение Л. (справочное) Библиографический список..... 103

Введение

Для формального описания алгоритмов переработки информации в устройствах управления разработаны специальные языки, которые нашли широкое применение в процессе проектирования и эксплуатации ЭВМ, вычислительных системах и сетях, а также при программировании многих задач.

На предмете «Математическая логика и теория алгоритмов» студентами рассматриваются некоторые из таких языков – операторные схемы алгоритмов, которые могут быть в некоторых случаях успешно применены для формального описания функционирования управляющих устройств.

Описание алгоритмов в терминах операторных схем является лишь первым этапом формализации. Практические задачи реализации этих алгоритмов в виде микропрограмм и программ ЭВМ всегда требуют оптимизации с целью сокращения оборудования и затрат рабочего времени. Отсюда вытекает необходимость на следующем этапе формализации производить разного рода равносильные преобразования первоначального вида алгоритмов для получения максимально экономного по числу элементарных выражений описания. Практика показывает, что во многих случаях при использовании операторных схем алгоритмов удается избежать полного перебора и получить минимальную по числу элементарных выражений логическую схему алгоритмов. [1]

К сожалению, на данный момент времени из-за отсутствия программной реализации лабораторной установки, построение схем студентами и проверка схем преподавателями происходит вручную, что требует большого количества затрат, поэтому было принято решение решить данную проблему в рамках данной выпускной квалификационной работы.

[1] Матвеев В.Д. Теория алгоритмов [Текст] / Матвеев В.Д., Матвеева Л.И – Киров: Изд-во ВятГУ, 2011 год. – 40 с.

1. Анализ состояния проблемы

В данном разделе производится обзор научной литературы о предметной области, анализ актуальности проблемы, обзор существующих аналогов, а также выбор направления разработки программы моделирования в рамках дипломного проекта.

1.1. Обзор аналогов и научной литературы

Актуальность проблемы заключается в том, что аналога программы, содержащей в себе средства для построения и проверки операторных схем алгоритмов, отсутствует, поэтому было принято решение ее реализовать.

В качестве литературы по теме было взято учебное пособие Матвеева В.Д. и Матвеевой Л.И «Теория алгоритмов» вторая часть. Данное учебное пособие имеет всю необходимую теоретическую информацию об операторных схемах алгоритмов, которая сопровождается примерами.

1.2. Направление разработки программы

Для определения направления разработки программы воспользуемся методом, основанным на множестве опорных точек зрения.[2] На рисунке 1 изображена диаграмма идентификаций точек зрения, на которой отображены три опорные точки: ГСА, МСА, ЛСА. Так же на диаграмме присутствуют сервисы по построению (добавление вершин, соединение вершин линиями, горячие клавиши), редактированию (перемещение вершин, выравнивание

[2] Долженкова М.Л. Технологии программирования [Текст] / Долженкова М.Л., Караваева О.В. – Киров: Изд-во ВятГУ, 2013 год – 127 с.

вершин, удаление линий, редактирование линий) и проверке (динамическая проверка схем, информация об ошибках с пояснением и по возможности, выделение ошибки).

Рисунок 1 – Диаграмма идентификаций точек зрения

Вывод

Выбранная тема является актуальной, поскольку отсутствует аналог программы, которая предоставляет средства для построения и проверки операторных схем алгоритмов, при этом присутствует лабораторная работа по данной теме, которая выполняется и проверяется вручную.

Было выбрано учебное пособие, по теоретической части которого будет реализовываться программа, а так же было выбрано направление развития программы с помощь метода опорных точек зрения. Данный метод помог сформировать сервисы, которые будут учтены при формировании технического задания для разрабатываемой программы.

2. Постановка задачи

В данном разделе будет сформулировано техническое задание на проектирование программы для построения и проверки операторных схем алгоритмов. Для формирования требований к функциональным характеристикам программы был использован подраздел 1.2.

2.1. Основание для разработки

Программа разрабатывается на основе учебного плана кафедры «Электронные вычислительные машины»

2.2. Назначение разработки

Основным назначением программы является облегчение и ускорение выполнения лабораторной работы, по предмету «Математическая логика и теория алгоритмов».

Данная программа полезна как для студентов (предоставляет средства для выполнения задания в одной программе), а так же для преподавателей (предоставляет средства проверки корректности выполнения).

2.3. Требования к программе

Требования к функциональным характеристикам:

- генерация задания в виде словесного описания алгоритма;

- предоставление средств для построения ГСА с возможностью выравнивания вершин;
- предоставление средств для построения МСА с возможностью использования горячих клавиш;
- предоставление средств для построения ЛСА с возможностью использования горячих клавиш;
- динамическая проверка ГСА с выделением ошибок;
- динамическая проверка МСА с пояснением ошибок;
- динамическая проверка ЛСА с пояснением ошибок.

Требования к минимальной конфигурации:

- минимальное разрешение экрана 1280x720;
- наличие компьютерной мыши;
- наличие клавиатуры.

Требования к программной совместимости:

- операционная система: Windows 7,8,10.

2.4. Требования к документации

Документация должна содержать:

- тексты программ со всеми необходимыми комментариями;
- пояснительную записку, содержащей описание разработки;
- руководство пользователя.

Вывод

В данном разделе были рассмотрены требования к функциональным характеристикам, которые включают в себя предоставление средств для построения и проверки схем с обоснованием ошибок, а так же требования к параметрам техническим средств, без соблюдения которых могут возникнуть проблемы при использовании программы.

3. Разработка модульной структуры программы

В данном разделе производится проектирование модульной структуры программы, которая будет реализованы в виде отдельных библиотек, чтобы при необходимости вносить изменения только в одном модуле, без повторной сборки проекта и остальных модулей.

На рисунке 2 представлена модульная структура программы, которая состоит из следующих модулей:

- главная программа (SchemeConstructor.exe);
- операторный алгоритм в виде класса (OperatorAlgorithm.dll);
- генератор задания (AlgorithmGenerator.dll);
- модуля классов объектов ГСА (ClassesGraph.dll);
- интерфейса взаимодействия с объектами ГСА (InterfaceGraph.dll).
- модуль проверки ГСА, МСА, ЛСА (CheckingTheScheme.dll);
- модуль ведения статистики (StatisticsTable.dll);

Связующим звеном является главная программа, которая с помощью модуля генератора задания получает задание в виде операторного алгоритма. Класс операторного алгоритма хранится в отдельном модуле для согласованного представления алгоритма в модулях генерации алгоритма и проверки схем алгоритма.

С помощью интерфейса главной программы происходит построение ГСА, МСА и ЛСА. Особенностью построения ГСА заключатся в том, что добавление объектов происходит через интерфейс главной программы, но взаимодействие с объектами графа происходит через специальный интерфейс. Классы объектов графа хранятся в отдельном модуле для согласованного представления

объектов в модулях интерфейса взаимодействия с объектами и с модулем проверки схем алгоритма.

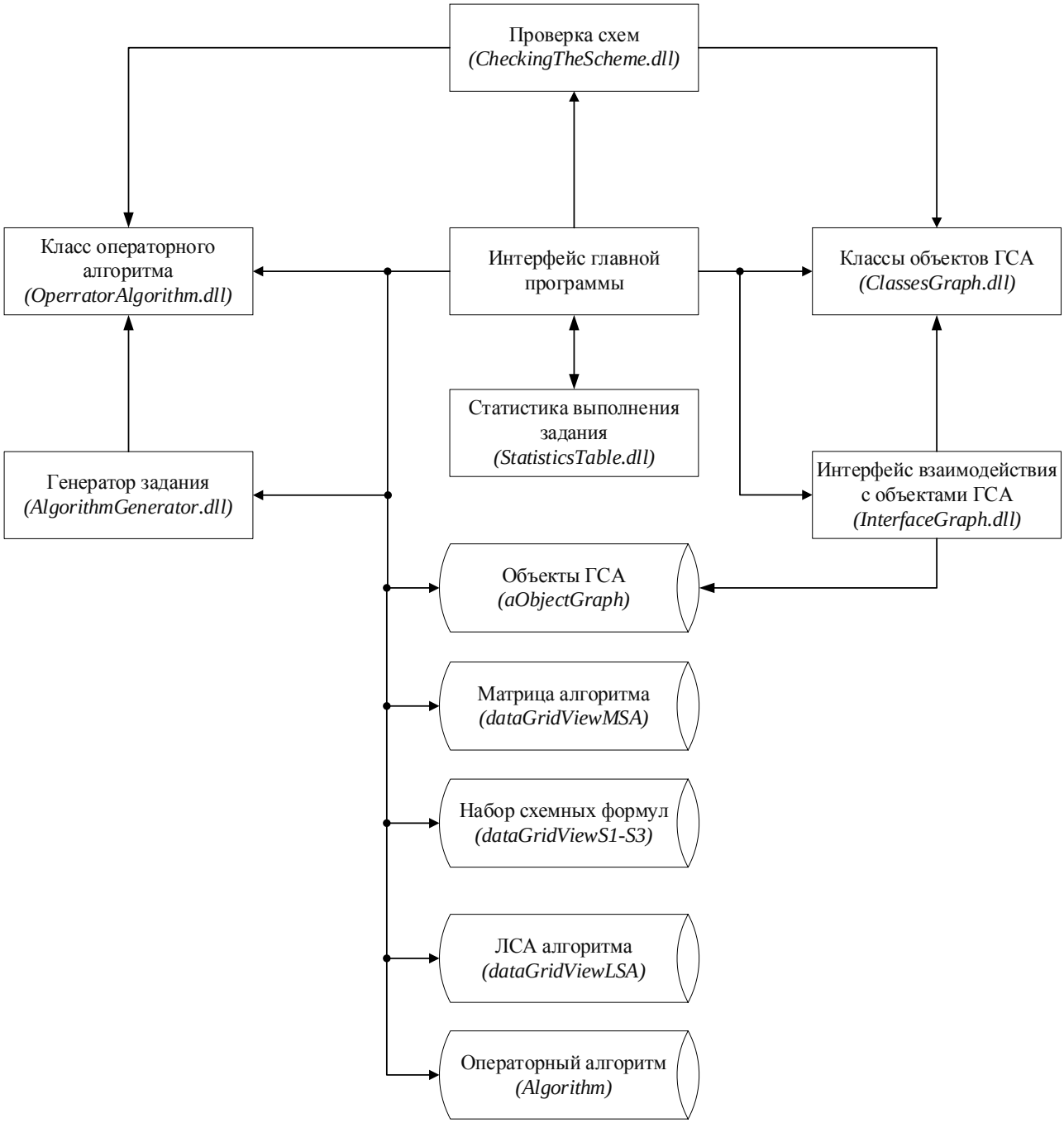


Рисунок 2 – Модульная структура программы

С помощью модуля проверки схем, происходит проверка ГСА, на основе соединительных линий, хранящихся в списке объектов графа, а так же

сгенерированного операторного алгоритма. Проверка МСА происходит на основе построенной матрицы и сгенерированного операторного алгоритма. Проверка ЛСА так же происходит на основе построенных формул и сгенерированного алгоритма.

С помощью модуля ведения статистики выполнения задания фиксируются результаты выполнения задания и посылаются в главную программу.

Вывод

В данном разделе была разработана структура программы, которая включает в себя ряд библиотек и основную программу. В ходе разработки структуры было выяснено то, что модули должны иметь единое представление операторной схемы алгоритма. То есть сгенерированный алгоритм одним модулей был понятен другим модулям. Для этого необходимо будет реализовать отдельный класс, вынесенный его в отдельную библиотеку.

Так же классы объектов графа нужно вынести в отдельный модуль для согласованного представления объектов в модулях интерфейса взаимодействия с объектами и с модулем проверки схем алгоритма.

4. Генерация задания

Модули программы должны иметь единое представление операторного алгоритма, поэтому помимо описания процесса генерации задания, в данном разделе описан еще и класс представления операторного алгоритма, который хранит в себе набор вершин алгоритма, которые представляют собой набор объектов определенного класса.

4.1. Вершины алгоритма

Операторный алгоритм состоит из двух типов вершин – операторной вершины и логического условия. Из операторной вершины можно провести переход в одну другую вершину, а из логического условия в две других вершины, рисунок 3.[3]

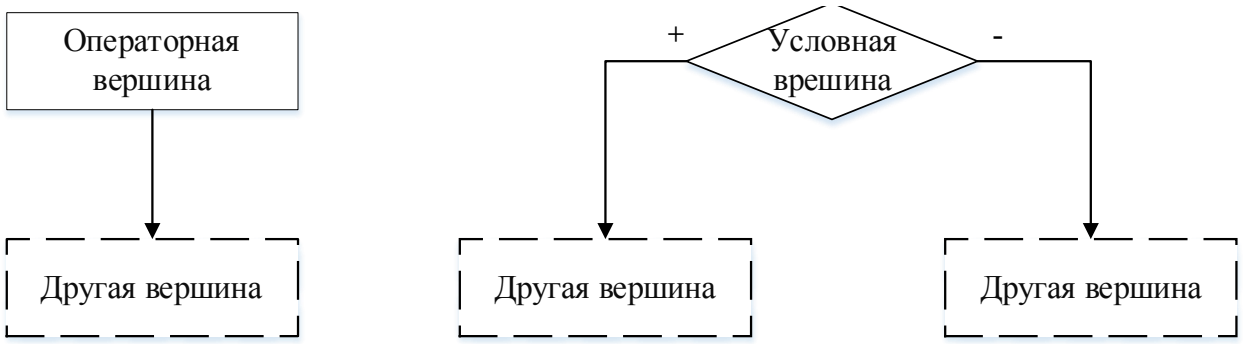


Рисунок 3 – Типы вершин операторного алгоритма

Так же у операторной вершины может вовсе не быть выхода, тогда данная вершина может считаться конечной вершиной. Так же в операторную

[3] Матвеев В.Д. Теория алгоритмов [Текст] / Матвеев В.Д., Матвеева Л.И – Киров: Изд-во ВятГУ, 2011 год. – 40 с.

вершину может не вести перехода, тогда данная вершина будет считаться начальной вершиной. На рисунке 4 представлены типы операторных вершин.

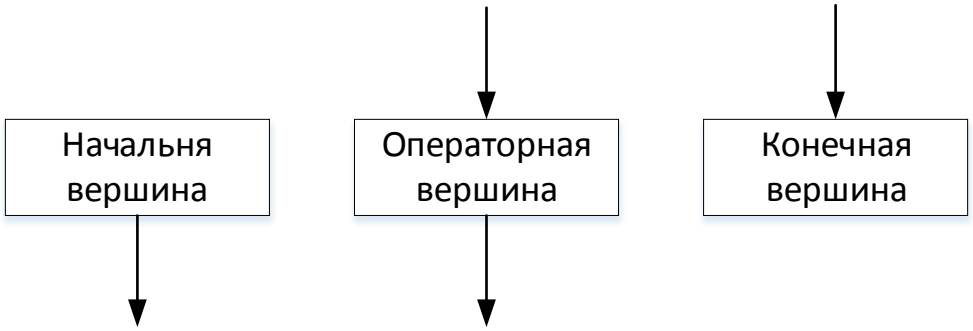


Рисунок 4 – Типы операторных вершин

Требования к вершинам следующие:

- из операторной вершины обязательно должен быть выход, кроме конечной вершины.
- из условной вершины обязательно должно быть два выхода;
- в начальную вершину не должно быть перехода.

4.2. Операторный алгоритм

В совокупности операторные и условные вершины составляют операторный алгоритм, к которому предъявляются определенные требования:

- имена всех вершин должны быть уникальными, из чего исключается неопределенность перехода.
- алгоритм должен иметь единственный начальный оператор;
- алгоритм должен иметь единственный конечный оператор;
- каждая вершина алгоритма должна быть достижимой от начального оператора;

- отсутствие циклов логических условий, то есть циклов, в которых присутствуют только условные вершины.

На рисунке 5 изображен пример условного цикла, который образуют условные вершины 3, 1 и 2.

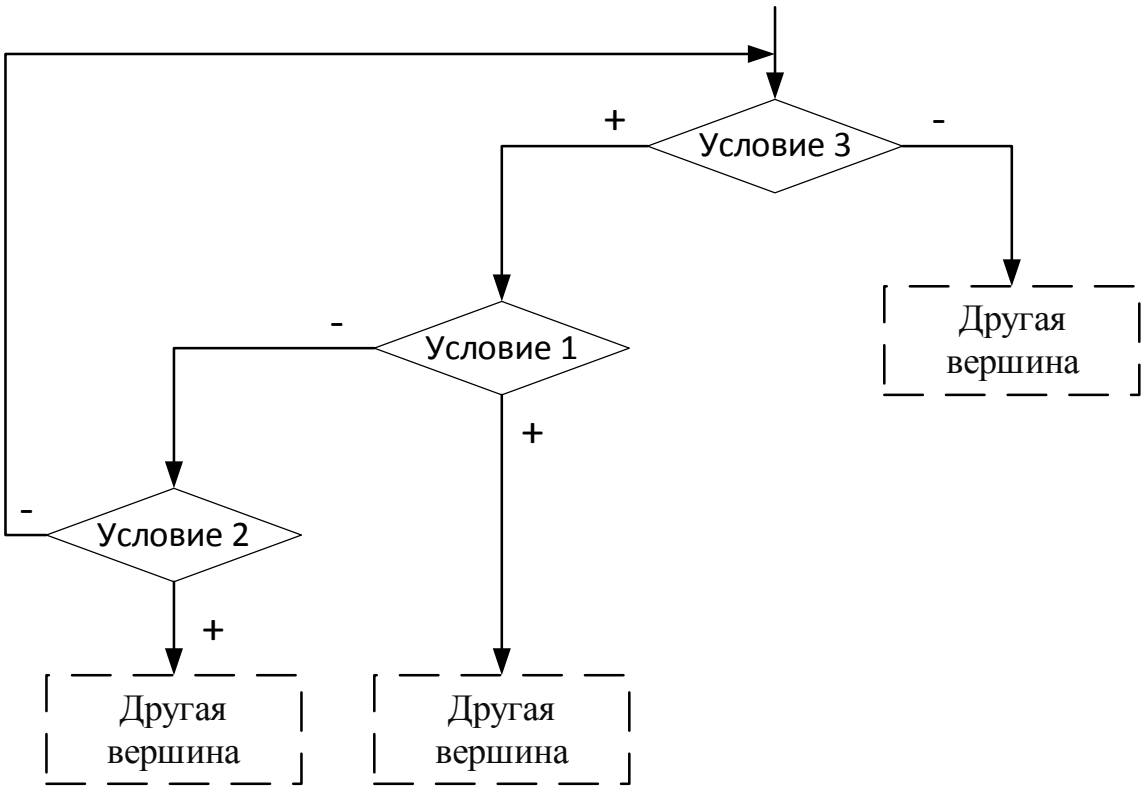


Рисунок 5 – Пример условных циклов

4.3. Генерация операторного алгоритма

Генерация алгоритма строится на трех списках:

1. список еще не присоединенных вершин к алгоритму;
2. список присоединенных вершин к алгоритму;
3. список присоединенных вершин к алгоритму, которые имеют свободные выходы.

Изначально алгоритм состоит из одной вершины – начальной вершины, поэтому данная вершина находится во втором списке, а все остальные вершины в первом списке. Так же третий список содержит один свободный выход – выход из начальной вершины.

Затем случайно берется вершина из третьего списка, из свободного выхода которой будет строиться переход.

Далее, в зависимости от того, сколько вершин находится в третьем списке, выбирается вершина, к которой будет осуществлен переход. Если список содержит одну текущую вершину, при этом список один еще не пуст, то выбирается случайная вершина (кроме конечной, если кроме нее в списке еще есть другие вершины) из списка еще не присоединенных вершин, иначе выбирается вершина либо из первого списка с вероятностью 40% (кроме начальной вершины), либо из второго списка.

При взятии вершины из списка еще не присоединенных вершин, данная вершина удаляется из этого списка и помещается в список присоединенных вершин к алгоритму и в список вершин, имеющих свободный выход. Если это условная вершина, то она дважды добавляется в этот список.

При построении пути из операторной вершины, она удаляется из третьего списка.

Концом генерации алгоритма будет ситуация, когда в первом списке не останется еще не присоединенных вершин к алгоритму и в третьем списке не останется свободных выходов из вершин.

На рисунках 7-8 изображена схема программы генерации.

Рассмотрим пример генерации алгоритма из 4 операторных вершин (A0,A1,A2,Ak) и 2 условных вершин (p0, p1).

Итерация 0:

Список 1: A1, A2, Ak, p0, p1.

Список 2: A0

Список 3: A0

- берем случайно вершину A0 из 3 списка;
- в списке 3 не остается вершин и список 1 не пуст, поэтому случайно выбираем из 1 списка вершину A2;
- добавление перехода из A0 в A2;
- добавление вершины A2 во список 2 и 3.

Итерация 1:

Список 1: A1, Ak, p0, p1

Список 2: A0, A2

Список 3: A2

- берем случайно вершину A2 из 3 списка;
- в списке 3 не остается вершин и список 1 не пуст, поэтому случайно выбираем из 1 списка вершину p1;
- добавление перехода из A2 в p1;
- добавление вершины p1 в 2 и 3 список.

Итерация 2:

Список 1: A1, Ak, p0,

Список 2: A0, A2, p1

Список 3: p1, p1

- берем случайно вершину p1 из 3 списка;
- в списке 3 остается вершина, поэтому случайно выбираем вершину из 1 или 2 списка. Выбрали вершину p0 из 1 списка.
- добавление перехода из p1 в p0;
- добавление вершины p0 в 2 и 3 список.

Итерация 3:

Список 1: A1, Ak,

Список 2: A0, A2, p1, p0

Список 3: p1, p0, p0

- берем случайно вершину p0 из 3 списка;
- в списке 3 остается вершина, поэтому случайно выбираем вершину из 1 и 2 списка. Выбрали вершину A2;
- добавление перехода из p0 в A2;
- так как вершину A2 выбрали из 2 списка, то добавление во 2 и в 3 список не происходит.

Итерация 4:

Список 1: A1, Ak,

Список 2: A0, A2, p1, p0

Список 3: p1, p0

- берем случайно вершину p1 из 3 списка;
- в списке 3 остается вершина, поэтому случайно выбираем вершину из 1 и 2 списка. Выбрали вершину Ak;
- добавление перехода из p1 в Ak;
- так как Ak – конечная вершина, то она добавляется только во 2 список.

Итерация 5:

Список 1: A1

Список 2: A0, A2, p1, p0, Ak

Список 3: p0

- берем случайно вершину p0 из 3 списка;

- в списке 3 не остается вершин и список 1 не пуст, поэтому случайно выбираем из 1 списка вершину A1;
- добавление перехода из p0 в A1;
- добавление вершины A1 в список 2 и 3.

Итерация 6:

Список 1:

Список 2: A0, A2, p1, p0, Ak

Список 3: A1

- берем случайно вершину A1 из 3 списка;
- в списке 3 не остается вершин и список 1 пуст, поэтому случайно выбираем из 2 списка вершину A2;
- добавление перехода из A1 в A2;
- так как вершину A2 выбрали из 2 списка, то добавление во 2 и в 3 список не происходит.

Так как список 1 и 3 пусты, то генерация закончена. После чего случайно выбираются условия переходов из условных вершин. На рисунке 6 изображен сгенерированный алгоритм.

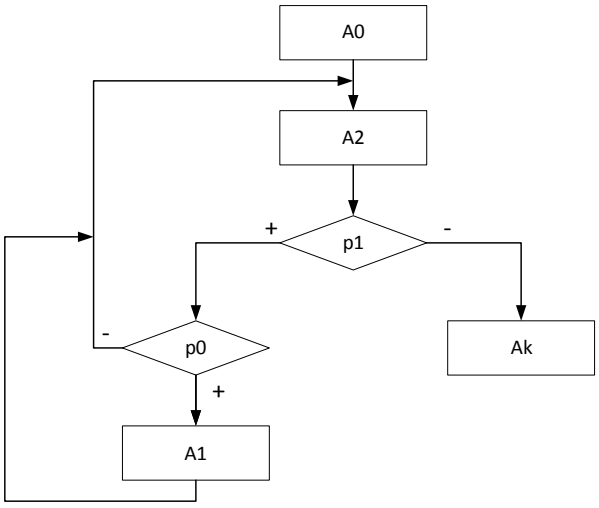


Рисунок 6 – Сгенерированный алгоритм

Рисунок 7 – Алгоритм генерации часть 1

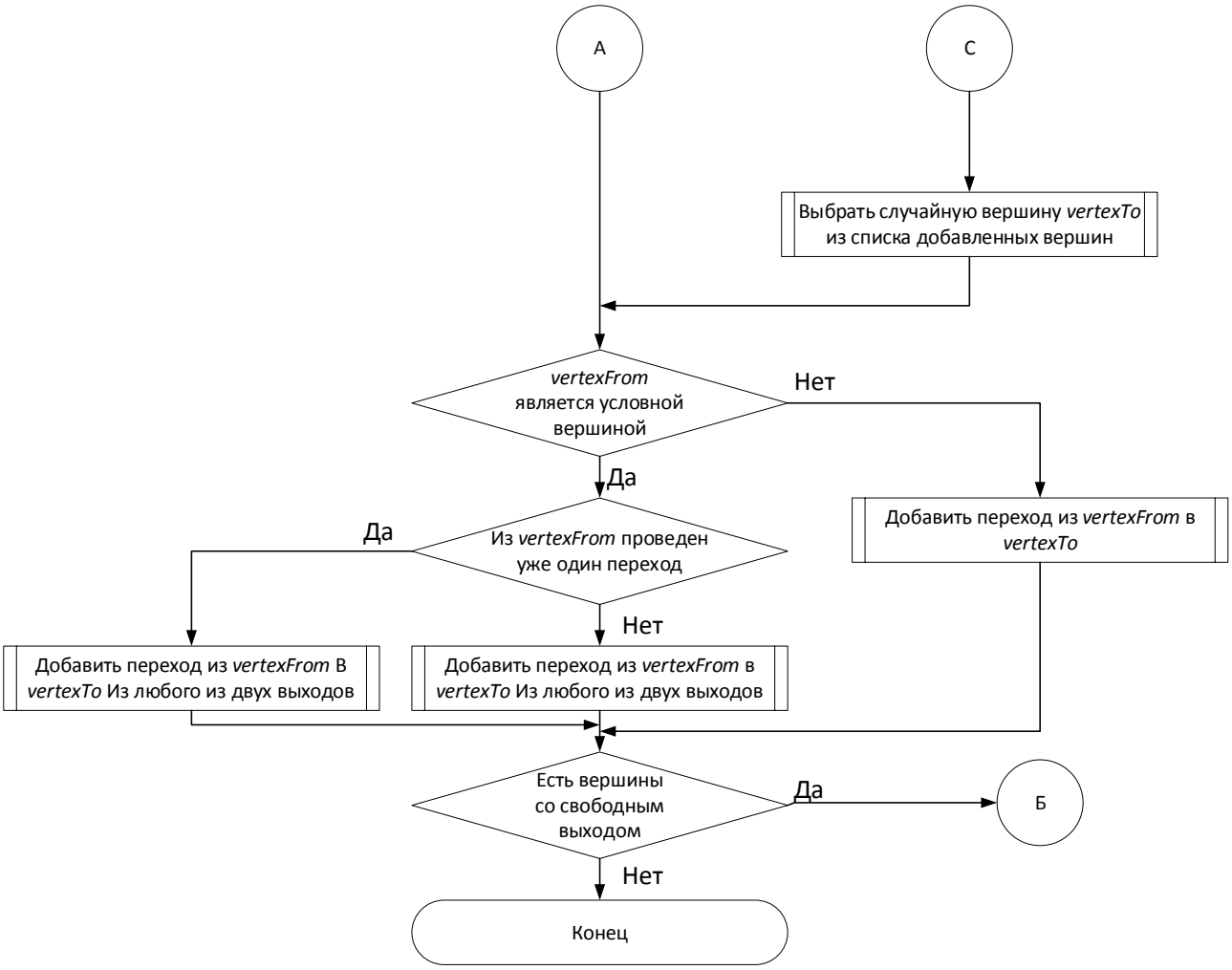


Рисунок 8 – Алгоритм генерации часть 2

Так как при генерации алгоритма может сформироваться цикл условных вершин, то необходимо произвести проверку наличия данных циклов и при необходимости устранить.

Для быстрого поиска и устранения циклов используется понятие безопасной вершины. Безопасная вершина - это та вершина, которая не ведет к условному циклу. Изначально операторные вершины считаются безопасными, так как они разрывают циклы условных вершин, а все условные вершины считаются не безопасными и требуют проверки. Это понятие ускоряет поиск из-за того, что, если при очередной проверке перехода из условной вершины

встречена условная вершина, которая до этого была проверена, она вновь не проверяется.

Изначально брались условные вершины по индексу и проверялись переходы из данных вершин. Данный подход оказался ошибочным, так как он не правильно определяет конец цикла и не правильно устранялся. На рисунке 9 изображен пример, когда проверка началась с условной вершины с индексом 1, из-за чего исправлялся переход из условной вершины с индексом 2 в условную вершину с индексом 1, тем самым оставляя условную вершину 1 недостижимой.

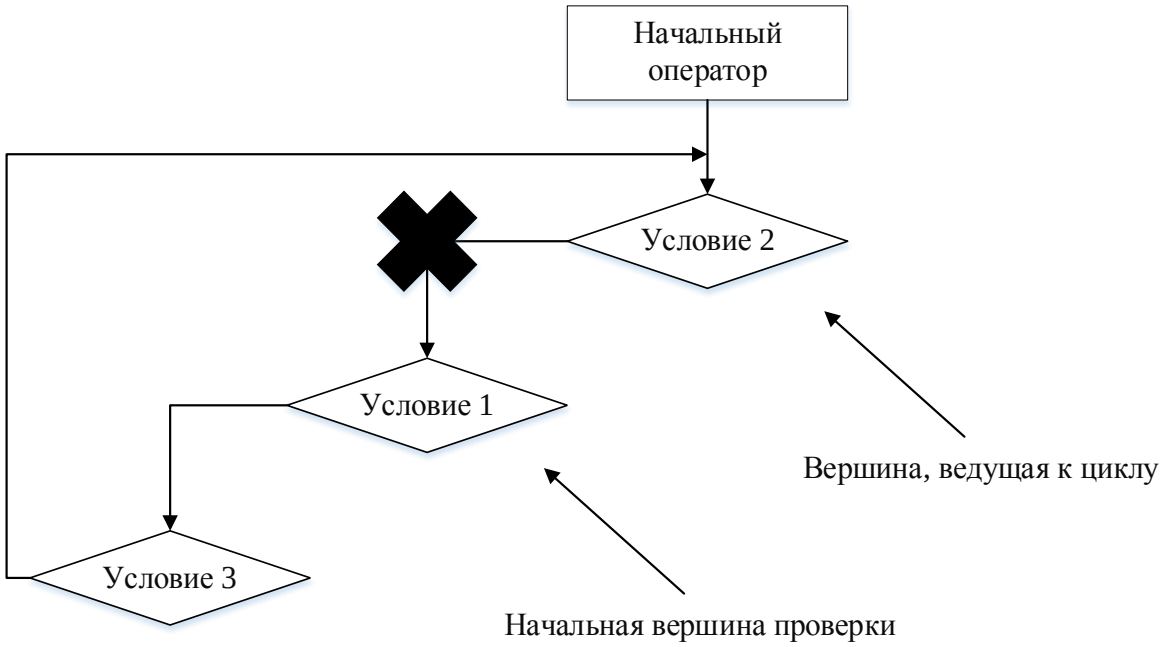


Рисунок 9 – Неправильное определение конца цикла

Поэтому проверку необходимо начинать только с тех условных вершин, на которые указывают операторные вершины, рисунок 10.

Проверка осуществляется рекурсивно, то есть, изначально берется условная вершина, в которую происходит переход из операторной вершины. Далее проверяется переход при истине. Если переход указывает на безопасную вершину, то проверяется переход при лжи. Если же переход указывает на не

безопасную вершину, то происходит переход к данной вершине, чтобы проверить, указывает ли она на безопасные вершины.

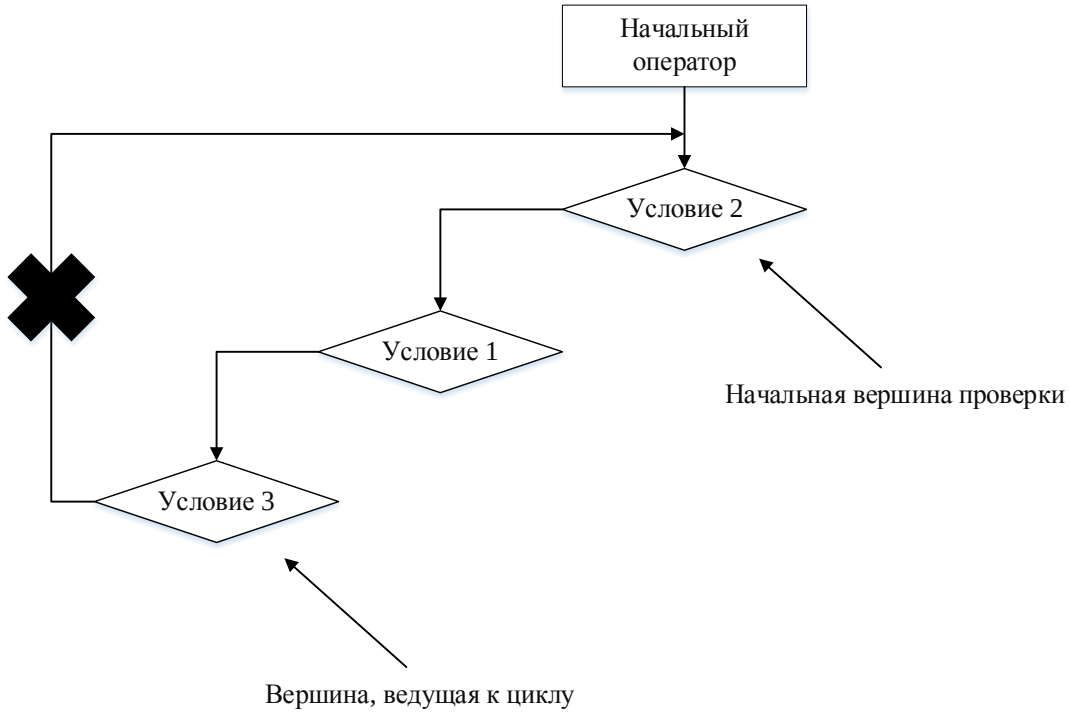


Рисунок 10 – Правильное определение конца цикла

В конечном счете, идя рекурсивно по небезопасным вершинам, мы найдем безопасную вершину, либо цикл. Цикл определяется на основе пройденного пути, то есть, если вершина указывает на условную вершину, в которой были до этого, то найден цикл, который устраняется путем перенаправления перехода на операторную вершину. Если же дошли до безопасной вершины, то отмечаем ее как безопасную и возвращаемся к вершине, в которой были до этого, чтобы проверить переход при лжи.

Рассмотрим пример поиска условных циклов. Допустим есть алгоритм, изображенный на рисунке 11.

Изначально вершины A0, A1, A2, A3, A4, A5 считаются безопасными, а вершины p0, p1, p2, p3, p4, p5, p6 нет.

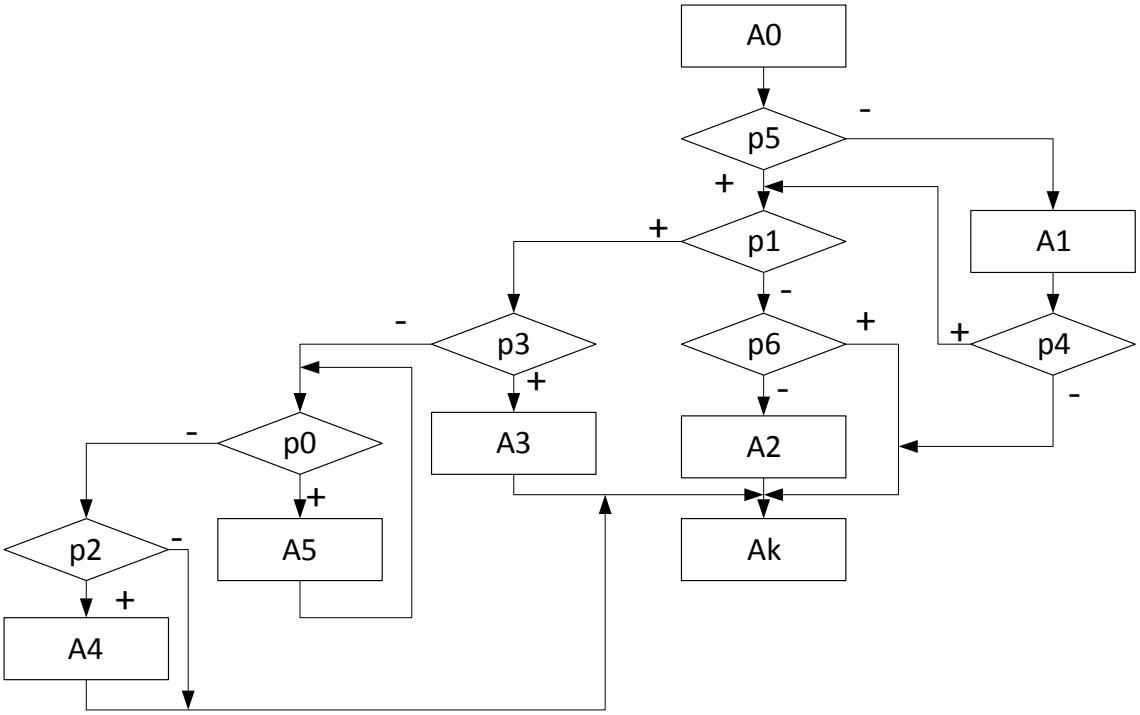


Рисунок 11 – Пример алгоритма для поиска условных циклов

Начинаем с вершины A0, которая указывает на вершину p5, которая является не безопасной. Поэтому проверяем вершину p5.

P5 указывает на P1 (О) и A1 (Б).

P1 указывает на P3 (О) и P6 (О).

P3 указывает на A3 (Б) и P0 (О).

P0 указывает на A4 (Б) и P2 (О).

P2 указывает на A5 (Б) и Ak (О) => P2 (Б). Возвращаемся.

P0 указывает на A4 (Б) и P2 (Б) => P0 (Б). Возвращаемся.

P3 указывает на A3 (Б) и P0 (Б) => P3 (Б). Возвращаемся.

P1 указывает на P3 (Б) и P6 (О).

P6 указывает на Ak (Б) и A2 (Б) => P6 (Б). Возвращаемся.

P1 указывает на P3 (Б) и P6 (Б) => P1 (Б). Возвращаемся.

P5 указывает на P1 (Б) и A1 (Б) = > P5 (Б). Возвращаемся.

Вернулись к A0, поэтому путь из A0 безопасный и вершины P2, P0, P3, P6, P1, P5 являются безопасными.

Далее смотрим вершину A1, которая ведет в вершину P4. Которая является не безопасной. Поэтому проверяем вершину P4.

P4 указывает на P1 (Б) и Ak (Б) => P4 (Б). Возвращаемся.

Вернулись к A1, поэтому путь из A1 безопасный и вершина P4 являются безопасной.

Вершины A2, A3, A4 указывают на безопасные операторные вершины, а вершина A5 указывает на P0, которая ранее проверялась и которая считается безопасной. В итоге все переходы из операторных вершин проверены и циклы не были обнаружены.

4.4. Программная реализация

Вершины алгоритма представляют собой объекты классов OperatorVertex и ConditionVertex. Конструктор классов принимает имя вершины, имя вершины назначения. Так же объект класса OperatorVertex имеет свойство, которое отвечает тип вершины: start, final и обычная вершина. Тип start изначально имеет значение false и устанавливается с помощью метода, а тип final устанавливается тогда, когда вершина vertexTo имеет значение null. Поля vertexTo_IfTrue и vertexTo_IfFalse класса ConditionVertex не могут иметь значение null, поэтому при попытке присвоения этого значения будет выдаваться исключение.

Подход вызова исключения вместо возврата значения null, позволяет очистить основную программу от различных проверок на null, а так же облегчает отладку программы, так как исключение будет вызвано сразу, а не через неопределенное время.

Класс Algorithm, вместе с классами вершин, находящийся в библиотеке OperatorAlgorithm.dll, при вызове конструктора которого происходит проверка

выполнения всех требований, описанных ранее. В случае если хотя бы одно требование будет нарушено, будет выдано исключение, чтобы в дальнейшем не нарушить работу других модулей, которые будут взаимодействовать с некорректным алгоритмом.

Так же с помощью публичных методов класса Algorithm происходит определение типа вершины по имени, получение количества вершин, получение начального и конечного оператора или других вершин по индексу или по имени. В случае если вершины с таким именем не будет найдено, то будет выдано исключение.

Класс генерации алгоритма Generate, находящийся в библиотеке AlgorithmGenerator.dll генерирует операторный алгоритм, в котором имена операторов и условных вершин состоят из идентификатора (captionOperator, captionCondition) и индекса (для начального и конечного оператора можно установить свои значения с помощью параметров indexStartOperator, indexFinalOperator). Все поля и методы класса Generate, кроме самого метода генерации, скрыты.

На рисунке 12 представлена диаграмма классов, участвующих в генерации задания.

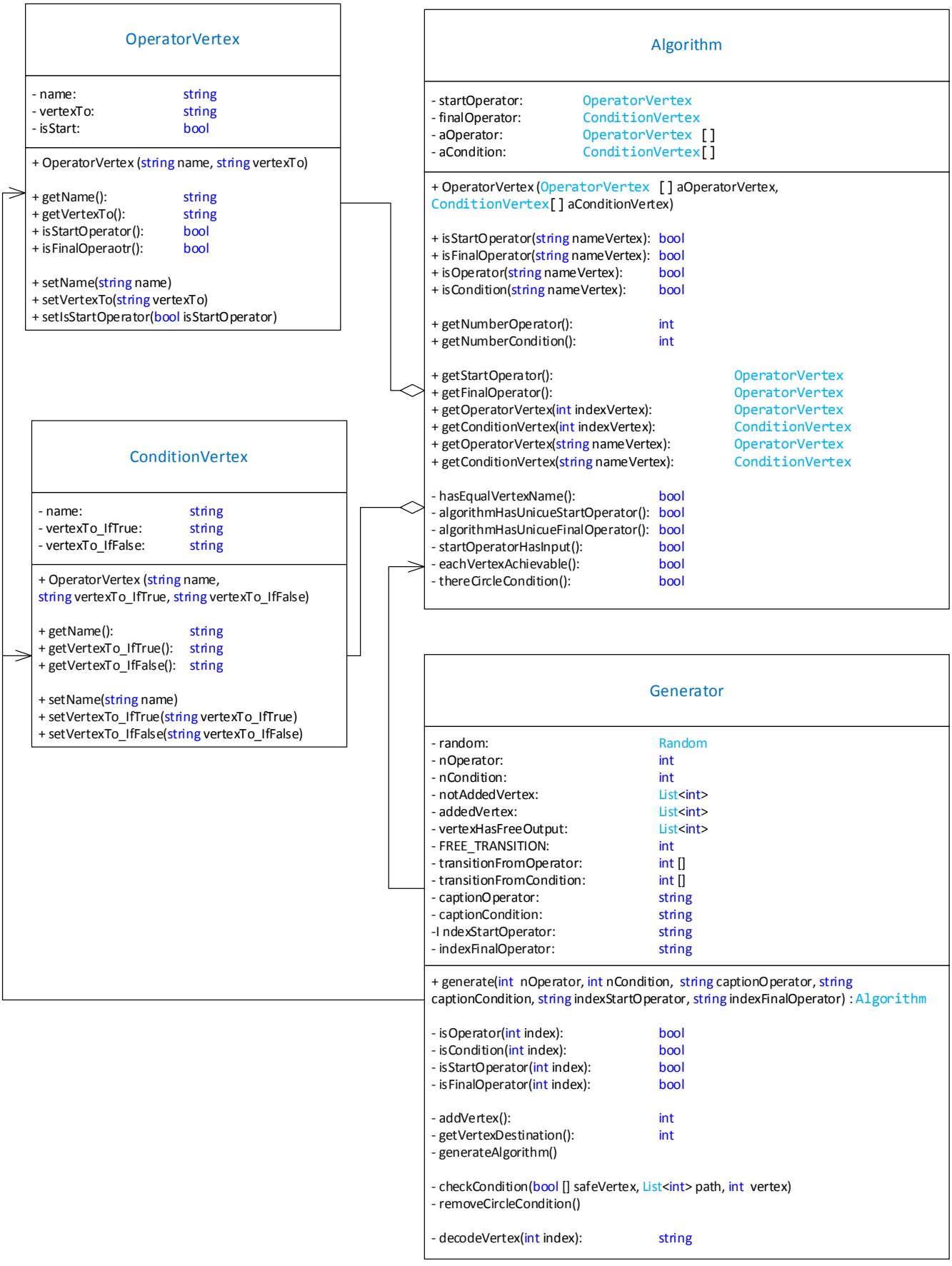


Рисунок 12 – Диаграмма классов, участвующих в генерации задания

Вывод

В данном разделе был разработан класс для генерации задания. Для реализации класса понадобилось вводить дополнительные классы: класс операторной и условной вершины и класс алгоритма, который представляет собой набор связанных операторный и условных вершин. Класс алгоритма позволяет использовать его без опасений на возникновение ошибки, так как при инициализации экземпляра класса происходит проверка соблюдения всех необходимых требований к алгоритму.

Был разработан алгоритм генерации задания с демонстрацией примера, который на основе трех списков создает переходы между вершинами. Так же после генерации происходит корректировка сгенерированного алгоритма для удовлетворения требований к алгоритму, а именно поиск и удаление условных вершин. Были раскрыты тонкости обнаружения правильного конца цикла, без соблюдения которых цикл не правильно устранялся, из-за чего появлялись вершины в алгоритме, которые были не достижимы.

5. Инструменты построения графической схемы алгоритма

Так как разрабатываемая программа предоставляет средства для построения схем, то необходимо разработать инструменты для построения графической схемы алгоритма, для чего необходимо разработать объекты графической схемы и интерфейс взаимодействия с данными объектами, о чем и пойдет речь в данном разделе.

5.1. Объекты графической схемы

Базовый класс

Базовым классом граф-схемы является объект, который имеет контур и заливку. Оба эти параметра имеют свойства цвета и видимости, которая определяет, видима ли заливка и контур. Вдобавок контур имеет еще свойство толщины линии и тип линии, который может быть сплошным, пунктирным, пунктирным с точкой и другие.

Помимо видимых параметров, объект обладает идентификатором, чтобы отличать его от других объектов и тип объекта. Второй параметр нужен для того, чтобы определять в общем списке объектов подкласс и дополнительные способы взаимодействия с ним.

Так же объект имеет область, в которую полностью входит объект. Данная область имеет координату верхней левой точки, которая по X имеет значение X самой левой точки контура и по Y имеет значение самой верхней токи контура, а так же данная область имеет ширину и высоту, которая является разностью между левой верхней точкой и правой нижней точкой.

На рисунке 13 изображен пример базового объекта.

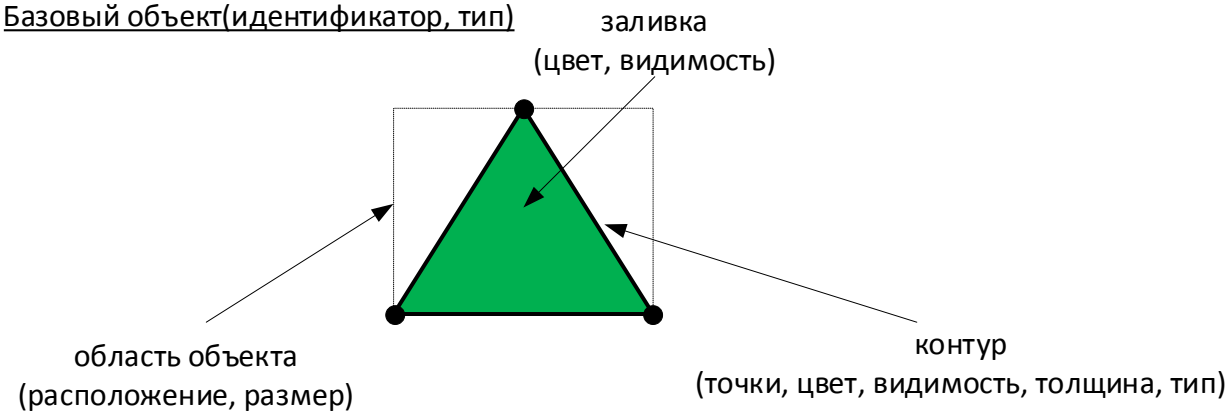


Рисунок 13 – Пример базового объекта

Класс текст

Данный класс является дочерним классом базового класса, и который наследует все свойства родительского класса и приобретает новые свойства. К объекту данного класса добавляются такие параметры как текст, шрифт текста, размер текста и цвет текста.

Область объекта определяется на основе самого текста, шрифта и размера текста, и каждый раз изменяется при изменении одного из параметров. Контур объекта отображает данную область.

К объекту класса текст можно прикрепить другой объект, например линию, поэтому объекту добавляется еще один параметр в виде прикрепленного объекта. На рисунке 14 изображен пример объекта класса.

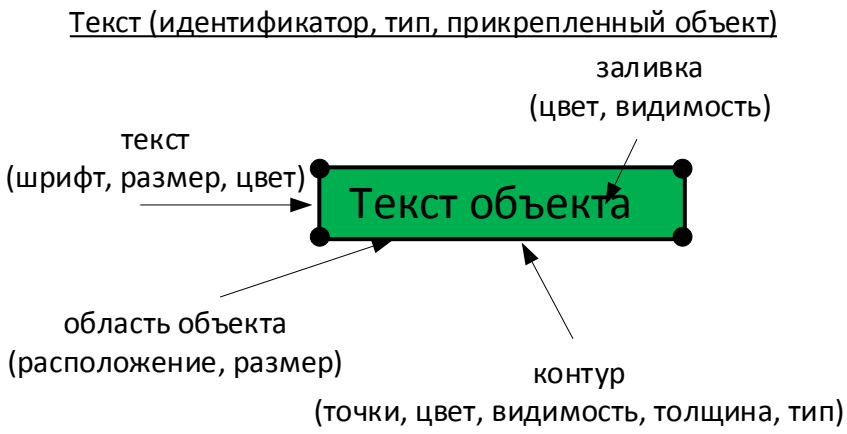


Рисунок 14 – Пример объекта класса текст

Класс места для вывода и присоединения линии

Данный класс предназначен для вывода линии из объекта или присоединения линии к объекту. Объектом данного класса является прямоугольник заданного размера и заданного расположения. Расположение задается координатами точки, откуда будет начинаться линия, а сам контур располагается так, чтобы данная точка была в центре.

Класс имеет ссылку на объект, к которому данное место привязано, а так же ссылку на линию, которая связана с данным местом. Линия может быть выходящей или входящей, поэтому присутствует соответствующий флаг. Так же класс имеет флаг блокировки места, который говорит о том, что место заблокировано и вывести линию из него нельзя.

На рисунке 15 изображен пример места для вываод и присоединения линии.

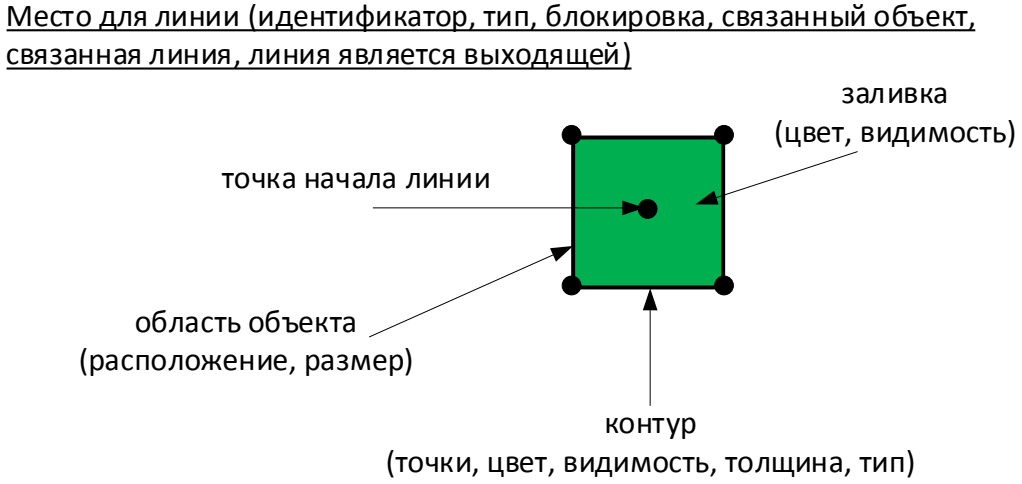


Рисунок 15 – Пример места для вывода и присоединения линии

Класс вершина

Данный класс объединяет два предыдущих класса в один объект. То есть объект имеет текст и места для вывода и соединения линий. Пример данного класса изображен на рисунке 16.

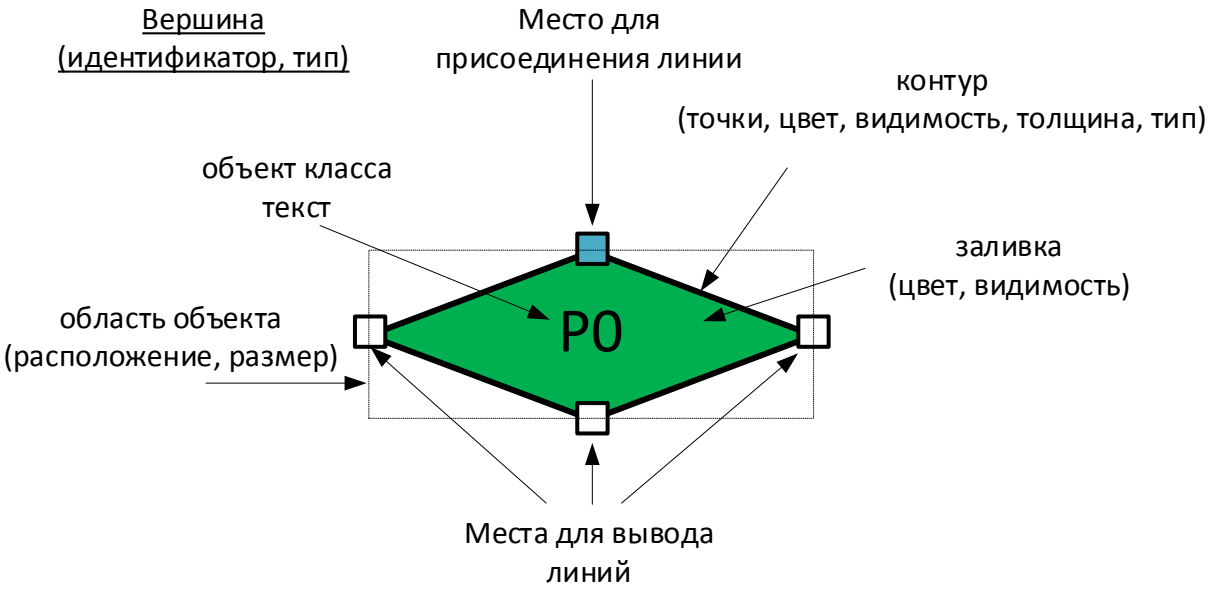


Рисунок 16 – Пример объекта класса вершина

Класс линии

Данный класс предназначен для связи объектов.

Линия имеет ссылки на объект, с которого выводится линия и объект, на который указывает линия. При наличии второго объекта появляется стрелка, которая имеет такие параметры как ширина, длина и отступ от конца линии. Обычно этими объектами являются места для линий, но также линия может указывать на другую линию. Так же линия может иметь текст. Пример данного класса изображен на рисунке 17.

Область линии определяется по-другому, чем у других объектов. Для этого сначала проверяется, принадлежит ли точка прямоугольнику, который образуют две точки лини. Затем определяется уравнение прямой, по которому будет найдено расстояние от точки до прямой, после чего будет сделан вывод, находится ли указанная точка в области объекта.

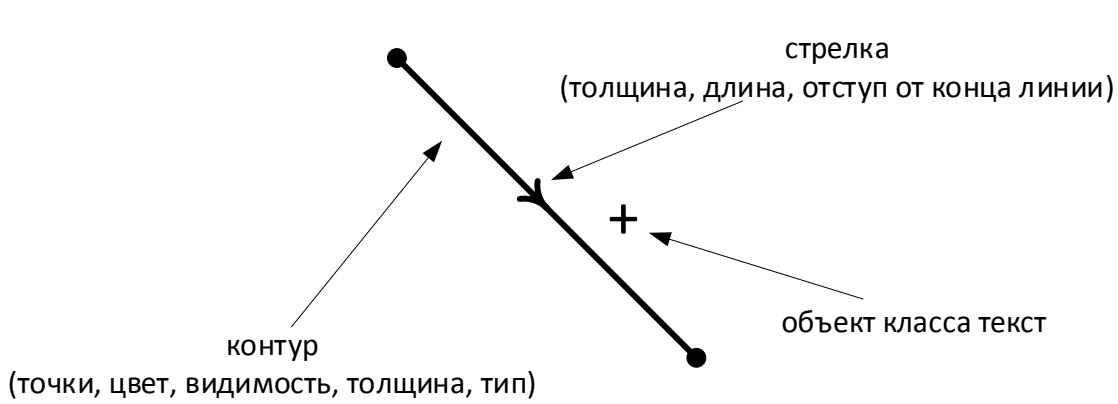


Рисунок 17 – Пример объекта класса линия

Класс прямоугольная линия

Данный класс является дочерним классом линии и состоит из списка сегментов линий класса линии. Помимо свойств, которые имеет класс линии, ломаная линия имеет еще место для продолжения линии. На рисунке 18 приведен пример ломаной линии.

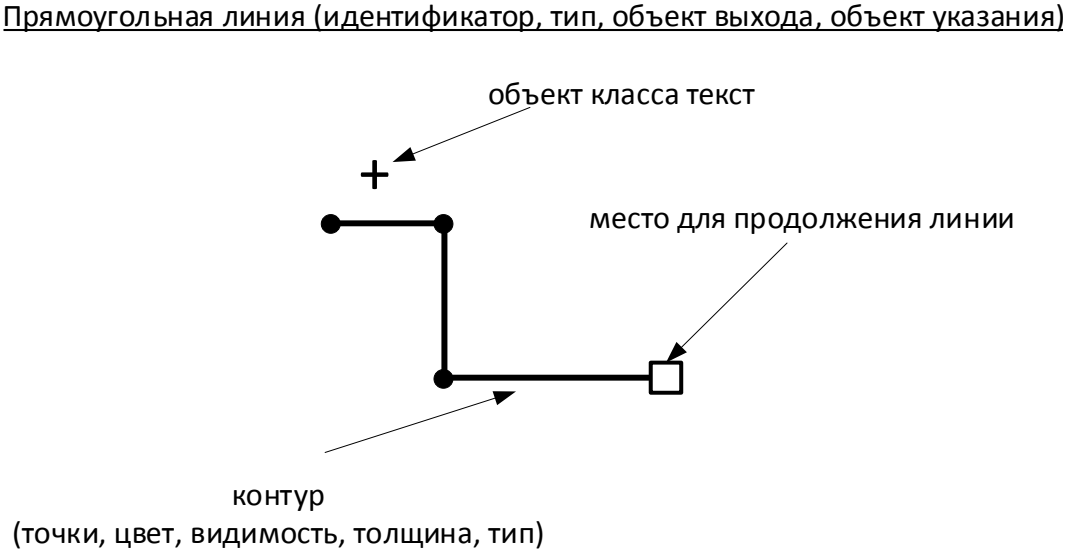


Рисунок 18 – Пример объекта класса прямоугольная линия

5.2. Программная реализация классов объектов

Класс ObjectGraph.

Данный класс реализует базовый класс, описанного в прошлом пункте.

Помимо методов для чтения и редактирования свойств объекта, присутствуют дополнительные методы, которые могут быть переопределены дочерними классами.

Метод отображения объекта drawObject, который отображает объект на холсте bitmap, передаваемого по ссылке. Функции для работы с графикой были взяты из пространства имен System.Drawing.[4] Метод не имеет понятия замкнутого контура, поэтому если контур должен быть замкнутым, то координаты последней точки должны совпадать с координатами первой точки. Метод отображен на рисунке 19.

```
public virtual void drawObject(Bitmap bmp)
{
    Graphics graph = Graphics.FromImage(bmp);

    //Отображение заливки.
    if (visibleBrush) {
        graph.FillPolygon(new SolidBrush(colorBrush), aPointContour);
    }

    //Отображение контура.
    if (visibleContour) {
        Pen pen = new Pen(colorContour, widthPenContour);
        switch (typeLineContour) {
            case 0: pen.DashStyle = System.Drawing.Drawing2D.DashStyle.Solid;       break;
            case 1: pen.DashStyle = System.Drawing.Drawing2D.DashStyle.Dash;        break;
            case 2: pen.DashStyle = System.Drawing.Drawing2D.DashStyle.Dot;         break;
            case 3: pen.DashStyle = System.Drawing.Drawing2D.DashStyle.DashDot;     break;
            case 4: pen.DashStyle = System.Drawing.Drawing2D.DashStyle.DashDotDot;  break;
            case 5: pen.DashStyle = System.Drawing.Drawing2D.DashStyle.Custom;      break;
            default: pen.DashStyle = System.Drawing.Drawing2D.DashStyle.Solid;      break;
        }
        graph.DrawLines(pen, aPointContour);
    }
 }
```

Рисунок 19 – Метод отображения объекта

[4] System.Drawing Namespace [Электронный ресурс]. URL: https://docs.microsoft.com

Метод перемещения объекта moveObject, который сдвигает точки объекта и область объекта на заданный вектор vectorMove, рисунок 20.

Метод обнаружения объекта cursorOnArea, который определяет вхождения курсора в область, в которую входит объект, рисунок 21

```
public virtual void moveObject(Point vectorMove)
{
    /*Сдвиг точек объекта*/
    for (int i = 0; i < aPointContour.Length; i++) {
        aPointContour[i].X = aPointContour[i].X + vectorMove.X;
        aPointContour[i].Y = aPointContour[i].Y + vectorMove.Y;
    }

    /*Сдвиг области*/
    pointArea.X = pointArea.X + vectorMove.X;
    pointArea.Y = pointArea.Y + vectorMove.Y;
}
```

Рисунок 20 – Метод перемещения объекта

```
public virtual bool cursorOnArea(Point cursor)
{
    if ((pointArea.X <= cursor.X) && (pointArea.X + sizeArea.Width >= cursor.X)) {
        if ((pointArea.Y <= cursor.Y) && (pointArea.Y + sizeArea.Height >= cursor.Y)) {
            return true;
        }
    }
    return false;
}
```

Рисунок 21 – Метод определения вхождения курсора в область объекта

Исключения, которые могут возникнуть при использовании данного класса: входной параметр конструктора aPointContour равен null.

Начальные значения параметров:

- видимость контура – включено;
- цвет контура – черный;
- толщина контура – 2 пикселя;
- тип линии контура – сплошная линия;
- видимость заливки – включено;
- цвет заливки – белый.

Класс TextGraph

Данный класс реализует класс текст, описанного в прошлом пункте.

Данный класс является дочерним классом ObjectGraph, и который наследует все методы родительского класса за исключением переопределенного метода отображения объекта, так как у объекта появляется текст, который нужно так же отображать.

Переопределяемый метод drawObject отображает сначала заливку и контур с помощью метода родительского класса и затем отображает текст с заданными характеристиками. При равенстве параметра *text* значению *null* возникает исключение, рисунок 22.

```
public override void drawObject(Bitmap bmp)
{
    Graphics graph = Graphics.FromImage(bmp);

    //Отрисовка фона с контуром.
    base.drawObject(bmp);

    //Отрисовка текста.
    graph.DrawString(text, new Font(fontText, sizeText), new SolidBrush(colorText), pointArea);
}
```

Рисунок 22 – Переопределенный метод отображения объекта

С помощью метода setArea определяются новые значения точек контура и области объекта при изменении текста, размера и шрифта, рисунок 23.

```
 Private void setArea()
{
    //Определение размера текста в пикселях.
    sizeArea = TextRenderer.MeasureText(text, new Font(fontText, sizeText));

    //Переопределение контура текста.
    aPointContour[1].X    = pointArea.X + sizeArea.Width;   //Верхняя правая точка.
    aPointContour[1].Y    = pointArea.Y;
    aPointContour[2].X    = pointArea.X + sizeArea.Width;   //Нижняя правая точка.
    aPointContour[2].Y    = pointArea.Y + sizeArea.Height;
    aPointContour[3].X    = pointArea.X;                      //Нижняя левая точка.
    aPointContour[3].Y    = pointArea.Y + sizeArea.Height;
 }
```

Рисунок 23 – Метод определения контура и области объекта

Так как конструктор родительского класса тоже наследуется, то для отправки параметра aPointContour используется статический метод getPointContourText, похожий на метод setArea.

Начальные значения параметров объекта:

- шрифт текста – Calibri;
- размер шрифта – 14;
- цвет текста – черный.

Класс PlaceForLine

Данный класс реализует класс места для вывода и присоединения линии.

Данный класс является дочерним классом ObjectGraph, и переопределяет метод перемещения, так как при перемещении данного объекта должна перемещаться первая или последняя точка линии, связанной с данным местом, так же переопределен метод отображения и определения вхождения курсора на объект, так как объект может быть заблокирован. То есть метод moveObject подразумевает, что у линии должен быть определен метод перемещения начальной и конечной точки, рисунок 24.

```
public override void moveObject(Point vectorMove)
{
    //Перемещение объекта.
    base.moveObject(vectorMove);

    //Перемещение точки для начала линии.
    pointForLine.X = pointForLine.X + vectorMove.X;
    pointForLine.Y = pointForLine.Y + vectorMove.Y;

    //Перемещение точки линии.
    if (line != null) {
        if (isOutputLine) {
            line.moveFirstPointLine(vectorMove);
        }
        else {
            line.moveLastPointLine(vectorMove);
        }
    }
}
```

Рисунок 24 – Переопределенный метод перемещения объекта

Класс VertexGraph

Данный класс реализует класс вершины, описанного в прошлом пункте.

Данный класс является дочерним классом ObjectGraph, и переопределяет методы отображения объекта, так как необходимо еще отображать текст, и метод перемещения объекта, так как помимо контура объекта перемещается еще текст и места для линий.

Текст присоединяется к объекту с помощью указания ссылки, которая может иметь значение null, при вызове конструктора, места для линий добавляются с помощью соответствующих методов. Места для вывода и присоединения линий добавляются и удаляются с помощью соответствующих методов.

Класс LineGraph

Данный класс реализует класс прямоугольной линии, описанного в прошлом пункте.

Так как линия является динамическим объектом в плане контура, то используется не массив точек, а список точек aPointLine.

Работа с объектом будет производиться на основе работы с сегментами линий, то есть метод определения вхождения курсора в область объекта будет проверять вхождение курсора на один из сегментов линии. На рисунке 25 изображен переопределенный метод определения вхождения курсора.

```
public override bool cursorOnArea(Point cursor)
{
    if (aPointLine.Count >= 2) {
        for (int i = 0; i < aPointLine.Count - 1; i++) {
            if (cursorOnSegmentLine(i, cursor) == true) {
                return true;
            }
        }
    }
    return false;
}
```

Рисунок 25 – Переопределенный метод определения вхождения курсора в область объекта

Если сегмент линии вертикальный, то курсор должен располагаться в пределах установленной дистанции в обе стороны от линии по X и в пределах координат двух точек сегмента линии по Y, если сегмент линии горизонтальный, то наоборот. Если линия наклонная, то производится проверка вхождения точки в прямоугольник, формируемый сегментом линии как диагональ, и если точка входит в прямоугольник, то высчитывается расстояние от точки до прямой, которое должно быть не больше установленной дистанции. Данное расстояние можно изменить с помощью метода setDistanceLine.

```
        public bool cursorOnSegmentLine(int indexSegment, Point cursor)
        {
            if (indexSegment < m_aPointLine.Count - 1) {
                Point point1 = m_aPointLine[indexSegment];
                Point point2 = m_aPointLine[indexSegment + 1];
                /*Если линия не вертикальная*/
                if (point1.X != point2.X) {
                    /*Если линия не горизонтальная*/
                    if (point2.Y != point2.Y) {
                        /*Если курсор лежит в интервале по X*/
                        if (Math.Sign(cursor.X - point1.X) != Math.Sign(cursor.X - point2.X)) {
                            /*Если курсор лежит в интервале по Y*/
                            if (Math.Sign(cursor.Y - point1.Y) != Math.Sign(cursor.Y - point2.Y)) {
                                int A = point2.Y - point1.Y;            //Ax + By + C = 0
                                int B = point1.X - point2.X;            //Уравнение прямой.
                                int C = point1.Y * (point2.X - point1.X) - point1.X * (point2.Y –
                                point1.Y);
                                double dist = Math.Abs(A * cursor.X + B * cursor.Y + C) /
                                              Math.Sqrt(A * A + B * B); //Расстоярие от точки до линии.
                                if (dist < m_distanceLine) {
                                    return true;
                                }
                            }
                        }
                    }
                    /*Если линия горизонтальная*/
                    else {
                        /*Если курсор в пределах линии по Y*/
                        if (Math.Abs(point1.Y - cursor.Y) < m_distanceLine) {
                            /*Если курсор лежит в интервале по X*/
                            if (Math.Sign(cursor.X - point1.X) != Math.Sign(cursor.X - point2.X)) {
                                return true;
                            }
                        }
                    }
                }
                /*Если линия вертикальная*/
                else {
                    /*Если курсор в пределах линии по X*/
                    if (Math.Abs(point1.X - cursor.X) < m_distanceLine) {
                        /*Если курсор лежит в интервале по Y*/
                        if (Math.Sign(cursor.Y - point1.Y) != Math.Sign(cursor.Y - point2.Y)) {
                            return true;
                        }
                    }
                }
            }
            return false;
        }
```

Рисунок 26 – Метод определения вхождения курсора на сегмент линии

Так же при добавлении, удалении и перемещении точек линий с помощью методов addToEndPoint, insertPoint, deleteLastPoint, deletePoint, moveLastPointLine, movePointLine происходит смещение места для продолжения линий, причем смещение происходит на место последней точки линии. Так же первую точку линии нельзя удалить, можно только переместить.

Рисунок 27 – Диаграмма классов объектов граф-схемы

5.3. Интерфейс взаимодействия с объектами

Управление объектами происходит через специальный интерфейс взаимодействия, который предоставляет функции обнаружения объекта, перемещения объекта, построение и редактировании линии.

Обнаружение объектов

Для обнаружения объектов используется класс DetectorObject. С помощью метода detect происходит проверка объекта под курсором. Если под курсором найден объект, то данный объект помещается в буфер, а на холсте отображается выделение данного объекта.

Реакция на попадание курсора на объект и ухода курсора с объекта определяется соответствующими Private методами. Если это вершина, то выделяется либо контур объекта, либо место для вывода линии, рисунок 28.

В случае, если обнаруженным объектом является линия, то происходит выделение либо место для вывода линии, либо тех сегментов линии, которые при клике будут удалены, рисунок 29.

При попадании курсора на другие объекты, происходит выделение области объекта пунктирными линиями, рисунок 30. В случае с попаданием курсора на текст линии, выделяется еще и линия, к которой принадлежит текст.

После вызова метода detect, нужно вызывать метод bitmapWasChange, чтобы определить, было ли изменение на холсте. Это сделано для того, чтобы не отображать при каждом сдвиге мыши не измененные холсты.

На рисунке 30 изображен UML блок класса DetectorObject.

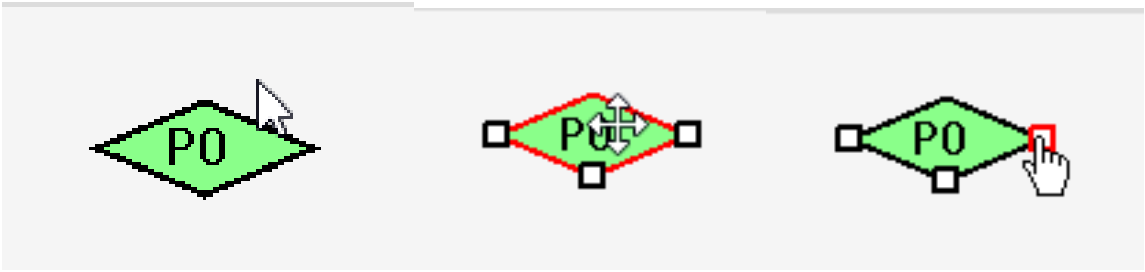


Рисунок 27 – Реакция VertexGraph на курсор

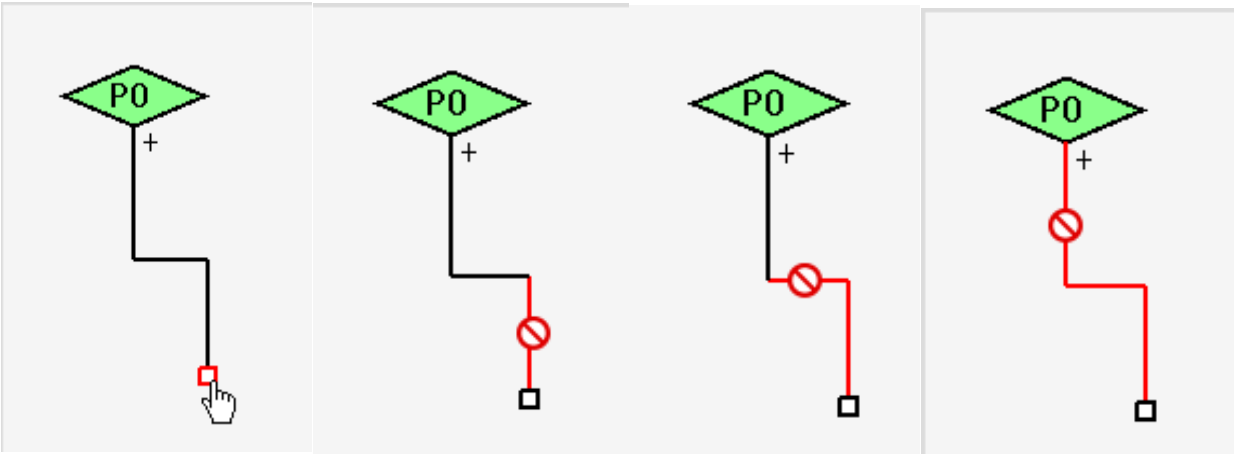


Рисунок 28 – Реакция LineGraph на курсор

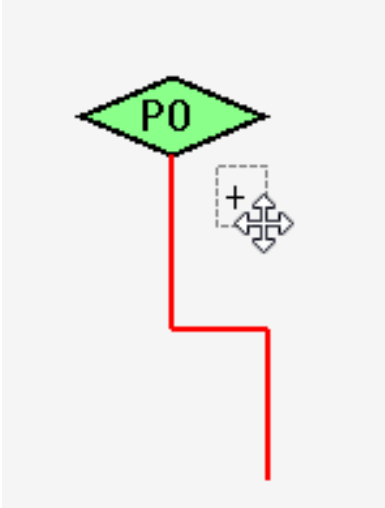


Рисунок 39 – Реакция ObjectGraph на курсор

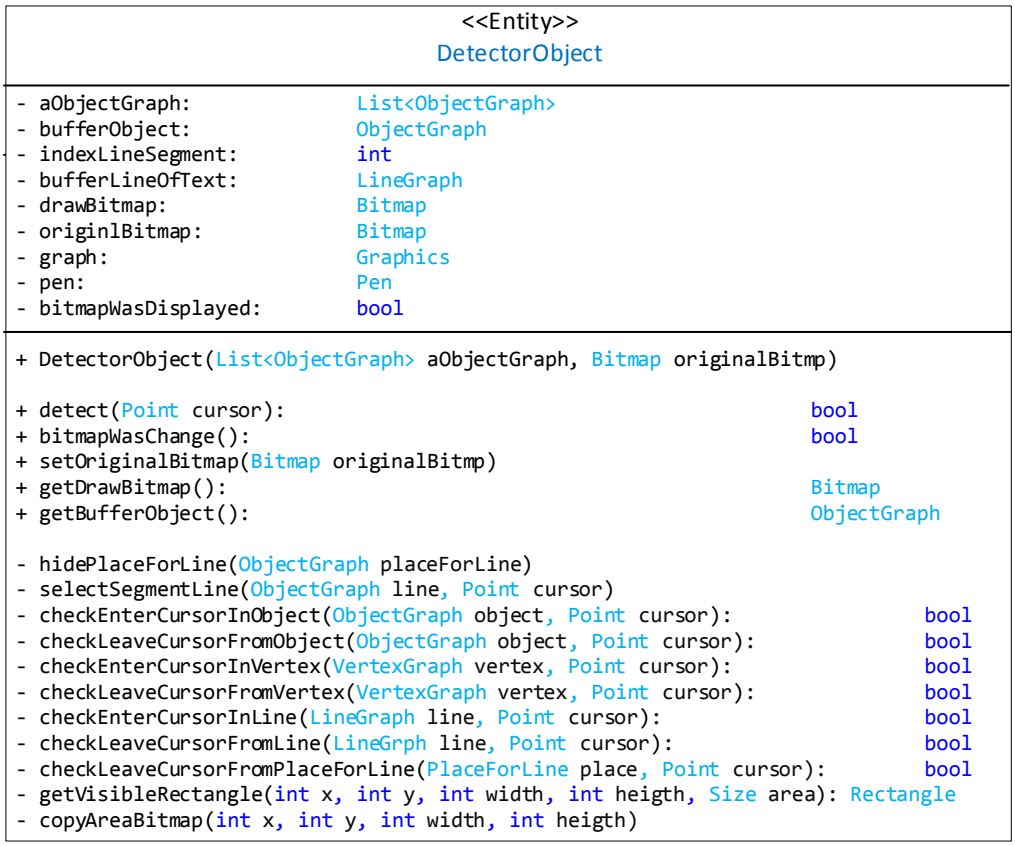


Рисунок 30 – Класс DetectorObject

Перемещение объектов

Для перемещения объекта используется класс MoveObject. Метод preparation вызывается, когда была зажата клавиша мыши на объекте, при этом входным параметром является перемещаемый объект. При перемещении мыши вызывается метод move, а после отжатия клавиши мыши вызывается метод finish.

При перемещении объекта перемещается только контур области объекта, чтобы не тратить время на перемещение всех точек контура объекта и связанных с данным объектом объектов. Так же при перемещении объекта происходит поиск объектов, с которыми может быть произведено вертикальное или горизонтальное выравнивание.

На рисунках 31-33 изображен процесс перемещения объекта, а на рисунке 34 изображен UML блок класса MoveObject.

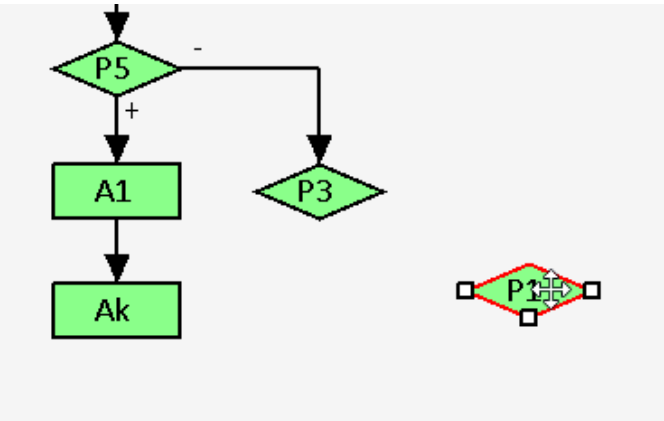


Рисунок 31 – Наведение на объект

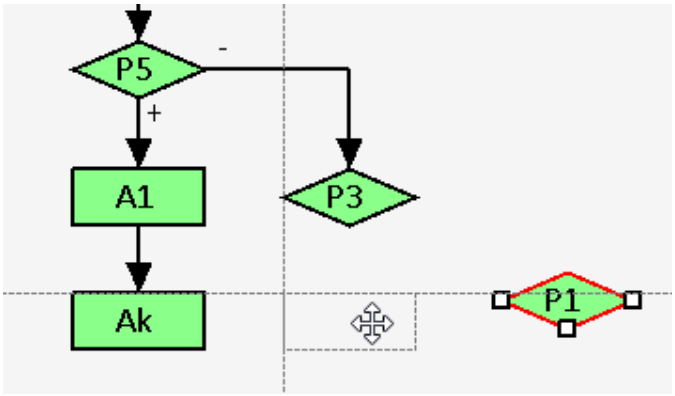


Рисунок 32 – Перемещение объекта с возможностью выравнивания его с другими объектами

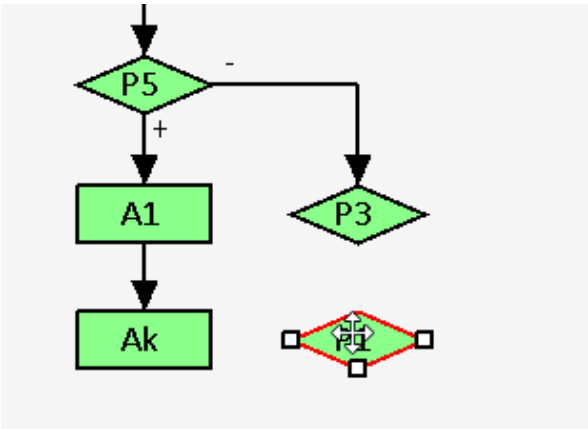


Рисунок 33 – Завершение перемещения

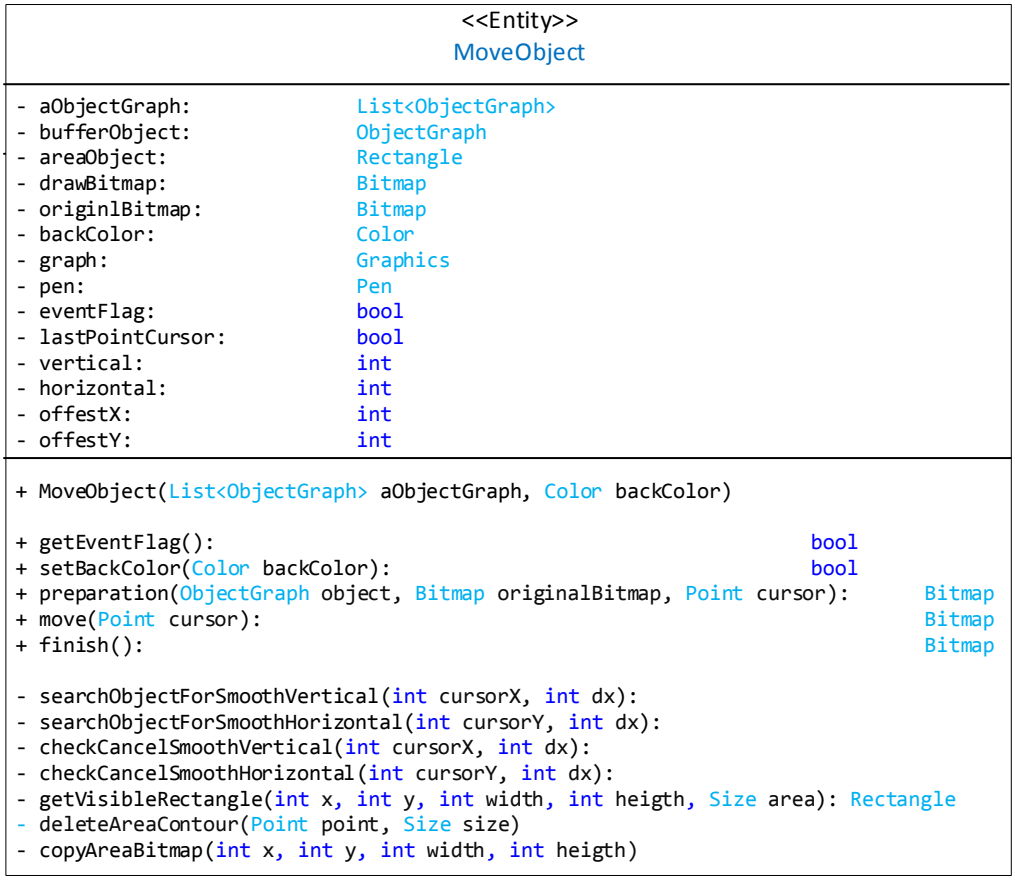


Рисунок 34 – Класс MoveObject

Построение линии

Для построения линии используется класс ConstructorLine. Метод preparation вызывается в том случае, если курсор был зажат на месте для продолжения линии. В случае, если курсор был зажат на месте для вывода линии из вершины, то сначала создается линий, а только затем вызывается данный метод. В момент перемещения курсора, строится либо горизонтальная линия, либо вертикальная, при этом проверяется, не указывает ли конец строящейся линии на место для соединения линии, либо на какую-либо линию. В момент, когда клавиша мыши отжимается, проверяется, не указывает ли линии на место для соединения линии, либо на линию. Если указывает, то происходит привязка линии к месту или к линии.

На рисунках 35-36 изображен процесс связывания вершин линией. В случае, если место для присоединения линии занято, то можно связать вершины через линию, рисунок 37.

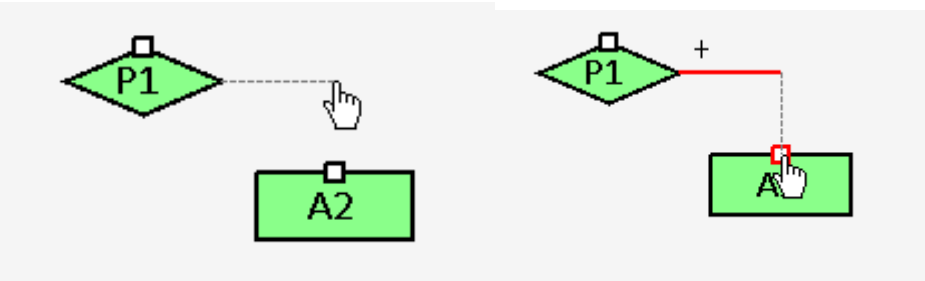


Рисунок 35 – Подведение линии к вершине

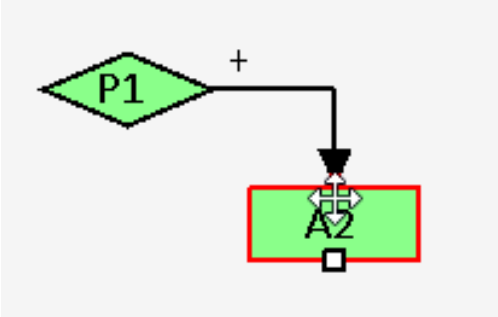


Рисунок 36 – Соединение линии с вершиной

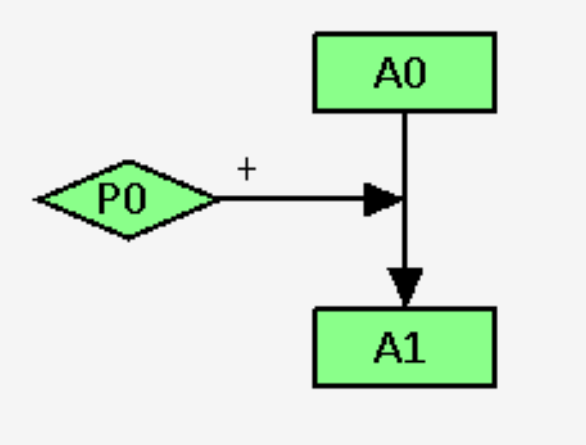


Рисунок 37 – Соединение вершину с вершиной через линию

Так же класс может заниматься удаление линии или ее сегмента, при этом производится отвязка других линий от данной линии, если данные линии больше не указывают на удаляемую линию. На рисунках 38-39 изображен процесс удаления линии.

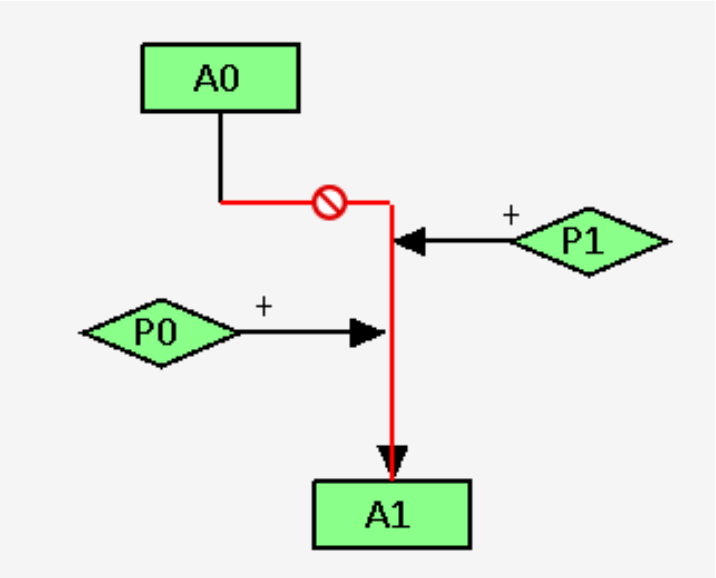


Рисунок 38 – Выбор сегмента для удаления

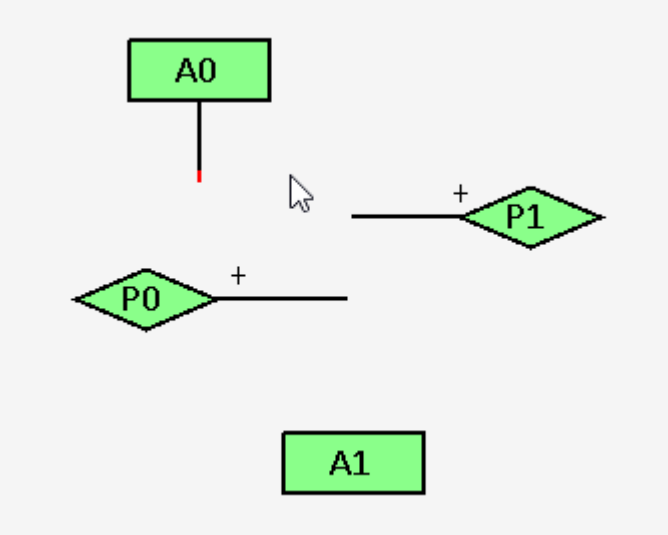


Рисунок 39 – Удаление сегментов линии с отвязкой других линий

Вывод

В данном разделе были разработаны классы объектов графа и интерфейс взаимодействия с ними. Классы и интерфейс находятся в отдельной библиотеке, чтобы можно было улучшать и исправлять ошибки. Для демонстрации возможностей классов были построены диаграммы классов, чтобы наглядно показать, какие поля и методы они имеют.

6. Проверка графической схемы алгоритма

В данном разделе будет рассмотрен принцип проверки графической схемы алгоритма. Проверка осуществляется на основе построенной линии, которая происходит динамически в после каждого соединения двух вершин линией.

Проверку построенного перехода выполняет класс CheckingTheGraph, который возвращает результат проверки в виде структуры, содержащих два публичных поля – поле наличие ошибки (hasErros) и поле обоснования ошибки (explanationError). Остальные классы проверки схем алгоритмов так же будут возвращать результат проверки с помощью данной структуры.

На рисунке 40 изображен класс CheckingTheGraph, который в конструкторе принимает операторный алгоритм и обозначения перехода при истине и перехода при лжи из условной вершины.

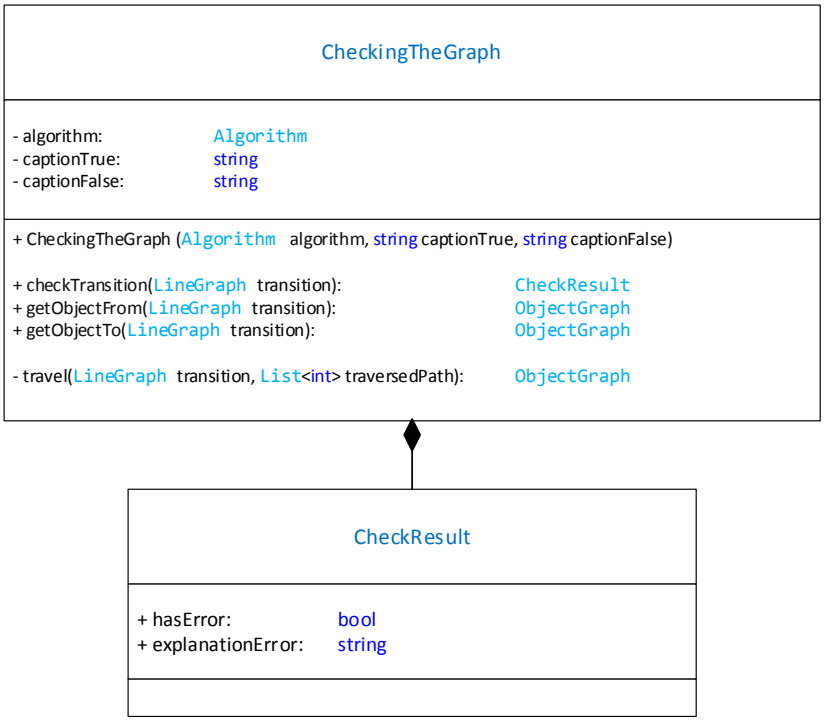


Рисунок 40 – Класс CheckingTheGraph

При соединении линии с объектом или другой линией, она имеет указатель на объект, из которой построен переход и объект либо линию, на которую указывает построенная линия. Если линия указывает на другую линию, то необходимо посмотреть, на какой объект указывает та линия. Та линия в свою очередь может указывать на другую линию и так далее, поэтому происходит проход по линиям. Проход по линиям продолжается до тех пор, пока не дойдем до объекта, либо не дойдем до линии, которая уже была в пройденном пути, либо не дойдем до линии, которая ни на что не указывает.

На рисунке 41 показан метод нахождения объекта, на который указывает линия. Метод работает рекурсивно и вызывает сам себя, при проходе по линиям. Так же, чтобы обнаружить замкнутый цикл линий, используется список идентификаторов линий, который хранит пройденных путь.

```
private ObjectGraph travel(LineGraph transition, List<int> traversedPath)
{
    //Если линии указывает на объект.
    if (transition.getObjectTo() != null) {
        //Если линия указвыает на линию.
        if (transition.getObjectTo().getType() == TYPE_OBJECT.LINE) {
            for (int i = 0; i < traversedPath.Count; i++) {
                if (traversedPath[i] == transition.getObjectTo().getID()) {
                    return null;
                }
            }
            //Переход к линии, на которую указвыает текущая линия.
            traversedPath.Add(transition.getObjectTo().getID());
            return travel((LineGraph)transition.getObjectTo(), traversedPath);
        }
        //Если линия указывает на объект, отличный от линии.
        else {
            return transition.getObjectTo();
        }
    }
    //Если линия не указывает на объект.
    else {
        return null;
    }
}
```

Рисунок 41 – Метод поиска объекта, на который указывает линия

Получив два объекта, соединенные линиями, определяется корректность построенного перехода. Во-первых, у обоих линий тип объекта должен быть VertexGraph, во-вторых, объекты должны иметь текст и в-третьих текст должен совпадать с одним из операторов или условий, которые присутствуют в сгенерированном алгоритме.

После проверки данных требований, происходит сравнение с сгенерированным алгоритмом. Если переход выходит из операторной вершины, то проверяется отсутствие текста у линии. Если текст отсутствует, то происходит поиск в алгоритме данного перехода и сравнивается вершина назначения. Если переход выходит из условной вершины, то проверяется наличие текста у линии, на основе которого выбирается один из путей в сгенерированном алгоритме, на основе которого происходит проверка.

В случае, если переход построен корректно из операторной вершины, либо оба перехода построены корректно в случае, если переходы идут из условной вершины, происходит выделение данной вершины в главной программе.

Так же, если линия была удалена, то все линии, указывающие на данную линию напрямую или через другие линии, являются не верными, из-за чего в главной программе так же снимается выделение с данных вершин.

На рисунке 42 изображен описанный алгоритм в виде графической схемы, в которой блоки о получении объектов, выполняют методы getObjectFrom и getObjectTo. А метод getObjectTo в свою очередь выполняет рекурсивное продвижение по линиям с помощью метода travel, описанного ранее.

Графическая схема алгоритма полностью считается корректной, если все переходы были построены, и все они были построены правильно. Так как класс CheckingTheGraph проверяет только отдельные переходы, то в главной программе происходит учет количества правильно построенных переходов и в главной программе проверяется завершение построения граф-схемы алгоритма.

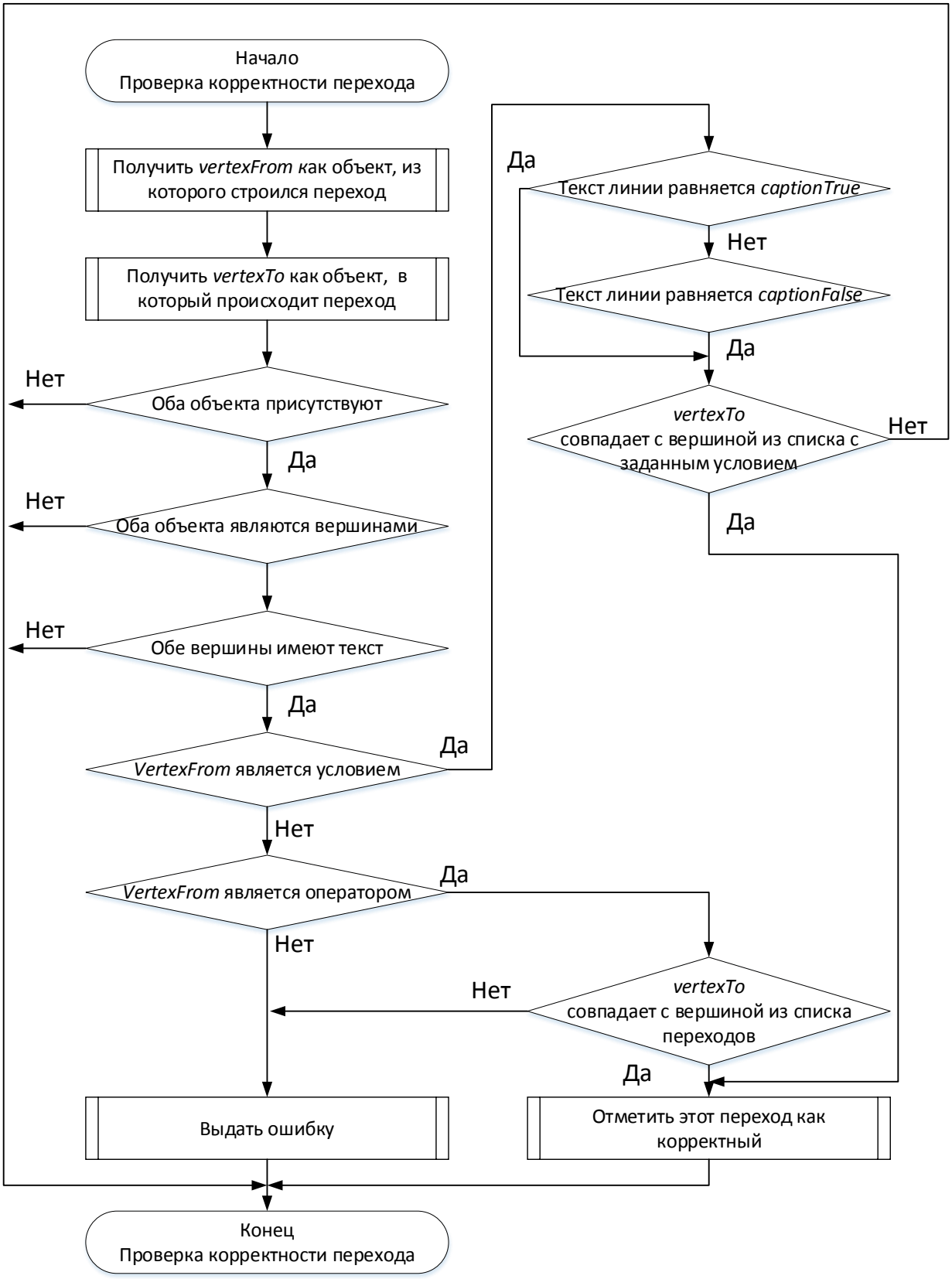


Рисунок 42 – Алгоритм проверки перехода

Вывод

В данном разделе был рассмотрен принцип проверки графической схемы алгоритмы, который требует проход по линиям до конечной вершины, так как линии могут указывать на другие линии. При этом сложность заключается в том, что при соединении одной линии к вершине, нужно проверить, не указывает ли какая-либо другая линия на данную линию, чтобы так же проверить переход.

7. Проверка матричной схемы алгоритма

В данном разделе описывается принцип проверки матричной схемы. Проверка осуществляется в три этапа. Первый этап – разбиение строки на лексемы. Второй этап – преобразование набора лексем в набор переходов и третий этап – проверки правильно построенных переходов. Весь данный процесс схож с работой первых трех ступеней компилятора, когда сначала производится лексический анализ, потом синтаксический анализ и затем семантический анализ.[5]

7.1. Разбиение строки на лексемы

Разбиением строки на лексемы занимается класс Lexer, который в конструкторе принимает операторный алгоритм для получения имен вершин. Набор имен вершин он сортирует по убыванию размера имен, чтобы при распознавании не принять подстроку более длинного имени за имя.

Под лексемой понимается допустимый набор символов, обозначающий либо зарезервированный символ (specialized), либо обозначающий имя операторной (nameOperator) или условной (nameCondition) вершины. Если набор символов не является допустимым, то есть набор символов не был распознан, то тип данной лексемы равняется notDefined. Помимо имени и типа лексемы имеется еще и информация о положении не распознанной лексемы в строке. На рисунке 43 изображено представление лексемы.

С помощью метода getListLexem происходит разбиение строки. В зависимости от типа схемы, выбирается свой список зарезервированных символов с помощью метода getSpecializedSymbol.

[5] Aho A. Compilers: Principles, Technizues, and Tools [Текст] /Aho. A.,Lam M. Ulman J. – Изд-во Addison-Wesley, 2006 год – 1184 с.

```
public enum TYPE_LEXEM
{
      notDefined,
      specialized,
      nameOperator,
      nameCondition
}

public struct Lexem
{
      public string       name;       //Сама лексема.
      public TYPE_LEXEM   type;       //Тип лексемы.
      public int          position;   //Позиция в строке.
}
```

Рисунок 43 – Представление лексемы

Всего 5 типов схем: MCA, S1, S2, S3, LSA:

- MCA: «¬», «V», «1», «-»;
- S1: «¬», «V»;
- S2: «¬», «V», «(», «)»;
- S3: «¬», «V», «↑», «↓», «^{0}», «^{1}», «^{2}», «^{3}», «^{4}», «^{5}», «^{6}», «^{7}», «^{8}», «^{9}», «*»;
- LSA: «¬», «V», «↑», «↓», «^{0}», «^{1}», «^{2}», «^{3}», «^{4}», «^{5}», «^{6}», «^{7}», «^{8}», «^{9}», «*», «ω».

На рисунке 44 изображен класс Lexer. Разбиение строки производится путем проверки очередного символа с зарезервированным символом. Если символ не является зарезервированным, то проверяются имена вершин с подстроками, длинами равными длине имен. Если не было найдено совпадение ни с одним именем вершины, то происходит получение не распознанной лексемы с помощью метода getNotRecognizedLexem, который идет по строке, пока не найдет распознанную лексему. При этом разбиение строки на лексемы не прекращается при обнаружении не распознанной лексемы.

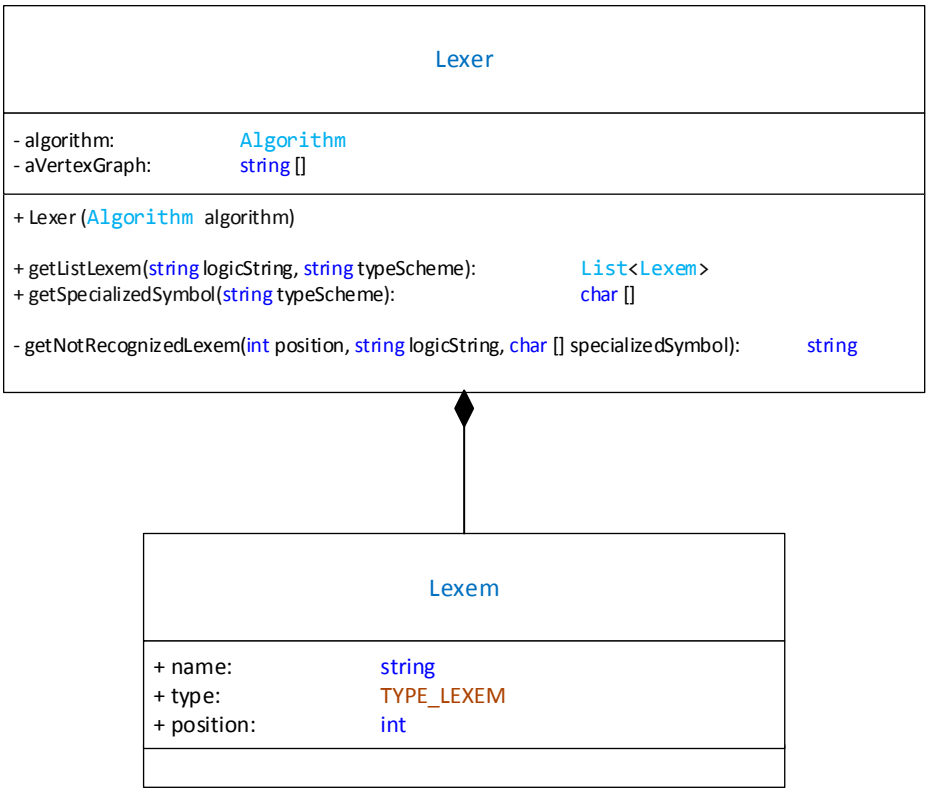


Рисунок 44 – Класс Lexer

7.2. Преобразование набора лексем в набор переходов

Ячейка матрицы содержит логическую функцию, связывающую два оператора между собой. Строка ячейки отвечает за операторную вершину, из которой происходит переход, а столбец ячейки отвечает за операторную вершину, в которую происходит переход. [6]

Связь между операторами может быть трех видов.

Связь между операторами может отсутствовать, тогда ячейка матрицы принимает значение «-».

Связь между операторами может быть прямая, то есть между операторами не находятся условные вершины, тогда ячейка принимает значение «1».

[6] Матвеев В.Д. Теория алгоритмов [Текст] / Матвеев В.Д., Матвеева Л.И – Киров: Изд-во ВятГУ, 2011 год. – 40 с.

Связь между операторами может быть косвенная, через условные вершины, тогда ячейка будет содержать логические условия.

Пример матричной схемы алгоритма приведен на рисунке 45.

	A1	A2	A3	A4	A5	A6	A7	A8	A9	Ak
A0	–	–	–	–	–	–	–	–	1	–
A1	–	1	–	–	–	–	–	–	–	–
A2	–	–	–	–	–	–	1	–	–	–
A3	–	–	–	–	–	1	–	–	–	–
A4	–	–	¬p2p3∨¬p2¬p3p5	–	–	¬p2¬p3¬p5¬p6	p2	–	–	¬p2¬p3 ¬p5p6
A5	–	–	1	–	–	–	–	–	–	–
A6	p1∨¬p1p4∨¬p1¬p4	–	–	–	–	–	–	–	–	–
A7	–	–	–	1	–	–	–	–	–	–
A8	–	–	–	–	1	–	–	–	–	–
A9	–	–	–	–	–	–	–	1	–	–

Рисунок 45 – Примет матричной схемы алгоритма

Обработку данных связей лучше всего выполнить с помощью детерминированного конечного автомата, с помощью которого можно не только наглядно представить процесс преобразования, что позволит быстрее находить и исправлять ошибки, но и позволит проверять корректность структуры строки.

В качестве автомата был взят автомат Мура, в котором выходной сигнал, зависит только от состояния, в котором находится автомат.[7]

На рисунке 46 представлены входные сигналы, которые могут поступить в автомат.

На рисунке 47 изображен автомат преобразования набора лексем в набор переходов, который содержит 10 состояний X0-X9 и который выдает 8 выходных сигналов в зависимости от состояния.

Ветвь X0-X1-X9 рассматривает первую ситуацию, когда после символа «-» должен идти конец строки.

[7] Ожиганов А.А. Теория автоматов [Текст] / Ожиганов А.А. – Санкт-Петербург: Изд-во ИТМО, 2013 год – 84 с.

Ветвь X0-X2-X9 рассматривает вторую ситуацию, когда после символа «1» должен идти конец строки. Причем, при виде данного символа должен создаться прямой переход между операторными вершинами.

Входные сигналы:	
1	- безусловный переход;
«1»	- текущая лексема является символом безусловного перехода;
«-»	- текущая лексема является обозначением отсутствия перехода;
«p*»	- текущая лексема является условной вершиной алгоритма;
«¬»	- текущая лексема является обозначением операции «НЕ»;
«V»	- текущая лексема является обозначением операции «ИЛИ»;
endLine	- флаг конца списка лексем;
successAdd	- флаг успешного добавления перехода/Переходов;

Рисунок 46 – Расшифровка входных сигналов

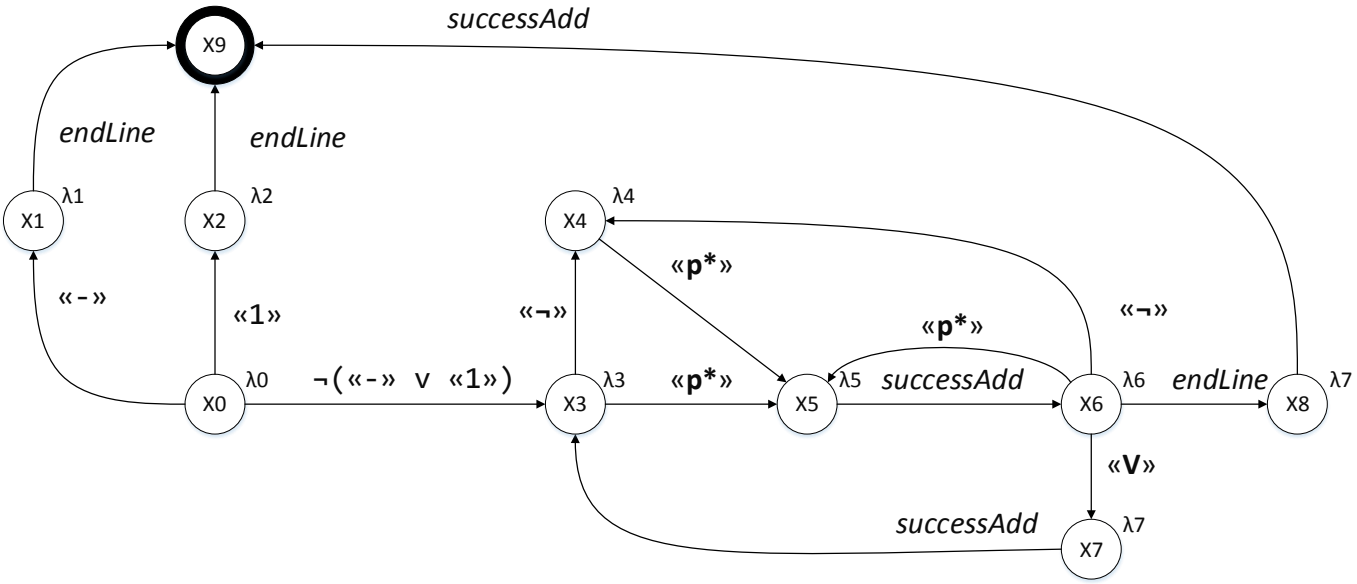


Рисунок 47 – Автомат преобразования набора лексем в набор переходов

Ветвь X0-X3 ведет к рассмотрению логической функции, которая состоит из условных вершин и символа отрицания.

Состояние X4 обрабатывает вершину отрицания, путем инверсии логического условия следующей условной вершины (изначально логическое

условие принимает истинное значение), а состояние X5 обрабатывает условную вершину, путем попытки добавления перехода. Попытка заключается в том, что начала двух и более путей может совпадать, поэтому они обязательно должны быть идентичными и в случае, если это не так, то возникает ошибка неоднозначного определения пути.

После успешного добавления, состояние X6 переходит к условной вершине, к которой в состоянии X5 был построен переход, при этом устанавливается логическое условие, которое является ложным, если до состояния X5 было состояние X4, иначе истинным.

В состоянии X6 может быть конец выражения, либо оно может продолжаться, причем продолжение может быть текущего пути, либо начала альтернативного, после встречи символа «ИЛИ», которое отправляет в состояние X7.

В состоянии X7 происходит построение перехода из текущей условной вершины в операторную вершину, к которой принадлежит столбец матрицы текущей ячейки. Состояние X8 делает то же самое, только вместо продолжения строки, оно ожидает конца.

В состоянии X3 происходит возврат к вершине, которой принадлежит строка матрицы и цикл повторяется.

На рисунке 48 представлено обоснование ошибок, в случае, если из состояния не будет найден выход.

Ошибки:

X0:	endLine:	Ячейка матрицы пуста;
	«V»:	Отсутствует левое выражение операции «ИЛИ»;
	«A*»:	Ячейка матрицы не может содержать операторную вершину ««*currentVertex*»;
X1:	При любой лексеме:	После символа «-» ожидался конец строки;
X2 :	При любой лексеме:	После символа «1» ожидался конец строки;
X3 :	endLine:	Отсутствует правое выражение операции «ИЛИ»;
X4:	endLine:	Отсутствует условная вершина после операции «НЕ»;
	«¬»:	Выражение «¬¬» является избыточным, необходимо его убрать;
	«A*»:	Ячейка матрицы не может содержать операторную вершину ««*currentVertex*»;
	При любых других лексемах:	Отсутствует условная вершина после операции «НЕ»;
X5:		Неоднозначно задан переход из вершины «*currentVertex» («currentLexem» и «addedVertex»);*
X6:	«-»:	Символ «-» не может находиться в строке с другими символами;
	«1»:	Присутсвтует символ «1», который не вяляется частью имени вершины и который не может находиться строке с другими символами;
	«A*»:	Ячейка матрицы не может содержать операторную вершину ««*currentVertex*»;
X7:		Неоднозначно задан переход из вершины «*currentVertex» («currentLexem» и «addedVertex»);*
X9:		Неоднозначно задан переход из вершины «*currentVertex» («currentLexem» и «addedVertex»);*

Расшифровка:
addedVertex – Вершина, в которую был построен переход до этого.

Рисунок 48 – Расшифровка ошибок

Выходные сигналы:

λ0:	*indexLexem*	= 0;
λ1:	*indexLexem*	= *indexLexem* + 1
λ2:	Добавить переход из *vertexFrom* в *vertexTo*	
	indexLexem	= *indexLexem* + 1
λ3:	*current*	= true;
	future	= true;
	currentVertex	= *vertexFrom*
λ4:	*future*	= *false*
	indexLexem	= *indexLexem* + 1
λ5:	Попытка добавления перехода из *currentVertex* В *currentLexem* под улсовием *current*	
λ6:	current	= future
	future	= true
	currentVertex	= currentLexem
	indexLexem	= *indexLexem* + 1
λ7:	Попытка добавления перехода из *currentVertex* В *vertexTo* под улсовием *current*	

Расшифровка:

currentLexem	*– Текущая лексема.*	
indexLexem	*– Индекс текущей лексемы в списке лексем.*	
Current	*– Логические условие текущей вершины*	*(… _ p* …)*
Future	*– Логическое условие следующей вершины*	*(… p* _p* …)*
currentVertex	*– Текущая вершина*	
vertexFrom	*- Операторная вершина, принадлежащая строке матрицы.*	
vertexTo	*- Операторная вершина, принадлежащая столбцу матрицы.*	

Рисунок 49 – Расшифровка выходных сигналов

7.3. Проверка набора переходов

При проверке нужно учитывать наличие одного из трех типов связей.

Если в исходном алгоритме из текущей операторной вершины есть прямой переход в другую операторную вершину, то проверяется, чтобы в построенных переходах так же был один переход, который вершину назначения имел ту же операторную вершину, что и в алгоритме.

Если в исходном алгоритме из текущей операторной вершины идет переход в условную вершину, то проверяется достижимость операторной вершины, принадлежащей столбцу матрицы. Если вершина недостижима, то список построенных переходов должен быть пуст. Если вершина достижима, то происходит проход по вершинам исходного алгоритма с проверкой соответствий переходов с построенными вершинами.

Из-за специфики матричной схемы, у некоторых условных вершин в списке построенных переходов будет отсутствовать переход, так как, по мнению пользователя, идя по этому переходу, нельзя достичь указанной операторной вершины. Поэтому происходит проверка, действительно ли указанная операторная вершина не достижима через условные переходы.

Вывод

Разбиение проверки схемы на три этапа позволило упростить проверку схемы, и позволило конкретизировать пояснение ошибки. То есть, если на этапе разбиения строки на лексеме возникнет ошибка, то будет сообщено об неопознанной лексеме. Если ошибка возникнет при преобразовании списка лексем в набор переходов, то будет сообщено об нарушении структуры строки. Если ошибка возникнет при проверке набора переходов, то будет сообщено об ошибке неправильно построенного перехода.

8. Проверка логической схемы алгоритма

В данном разделе будет рассмотрен принцип проверки логической схемы алгоритма, который, как и проверки матричной схемы, будет производиться в три этапа. Первый этап был описан в прошлом разделе, а в данном разделе будут описаны только этапы преобразования лексем в набор переходов и проверка наборов переходов. Преобразование делает автомат, который изображен в приложении Д, а в приложении К представлены выходные сигналы автомата.

8.1. Преобразования набора лексем в набор переходов

Рассмотрим пример логической схемы алгоритма, которая содержит 11 операторных вершин и 7 условных вершин. Пример изображен на рисунке 50.

$$\mathcal{U} = \text{A0}\downarrow^{8}\ \text{A8}\downarrow^{6}\ \text{A7}\ \text{p5}\uparrow^{1}\downarrow^{7}\ \text{A9}\ \text{A10}\ \text{A3}\downarrow^{3}\ \text{p6}\uparrow^{2}\ \omega\uparrow^{8}$$
$$\downarrow^{1}\ \text{A2}\omega\uparrow^{7}\downarrow^{2}\ \text{A5}\downarrow^{4}\ \neg\text{p1}\uparrow^{3}\ \text{A4}\ \neg\text{p2}\uparrow^{4}\ \text{A1}\ A6\ \text{p4}\uparrow^{1}\ \text{p3}\uparrow^{6}\ \text{Ak}$$

Рисунок 50 – Пример логической схемы алгоритма

В данном примере можно проследить ряд правил.

Во-первых, строка должна начинаться с начальной операторной вершины.

Во-вторых, после условной вершины должен идти символ «↑» с индексом, состоящим из цифр верхнего регистра, который отвечает за один из выходов условной вершины.

Во-третьих, должен быть символ «↓» с соответствующим индексом, который указывает на альтернативный путь. Индекс каждого альтернативного выхода должен быть уникальным.

В-четвертых, после символа «ω» так же должен идти символ «↑» с индексом, состоящим из цифр верхнего регистра. Этот символ отвечает за безусловный переход, что означает отсутствие связей между правой ли левой вершиной между данным символом. По этой причине, после символа «ω» должен идти символ «↓», либо конце строки, иначе, если после символа сразу будет идти вершина, то данная вершина будет не достижима ни из одной другой вершины.

В-пятых, после конечной вершины не может идти вершина, так как конечная вершина не имеет выхода. Для этого, после конечной вершины должен идти конец строки, либо символ «*», означающий разрыв.

Поэтому для выполнения первого правила, необходимо для начала проверить, является ли первая лексема начальным оператором. Это происходит в состоянии X0, рисунок 51. При переходе в состояние X1 текущая лексема становится текущей вершиной, для которой будет определяться переход. Поэтому обязательно после этого должна быть любая другая лексем, иначе будет отсутствовать вершина для перехода.

Если встречена любая лексема, кроме конца строки, то происходит переход в состояние X2.

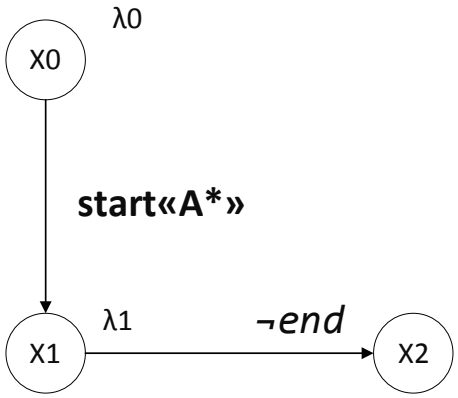


Рисунок 51 – Состояния X0, X1, X2

В состоянии X2 определяется один из четырех вариантов: либо текущая лексема является операторной вершиной, и тогда переход происходит в состояние X3, либо условной вершиной, и тогда переход происходит в состояние X6, либо символом альтернативного пути, тогда происходит переход в состояние X14, либо символом безусловного перехода, тогда переход происходит в состояние X20, рисунок 52.

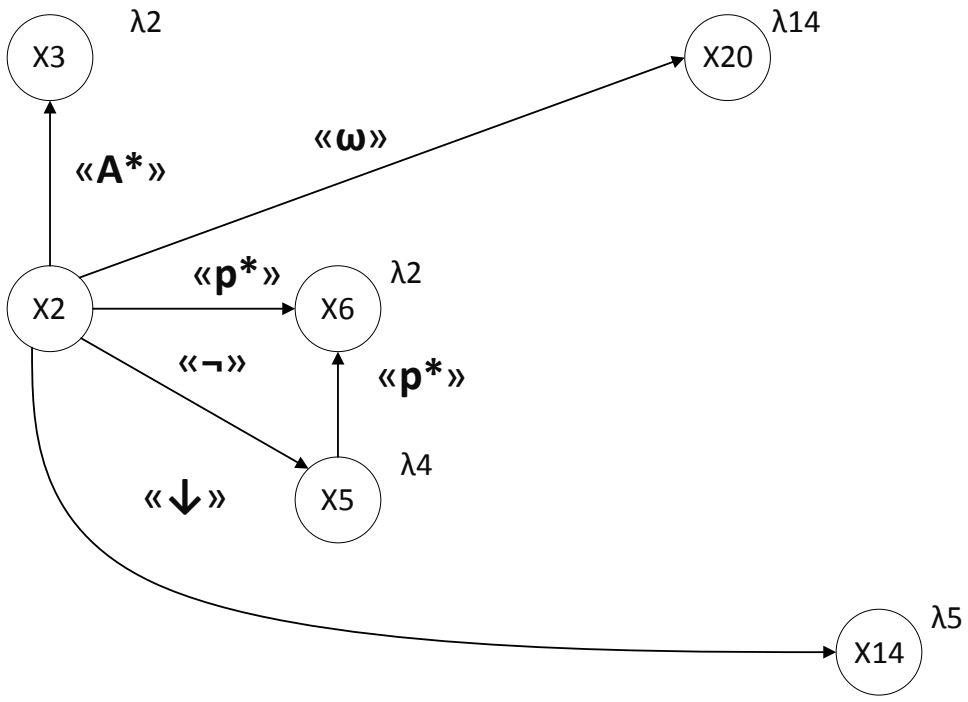


Рисунок 52 – Состояния X2, X3, X5, X6, X14, X20

Рассмотрим ситуацию, когда текущая лексема является операторной вершиной.

После перехода в состояние X3, происходит добавление перехода из текущей вершины в текущую лексему, которая содержит в себе имя операторной вершины. После успешного добавления происходит переход в состояние X4.

В состоянии X4 определяется, является ли операторная вершина конечной. Если не является, то происходит установка текущей лексемы как

текущей вершины с помощью состояния X1 и все продолжается заново. Если вершина является конечной, то происходит переход в состояние X21.

В состоянии X21 ожидается конец строки, тогда происходит переход в конечное состояние X24, либо символ «*», который говорит о том, что следующая идущая вершина не является вершиной назначения из текущей вершины, то есть, из конечной вершины. С помощью данной ветви автомата проверяется соблюдение пятого правила логической схемы алгоритма, рисунок 53.

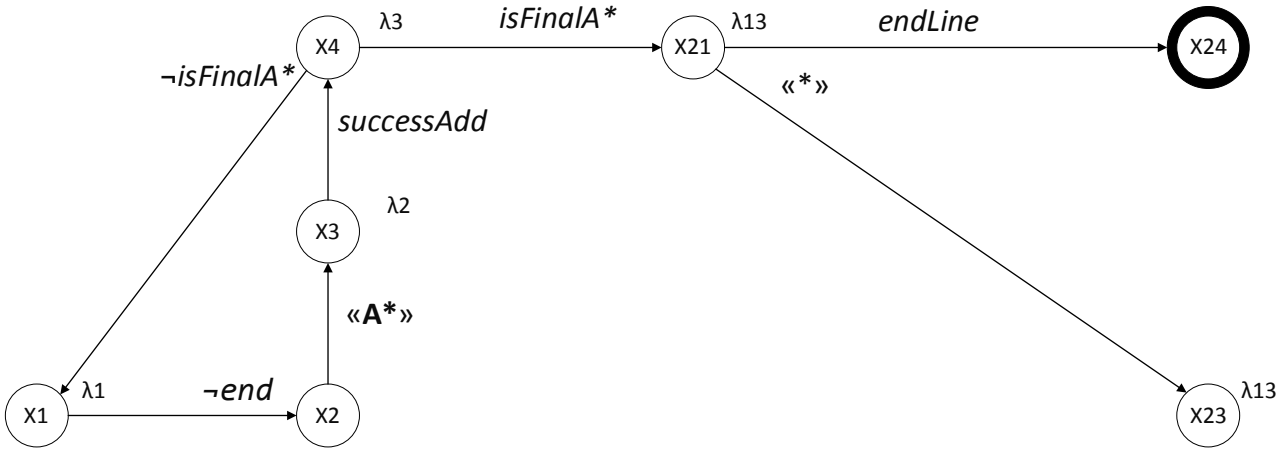


Рисунок 53 – Состояния X1, X2, X3, X4, X21, X23,24

Рассмотрим теперь ситуацию, когда текущая лексема является условной вершиной. При этом, если встречен символ отрицания, то это предполагает так же, что следом за данным символом будет условная вершина, просто в состоянии X5 будет установлено ложное логическое условие будущей условной вершины.

После успешного добавления перехода из текущей вершины с помощью состояния X6 и сдвига указателя лексемы с помощью состояния X7, ожидается символ «↑» с объявлением индекса данного перехода. Состояние X18 проверяет, чтобы после символа «↑» шли цифры верхнего регистра.

Если цифра является нулем, то происходит переход в состояние X10, если цифра отличная от нуля, то происходит переход к состоянию X9, который все цифры верхнего регистра будет собирать в буфер, так как все цифры верхнего регистра расположены в списке лексем как отдельные лексемы. Как только будет найдена лексема, отличная от цифры верхнего регистра, будет осуществлен переход к состоянию X10, рисунок 54.

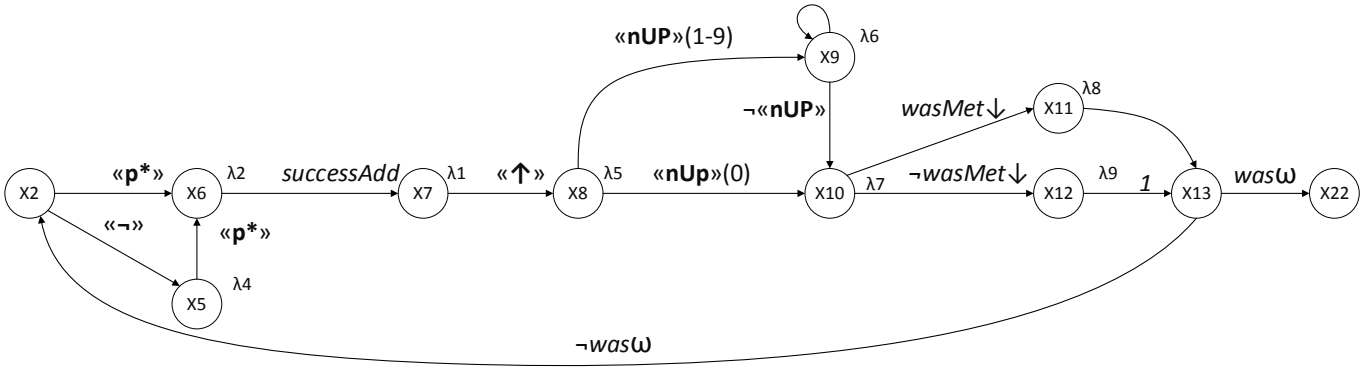


Рисунок 54 – Состояния X2, X5, X6, X7, X8, X9, X10, X11, X12, X13

После прочтения индекса, в состоянии X10 происходит проверка, не встречался альтернативный путь с данным индексом.

Если встречался, то происходит переход в состояние X11, в котором происходит добавление перехода из текущей условной вершины по инвертированному условию. То есть, если в состоянии X5 был определено то, что условие инвертировано, то альтернативный путь будет построен из пути по истине. Если же до вершины X6 были в состоянии X2, то альтернативный путь будет присоединен к пути по лжи.

Если же альтернативный путь еще не встречался, то происходит переход в состояние X12, в котором происходит добавление текущей вершины в список ожидающих вершин. Запись в списке имеет вид в виде ожидающей вершины, условия, при котором происходит переход (инвертированное текущее условие) и индекс альтернативного перехода.

Так как текущая вершина является вершина, из которой идет символ «↑» (это было установлено состоянии X7), и символ принадлежит условной вершине, то происходит переход к состоянию X2, в котором будет определяться переход из текущей условной вершины.

Рассмотрим теперь ситуацию, когда текущая лексема является символом «↓». В состоянии X14 проверяется, чтобы после символа «↓»шли цифры верхнего регистра. Состояния X14, X15, X16 схожи с состояниями X8, X9, X10, которые так же загружают в буфер цифры верхнего регистра.

В состоянии X16 происходит проверка на уникальность индекса текущего альтернативного перехода. Если индекс уникален, то с помощью состояния X18 происходит добавления переходов из всех тех вершин, которые ожидали данный альтернативный переход. Поэтому важно в этот момент еще провести проверку, чтобы после индекса шла либо операторная, либо условная вершина, на которую будет указывать данный альтернативный путь. Вершины находятся в ожидании в том случае, если сначала была встречена вершина, а потом встречен альтернативный переход. Данный список заполняется в состоянии X12, который описан был ранее.

Состояние X19 добавляет текущий альтернативный переход в список альтернативных переходов. Добавляет альтернативный переход он в виде индекса данного перехода и вершины, в которую он ведет. Список этот используется для проверки на уникальность, а так же для проведения перехода, если данный переход был объявлен до встречи вершин, которые будут переходить к этому альтернативному переходу.

Дальше происходит переход к состоянию X2, который снова выбирает один из пяти путей. Важным моментом является то, что переход из вершины, стоящей до символа «↓» не был определен, поэтому лексема не сдвигается и не происходит переход к другой вершине, а в очередном цикле будет определяться переход из вершины, встреченной до альтернативного перехода. Рисунок 55.

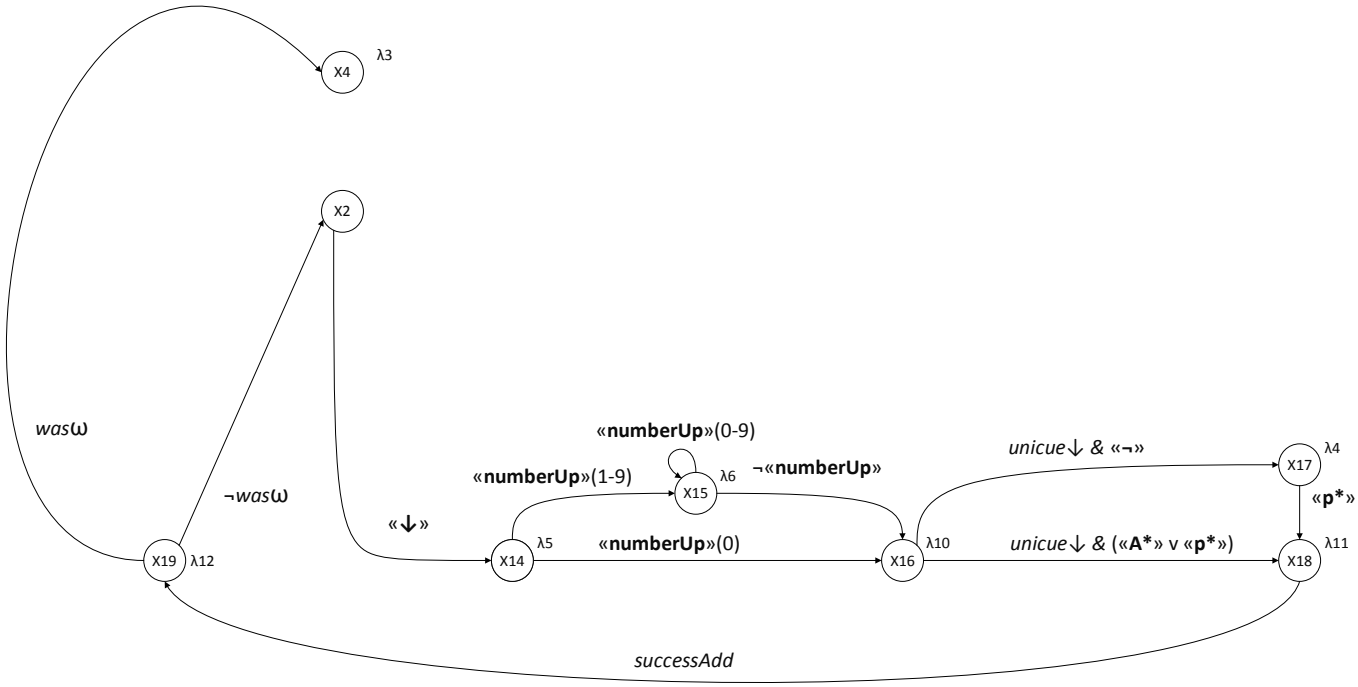


Рисунок 55 – Состояния X2, X14, X15, X16, X17, X18, X19, X4

И осталась последняя ситуация, когда текущая лексема является символом «ω». В данной ситуации происходит переход к состоянию X20, где происходит установка флага того, что был встречен безусловный переход.

Далее обработка схожа с обработкой условной вершины, то есть так же происходит переход по состояниям X8, X9, X10, X11, X12, X13, за рядом исключений.

Во-первых, данный символ может идти после операторной вершины, поэтому альтернативный переход будет не альтернативным, а прямым.

Во-вторых, в список ждущих вершин, в случае, если перед символом шла операторная вершина, будет записана именно операторная вершина.

В-третьих, после безусловного перехода ожидается либо конец строки, либо символ «↓», иначе вершина, идущая после безусловного перехода, будет недостижимой. Все это проверяется в состояниях X22 и X23. Так же разрешается символ «*», после которого все равно должен идти «↓», либо конец строки.

При встрече символа «↓» в состоянии X23 происходит переход в состояние X14, которое было описано ранее, с одним исключением. Из состояния X19 будет осуществлен переход в состояние X4, а не в X2, так как текущая вершина не определена, потому что безусловный переход говорит о том, что вершина, идущая перед безусловным переходом и после безусловного перехода, никак не связаны. Так как указатель индекса лексемы, находится на вершине, идущей после безусловного перехода, то текущей вершиной становится именно эта лексема, а так как текущая лексема может быть конечной вершиной, то происходит переход именно в состояние X4, где проверяется, является ли текущая лексема конечным оператором, рисунок 56.

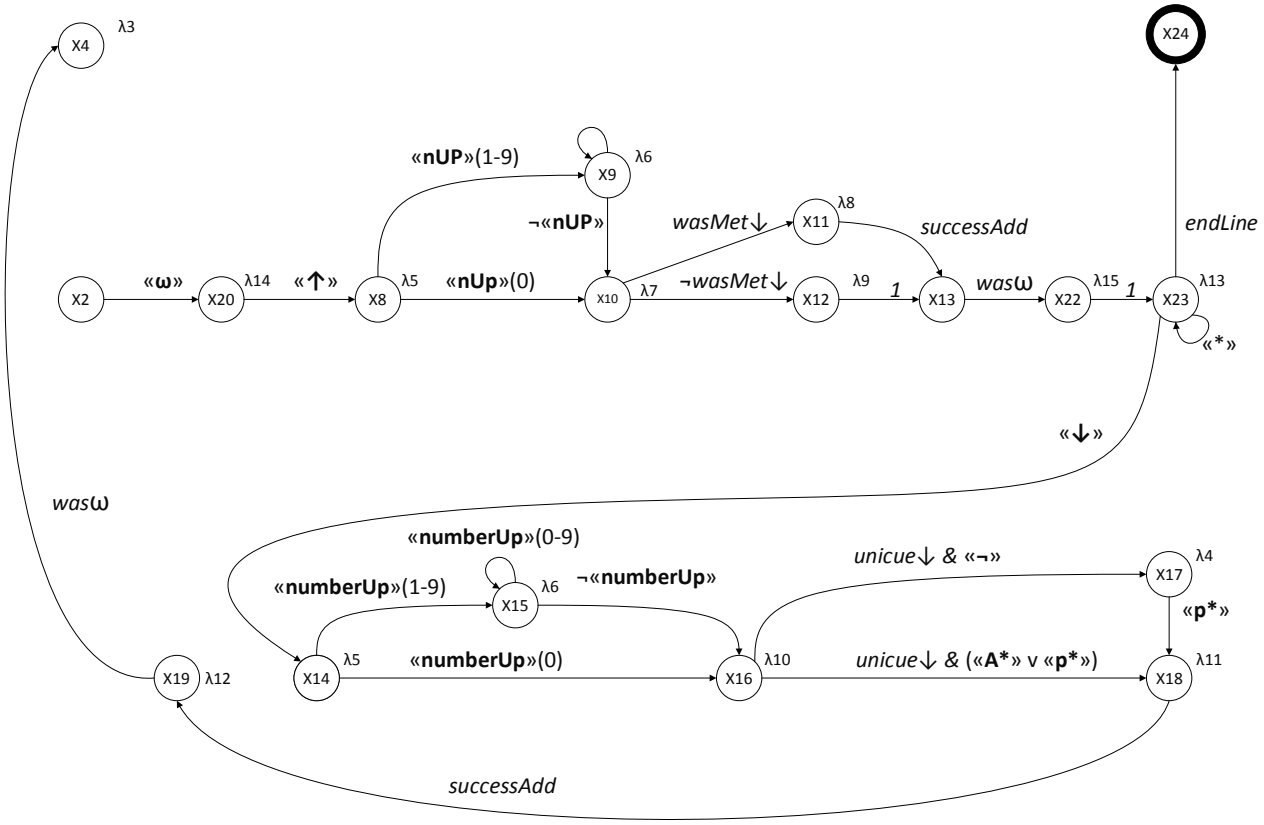


Рисунок 56 – Пути обработки символа «*»

8.2. Проверка наборов переходов

В логической схеме, в отличие от матричной, в строке должны присутствовать все переходы, поэтому проверка переходов осуществляет не путем обхода вершин алгоритма с проверкой на достижимость, а путем проверки наличия всех переходов, которые есть в алгоритме. Перед этим проверяется, чтобы все вершины были объявлены один раз в строке, а так же проверяется отсутствие тех условных вершин, которые ведут при истине и лжи к одной и той же вершине, так как логическая схема будет не оптимальна при их присутствии.

Вывод

В данном разделе был разработан автомат для проверки логической схемы алгоритма. Использование детерминированного конечного автомата позволило ускорить процесс разработки преобразователя, а так же позволило не только производить преобразования логической схемы в набор переходов, но и проверять структуру схемы и выдавать ошибки с пояснением.

9. Разработка пользовательского интерфейса

В данном разделе будет разработан эскиз пользовательского интерфейса. Главная задача состоит в том, чтобы расположить компоненты максимально комфортно и эффективно для использования.

Эскиз окна главного меню изображен на рисунке 57. Окно содержит всего две области: область словесного описания алгоритма и области кнопок открытия схем алгоритма.

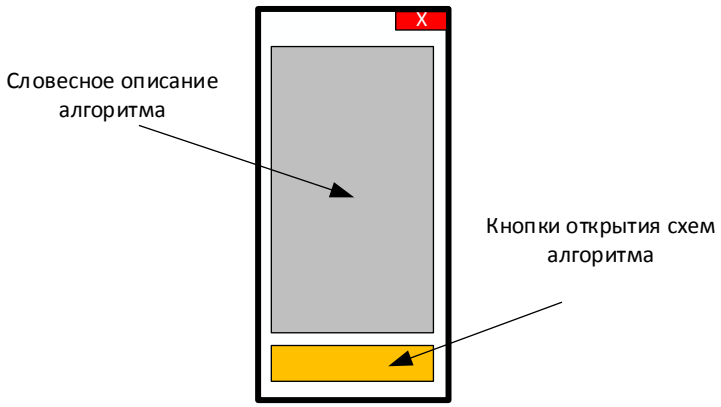


Рисунок 57 – Эскиз окна главного меню

Схемы будут открываться в отдельных окнах, что позволит больше места выделять под схемы. Плюс это позволит пользователю визуально иметь доступ к двум схемам одновременно, нежели, если бы схемы располагались в одном окне.

Эскиз окна построения графической схемы алгоритма изображен на рисунке 58. Окно содержит всего три области: область панели управления, область построения схемы и область информации о выполнении задания.

Панель управления будет содержать только список вершин, которые могут быть добавлены в графическую схему.

Было принято решение, что удобнее будет выбирать из списка имя вершины, нежели вводить его с помощью клавиатуры.

Снизу окна будет находиться информационное табло, которое будет содержать в себе информацию о состоянии выполнения задания (выполнено или нет), прогресс выполнения задания, количество совершенных ошибок и время выполнения текущего задания.

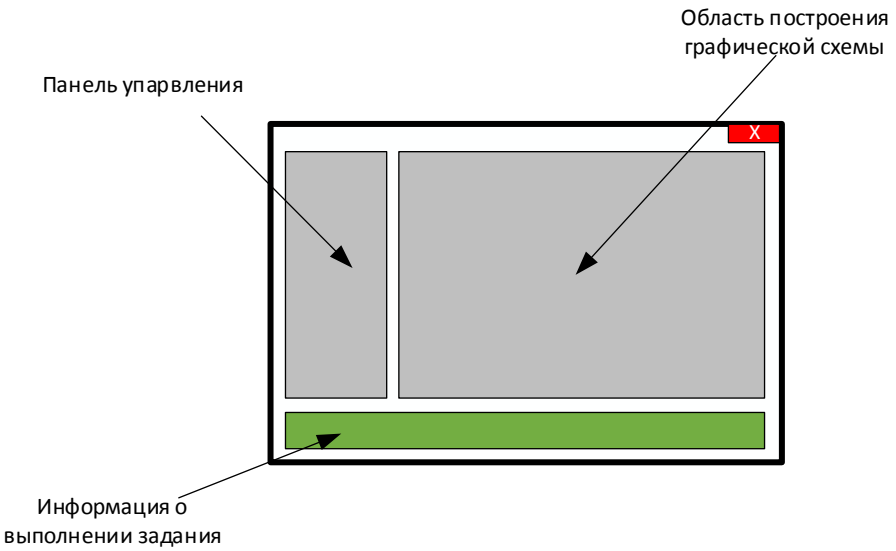


Рисунок 58 – Эскиз окна построения графической схемы алгоритма

На рисунке 59 изображен эскиз окна построения матричной схемы алгоритма. Так как над матричной схемой выполняются действия только выбора ячейки и ввода значения в ячейку, то окно для построения матричной схемы алгоритма содержит только область построения схемы и область информации о выполнении задания.

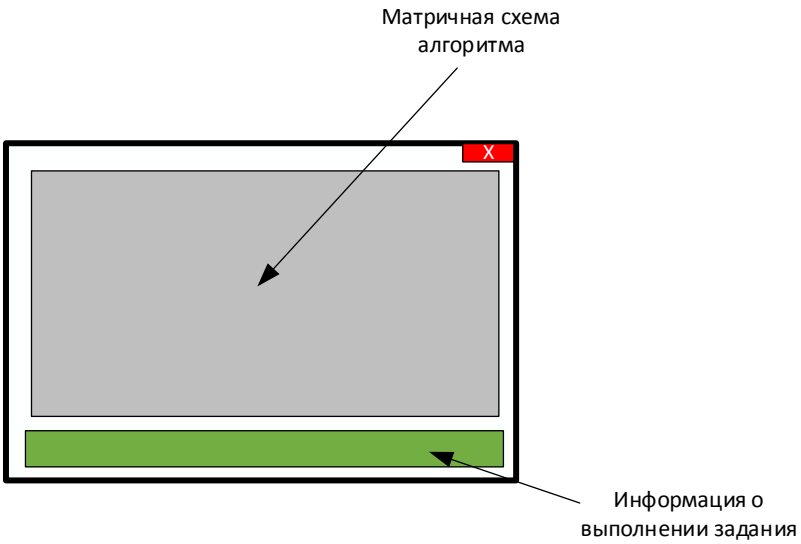


Рисунок 59 – Эскиз окна построения матричной схемы алгоритма

Логическая схема алгоритма будет разделена на три этапа: построение формул переходов, минимизация схемной формулы и построение самой логической схемы. Так как все этапы не поместятся в одном окне, то необходимо либо добавлять возможность прокручивать схемы, либо поместить этапы во вкладки. Был выбран второй вариант. На рисунке 60 изображен эскиз окна построения логической схемы алгоритма.

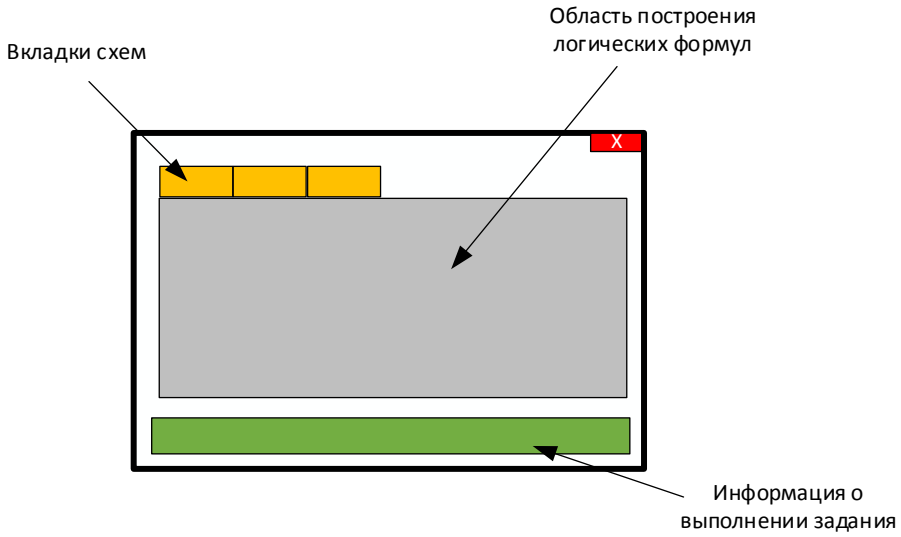


Рисунок 60 – Эскиз окна построения логической схемы алгоритма

Вывод

В данном разделе был разработан пользовательский интерфейс программы. Стиль оформления был выбран минималистичный, то есть на экране будет малое количество компонентов. Данный стиль позволит иметь больше места для самих схем, а так же позволит без больших усилий разобраться в интерфейсе. Так же было принято решение для каждой схемы выделить свое собственное окно, что позволит, при необходимости, получить доступ к нескольким схемам одновременно, плюс это позволит пользователю самому управлять пространством экрана, оставляя на экране только те схемы, которые необходимы ему на данный момент.

10. Построение автоматов для преобразования формул переходов

В данном разделе будут разработаны автоматы для преобразования формул переходов в набор переходов. Процесс преобразования так же будет выполняться в три этапа: разбиение формулы на лексемы, проверка структуры формулы и преобразование ее в набор переходов, проверка набора переходов. Все автоматы находятся в приложениях Б-Г, а в приложениях З-Й находятся выходные сигналы данных автоматов.

10.1. Система формул переходов S1

Рассмотрим пример системы формул переходов S1, который содержит 11 операторных вершин и 7 условных вершин. Пример изображен на рисунке 61.

A0 → A8
A8 → A7
A7 → p5A9 ∨ ¬ p5A2
A2 → A9
A9 → A10
A10 → A3
A3 → p6A8 ∨ ¬ p6A5
A5 → p1p6A8 ∨ p1¬ p6A5 ∨ ¬ p1A4
A4 → p2 p1p6A8 ∨ p2p1¬ p6A5 ∨ p2¬ p1A4 ∨ ¬ p2A1
A1 → A6
A6 → p4p3Ak ∨ p4¬ p3A7 ∨ ¬ p4A2

Рисунок 61 – Пример системы формул переходов

Как видно из примера, из операторной вершины может быть два типа перехода. Первый из них – это безусловный переход. Примеры изображены на рисунке 62.

A0 → A8
A8 → A7
A2 → A9
A9 → A10
A10 → A3
A1 → A6

Рисунок 62 – Пример безусловных переходов

В начальном состоянии X0, при встрече операторной вершины, происходит переход в состояние X1, в котором ожидается конец строки. При этом в состоянии X1 происходит построение перехода из операторной вершины, которой принадлежит логическая функция, в операторную вершину, которая указана в логической функции. Рисунок 63.

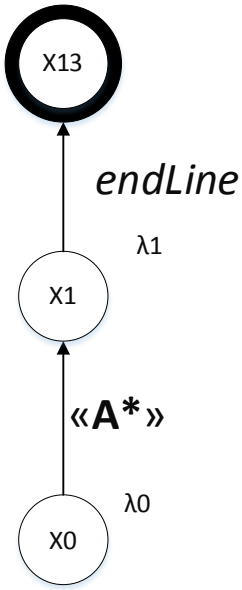


Рисунок 63 – Состояния X0, X1, X13

Второй тип – это переход при выполнении условий.

A7 → **p5**A9 ∨ ¬ **p5**A2
A3 → **p6**A8 ∨ ¬ **p6**A5
A5 → **p1**p6A8 ∨ **p1**¬ p6A5 ∨ ¬ **p1**A4
A4 → **p2** p1p6A8 ∨ **p2**p1¬ p6A5 ∨ **p2**¬ p1A4 ∨ ¬ **p2**A1
A6 → **p4**p3Ak ∨ **p4**¬ p3A7 ∨ ¬ **p4**A2

Рисунок 64 – Пример условных переходов

Данная ситуация рассматривается, перейдя в состояние X3, перед этим может быть осуществлен переход в X2, если условие будет инвертировано, рисунок 65.

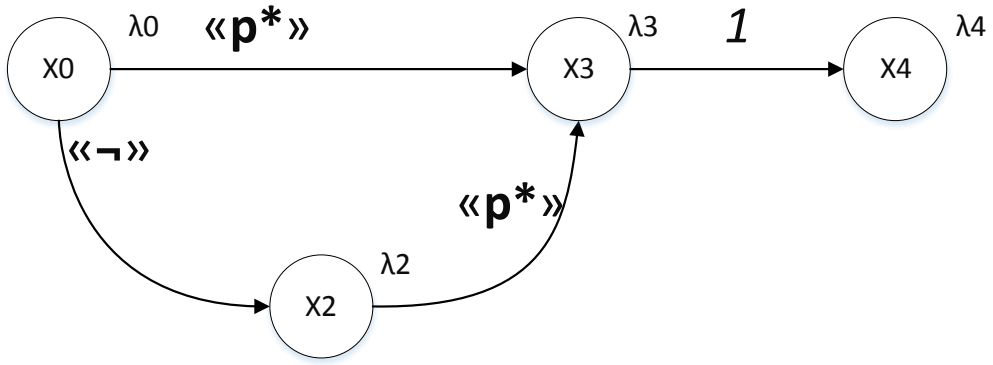


Рисунок 65 – Состояния X0, X2, X3

В состоянии X3 происходит построение перехода из операторной вершины, к которой принадлежит логическая функция, в первую условную вершину. Далее происходит безусловный переход к состоянию X4, который является началом цикла обработки логической функции с условными переходами.

В состоянии X4 происходит переход к вершине, которая была встречена до этого, то есть текущая лексема.

Из состояния X4 есть три перехода, рисунок 66.

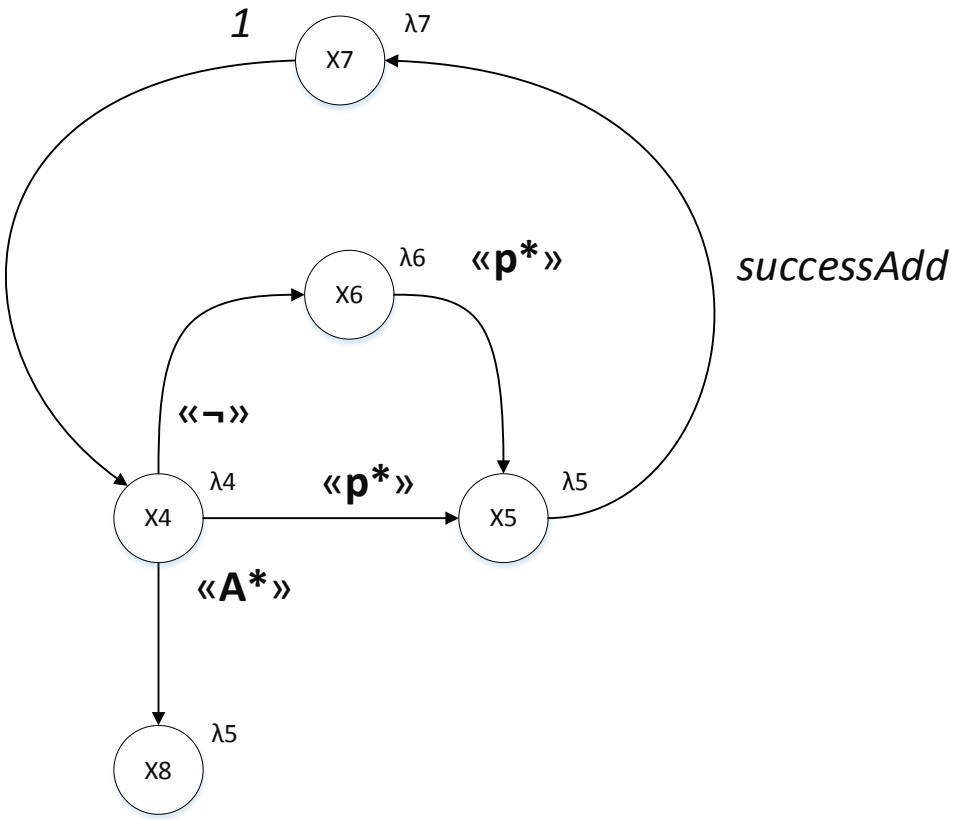


Рисунок 66 – Состояния X4, X5, X6, X7, X8

Первый при встрече символа отрицания переход ведет к состоянию X6, которое инвертирует логическое условие следующей условной вершины. Второй переход при встрече условной вершины ведет к вершине X5, которая производит попытку добавления перехода. Попытка заключается в том, что начала цепочек путей могут совпадать. На рисунке 64 это продемонстрировано.

Поэтому необходимо, чтобы вновь добавляемый переход либо был новый, либо совпадал с уже построенным переходом, иначе будет ошибка неоднозначного определения перехода.

После того, как переход был успешно добавлен, происходит переход к состоянию X7, который присваивает логическое условие, измененное в состоянии X6, либо равное истине, если перед состоянием X5 было состояние

X4. После состояния X7 происходит безусловный переход к состоянию X4 и цикл повторяется.

То есть цикл состояний X4, X5, X6, X7 обрабатывает ситуацию, когда условные вершины идут друг за другом.

В случае, если после очередного логического условия встречена операторная вершина, то происходит переход в состояние X8, которое производит попытку добавления перехода из текущей условной вершины в операторную вершину.

Из-за специфики схемы S1, после операторной вершины в логической функции не может идти следом другая вершина, поэтому ожидается конец строки, либо символ операции «ИЛИ», который говорит о том, что есть альтернативный путь в эту же или в другую операторную вершину.

Состояния X9, X10, X11, X12 производят подготовку к обработке альтернативного пути.

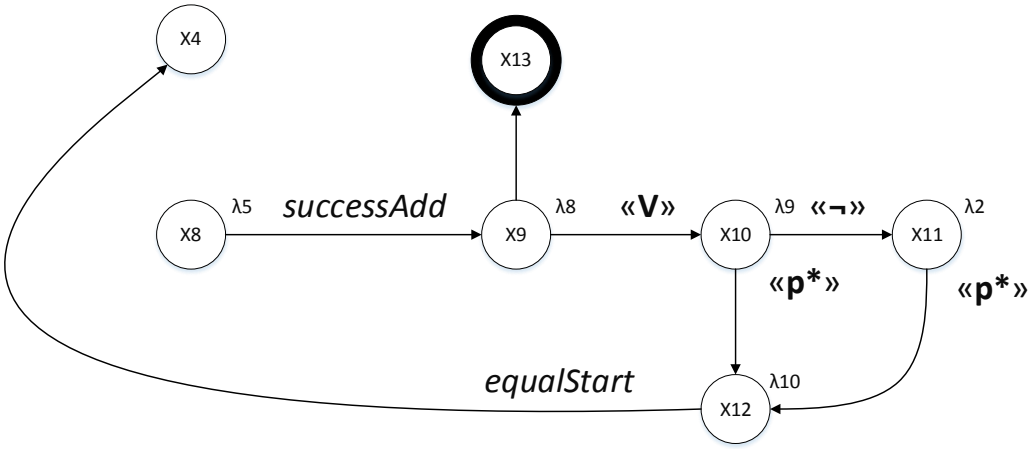


Рисунок 67 – Состояния X9, X10, X11, X12

Во-первых, ожидается символ операции «ИЛИ», затем ожидается либо символ отрицания, после которого должна идти условная вершина, либо условная вершина. Далее эта условная вершина должна совпадать с началом других альтернативных путей. На рисунке 64 начала путей отмечены жирным

шрифтом. Только после выполнения всех этих моментов, происходит переход в состояние X4, которое начинает цикл обработки альтернативного пути.

10.2. Система скобочных формул переходов S2

Рассмотрим пример системы скобочных формул переходов S2, которая является преобразованием формулы S1 из рисунка 61.

A0 → A8
A8 → A7
A7 → p5A9 ∨ ¬ p5A2
A2 → A9
A9 → A10
A10 → A3
A3 → p6A8 ∨ ¬ p6A5
A5 → p1 (p6A8 ∨ ¬ p6A5) ∨ ¬ p1A4
A4 → p2 (p1(p6A8 ∨ ¬ p6A5)∨ ¬ p1A4)∨ ¬ p2A1
A1 → A6
A6 → p4 (p3Ak ∨ ¬ p3A7) ∨ ¬ p4A2

Рисунок 68 – Пример системы скобочных формул переходов

В S2 так же, как и в S1, присутствуют два типа переходов – безусловный и переход при выполнении логических условий. Обработка первого типа ничем не отличается от S1, а отличается только второй тип.

Отличие заключается в том, что появляются подмножества, в которые происходит переход из условных вершин. Данные подмножества выделены скобками на рисунке 68.

В данных логических функциях можно проследить ряд правил.

Во-первых, переход осуществляется напрямую из условной вершины, то подмножество не объявляется. Это видно из примера, в котором есть переходы p5A9, ¬ p1A4 и другие. Если же из условной вершины происходит переход в другую условную вершину, то объявляется подмножество.

Во-вторых, в одном подмножестве может быть только два выражения, над которыми выполняется операция «ИЛИ». Возьмем из примера ситуацию формулы операторной вершины A4:

Подмножество 0: **p2**(…) ∨ ¬ **p2**A1

Подмножество 1: **p1**(…)∨ ¬ **p1**A4

Подмножество 2: **p6**A8 ∨ ¬ **p6**A5

Из второго правила вытекает третье правило, которое заключается в том, что первые вершины двух путей должны совпадать. Данные вершины отмечены жирным шрифтом в подмножествах.

И четвертое правило заключается в том, что логические условия в одном подмножестве должны быть различны, при этом не обязательно, в каком порядке они идут, то есть следующая формула эквивалентна формуле из операторной вершины A4:

$$A4 \rightarrow \neg p2A1 \vee p2\,(\neg p1A4 \vee p1(p6A8 \vee \neg p6A5))$$

В итоге, из-за перечисленных требований необходимо вводить магазинную память в автомат, в который будет загружаться флаг о том, какое выражение в подмножестве будет обработано (первое или второе), логическое условие первой вершины пути подмножества, а также сама условная вершина подмножества. Именно эти данные загружаются в магазинную память в виде стека в состоянии X4. Рисунок 69.

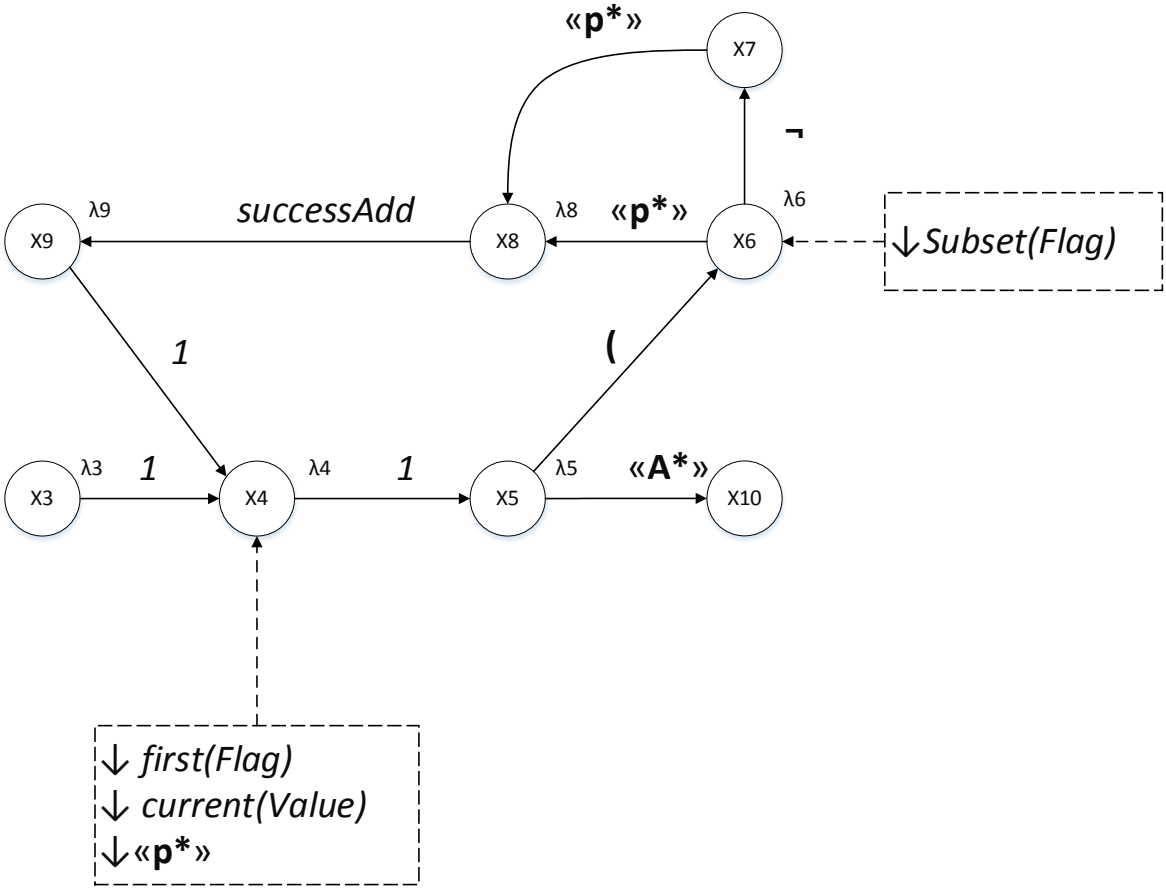


Рисунок 69 – Состояния X3, X4, X5, X6, X7, X8, X9, X10

Далее в состоянии X5 происходит переход к условной вершине, которая записана в текущей лексеме. После чего ожидается два варианта, либо открывающая скобка подмножества, либо операторная вершина.

Если текущая лексема равняется открывающей скобке, то происходит переход из состояния X5 в состояние X6.

В состоянии X6 в стек записывается флаг того, что было вхождение в подмножество. Далее ожидается первая условная вершина подмножества, в которую строиться переход из вершины, стоящей перед объявлением подмножества, но перед этим, при необходимости, происходит переход в состояние X7 для инвертирования логического условия вершины будущей вершины.

После успешного добавления перехода из текущей условной вершины в вершину, которая находится в текущей лексеме, происходит переход в состоянии X9. После этого происходит переход в состояния X4 и X5, так как снова происходит обработка начала подмножества.

Если в состоянии X5 встречена операторная вершина, происходит переход в состояние X10, в котором происходит попытка добавления перехода из текущей условной вершины в операторную вершину. При успешном добавлении перехода происходит переход в состояние X11.

В состоянии X11 происходит чтение данных из стека. В конце стека может находиться либо флаг, говорящий о том, что было обработано первое выражение подмножества, либо флаг о том, что было обработано второе выражение подмножества, рисунок 70.

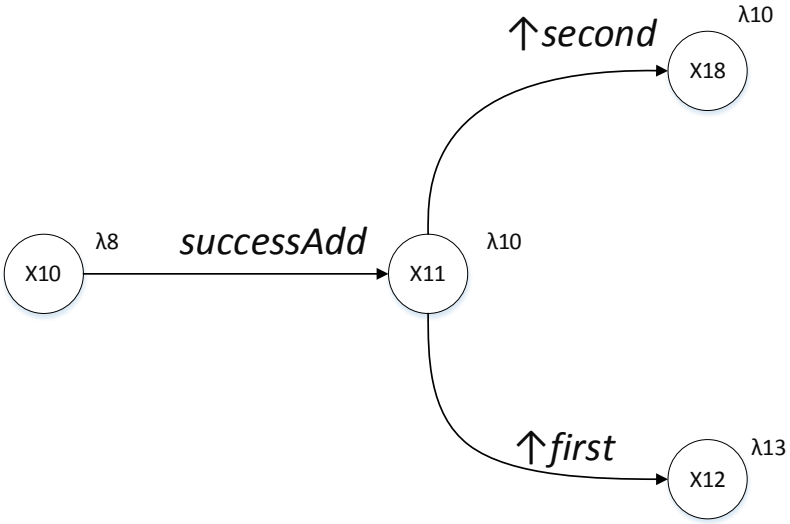


Рисунок 70 – Состояния X10, X11, X12, X18

Если было первое выражение подмножества, то ожидается символ операции «ИЛИ», после чего ожидается условная вершина, либо символ отрицания, после которого должна идти условная вершина.

В состоянии X15 вновь происходит чтение из стека. Прочитанное значение должно быть отлично от текущего логического условия, иначе будет

нарушено четвертое требование, описанное ранее. Далее в состоянии X16 вновь происходит чтение из стека и прочитанное значение должно совпадать с текущей вершиной, иначе будет нарушено третье требование, описанное ранее, рисунок 71.

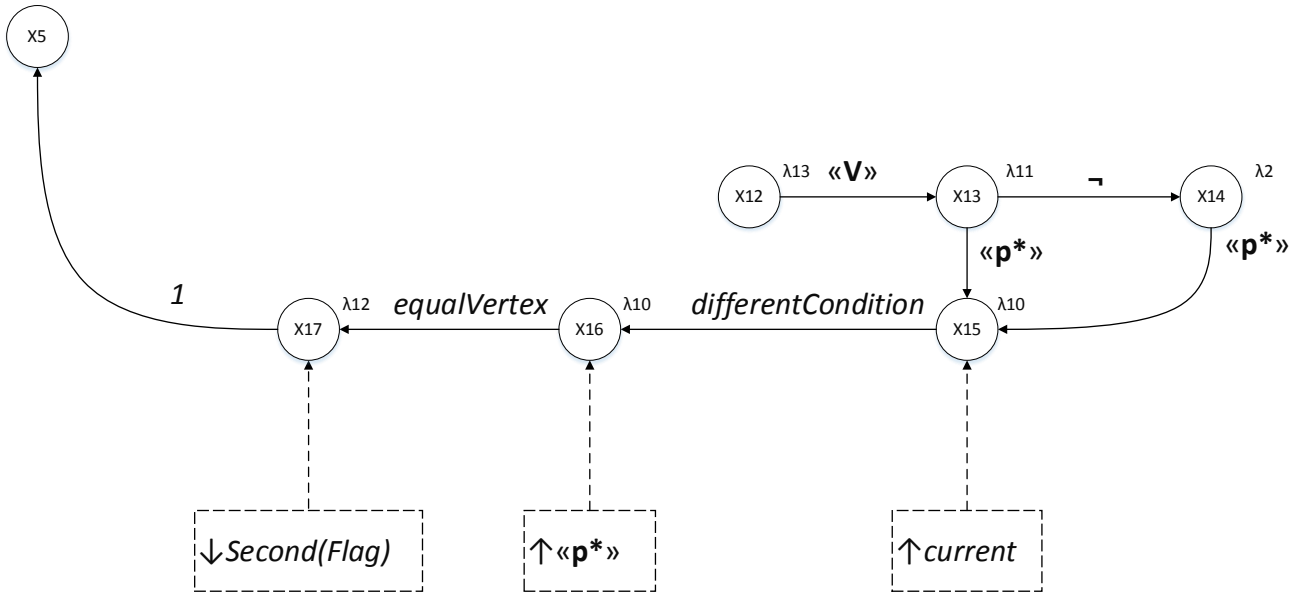


Рисунок 71 – Состояния X12, X13, X14, X15, X16, X17, X3

Если все требования соблюдены, то происходит переход в состояние X17, где записывается флаг того, что будет обработано второе выражение подмножества, после чего происходит возврат в состояние X5, к котором начинается повторный цикл для обработки второго выражения подмножества.

После обработки второго подмножества в состоянии X11 будет прочитан флаг того, что было обработано второе подмножество, после чего произойдет переход в состояние X18.

В состоянии X18 происходит чтение из стека. Если из стека не было прочитано никакого значения, то есть стек был пуст, то происходит переход в состояние X20, где ожидается конец выражения. Если из стека было прочитано значение, которое обязательно должно равняться флагом того, что было

вхождение в подмножество, то ожидается закрывающая скобка, говорящая о том, что происходит выход из подмножества, рисунок 72.

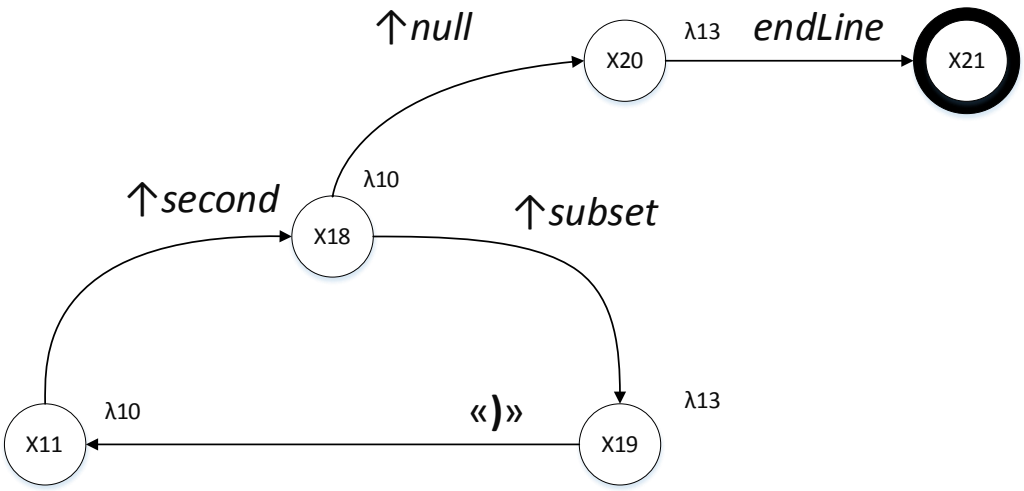


Рисунок 72 – Состояния X11, X18, X19, X20

После выхода из подмножества происходит возврат в состояние X11, в котором достается из стека значение. Если значение будет равняться флагу того, что было обработано первое выражение, это означает, что подмножество, из которого был осуществлен выход, принадлежит первому выражению, и ожидается второе выражение подмножества, в которое было осуществлено возвращение при встрече закрывающейся скобки.

10.3. Система схемных формул переходов S3

Рассмотрим пример системы схемных формул переходов S3, которая является преобразованием формулы S2 из рисунка 68.

Как в S1 и S2 есть ветвь автомата, состоящая из состояний X0, X1 и X24, которая обрабатывает безусловный переход, и есть ветвь, которая обрабатывает переход при выполнении условий.

Как и у схемы S2, у схемы S3 есть свои требования.

$$A0 \rightarrow A8$$
$$A8 \rightarrow A7$$
$$A7 \rightarrow p5 \uparrow^{1} A9 * \downarrow^{1} A2$$
$$A2 \rightarrow A9$$
$$A9 \rightarrow A10$$
$$A10 \rightarrow A3$$
$$A3 \rightarrow p6 \uparrow^{2} A8 * \downarrow^{2} A5$$
$$A5 \rightarrow p1 \uparrow^{3} p6 \uparrow^{4} A8 * \downarrow^{4} A5 * \downarrow^{3} A4$$
$$A4 \rightarrow p2 \uparrow^{6} p1 \uparrow^{7} p6 \uparrow^{8} A8 * \downarrow^{8} A5 * \downarrow^{7} A4 * \downarrow^{6} A1$$
$$A1 \rightarrow A6$$
$$A6 \rightarrow p4 \uparrow^{9} p3 \uparrow^{10} Ak * \downarrow^{10} A7 * \downarrow^{9} A2$$

Рисунок 73 – Пример системы схемных формул переходов

Во-первых, после условной вершины должен идти символ «↑» с индексом, состоящим из цифр верхнего регистра, который отвечает за один из выходов условной вершины.

Во-вторых, должен быть символ «↓» с соответствующим индексом, который указывает на альтернативный путь. Индекс каждого альтернативного выхода должен быть уникальным.

Поэтому состояния X4, X5, X6, X7 обрабатывают первое требование, рисунок 74, а состояния X16, X17, X18, X19 обрабатывают второе требование, рисунок 75.

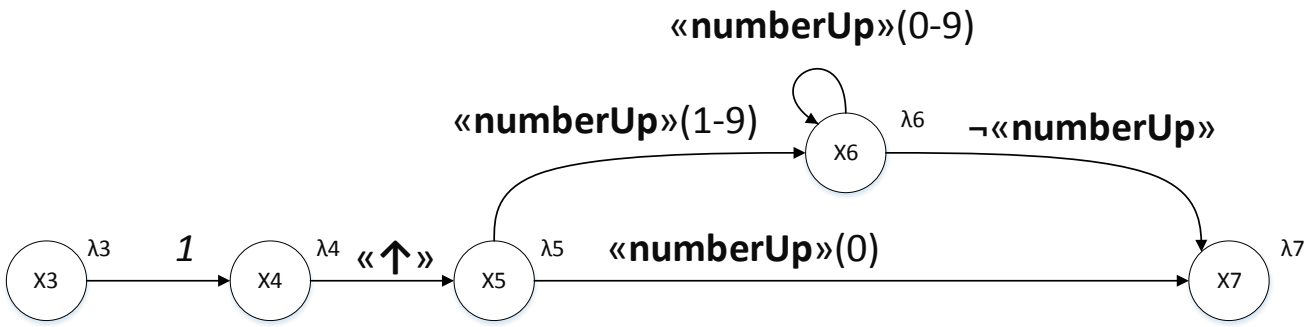


Рисунок 74 – Состояния X3, X4, X5, X6, X7

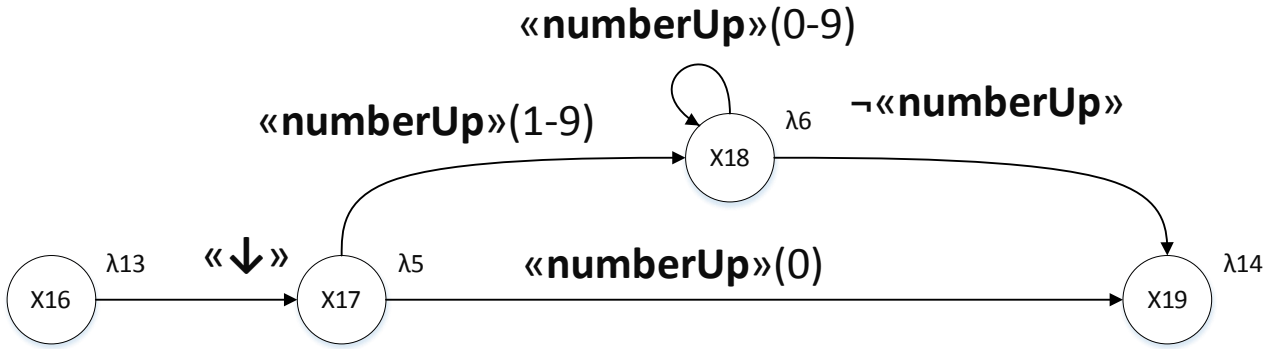


Рисунок 75 – Состояния X16, X17, X18, X19

Для того чтобы проверить уникальность стрелок, необходимо вводить дополнительный список, который будет хранить встреченные стрелки до этого. В состоянии X19 проверяется уникальность стрелки, а в состоянии X22 происходит добавление стрелки в список. При этом необходимо знать, на какую вершину указывает эта стрелка, чтобы потом в случае встречи для условной вершины, которая ведет к этому альтернативному пути, сразу получить вершину назначения. Поэтому в состояниях X21 и X22 определяется вершина, идущая после стрелки, рисунок 76.

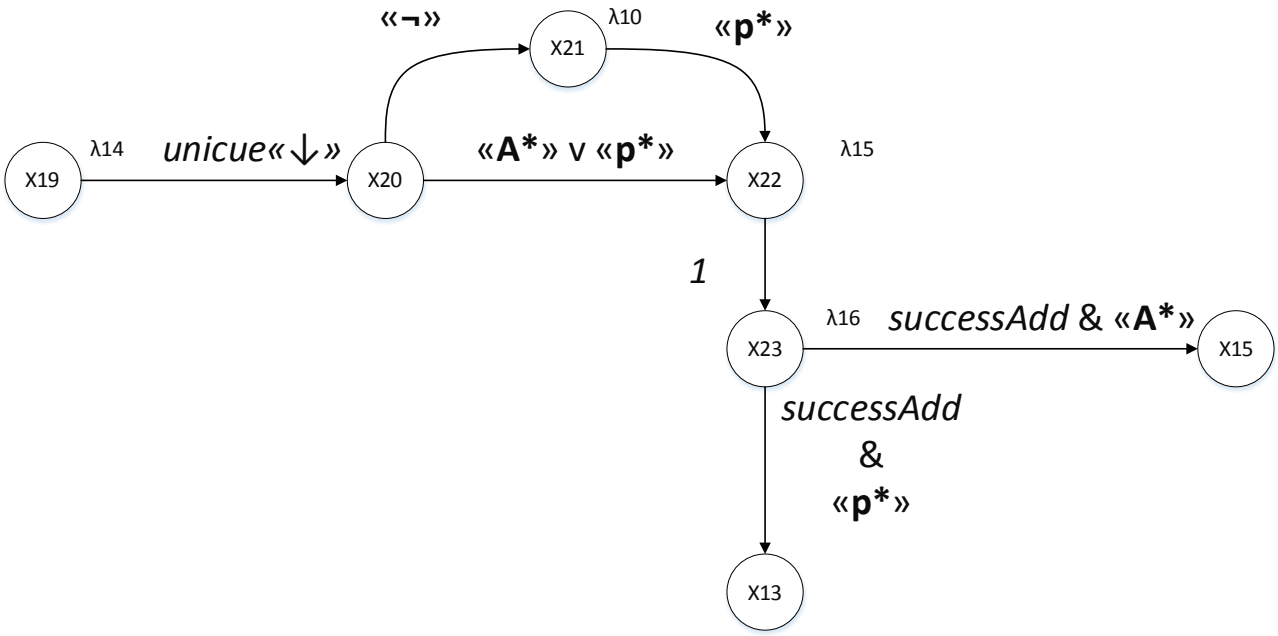


Рисунок 76 – Состояния X19, X20, X21, X22, X23, X13, X15

В список ждущих вершин происходит запись в состоянии X8, в которое происходит переход из состояния X7 из-за того, что не был встречен альтернативный путь. Если же был встречен альтернативный путь, то происходит переход в состояние X9, где происходит попытка добавления перехода из текущей условной вершины в вершину, на которую указывает стрелка в списке встреченных альтернативных путей, рисунок 77.

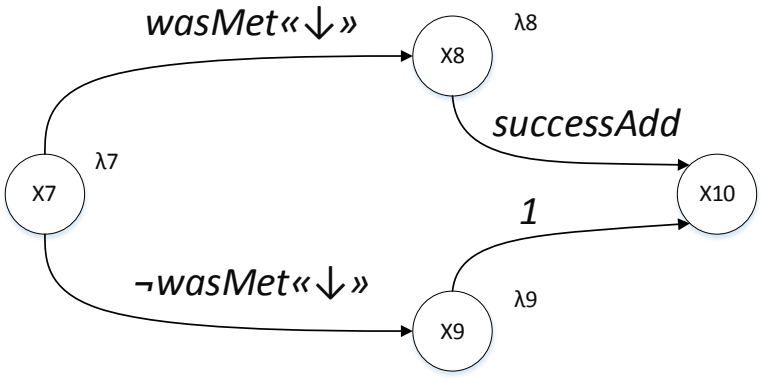


Рисунок 77 – Состояния X7, X8, X9, X10

Еще важным моментом является то, что после операторной вершины должен идти либо конец строки, либо символ «*», так как логическая формула должна содержать только переходы из операторной вершины, которой принадлежит логическая функция, в операторные вершины, которые достижимы через условные вершины. Данный момент обрабатывается в состоянии X15.

Вывод

В данном разделе были разработаны автоматы для преобразования списка лексем в набор переходов. Первым этапом построения служило выявление особенностей структуры логических функций, после чего добавлялись отдельные части в автомата для проверки данных особенностей. Все состояния и все ключевые моменты были описаны в разделе с демонстрацией частей автоматов, а так же с примерами самих формул.

Заключение

Результатом данной выпускной квалификационной работы стала программа, написанная на языке C#.[8] Программа позволяет строить и проверять операторные схемы алгоритмов. Для этого нужно было решить ряд проблем, которые, благодаря знаниям, полученным в ходе обучения, удалось решить.

Во-первых, нужно было решить проблему с предоставлением средств для построения графической схемы алгоритма. Для этого были использованы знания объектно-ориентированного подхода, то есть все элементы графической схемы являются объектами определенных классов. Возможность наследовать помогло отделить контур фигур от других особенностей объектов. Полиморфизм помог переопределять методы дочерних классов, а инкапсуляция помогла скрыть всю сложность функционала классов, оставив на виду только методы, с помощью которых можно взаимодействовать с объектами.[9] В принципе вся программа строится на данном подходе, так как генерация задания и проверка схем так же реализованы в виде классов.

Во-вторых, нужно было решить проблему с проверкой схем алгоритмов, которые представляются в текстовом виде. Мало того, что нужно понять, какую информацию скрывает в себе текст, так еще нужно разобраться, в чем заключается ошибка и пояснить ее. Справиться с этой проблемой помогли знания по работе компиляторов, а именно первых трех этапов: лексического анализа, синтаксического анализа и семантического анализа. То есть в первую очередь текст разбивается на лексемы, которые представляют собой осмысленный набор символов. Далее, как в синтаксическом анализе, при котором проверяется корректность структуры строки кода с формированием

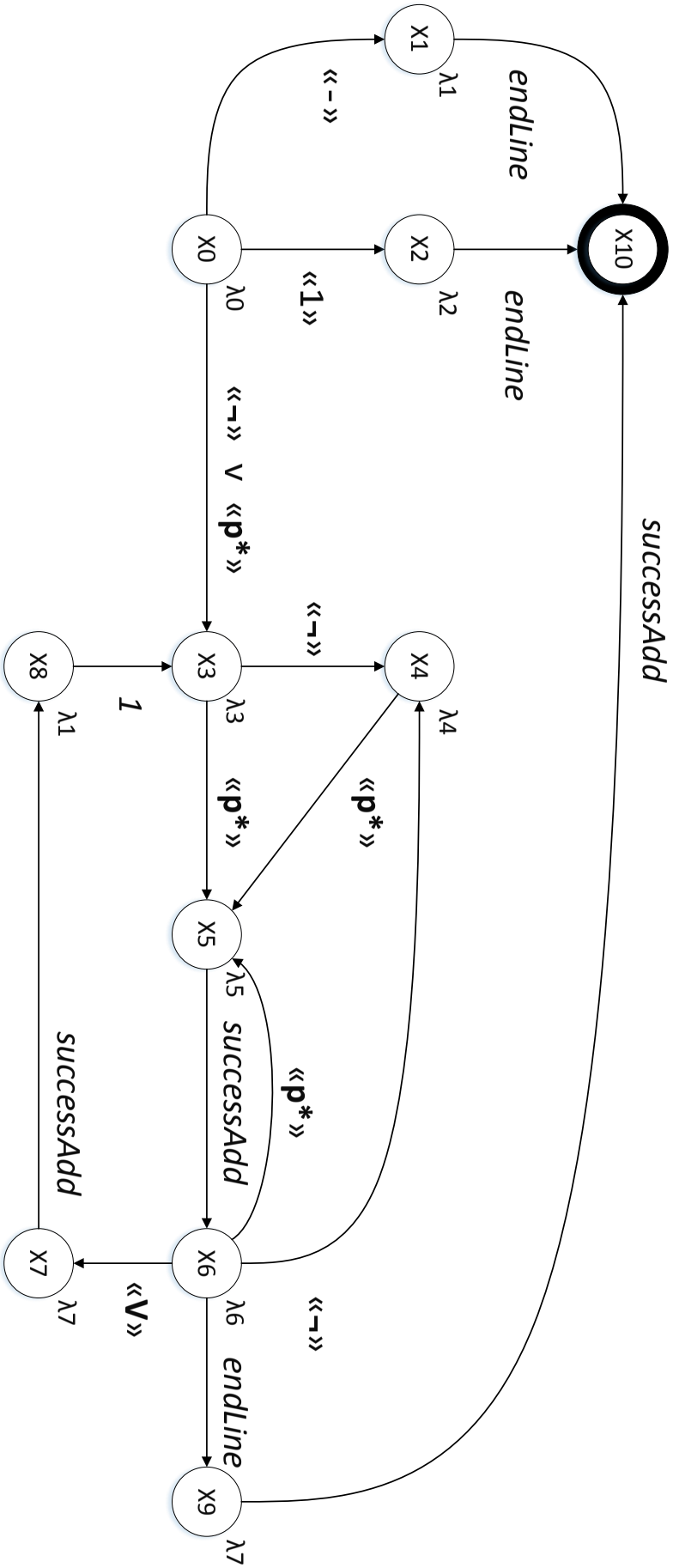
[8] Herbert S. C#: The Complete Reference [Текст] / Herbert S. – Изд-во McGraw-Hill Osborne Media, 2010 год. – 949 с.

[9] Weisfeld M. The Object-Oriented Thought Process [Текст] / Weisfeld M. – Изд-во Addison-Wesley, 2009 год – 304 с.

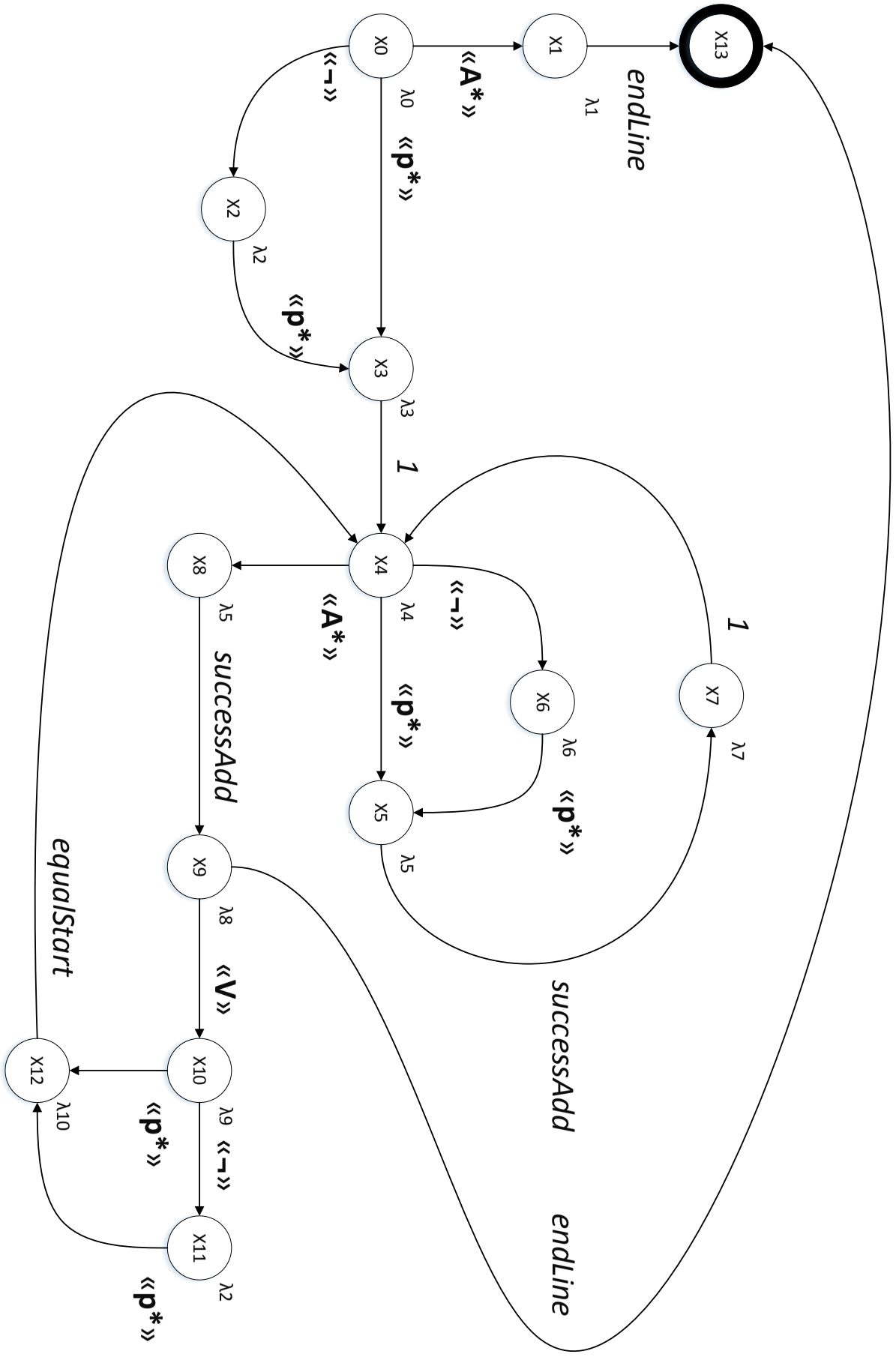
триад, в программе проверяется структура лексем, во время проверки которой происходит еще и построение переходов. И последним этапом является проверка информации, которая скрывает в себе текст.

Заключительная проблема заключалась в выборе способа проверки структуры текста. Данная проблема была решена с помощью навыков проектирования автоматов, полученных на предмете «теории автоматов». Первым этапом было определение списка входных сигналов. Далее выявлялись особенности в структуре каждой из схем, и, в конечном счете, происходило добавление состояний и переходов для обработки данных особенностей.

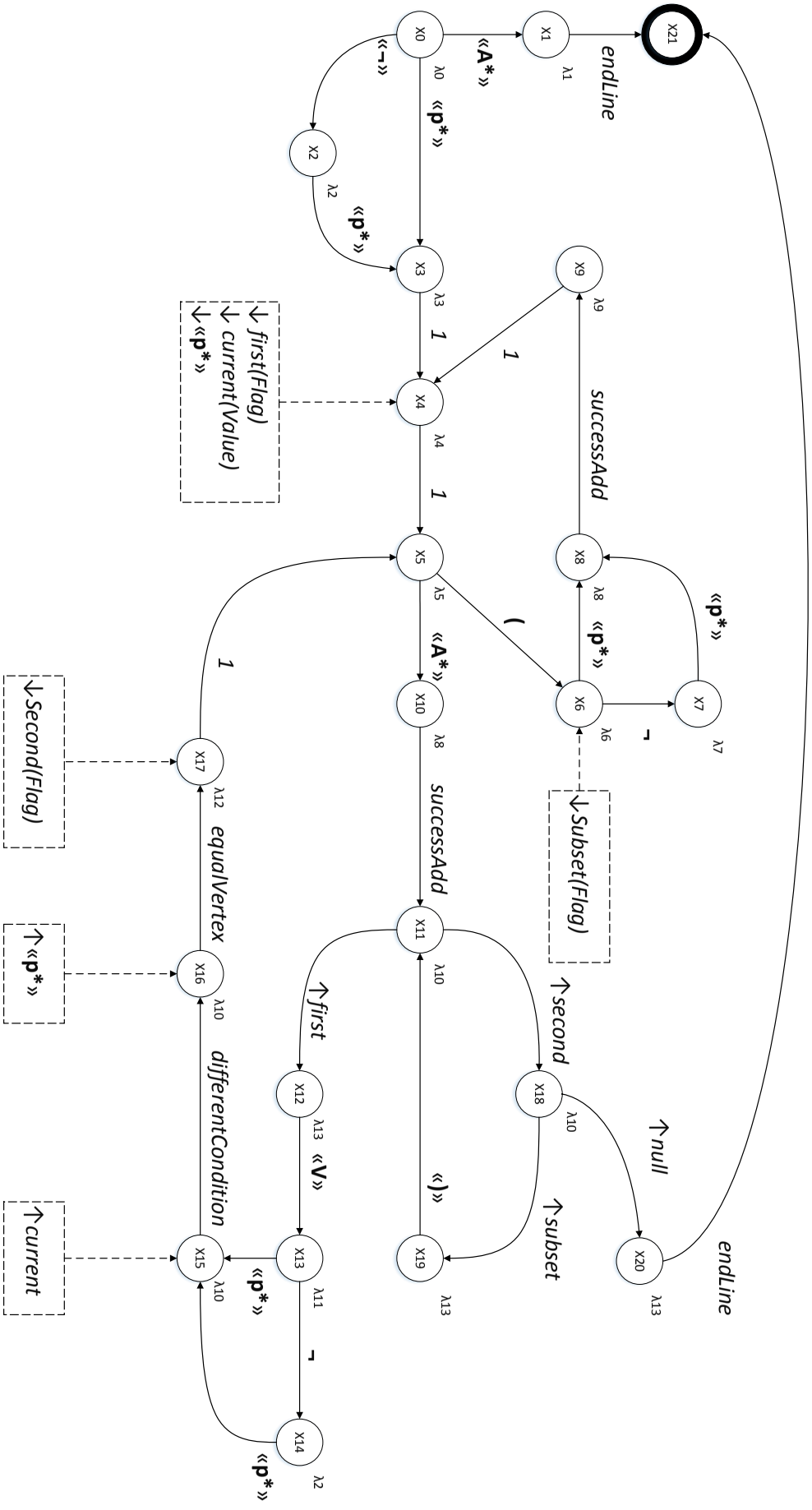
Приложение А. Автомат преобразования ячейки матрицы



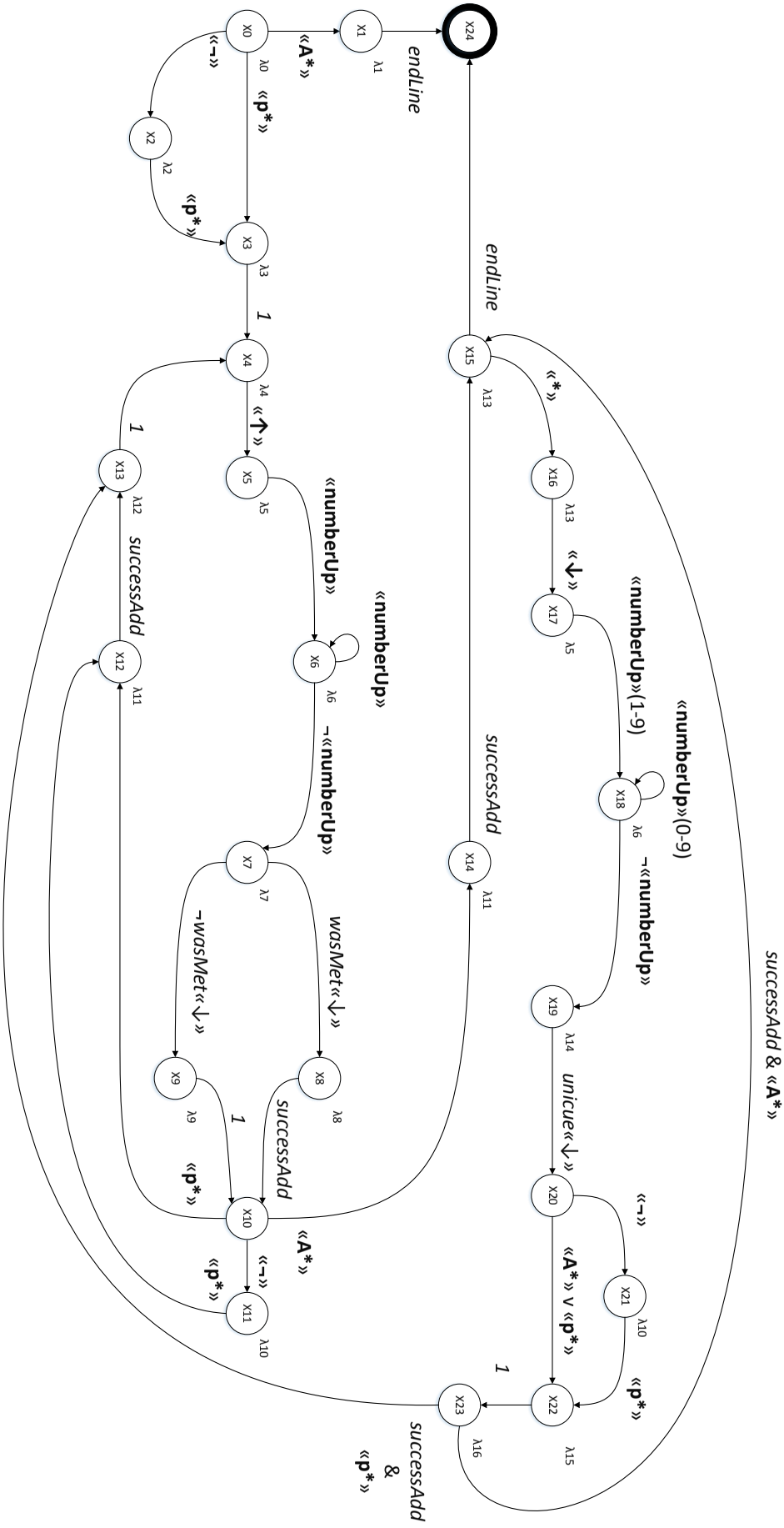
Приложение Б. Автомат преобразования системы формул переходов



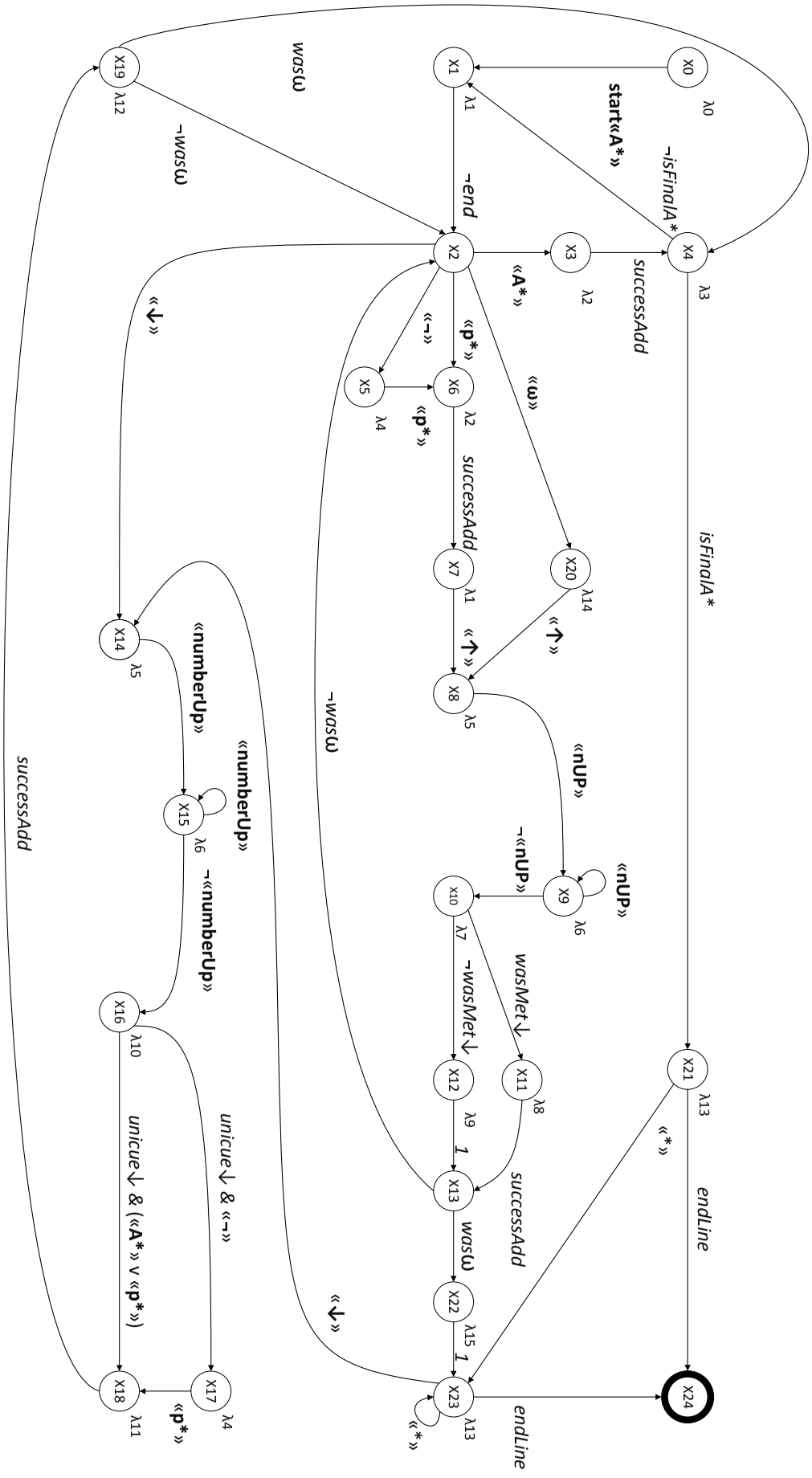
Приложение В. Автомат преобразования системы скобочных формул



Приложение Г. Автомат преобразования системы схемных формул



Приложение Д. Автомат преобразования логической схемы алгоритма



Изм.	Лист	№ докум.	Подпись	Дата	ТПЖА.090301.014 ПЗ	Лист 96

Приложение Е. Входные сигналы автоматов

Входные сигналы:

1	- безусловный переход;
«1»	- текущая лексема является символом безусловного перехода;
«-»	- текущая лексема является обозначением отсутствия перехода;
«p*»	- текущая лексема является условной вершиной алгоритма;
«¬»	- текущая лексема является обозначением операции «НЕ»;
«V»	- текущая лексема является обозначением операции «ИЛИ»;
«(»	- текущая лексема является началом подмножества;
«)»	- текущая лексема является концом подмножества;
«↑»	- текущая лексема является указателем перехода условной вершины;
«↓»	- текущая лексема является указателем альтернативного перехода условной вершины;
«*»	- текущая лексема является концом одного из путей;
«**ω**»	- текущая лексема является символом безусловного перехода;
«NumberUp»	- текущая лексема является цифрой в верхнем регистре;
start«A*»	- текущая лексема является начальной вершиной;
endLine	- флаг конца списка лексем;
successAdd	- флаг успешного добавления перехода/Переходов;
equalStart	- текущая лексема совпадает с startVertex;
differentCondition	- флаг того, что первые элементы двух конъюнкций в одном подмножестве имеют разные условия;
equalVertex	- флаг того, что первые элементы двух конъюнкций в одном подмножестве равны;
↑null	- стек пуст.
unicue<«↓»	- флаг уникальности «↓» с соответствующим индексом;
wasMet<«↓»	- флаг того, что «↓» с соответствующим индексом уже была встречена до этого;
isFinalA*	- флаг того, что текущая лексема является конечной вершиной.
*was*ω	- Флаг того, что был встречен символ безусловного перехода.

Приложение Ж. Выходные сигналы автомата преобразования ячейки матрицы

Выходные сигналы:

λ0: *indexLexem* = 0;

λ1: *indexLexem* = *indexLexem* + 1

λ2: Добавить переход из *vertexFrom* в *vertexTo*
indexLexem = *indexLexem* + 1

λ3: *current* = true;
future = true;
currentVertex = *vertexFrom*

λ4: *future* = *false*
indexLexem = *indexLexem* + 1

λ5: Попытка добавления перехода из *currentVertex* В *currentLexem* под улсовием *current*

λ6: current = future
future = true
currentVertex = currentLexem
indexLexem = *indexLexem* + 1

λ7: Попытка добавления перехода из *currentVertex* В *vertexTo* под улсовием *current*

Расшифровка:
currentLexem – Текущая лексема.
indexLexem – Индекс текущей лексемы в списке лексем.
Current – Логические условие текущей вершины (… _ p …)*
Future – Логическое условие следующей вершины (… p _p* …)*
currentVertex – Текущая вершина
vertexFrom - Операторная вершина, принадлежащая строке матрицы.
vertexTo - Операторная вершина, принадлежащая столбцу матрицы.

Приложение 3. Выходные сигналы автомата преобразования системы формул переходов

Выходные сигналы:

λ0: *current* = true;
future = true;
indexLexem = 0;

λ1: *startVertex* = *currentLexem*
indexLexem = *indexLexem* + 1

λ2: *current* = false
indexLexem = *indexLexem* + 1

λ3: *startVertex* = *currentLexem*

λ4: *currentVertex* = *currentLexem*
indexLexem = *indexLexem* + 1

λ5: Попытка добавления перехода из *currentVertex* В *currentLexem* под условием *current*.

λ6: *future* = false
indexLexem = *indexLexem* + 1

λ7: *current* = *future*;
future = true;

λ8: *current* = true;
future = true;
indexLexem = *indexLexem* + 1

λ9: *indexLexem* = *indexLexem* + 1

λ10: Проверка совпадения *currentLexem* и *startVertex*

Расшифровка:

currentLexem	*– Текущая лексема.*	
indexLexem	*– Индекс текущей лексемы в списке лексем.*	
Current	*– Логические условие текущей вершины*	*(… _ p* …)*
Future	*– Логическое условие следующей вершины*	*(… p* _p* …)*
startVertex	*– Первая вершина, идущая после операторной вершины*	*(A → startVertex)*
currentVertex	*– Текущая вершина*	

Приложение И. Выходные сигналы автомата преобразования системы скобочных формул

Выходные сигналы:

λ0: *current* = true;
future = true;
indexLexem = 0;

λ1: *startVertex* = *currentLexem*
indexLexem = *indexLexem* + 1

λ2: *current* = false
indexLexem = *indexLexem* + 1

λ3: *startVertex* = *currentLexem*

λ4: PUSH(«p*»)
PUSH(*current*)
PUSH(first)

λ5: *currentVertex* = *currentLexem*
indexLexem = *indexLexem* + 1

λ6: PUSH(subset)
indexLexem = *indexLexem* + 1

λ7: *future* = false
indexLexem = *indexLexem* + 1

λ8: Попытка добавления перехода из *currentVertex* В *currentLexem* под условием c*urrent*.

λ9: current = future;
future = true;

λ10: POP()

λ11: *current* = true;
future = true;
indexLexem = *indexLexem* + 1

λ12: PUSH(second)

λ13: *indexLexem* = *indexLexem* + 1

Расшифровка:

currentLexem – *Текущая лексема.*
indexLexem – *Индекс текущей лексемы в списке лексем.*
Current – *Логические условие текущей вершины* *(… _ p* …)*
Future – *Логическое условие следующей вершины* *(… p* _p* …)*
startVertex – *Первая вершина, идущая после операторной вершины* *(A → startVertex)*
currentVertex – *Текущая вершина*

Приложение Й. Выходные сигналы автомата преобразования системы схемных формул

Выходные сигналы:

λ0: *current* = true;
future = true;
indexLexem = 0;

λ1: *startVertex* = *currentLexem*
indexLexem = *indexLexem* + 1

λ2: *current* = false
indexLexem = *indexLexem* + 1

λ3: *startVertex* = *currentLexem*

λ4: *currentVertex* = *currentLexem*
indexLexem = *indexLexem* + 1

λ5: Очистить *bufferNumberUp*
indexLexem = *indexLexem* + 1

λ6: *bufferNumberUp* = *bufferNumberUp* + *currentLexem*
indexLexem = *indexLexem* + 1

λ7: Поиск стрелки «↓» с индексом *bufferNumberUp* в списке уже встреченных.

λ8: Попытка добавления перехода из *currentVertex* в вершину, которую указывает стрелка «↓» с индекосм *bufferNumberUp* в списке уже встреченных под условием ¬*current*.

λ9: Вершину *currentVertex* добавить в список ждущих вершин как вершину, ждущую стрелку «↓» с индекосм *bufferNumberUp*

λ10: *future* = false
indexLexem = *indexLexem* + 1

λ11: Попытка добавления перехода из *currentVertex* В *currentLexem* под условием *current*.

λ12: *current* = *future*;
future = true;

λ13: *indexLexem* = *indexLexem* + 1

λ14: Проверка уникальности индекса «↓» с индексом *bufferNumberUp* в списке уже встреченных.

λ15: Добавить индекс «↓» С индексом *bufferNumberUp* в список уже встреченных.

λ16: Попытка добавления переходов в current*Lexem* Из тех вершин, которые ожидали «↓» С индекосм *bufferNumberUp*

Расшифровка:
currentLexem – Текущая лексема.
indexLexem – Индекс текущей лексемы в списке лексем.
Current – Логические условие текущей вершины *(… _ p* …)*
Future – Логическое условие следующей вершины *(… p* _p* …)*
startVertex – Первая вершина, идущая после операторной вершины *(A → startVertex)*
currentVertex – Текущая вершина
bufferNumber – Буфер прочитанных цифр верхнего регистра.

Приложение К. Выходные сигналы автомата преобразования логической схемы алгоритма

Выходные сигналы:

λ0: *current* = true;
future = true;
indexLexem = 0;

λ1: *currentVertex* = *currentLexem*
current = future
future = true
indexLexem = *indexLexem* + 1
was»ω» = false

λ2: Попытка добавления перехода из *currentVertex* В *currentLexem*.

λ3: Проверка, является ли *currentLexem* Конечным оператором.
was»ω» = true

λ4: *future* = false
indexLexem = *indexLexem* + 1

λ5: Очистить *bufferNumberUp*
indexLexem = *indexLexem* + 1

λ6: *bufferNumberUp* = *bufferNumberUp* + *currentLexem*
indexLexem = *indexLexem* + 1

λ7: Поиск стрелки «↓» с индексом *bufferNumberUp* в списке уже встреченных.

λ8: Попытка добавления перехода из *currentVertex* в вершину, которую указывает стрелка «↓» с индекосм *bufferNumberUp* в списке уже встреченных под условием ¬*current*.

λ9: Вершину *currentVertex* добавить в список ждущих вершин как вершину, ждущую стрелку «↓» с индекосм *bufferNumberUp*

λ10: Проверка уникальности индекса «↓» с индексом *bufferNumberUp* в списке уже встреченных.

λ11: Попытка добавления переходов в current*Lexem* Из тех вершин, которые ожидали «↓» С индекосм *bufferNumberUp*

λ12: Добавить индекс «↓» С индексом *bufferNumberUp* в список уже встреченных.

λ13: *indexLexem* = *indexLexem* + 1

λ14: *was»ω»* = true
λ14: *Current* = ¬*current*

λ15: *indexLexem* = *indexLexem* - 1

Расшифровка:
currentLexem – Текущая лексема.
indexLexem – Индекс текущей лексемы в списке лексем.
Current – Логические условие текущей вершины *(… _ p* …)*
Future – Логическое условие следующей вершины *(… p*_p* …)*
currentVertex – Текущая вершина
bufferNumber – Буфер прочитанных цифр верхнего регистра.

Приложение Л. (справочное) Библиографический список

1. Матвеев В.Д. Теория алгоритмов [Текст] / Матвеев В.Д., Матвеева Л.И – Киров: Изд-во ВятГУ, 2011 год. – 40 с.

2. Herbert S. C#: The Complete Reference [Текст] / Herbert S. – Изд-во McGraw-Hill Osborne Media, 2010 год. – 949 с.

3. System.Drawing Namespace [Электронный ресурс]. URL: https://docs.microsoft.com

4. Aho A. Compilers: Principles, Technizues, and Tools [Текст] /Aho. A.,Lam M. Ulman J. – Изд-во Addison-Wesley, 2006 год – 1184 с.

5. Ожиганов А.А. Теория автоматов [Текст] / Ожиганов А.А. – Санкт-Петербург: Изд-во ИТМО, 2013 год – 84 с.

6. Мельцов В. Ю. Синтез микропрограммных управляющих автоматов [Текст] / Мельцов В. Ю., Фадеева Т. Р. – Киров: ВятГУ, 2000 год. – 56 с.

7. Weisfeld M. The Object-Oriented Thought Process [Текст] / Weisfeld M. – Изд-во Addison-Wesley, 2009 год – 304 с.

9. Долженкова М.Л. Технологии программирования [Текст] / Долженкова М.Л., Караваева О.В. – Киров: Изд-во ВятГУ, 2013 год – 127 с.

9. 90 рекомендаций по стилю написания программ C++ [Электронный ресурс]. URL: https://habr.com/ru/post/172091/

10. 10 приемов, разрушающих хрупкую красоту кода [Электронный ресурс]. URL: https://habr.com/ru/post/59570/