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

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ***

***НА ТЕМУ:***

|  |
| --- |
| Метод верификации графовой информационной модели |

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Студент | ИУ7-82Б |  |  |  | Т.М. Чуйкова |

(Группа) (Подпись, дата) (И.О.Фамилия)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Руководитель ВКР |  |  |  | Г.А. Щетинин |

(Подпись, дата) (И.О.Фамилия)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Нормоконтролер |  |  |  | Д.Ю. Мальцева |

(Подпись, дата) (И.О.Фамилия)

*2022 г.*

**РЕФЕРАТ**

Расчетно-пояснительная записка содержит 65 страниц, 18 рисунков, 14 листингов, 16 источников, 4 приложения.

ГРАФОВАЯ ИНФОРМАЦИОННАЯ МОДЕЛЬ, ВЕРИФИКАЦИЯ, ЗАДАЧА ВЕРИФИКАЦИИ МОДЕЛЕЙ, КЛАССИФИКАЦИЯ МЕТОДОВ ВЕРИФИКАЦИИ.

Объектом разработки является метод верификации графовой информационной модели.

Цель выпускной квалификационной работы – разработать и реализовать метод верификации информационной модели, описанную как граф.

Для достижения поставленной цели были решены следующие задачи:

- выполнен обзор основных проблем в области информационного моделирования;

- рассмотрены вариации задач верификации ПО;

- проанализировать существующие способы верификации ПО и их применимость к информационным моделям;

- разработаны метод верификации графовой информационной модели;

- реализованы предложенный метод и проверить его работу.

СОДЕРЖАНИЕ

стр.

[Определения 8](#_Toc106169640)

[Обозначения и сокращения 9](#_Toc106169641)

[Введение 10](#_Toc106169642)

[1 Аналитический раздел 12](#_Toc106169643)

[1.1 Использование информационной модели 12](#_Toc106169644)

[1.2 Вариации задач о верификации 15](#_Toc106169645)

[1.3 Анализ методов верификации моделей и алгоритмов 15](#_Toc106169646)

[1.3.1 Априорная верификация 15](#_Toc106169647)

[1.3.2 Апостериорная верификация 19](#_Toc106169648)

[1.4 Сравнительные характеристики методов верификации 20](#_Toc106169649)

[1.5 Вывод 21](#_Toc106169650)

[2 Конструкторский раздел 23](#_Toc106169651)

[2.1 Формализация задачи 23](#_Toc106169652)

[2.1.1 Требования к информационной модели 23](#_Toc106169653)

[2.1.2 Требования к методу верификации 23](#_Toc106169654)

[2.1.3 *IDF0* 23](#_Toc106169655)

[2.2 *Use-case* диаграмма 25](#_Toc106169656)

[2.3 Предполагаемый метод 26](#_Toc106169657)

[2.4 Вывод 28](#_Toc106169658)

[3 Технологический раздел 30](#_Toc106169659)

[3.1 Выбор и обоснование языка программирования и среды разработки 30](#_Toc106169660)

[3.2 Собственная реализация 30](#_Toc106169661)

[3.2.1 Представление информационной модели 30](#_Toc106169662)

[3.2.2 Булева матрица переходов 34](#_Toc106169663)

[3.2.3 Алгоритм верификации 37](#_Toc106169664)

[3.2.4 Алгоритм вывода графовой модели 40](#_Toc106169665)

[3.3 Исследование временных характеристик 42](#_Toc106169666)

[3.4 Пример работы программы 44](#_Toc106169667)

[3.4 Вывод 46](#_Toc106169668)

[4 Исследовательский раздел 47](#_Toc106169669)

[4.1 Исследование скорости работы алгоритма для графа с безусловными переходами 47](#_Toc106169670)

[4.2 Исследование скорости работы алгоритма для графа с условными переходами 48](#_Toc106169671)

[4.3 Вывод 49](#_Toc106169672)

[Заключение 50](#_Toc106169673)

[Список использованных источников 51](#_Toc106169674)

[Приложение А. Построение ориантированного графа 54](#_Toc106169675)

[Приложение Б. Верификация ориентированного графа, представленного матрицей переходов 56](#_Toc106169676)

[Приложение В. Верификация ориентированного графа, представленного списками смежности 60](#_Toc106169677)

[Приложение Г. Исследование временных характеристик 64](#_Toc106169678)

# ОПРЕДЕЛЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие термины с соответствующими определениями:

Артефакт программного обеспечения- исследуемый участок кода программы.

Верификация — оценка результатов процесса с целью гарантии корректности и непротиворечивости в отношении входов и стандартов, существующих для данного процесса.

Информационная модель — [модель](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C) объекта, представленная в виде [информации](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D1%8F), описывающей существенные для данного рассмотрения [параметры](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80) и переменные величины объекта, связи между ними, входы и выходы объекта и позволяющая путём подачи на модель информации об изменениях входных величин моделировать возможные состояния объекта [1].

# ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

АСУ – автоматизированные системы управления;

БДР – бинарная диаграмма решений;

ДПС – диаграмма переходов состояний;

ПО – программное обеспечение;

СЭД – система электронного документооборота;

BPMN – Business Process Management Notation;

UML – Unified Modeling Language.

# ВВЕДЕНИЕ

Развитие информационного общества тесно связано с повсеместным внедрением вычислительных систем во все сферы деятельности. Одним из эффективных инструментов планирования при разработке подобных систем является информационная модель. Данная модель представляет собой структурированную информацию о сущности, которую можно в дальнейшем представить в виде параметров и переменных и их взаимосвязи.

Отличительной особенностью моделирования сложных систем является их многофункциональность и многообразие способов использования. Создание информационной модели для подобных проектов позволит прояснить цели и задачи проекта и сформулировать к нему необходимые требования.

Повышающийся уровень сложность программного обеспечения (ПО) вызывает рост числа ошибок при разработке, а увеличение количества и критичности выполняемых их функций вызывает рост ущерба от этих ошибок. Построение информационных систем определенного уровня сложности не может проходить безошибочно, возрастающая при этом сложность проектирования предоставляет все больше возможностей для ошибок и затрудняет их своевременное обнаружение. Для обеспечения адекватности и надежности работы сложных систем необходимо иметь возможность их верификации уже на ранних этапах разработки.

В связи с этим верификация сложных информационных моделей является актуальной задачей.

Целью данной выпускной квалификационной работы является разработка метода верификации графовой информационной модели. При разработке данного метода были поставлены следующие задачи:

- выполнить обзор основных проблем в области информационного моделирования;

- рассмотреть вариации задач верификации ПО;

- проанализировать существующие способы верификации ПО и их применимость к информационным моделям;

- разработать метод верификации графовой информационной модели;

- реализовать предложенный метод и проверить его работу.

## 1 Аналитический раздел

### 1.1 Использование информационной модели

При разработке ПО для встроенных систем для моделирования ее работы часто используются информационные модели, на основе которых формируются основные требования к проекту и отображаются в частном техническом задании [2].

Применение информационной модели позволяет доступно охарактеризовать связь между данными, их структурой и процедурой обработки, определяя сущность описанного объекта. Данная модель описывает состояния и свойства объектов и их взаимодействия.

Общая классификация строится исходя из определения [3]:

- в зависимости от охватываемого момента времени делятся на динамические и статические;

- вид разделения по способу описания представлен на рисунке 1;

- в зависимости от целей построения делятся на графовые, графические, идеографические, текстовые, алгоритмические, табличные.

Существуют следующие способы описания структуры информационной модели [4]:

- описательная – модель описывается на естественном языке, имеет произвольную структуру и служит для удобства восприятия информации конкретной группы людей;

- формальная – модель создается на формальном языке. Может иметь форму графа, таблицы, формулы и другие виды представления структурной информации;

- хроматическая (цветовая) – модель описывает способ представления цветов, создается с применением естественного языка семантики цветовых концептов.

Перед разработкой информационной модели проводится анализ предметной области, выделение ключевых сущностей и соответствующей им наиболее ценной для исследования информации. Разработка модели заключается в анализе полученных данных, их структурировании и, в зависимости от поставленной задачи, построении необходимой модели. Завершающим этапом при проектировании является поиск несостыковок и логических ошибок и их устранение [5].



Рисунок 1 – Классификация информационных моделей по способу описания

Использование информационных моделей для описания сложных систем при проектировании ПО [6] позволяет смоделировать работу будущей системы и своевременно найти возможные несостыковки.

Примером проектирования с использованием информационного моделирования является описание функционала работы предприятия для автоматизированных систем управления (АСУ).

Часто для представления данных систем используются схемы потоков информации, *UML*-диаграммы и диаграммы переходов состояний (ДПС), пример которой представлен ниже на рисунке 2.

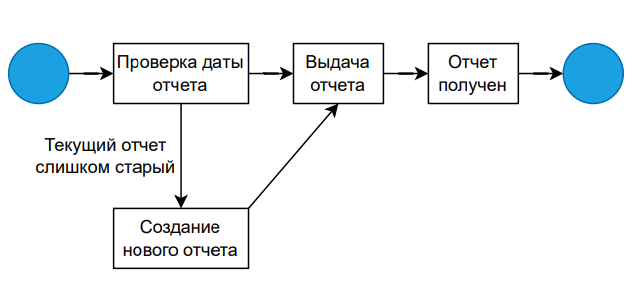


Рисунок 2 – Вид ДПС

Представленная на рисунке 2 диаграмма является информационной моделью сущности отчета и описана в виде ориентированного графа. Данная модель является небольшой, однако при проектировании сложных систем количество вершин и переходов может составлять несколько десятков, что существенно усложнит проверку поиска несостыковок и логических ошибок экспертами.

Ошибки на этапе построения информационной модели неочевидны, они могут долгое время накапливаться и, в конечном счете, привести к неисправности работы системы.

Возникает необходимость верификации информационных моделей подобно верификации программного обеспечения.

### 1.2 Вариации задач о верификации

При проектировании ПО для гарантии корректности разрабатываемой модели одним из основных этапов является ее верификация. Данный процесс предназначен для проверки адекватности реализуемой системы на начальных этапах разработки. Целью проверки является выявление конструктивных ошибок, связанных с противоречащими требованиям в формальном описании системы.

Можно выделить следующие задачи [7], решаемые при верификации ПО в процессе его разработки:

- выявление дефектов различных артефактов разработки ПО (требований, проектных решений, документации или кода);

- выявление наиболее критичных и наиболее подверженных ошибкам частей создаваемой или сопровождаемой системы;

- контроль и оценка качества ПО во всех его аспектах;

- предоставление всем заинтересованным лицам (руководителям, заказчикам, пользователям и пр.) информации о текущем состоянии проекта и характеристиках его результатов [8];

- предоставление руководству проекта и разработчикам информации для планирования дальнейших работ, а также для принятия решений о продолжении проекта, его прекращении или передаче результатов заказчику.

На этапе построения информационной модели из всех перечисленных возможно только задача выявление дефектов в соответствии с требованиями.

1.3 Анализ методов верификации моделей и алгоритмов

1.3.1 Априорная верификация

При помощи методов априорной верификации осуществляется проверка всех формальных требований к модели [9]. Данная проверка осуществляется при специальных условиях с заранее определенными средствами измерений. На рисунке 3 отражена классификация данных методов.

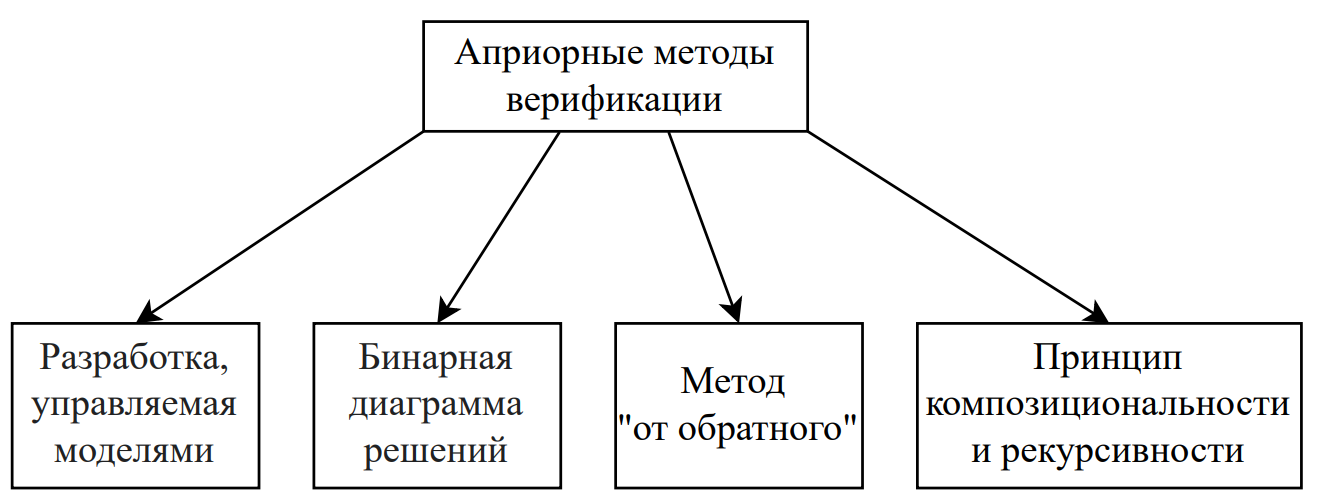


Рисунок 3 – Классификация основных априорных методов верификации

Разработка, управляемая моделями (*Model Driven Development*) – метод верификации, при котором модели выполняют роль основных артефактов разработки. Данный метод приведен в работе [10]. Проектирование при этом включает в себя следующие этапы:

- спецификация свойств;

- разработка основной модели информационной системы;

- разработка тестов системы;

- верификация модели;

- имитационное моделирование;

- окончательное оформление проекта.

При данном методе информационная система приобретает верификацию на проектной стадии. В этом основное отличие данного метода от остальных, которые верифицируются непосредственно при использовании информационной системы или после реализации.

Бинарная диаграмма решений (БДР) (*Binary Decision Diagrams*) – метод верификации, представленный направленным ациклическим графом, описанным булевой функцией от нескольких переменных. Данным граф образован узлами решений (переменными ) и двумя терминальными узлами.

У каждого узла решений есть два дочерних узла (потомка). Каждый терминальный узел определяется одним из двух значений булевой функции (0 или 1).

Пример применения булевых функций представлен в работе [11]. Вид БДР представлен на рисунке 4.

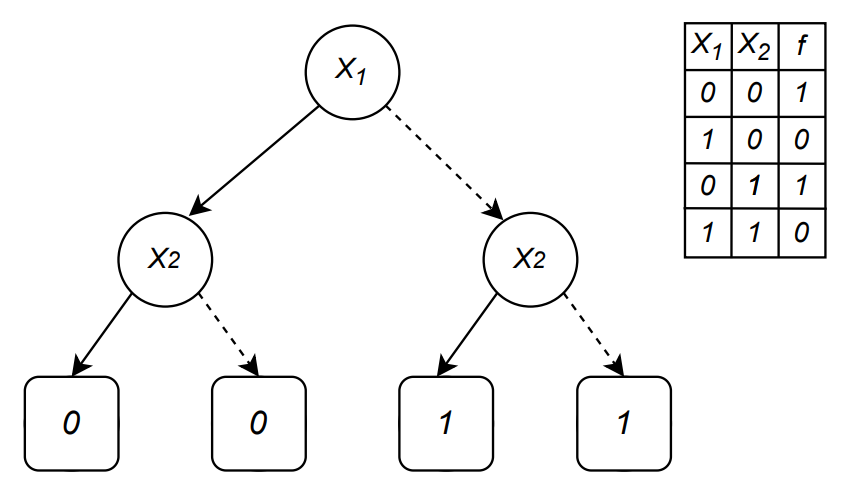


Рисунок 4 – Вид БДР

Недостатком метода считается его неприменимость к системам, в которых функции не могут быть интерпретируемы как булевы.

Достоинством данного метода является то, что его применение в алгоритмах верификации позволяет существенно усложнить верифицируемую систему. При этом в информационной системе увеличивается число состояний структуры Крипке (достигает ).

Метод верификации “от обратного” обычно сводится к определению тупиковых состояний информационной системы. При этом не производится проверка всех возможных состояний.

Пример модели состояний с одной тупиковой вершиной представлен на рисунке 5. Тупиковая вершина заштрихована.

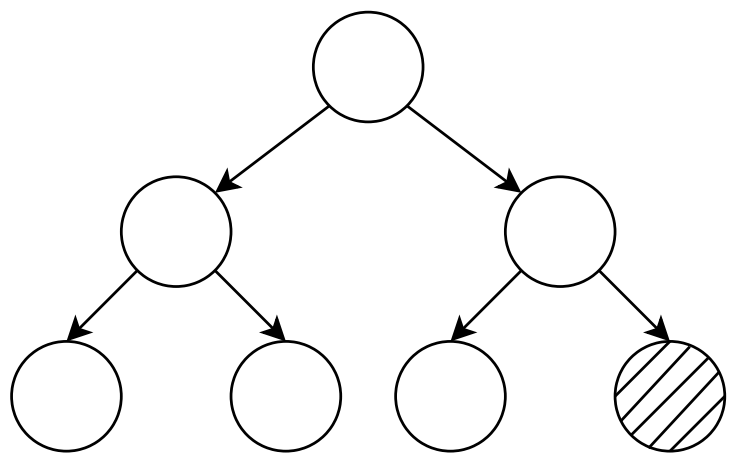


Рисунок 5 – Вид модели состояний с тупиковой вершиной

Принцип композициональности и рекурсивности [10] – принцип, при котором итоговое функционирование информационной системы определяется исходя из функционирование ее составных частей. Вид данного метода приведен на рисунке 6. Рекурсивная структура в данном случае состоит из однообразных модулей, связанных между собой [11].

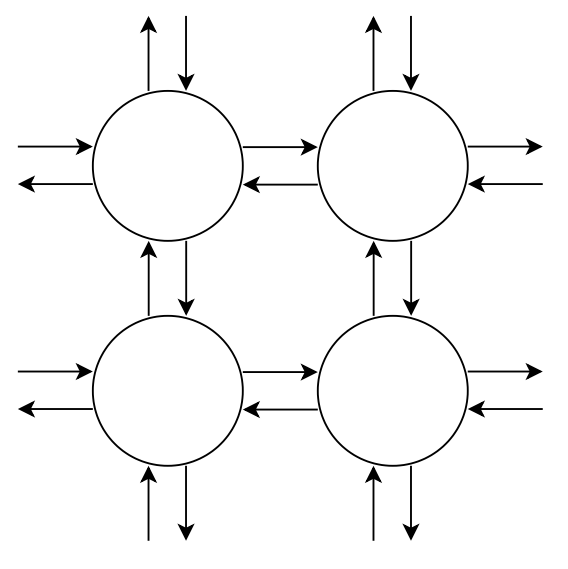


Рисунок 6 – Вид компонентной рекурсивной структуры

Данная идея имеет применение в теории фракталов. Теория фракталов описана в источнике [12]. Фракталом можно считать любой предмет, который обладает одним из следующих свойств:

- нетривиальность структуры фрактала;

- самоподобие (приближенное) фрактала;

- фрактал обладает дробной метрической размерностью;

- фрактал обладает рекурсивной структурой.

1.3.2 Апостериорная верификация

Методы апостериорной верификации базируются на сведениях об информационных системах и основных требованиях к ним.

Одним из методов апостериорной верификации является метод проверки моделей (*model checking*) [13], в основе которого лежит определение тупиковых ситуаций.

В качестве модели применяется модель Крипке, которая описывается следующим соотношением:

|  |  |
| --- | --- |
|  | (1) |

где – множество состояний;

– множество начальных состояний;

– отношение переходов (;

– функция разметки ().

Данный метод включает следующие этапы [14]:

- создание формальной модели информационной системы;

- создание формального языка требований;

- проверка формальной модели с применением разработанного языка требований.

Одним из недостатков данного метода верификации [15] является огромное количество проверок и расчетов.

Вид метода проверки моделей представлен на рисунке 7. На данном рисунке – начальное состояние модели; – состояния 1 и 2 соответственно. Переход → отражает нормальное функционирование, переход → ведет к тупиковой ситуации – ошибочное функционирование.

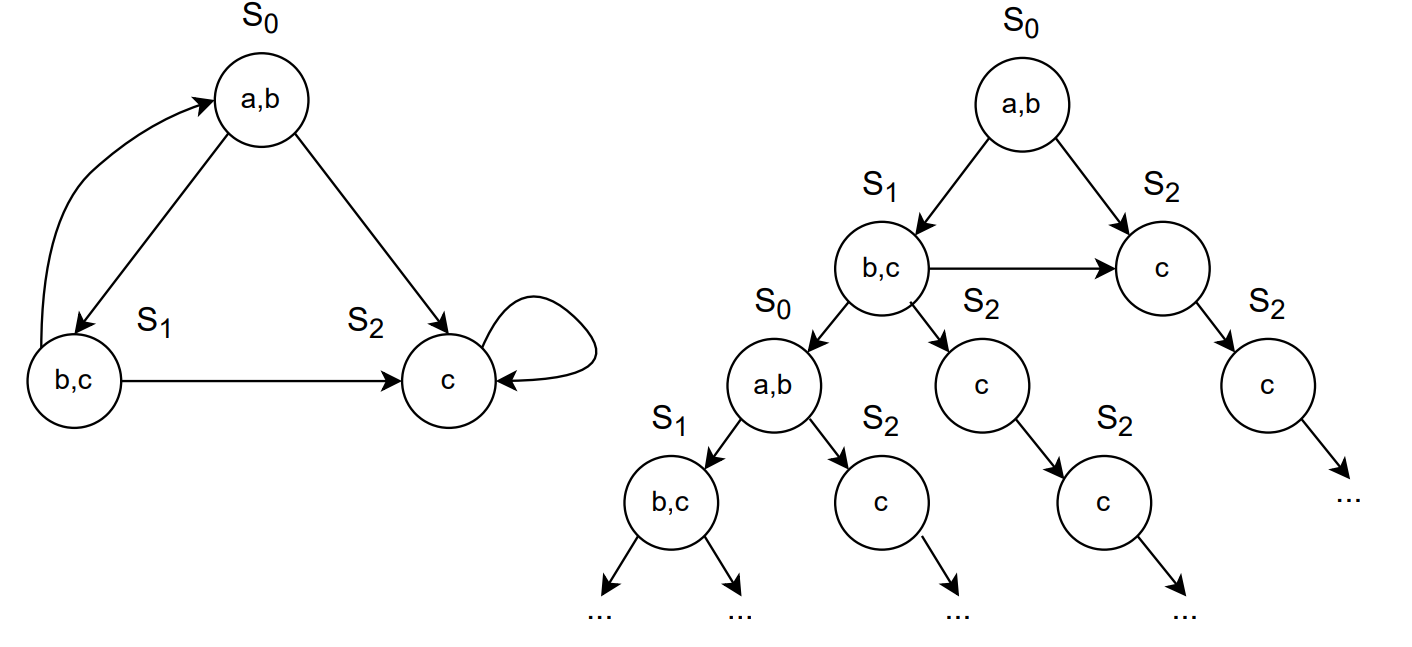


Рисунок 7 – Вид схемы проверки

1.4 Сравнительные характеристики методов верификации

Методы верификации ПО в зависимости от степени достоверности можно поделить на эмпирические (с использованием экспертизы), формальные (применяющие математический аппарат) и динамические (определяющие правильность работы при непосредственном запуске).

При этом выделяют следующие показатели, использующиеся при верификации [16]:

- вид метода;

- уровень автоматизации;

- уровень функциональной пригодности;

- точность;

- типы обнаруживаемых ошибок;

- эффективность;

- область применимости;

- время выполнения;

- способ достижения результата.

Сравнение формальных методов верификации по данным критериям представлены в таблице 1.

Таблица 1 – Сравнение формальных методов верификации

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Вид метода | Проверка моделей | Дедуктивный анализ | Проверка согласований | Абстрактная интерпретация |
| Продуктивность | Высокая | Средняя | Средняя | Высокая |
| Время выполнения | Среднее | Высокая | Малое | Высокая |
| Способ достижения результата | Информационная модель | Анализ пред и постусловий | Программный код | Абстрактный семантический домен |
| Уровень автоматизации | Автоматизированные | | | |
| Точность | Высокая | | | |
| Типы ошибок | Логические ошибки и сценарии, приводящие к недокументированному поведению | | | |

### 1.5 Вывод

Проведен обзор основных проблем в области информационного моделирования и анализ основных априорных и апостериорных методов верификации ПО. Приведено сравнение формальных методов верификации ПО.

По результатам проведенного анализа принято решение о разработке метода верификации с использованием подхода «от обратного». Данный подход позволяет избежать проверки всех возможных состояний системы и сводится к исследованию тупиковых состояний информационной системы.

## 2 Конструкторский раздел

### 2.1 Формализация задачи

### 2.1.1 Требования к информационной модели

В рамках разрабатываемого метода информационная модель должна обладать следующими свойствами:

- сущности анализируемой системы обладают однообразным набором свойств;

- структурная модель описываемой системы может быть представлена ориентированным графом;

- граф каждой сущности задается списком переходов (пример списка переходов: вершина\_1 вершина\_2; вершина\_2 вершина\_3; вершина\_1 вершина\_3; …) и списком помеченных вершин.

### 2.1.2 Требования к методу верификации

Для адекватности результата работы программы к реализуемому методу верификации предъявляется следующий список требований:

- время работы конечно;

- метод должен работать с ориентированными графами (необходимо присвоить направления ребрам графа);

- метод должен работать с динамическими системами (необходимо обозначить временную зависимость каждого элемента в пространстве);

- результатом работы метода является вывод о достижимости помеченных вершин из других.

### 2.1.3 *IDF0*

Суть поставленной задачи – проверить в графе, заданным списком переходов и списком помеченных вершин, достижимость помеченных вершин из других и отобразить графически «тупиковые» вершины. Ход работы: считать данные, верифицировать графы, отобразить результат.

На рисунках 8 – 9 представлена *IDF*0 диаграмма ПО.



Рисунок 8 – *IDF*0 диаграмма ветка А-0

Описание модели включает в себя такие поля как:

- список сущностей;

- список переходов для каждой сущности;

- список помеченных вершин в каждом графе.

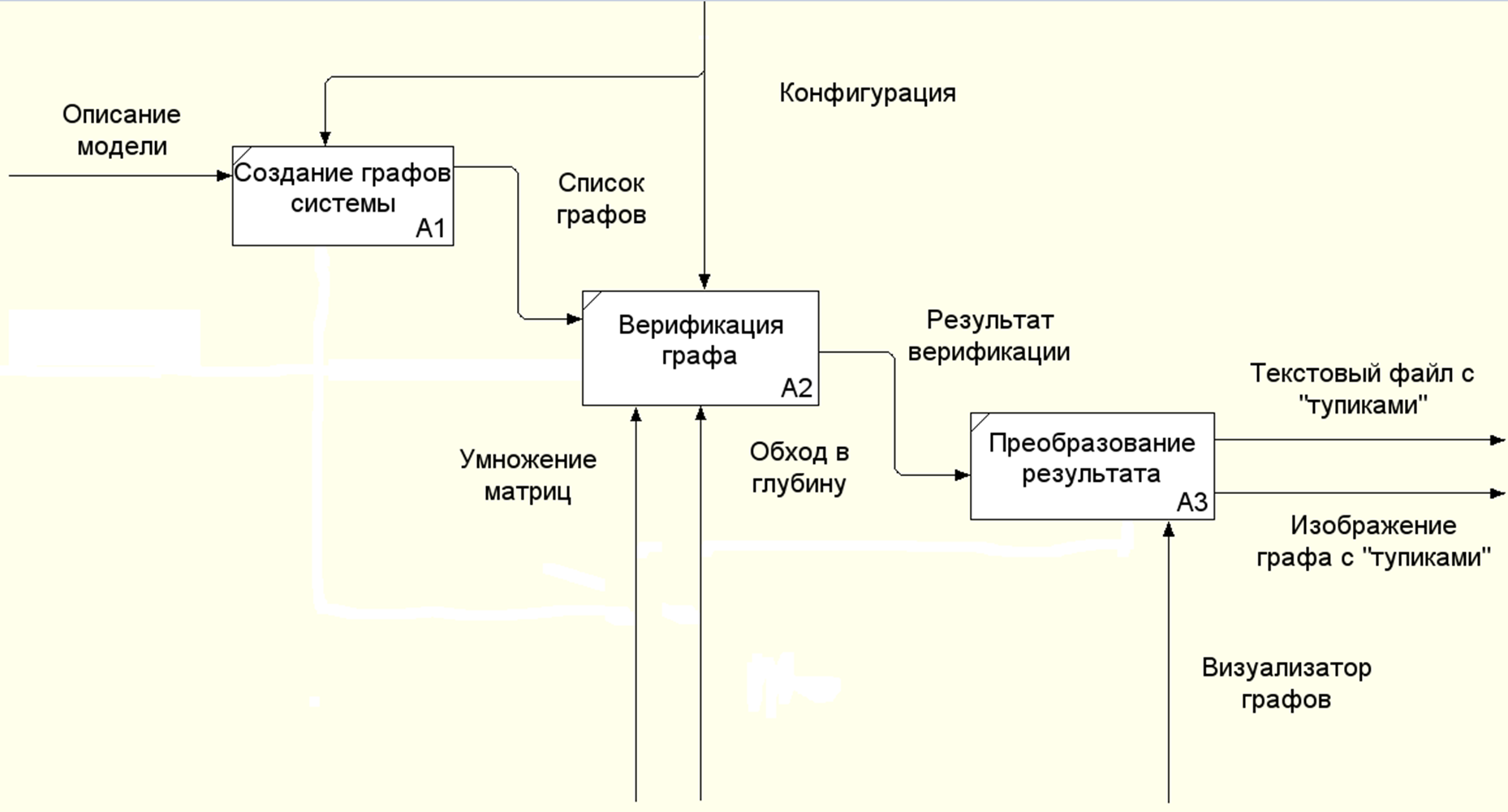


Рисунок 9 – *IDF*0 диаграмма ветка А0

Настройки конфигурации задаются следующим образом:

- список типов вершин;

- путь к директории для сохранения результата;

- список используемых опций.

Этап создания графов системы путем считывания описания модели и конфигурации, предобработка объектов лежит на *Python*. Данный шаг состоит из следующих трех этапов:

- анализ списка переходов и удаление ошибочных;

- создание матрицы переходов для каждой моделируемого объекта;

- создание ассоциативного массива для вершин и их меток.

### 2.2 *Use-case* диаграмма

На рисунке 10 приведена *use-case* диаграмма (диаграмма вариантов использования) ПО.

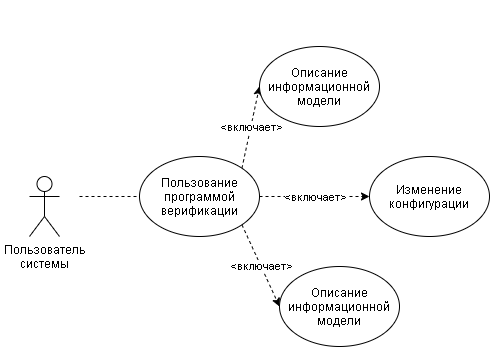


Рисунок 10 – *Use-case* диаграмма программы

Пользователь системы структурирует исходные данные и при запуске программы заполняет необходимые поля. Наряду с этим пользователь может изменять конфигурацию системы.

В данной системе представлены следующие группы пользователей:

- акторы (множество логически согласованных ролей);

- прецеденты (порядок действий).

По мере прогресса верификации, пользователю будет доступна информация о найденных «тупиках».

### 2.3 Предполагаемый метод

Подробная блок-схема разрабатываемого метода для безусловных переходов приведена на рисунке 11.

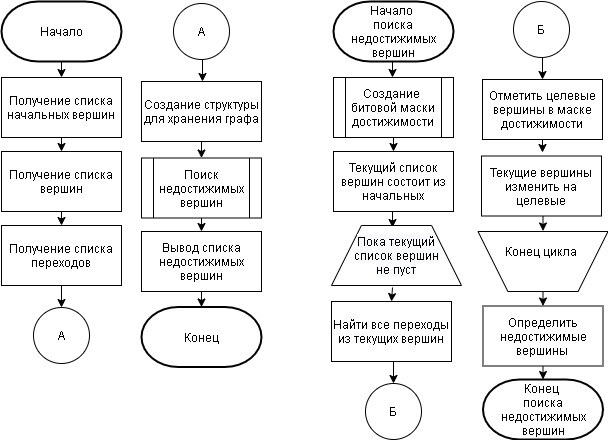


Рисунок 11 – Блок-схема разрабатываемого метода

На рисунке 12 приведена схема верификации графа с утловными переходами.

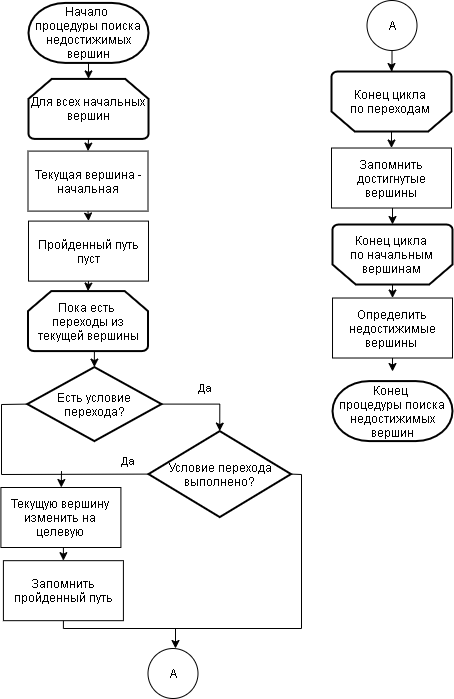


Рисунок 12 – Схема верификации графа с утловными переходами

Метод верификации графа включает в себя следующие шаги:

- создание булевой матрицы переходов;

- вычисление матрицы всех путей длины 1 путем перемножения матицы переход на саму себя;

- вычисление всех путей длины от 2 до *N*-1 аналогичным образом, где *N* – число состояний графа;

- анализ вычисленных матриц путей, который в свою очередь включает в себя следующие этапы:

- проверка недостижимости начальных вершин во всех путях для всех вершин,

- проверка достижимости конечных вершин для неконечных.

- возвращение результата верификации.

Предложенный алгоритм имеет сложность обхода аналогично методу обхода графа в глубину, то есть О(n), тогда результирующая сложность будет составлять О(n2), так как в худшем случае если на каждом шаге метод будет анализировать метка перехода, то ему потребуется рассматривать текущий путь, следовательно, выполнять О(n) сравнений. В данном случае n – число ребер графа, в котором число вершин не может превышать корня из n.

Как следствие, можем заключить, что для графа с n числом вершин сложность метода будет составлять О(n4). Сложность такого порядка говорит о том, что в перспективе необходимо будет оптимизировать данный метод, возможно комбинируя представления графа как матрицы переходов и списка смежности. Возможность параллельной верификации каждого отдельно взятого графа тоже позволит ускорить данный метод.

### 2.4 Вывод

В конструкторском разделе рассмотрены требования к графовой информационной модели и методу верификации.

Предоставлены схемы алгоритмов, описаны собственные методы решения задачи, приведена структура приложения. Приведены функциональная модель предполагаемого метода (*IDF*0 диаграмма ветка А-0 и *IDF*0 диаграмма ветка А0). Приведена подробная схема разрабатываемого метода верификации. Описаны основные шаги верификации графовой модели.

## 3 Технологический раздел

### 3.1 Выбор и обоснование языка программирования и среды разработки

В качестве языка программирования был выбран Python из-за следующих преимуществ:

- имеется достаточный опыт программирования на этом языке, что сократит время написания программы;

- данный язык программирования кроссплатформенный и объектно-ориентированный, что даст в полной мере использовать наследование, полиморфизм и т.д.

В качестве среды разработки была выбрана «*PyCharm*» по следующим причинам:

- она бесплатна в пользовании студентами;

- она имеет множество инструментов, облегчающих процесс написания и отладки кода;

- имеется достаточный опыт программирования при помощи данной среды разработки, что сократит время изучения возможностей.

### 3.2 Собственная реализация

### 3.2.1 Представление информационной модели

В данной реализации представленная система описывается классом *Model*, содержащим список графов и матрицу переходов системы. На вход последовательно поступают списки переходов каждой моделируемой сущности и, после проверки их корректности, создается граф и сохраняется в списке графов. Интерфейс класса позволяет динамически добавлять и удалять графы и переходы между графами.

Листинг 1 – Поля и методы класса *Model*

class Model:

def \_\_init\_\_(self):

self.name = "GraphModel"

self.schemes = list()

self.graph\_matrix = dict()

def add\_graph(self, name, list\_transitions, list\_marked\_statuses)…

def delete\_graph(self, name) …

def add\_graph\_transition(self, string) …

def delete\_graph\_transition(self, string) …

Каждая сущность системы описывается классом *Graph*, где сам граф хранится как матрица переходов. Также в этом классе хранится информация о самих вершинах и соответствующей им метки (если она есть). Интерфейс класса предоставляет возможность задания матрицы переходов и хранения информации о вершинах.

Листинг 2 – Поля и методы класса *Graph*

class Graph:

def \_\_init\_\_(self, name, transitions, marked\_statuses):

self.name = name

self.statuses = dict()

self.matrix = dict()

self.matrix\_filling(transitions)

self.marks\_filling(marked\_statuses)

def matrix\_filling(self, transitions): …

def marks\_filling(self, marked\_statuses): …

Все переходы в графах описываются классом *Transition*. В нем хранится информация о исходной и целевой вершинах, а также опция, связанная с этим переходом.

Листинг 3 – Поля и методы класса *Transition*

class Transition:

def \_\_init\_\_(self, status, next\_status, options=None):

self.status = str

self.next\_status = str

self.options = list

self.status = status

self.next\_status = next\_status

self.options = options

Кроме способности описывать динамические изменения в рамках одной сущности была предоставлена возможность хранить информацию о взаимодействиях между графами с помощью класса *GraphTransition*. Данный класс хранит информацию о исходной и целевой вершинах в разных графах, отождествляющихся с сущностями системы по именам.

Листинг 4 – Поля и методы класса *GraphTransition*

class GraphTransition(Transition):

def \_\_init\_\_(self, graph, status, next\_graph, next\_status, options=None):

super().\_\_init\_\_(status, next\_status, options)

self.graph = str

self.next\_graph = str

self.graph = graph

self.next\_graph = next\_graph

Метки вершин хранятся в множестве *Mark*, которое может быть переопределено для каждой системы в соответствии с конфигурацией.

Листинг 5 – Множество *Mark*

class Mark(enum.Enum):

start = 'start'

end = 'end'

Условия переходов хранятся в классе *BaseOption*, где хранится команда перехода и передаваемые ей параметры (условия перехода).

Листинг 6 – Поля и методы класса *BaseOption*

class BaseOption:

def \_\_init\_\_(self):

self.command = Command

self.parameters = list or None

def exec(self):

self.command.exec(self.parameters)

class Option(BaseOption):

docOptions = dict(all=AllCommand(), any=AnyCommand())

def find\_command\_in\_dicts(self, command):

command = docOptions.get(command)

if command:

return command

def \_\_init\_\_(self, command, args):

finding\_command = self.find\_command\_in\_dicts(command)

if finding\_command:

super().\_\_init\_\_()

self.parameters = args

self.command = finding\_command

Было реализовано 2 команды: *AnyCommand* и *AnyCommand*. Первая команда вернет правду, если в передаваемых условиях будет хотя бы одно совпадение с параметрами команды. Вторая команда вернет правду, если параметрами команды совпадают с последовательностью передаваемых условий.

Листинг 7 – Поля и методы классов *AnyCommand* и *AnyCommand*

class AnyCommand(Command):

def exec(self, parameters, conditions):

return set(conditions) - set(parameters)

class AllCommand(Command):

def exec(self, parameters, conditions):

return set(conditions) & set(parameters)

### 3.2.2 Булева матрица переходов

Данная функция получает на вход опциональную матрицу переходов и создает аналогичную булевою.

Листинг 8 – Функция создания булевой матрицы

def create\_bool\_matrix(graph\_matrix):

matrix = []

for status, values in graph\_matrix.items():

row = []

for next\_status, options in values.items():

if options is not None:

row.append(1)

else:

row.append(0)

matrix.append(row)

return matrix

Использование булевой матрицы при поиске достижимости вершин уменьшает время работы алгоритма верификации. С помощью булевой матрицы можно найти все возможные пути в графе, что облегчает проверку выполнений условий перехода. Все возможные пути находятся путем перемножения матрицы саму на себя столько раз, сколько вершин в графе. Листинг 9 умножения матрицы приведен ниже.

Листинг 9 – Класс перемножения булевой матрицы

class MultiplicationMatrix:

@staticmethod

def direct\_multiply(matrix\_a, matrix\_b):

n = len(matrix\_a)

l1 = len(matrix\_a[0])

l2 = len(matrix\_b)

m = len(matrix\_b[0])

if l1 != l2:

return IOError

matrix\_product = np.zeros((n, m), dtype="i").tolist()

for row in range(0, n):

for col in range(0, m):

for k in range(0, l1):

matrix\_product[row][col]=matrix\_product[row][col] or matrix\_a[row][k] \* matrix\_b[k][col]

return matrix\_product

Пример задания тестовых данных в коде программы приведен ниже в листинге.

Листинг 10 – Пример представления тестовых данных

trans\_1 = ["Draft New", "New Signed any:ParametersAny", "New Refused all:ParametersAll", "Signed Close"]

trans\_2 = ["Draft New any:ParametersAny all:ParametersAll"]

trans\_3 = ["1 2", "2 3", "3 4", "4 2"]

trans\_4 = ["1 2", "2 3", "3 4", "4 2", "3 5"]

trans\_5 = ["1 2", "2 3", "3 4", "4 2", "3 5", "1 6", "6 7", "7 8", "6 8", "8 4"]

trans\_f = ['1 2 any:1 all:1', '2 3 any:2', '1 3', '2 4', '3 5']

trans\_s = ['6 7', '7 8', '9 7']

mark\_1 = ['start Draft', 'end Refused', 'end Close']

mark\_2 = ['start Draft']

mark\_3 = ['start 1']

mark\_4 = ['start 1', 'end 5']

mark\_5 = ['start 1', 'end 5']

mark\_11 = ['start Draft', 'end Refused']

mark\_f = ['start 1', 'end 5']

mark\_s = ['start 6', 'start 9', 'end 8']

model = Model()

model.add\_graph("DocType1", trans\_1, mark\_1)

model.add\_graph("DocType2", trans\_2, mark\_2)

model.add\_graph("DocType3", trans\_3, mark\_3)

model.add\_graph("DocType4", trans\_4, mark\_4)

model.add\_graph("DocType5", trans\_5, mark\_5)

model.add\_graph("First", trans\_f, mark\_f)

model.add\_graph("Second", trans\_s, mark\_s)

model.add\_graph\_transition('First 4 Second 6')

3.2.3 Алгоритм верификации

Ниже представлен листинг функции, которая принимает решение о адекватности хранимой графовой модели. На вход она получает объект класса класса *Model*, а возвращает булевое значение.

Листинг 11 – Функция проверки адекватности модели

def check (information\_model):

result = False

if isinstance(information\_model, Model):

for graph in information\_model.schemes:

result = False

start\_statuses = []

end\_statuses = []

verifiable\_statuses = []

for status, mark in graph.statuses.items():

if mark == 'start':

start\_statuses.append(status)

elif mark == 'end':

end\_statuses.append(status)

else:

verifiable\_statuses.append(status)

i = 0

start\_statuses\_index = []

end\_statuses\_index = []

verifiable\_statuses\_index = []

for status, mark in graph.statuses.items():

if mark == 'start':

start\_statuses\_index.append(i)

elif mark == 'end':

end\_statuses\_index.append(i)

else:

verifiable\_statuses\_index.append(i)

i += 1

bool\_matrix = create\_adjacency\_matrix(graph.matrix)

steps = [bool\_matrix]

for i in range(len(graph.matrix) - 1):

steps.append(MultiplicationMatrix.direct\_multiply(steps[0], steps[-1]))

result = True

marked = None

status = None

for start in start\_statuses\_index:

for step\_matrix in steps:

for i in range(len(bool\_matrix)):

if step\_matrix[i][start] == 1:

result = False

marked = start

status = i

break

if result is False:

break

if result is False:

break

if result is False:

print('Error graph', graph.name, 'in start status', marked, 'from status', status)

return False

for end in end\_statuses\_index:

for i in range(len(bool\_matrix)):

if i == end or end\_statuses\_index.count(i) > 0:

continue

result = False

for step\_matrix in steps:

if step\_matrix[i][end] == 1:

result = True

break

if result is False:

marked = end

status = i

break

if result is True:

break

if result is False:

print('Error graph', graph.name, 'in end status', marked, 'from status', status)

return False

return result

3.2.4 Алгоритм вывода графовой модели

Ниже представлен листинг функции, которая вывод список переходов для разного представления графа. На вход она получает объект класса класса *Model*, ничего не возвращает.

Листинг 12 – Функция вывода информации о модели

def output\_model(model):

# Вывод матрицы переходов системы

for initial, value in model.graph\_matrix.items():

print('%16s' % initial, end='| ')

for purpose, options in value.items():

print(purpose, end=' ')

if options is not None:

if len(options) == 0:

print('pass', end=' ')

else:

for opt in options:

opt.exec()

print('op', end='+')

print( end=' ')

else:

print('None', end=' ')

print()

for graph in model.schemes:

print('\n\*\*\*', graph.name, '\*\*\*')

print('.....Transitions.....')

# Вывод списков смежности

for status, adjacency in graph.adjacency\_list.items():

for next\_status, options in adjacency.items():

if options is not None:

print(status, next\_status, end=' ')

if len(options) == 0:

print('pass')

else:

for opt in options:

opt.exec()

print(end=' ')

print('')

#Вывод матрицы переходов

for status, value in graph.matrix.items():

for next\_status, options in value.items():

if options is not None:

print(status, next\_status, end=' ')

if len(options) == 0:

print('pass')

else:

for opt in options:

opt.exec()

print(end=' ')

print('')

print('...............\n')

print('----Marked statuses-------')

for status, mark in graph.statuses.items():

print(status, mark)

print('--------------')

### 3.3 Исследование временных характеристик

Для подмены функционала разных способов верификации был разработан класс *Checker*, листинг которого представлен ниже.

Листинг 13 – Поля и методы класса *Checker*

class Checker:

def \_\_init\_\_(self, verifier):

self.checker = verifier#GraphAdjacencyLists()#GraphMatrixVerifier()

def check(self, model):

return self.checker.check\_model(model)

def get\_errors(self):

return self.checker.errors

def get\_warnings(self):

return self.checker.warnings

Ниже представлен листинг функции, которая сравнивает реализации и строит графики.

Листинг 14 – Функция построения графиков

def modeling():

adj\_verifier = GraphAdjacencyLists()

matr\_verifier = GraphMatrixVerifier()

repeat = 50

max\_ = 180

time\_adjacency = [0]

time\_matrix = [0]

for n in range(max\_):

check\_model = Model()

trans = [str(i) + ' ' + str(randint(0, n)) + 'any:' + str(i) for i in range(n)]

starts = ['start ' + str(i) for i in range(int(n \* 0.95))]

ends = ['ends ' + str(i) for i in range(int(n \* 0.05))]

marks = starts + ends

check\_model.add\_graph("Check", trans, marks)

start\_time = datetime.now()

for \_ in range(repeat):

adj\_verifier.check\_model(check\_model)

end\_time = datetime.now()

time\_adjacency.append(max(((end\_time - start\_time) / repeat).microseconds // 1000, time\_adjacency[-1]))

start\_time = datetime.now()

for \_ in range(repeat):

matr\_verifier.check\_model\_mark(check\_model)

end\_time = datetime.now()

time\_matrix.append(((end\_time - start\_time) / repeat).microseconds // 1000)

time\_matrix[-1] \*= 3

fig = plt.figure()

ax = fig.add\_subplot(111)

ax.plot(time\_matrix, [n for n in range(max\_ + 1)], label=u'Матрица смежности')

ax.plot(time\_adjacency, [n for n in range(max\_ + 1)], label=u'Список смежности', linestyle='--', linewidth=1)

for axis in [ax.xaxis, ax.yaxis]:

axis.set\_major\_locator(ticker.MaxNLocator(integer=True))

ax.set\_ylabel('Количество ребер')

ax.set\_xlabel('Время (мс)')

plt.title(u'Верификация графов с безусловными переходами')

plt.legend() # легенда для всего рисунка fig

plt.show()

### 3.4 Пример работы программы

Реализованный метод позволяет верифицировать различные диаграммы. Например, представленная на рисунке 13 ДПС может быть проверена на основе данного метода.

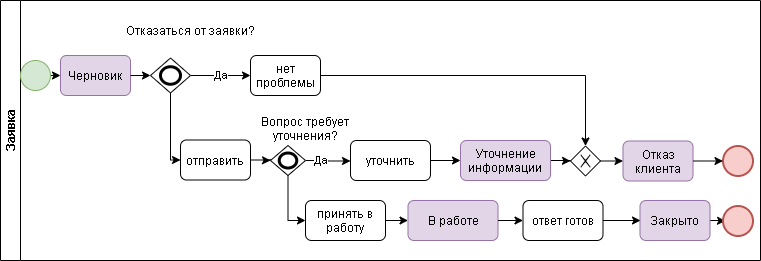


Рисунок 13 – Пример верифицируемой диаграммы

На вход программе подаются текстовые сведения о графе. Пример входных данных представлен на рисунке 14.

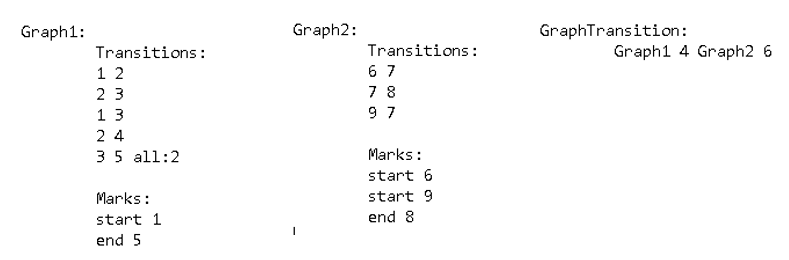


Рисунок 14 – Пример входных данных программы

На рисунке 15 представлен графический результат работы программы. Вершины 3 была отмечена как ошибочная, так как при переходе из 1 в 3 будет невозможен переход в конечную вершину 5. Также вершина 4 была отмечена как потенциально опасная.

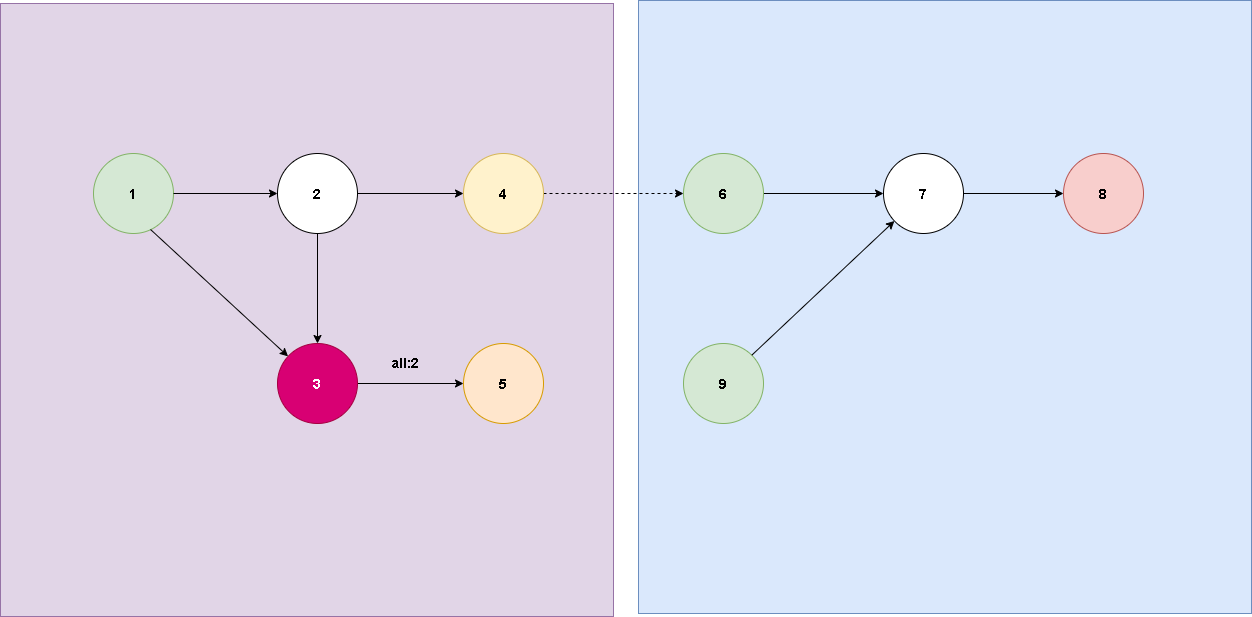
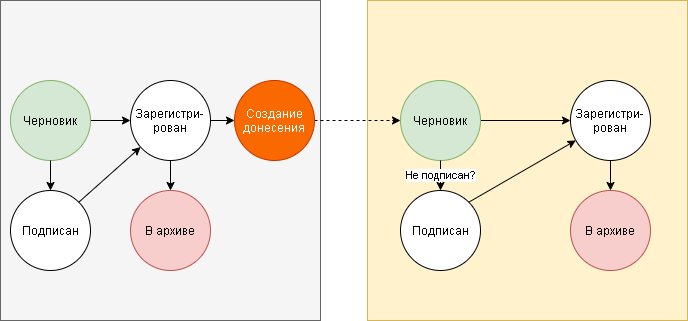


Рисунок 15 – Результат работы программы

На рисунке 16 представлен графический результат верификации ДПС, из которого видно, что вершина «Создание донесения» отмечена как ошибочная. Слева на картинке расположен вид отчета, справа – вид донесения.

Рисунок 16 – Результат верификации ДПС

### 3.4 Вывод

В технологическом разделе были выбраны средства реализации, представлены используемые структуры и приведены листинги реализуемых методов. Для анализа работы метода граф может хранится в виде двух структур данных: матрицей переходов и списком смежности. Информационная модель описывается списком графов, каждый из которых может быть связан переходом с другим графом. Также приведен пример работы метода.

## 4 Исследовательский раздел

### 4.1 Исследование скорости работы алгоритма для графа с безусловными переходами

На рисунке 17 представлено исследование скорости работы метода с безусловными переходами. Анализируя полученные результаты заметить, что для модели с безусловными переходами лучшей структурой для работы будет список смежности.

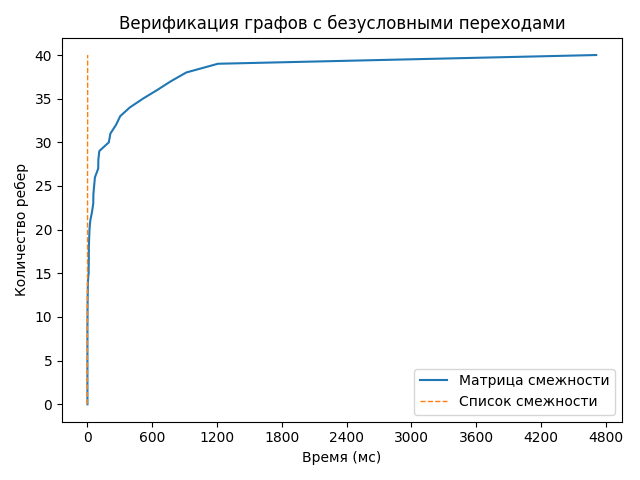


Рисунок 17 –Исследования скорости работы метода с безусловными переходами

Примером модели с такими переходами может служить ДПС. Для любого графа максимально возможное количество ребер будет не больше квадрата количества состояний.

### 4.2 Исследование скорости работы алгоритма для графа с условными переходами

На рисунке 18 представлено исследование работы метода с условными переходами.

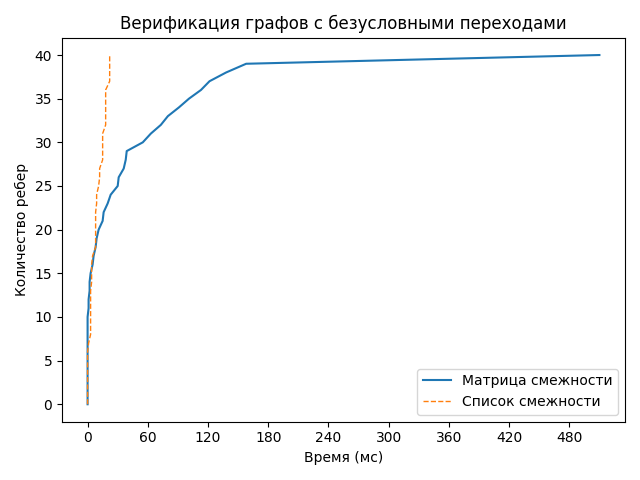


Рисунок 18 –Исследования скорости работы метода с условными переходами

При использовании дополнительных условий на переходах можно заметить, что время работы метода уменьшилось, так как уменьшается количество возможных переходов. Примером модели с такими переходами может служить блок-схема программы.

### 4.3 Вывод

В исследовательском разделе были проанализированы две структуры данных для хранения графа. Представление графа в виде списка смежности дает выигрыш по времени для любых переходов в графе.

# ЗАКЛЮЧЕНИЕ

Во время выполнения практики была достигнута поставленная цель: разработан метод верификации графовой информационной модели.

В ходе выполнения поставленной цели, были решены следующие задачи:

- выполнен анализ области информационного моделирования;

- рассмотрены вариации задач верификации ПО;

- проанализировать существующие способы верификации ПО и их применимость к информационным моделям;

- разработаны метод верификации графовой информационной модели;

- реализованы предложенный метод и проверить его работу.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 34.003-90. Информационная технология (ИТ). Комплекс стандартов на автоматизированные системы. Термины и определения // Москва: Стандартинформ. 2009. С. 14.
2. Большаков Александр Сергеевич, Раковский Дмитрий Игоревич. Программное обеспечение моделирования угроз безопасности информации в информационных системах. // [Правовая информатика](https://cyberleninka.ru/journal/n/pravovaya-informatika). – 2020.– С. 25-48.
3. Вараница-Городовская Жанна Игоревна, Иваньо Ярослав Михайлович, Асалханов Петр Георгиевич. Информационное и математическое обеспечение программного комплекса «Моделирование трудозатрат для сельскохозяйственных предприятий». // Информационные и математические технологии в науке и управлении. – 2018.– С. 156-165.
4. Киселев Денис Викторович, Семенов Сергей Сергеевич, Педан Алексей Викторович. Постановка задачи на моделирование процесса эксплуатации программно-аппаратных комплексов связи. // T-Comm - Телекоммуникации и Транспорт. – 2018.– С. 46-52.
5. Бурякова Н.А., Чернов А.В. Классификация частично формализованных и формальных моделей и методов верификации программного обеспечения // Инженерный Вестник Дона. 2010. N 4. С. 129.
6. Staroletov S., Tengis T. Model-driven development methods and formal verification algorithms for cyber-physical systems as applied to system software of unmanned aerial vehicles // Grand Altai Research & Education. 2020. С.75–83.
7. Бибило П.Н., Романов В.И. Интеграция САПР для синтеза логических схем с использованием глобальной оптимизации // Программные продукты и системы. 2019. С. 1–8.
8. Елсуков П.Ю. Принципы верификации информационных моделей и алгоритмов // Образовательные ресурсы и технологии. 2017. N 2 (19). С. 81–86.
9. Рудаков И.В., Гурин Р.Е. Разработка и исследование синтетического метода верификации программы с помощью SMT-решателей. // Вестник Московского государственного технического университета им. Н. Э. Баумана. Серия «Приборостроение». – 2020.– С. 49-64.
10. И.В. Ковалев, M.В. Сарамуд, В.В. Лосев, А.А. Колташев. Метод и инструментарий верификации кроссплатформенного бортового программного обеспечения. // Современные инновации, системы и технологии. – 2021.– С. 22-34.
11. Дергачев А.М., Садырин Д.С., Ильина А.Г., Логинов И.П., Кореньков Ю.Д. Подход к верификации аллокаторов динамической памяти, основанный на символьном выполнении программ. // Научно-технический вестник информационных технологий, механики и оптики. – 2020.– С. 101-110.
12. Общие сведения о фрактальном анализе сигналов [Электронный ресурс]. Москва, 2017. URL: https://bstudy.net/641283/tehnika/obschie\_svedeniya\_ fraktalnom\_analize\_signalov (дата обращения 01.03.2022).
13. Короченцев Д.А., Куфинов Д.Ю. Исследование вопросов защиты исполняемых файлов программных реализаций на основе виртуализации. // Colloquium-journal. – 2021.– С. 37-39.
14. Смирнова В.С., Шаламов В.В., Ефимова В.А., Фильченков А.А. Оптимизация гиперпараметров на основе объединения априорных и апостериорных знаний о задаче классификации. // Научно-технический вестник информационных технологий, механики и оптики. – 2020. – С.828-834.
15. Общие сведения о фрактальном анализе сигналов. [Электронный ресурс]. Москва, 2013. https://studme.org/327909/tehnika/apriornaya \_aposteriornaya\_informatsiya (дата обращения 01.03.2022).
16. Гурин Р.Е., Рудаков И.В., Ребриков А.В. Методы верификации программного обеспечения. // Машиностроение и компьютерные технологии. – 2015. – С.1-17.

# ПРИЛОЖЕНИЕ А

Построение ориентированного графа

class Graph:

def \_\_init\_\_(self, name, transitions, marked\_statuses):

self.name = name

self.statuses = dict()

self.matrix = dict()

self.adjacency\_list = dict()

self.matrix\_filling(transitions)

self.marks\_filling(marked\_statuses)

self.lists\_filling(transitions)

def matrix\_filling(self, transitions):

for transit in transitions:

if isinstance(transit, Transition):

if list(self.statuses.keys()).count(transit.status) == 0:

self.statuses.update({transit.status: None})

self.matrix.update({transit.status: dict.fromkeys(self.statuses.keys())})

if list(self.statuses.keys()).count(transit.next\_status) == 0:

self.statuses.update({transit.next\_status: None})

self.matrix.update({transit.next\_status: dict.fromkeys(self.statuses.keys())})

for row in self.matrix.keys():

columns = list(self.matrix.get(row).keys())

if columns.count(transit.status) == 0:

self.matrix.get(row).update({transit.status: None})

if columns.count(transit.next\_status) == 0:

self.matrix.get(row).update({transit.next\_status: None})

row = self.matrix.get(transit.status)

row.update({transit.next\_status: transit.options})

def marks\_filling(self, marked\_statuses):

for marked\_status in marked\_statuses:

if isinstance(marked\_status, MarkedStatus):

self.statuses.update({marked\_status.status: marked\_status.mark})

def lists\_filling(self, transitions):

for transit in transitions:

if isinstance(transit, Transition):

if list(self.adjacency\_list.keys()).count(transit.status) == 0:

self.adjacency\_list.update({transit.status: dict()})

self.adjacency\_list.get(transit.status).update({transit.next\_status: transit.options})

class Transition:

def \_\_init\_\_(self, status, next\_status, options=None):

self.status = str

self.next\_status = str

self.options = list

self.status = status

self.next\_status = next\_status

self.options = options

class GraphTransition(Transition):

def \_\_init\_\_(self, graph, status, next\_graph, next\_status, options=None):

super().\_\_init\_\_(status, next\_status, options)

self.graph = str

self.next\_graph = str

self.graph = graph

self.next\_graph = next\_graph

class Mark(enum.Enum):

start = 'start'

end = 'end'

class MarkedStatus:

def \_\_init\_\_(self, mark, status):

self.mark = Mark

self.status = str

self.mark = mark

self.status = status

# ПРИЛОЖЕНИЕ Б

Верификация ориентированного графа, представленного матрицей переходов

class MultiplicationMatrix:

@staticmethod

def direct\_multiply(matrix\_a, matrix\_b):

n = len(matrix\_a)

l1 = len(matrix\_a[0])

l2 = len(matrix\_b)

m = len(matrix\_b[0])

if l1 != l2:

return IOError

matrix\_product = np.zeros((n, m), dtype="i").tolist()

for row in range(0, n):

for col in range(0, m):

for k in range(0, l1):

matrix\_product[row][col]=matrix\_product[row][col] or matrix\_a[row][k] \* matrix\_b[k][col]

return matrix\_product

class GraphMatrixVerifier:

@staticmethod

def \_create\_adjacency\_matrix(graph\_matrix):

matrix = []

for status, values in graph\_matrix.items():

row = []

for next\_status, options in values.items():

if options is not None:

row.append(1)

else:

row.append(0)

matrix.append(row)

return matrix

def \_verify\_start\_graph(self, graph\_name, start\_status\_dict, status\_dict, steps):

result = True

for start\_index, start in start\_status\_dict.items():

for step\_matrix in steps:

for i in range(len(status\_dict)):

if step\_matrix[i][start\_index] == 1: # найден переход в начальный статус

self.add\_error(graph\_name, 'start', start, status\_dict[i])

result = False

return result

def \_verify\_end\_graph(self, graph\_name, end\_status\_dict, status\_dict, steps):

result = True

for i in range(len(status\_dict)):

if end\_status\_dict.get(i) is not None:

continue

result = False

for end\_index, end in end\_status\_dict.items():

for step\_matrix in steps:

if step\_matrix[i][end\_index] == 1:

result = True

break

if result is True:

break

if result is False:

self.add\_warning(graph\_name, 'end', status\_dict.get(i))

return result

def \_find\_all\_paths(self, matrix):

bool\_matrix = self.\_create\_adjacency\_matrix(matrix)

steps = [bool\_matrix]

for i in range(len(matrix) - 1):

steps.append(MultiplicationMatrix.direct\_multiply(steps[0], steps[-1]))

return steps

def \_check\_graph(self, graph\_statuses, graph\_matrix, graph\_name):

start\_statuses = self.\_determinate\_marked\_statuses(graph\_statuses, 'start')

end\_statuses = self.\_determinate\_marked\_statuses(graph\_statuses, 'end')

index\_statuses = self.\_determinate\_marked\_statuses(graph\_statuses, None)

index\_statuses.update(start\_statuses)

index\_statuses.update(end\_statuses)

steps = self.\_find\_all\_paths(graph\_matrix)

result = self.\_verify\_start\_graph(graph\_name, start\_statuses, index\_statuses, steps)

result = self.\_verify\_end\_graph(graph\_name, end\_statuses, index\_statuses, steps) and result

return result

def check\_model(self, information\_model):

self.warnings.clear()

self.errors.clear()

#self.successes.clear()

result = True

statuses = dict()

if isinstance(information\_model, Model):

for graph in information\_model.schemes:

result = self.\_check\_graph(graph.statuses, graph.matrix, graph.name) and result

for status, mark in graph.statuses.items():

if mark == 'start':

mark = None

statuses.update({graph.name+status: mark})

result = self.\_check\_graph(statuses, information\_model.graph\_matrix, information\_model.name)

return result

# ПРИЛОЖЕНИЕ В

Верификация ориентированного графа, представленного списками смежности

class GraphAdjacencyLists(BaseVerifier):

def \_bfs(self, graph, starts):

nodes = [node for node in starts]

visited = list(nodes)

queue = [nodes]

lasts = []

while queue:

temps = queue.pop()

nodes = []

for t in temps:

if graph.get(t) is None:

lasts.append(t)

else:

for neighbour in graph.get(t):

if neighbour not in visited and neighbour not in nodes:

nodes.append(neighbour)

if len(nodes) > 0:

queue.append(nodes)

visited += nodes

return visited, lasts

def \_dfs(self, graph, start, path=None, visited=None):

if visited is None:

visited = set()

path = list()

visited.add(start)

path.append(start)

if graph.get(start) is None:

return start

for next in set(graph.get(start)) - visited:

options = graph[start][next]

flag = len(options) == 0

for opt in options:

if isinstance(opt.command, AllCommand):

for p in opt.parameters:

if path.count(p) > 0:

flag = True

if isinstance(opt.command, AnyCommand):

for p in opt.parameters:

if path.count(p) > 0:

flag = True

if flag:

self.\_dfs(graph, next, path, visited)

return visited, path[-1]

def \_check\_graph(self, statuses, adjacency\_list):

start\_statuses = list(self.\_determinate\_marked\_statuses(statuses, 'start').values())

end\_statuses = list(self.\_determinate\_marked\_statuses(statuses, 'end').values())

all\_statuses = set(statuses.keys())

visited, lasts = self.\_bfs(adjacency\_list, start\_statuses)

result = set(visited) == set(statuses.keys())

err\_starts = set(statuses.keys()) - set(visited)

err\_ends = set(lasts) - set(end\_statuses)

return result, err\_starts, err\_ends

def \_check\_graph\_mark(self, statuses, adjacency\_list):

start\_statuses = list(self.\_determinate\_marked\_statuses(statuses, 'start').values())

end\_statuses = list(self.\_determinate\_marked\_statuses(statuses, 'end').values())

all\_statuses = set(statuses.keys())

visitedes = []

ends = []

for start in start\_statuses:

visited, end = self.\_dfs(adjacency\_list, start)

visitedes.append(visited)

ends.append(end)

return True, ends

def check\_model(self, information\_model):

self.warnings.clear()

self.errors.clear()

result = True

statuses = dict()

lists = dict()

if isinstance(information\_model, Model, method):

for graph in information\_model.schemes:

res, starts, ends = method(graph.statuses, graph.adjacency\_list)

result = res and result

for warn in starts:

self.add\_warning(graph.name, 'start', warn)

for warn in ends:

self.add\_warning(graph.name, 'end', warn)

for status, value in graph.adjacency\_list.items():

t = dict()

for next\_status, options in value.items():

t.update({graph.name + next\_status: options})

lists.update({graph.name + status: t})

for status, mark in graph.statuses.items():

statuses.update({graph.name + status: mark})

for trans in information\_model.graph\_transition\_list:

lists.update({trans.graph + trans.status: {trans.graph + trans.status: trans.options}})

result, starts, ends = self.\_check\_graph(statuses, lists)

for err in starts:

self.add\_error(information\_model.name, 'start', err)

# ПРИЛОЖЕНИЕ Г

Исследование временных характеристик

def modeling():

adj\_verifier = GraphAdjacencyLists()

matr\_verifier = GraphMatrixVerifier()

repeat = 10

max\_ = 180

time\_adjacency = [0]

time\_matrix = [0]

for n in range(max\_):

check\_model = Model()

trans = [str(i) + ' ' + str(randint(0, n)) + 'any:' + str(i) for i in range(n)]

starts = ['start ' + str(i) for i in range(int(n \* 0.95))]

ends = ['ends ' + str(i) for i in range(int(n \* 0.05))]

marks = starts + ends

check\_model.add\_graph("Check", trans, marks)

start\_time = datetime.now()

for \_ in range(repeat):

adj\_verifier.check\_model(check\_model)

end\_time = datetime.now()

time\_adjacency.append(max(((end\_time - start\_time) / repeat).microseconds // 1000, time\_adjacency[-1]))

start\_time = datetime.now()

for \_ in range(repeat):

matr\_verifier.check\_model\_mark(check\_model)

end\_time = datetime.now()

time\_matrix.append(((end\_time - start\_time) / repeat).microseconds // 1000)

time\_matrix[-1] \*= 3

fig = plt.figure()

ax = fig.add\_subplot(111)

ax.plot(time\_matrix, [n for n in range(max\_ + 1)], label=u'Матрица смежности')

ax.plot(time\_adjacency, [n for n in range(max\_ + 1)], label=u'Список смежности', linestyle='--', linewidth=1)

for axis in [ax.xaxis, ax.yaxis]:

axis.set\_major\_locator(ticker.MaxNLocator(integer=True))

ax.set\_ylabel('Количество ребер')

ax.set\_xlabel('Время (мс)')

plt.title(u'Верификация графов с безусловными переходами')

plt.legend() # легенда для всего рисунка fig

plt.show()