



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

ФАКУЛЬТЕТ «Робототехника и комплексная автоматизация»

КАФЕДРА «Системы автоматизированного проектирования (РК-6)»

РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к выпускной квалификационной работе

на тему

«Разработка компонентов графоориентированного
программного каркаса для реализации сложных
вычислительных методов»

Студент РК6-81Б
 группа

подпись, дата

Тришин И.В.
 ФИО

Руководитель ВКР

подпись, дата

Соколов А.П.
 ФИО

Консультант

подпись, дата

Першин А.Ю.
 ФИО

Нормоконтролёр

подпись, дата

Грошев С.В.
 ФИО

Москва, 2022

РЕФЕРАТ

выпускная квалификационная работа: 54 с., 20 рис., 0 табл., 13 источн.

CASE-СИСТЕМЫ, РАСПРЕДЕЛЁННЫЕ ВЫЧИСЛЕНИЯ, ГРАФООРИЕНТИРОВАННЫЙ ПОДХОД, СЛОЖНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ МЕТОДЫ, ОПИСАНИЕ БИЗНЕС-ЛОГИКИ.

Данная работа посвящена разработке программного инструментария, позволяющего описывать и реализовать логику решения различных задач, требующих большого количества трудоёмких вычислений. При описании применяется т.н. графоориентированный подход, который позволяет пользователю задавать действия алгоритма или вычислительного метода в виде переходов между состояниями данных. Формируемое описание затем интерпретируется и выполняется с применением стандартных или пользовательских реализаций каждого из переходов.

Реализованные программные средства позволяют структурировать и ускорить разработку наукоёмкого программного обеспечения, применяемого при анализе больших объёмов данных и научно-технических исследованиях.

Тип работы: выпускная квалификационная работа.

Тема работы: *«Разработка компонентов графоориентированного программного каркаса для реализации сложных вычислительных методов».*

Объект исследования: методы описания бизнес-логики в системах автоматизированной разработки программного обеспечения.

Основная задача, на решение которой направлена работа: @Основная задача, на решение которой направлена работа@.

Цели работы: создание новых программных средств для описания и представления графовых моделей сложных вычислительных методов и их обхода

В результате выполнения работы: 1) в ходе аналитического обзора литературы и сравнительного анализа разработки с аналогичной обоснована её актуальность; 2) обозначены теоретические основы подхода GBSE к построению описаний алгоритмов; 3) сформулированы требования к алгоритму интерпретации описаний, построенных по методологии GBSE; 4) спроектированы структуры данных, отвечающие за программное

представление описаний алгоритмов и их элементов в программном каркасе comsdk; 5) спроектированные структуры данных были реализованы на языке C++.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	11
1 Аналитический обзор	16
2 Постановка задачи	26
3 Архитектура программной реализация	29
3.1 Требования к алгоритму обхода графовых моделей	29
3.2 Функциональные структуры данных	31
3.3 Информационные структуры данных	35
3.3.1 Класс вершины графовой модели	35
3.3.2 Класс ребра графовой модели	36
3.3.3 Класс графовой модели	37
3.3.4 Классы операций с вершинами и рёбрами графовой модели	39
3.4 Управляющие структуры данных	42
4 Описание реализованных средств	44
4.1 Используемые технологии	44
4.2 Алгоритмы	44
4.3 Сборка и тестирование	47
ЗАКЛЮЧЕНИЕ	49
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	50
ПРИЛОЖЕНИЯ	53
А	53

ВВЕДЕНИЕ

Современные научно-технические исследования зачастую включают в себя задачи, при решении которых требуется большое количество вычислений, для которых задействуются большие вычислительные мощности. К таким задачам относятся, например, задачи анализа, определения характеристик материалов или технических объектов, моделирования сложных динамических процессов. Как правило, для решения подобных задач применяется или разрабатывается специализированное программное обеспечение (далее – ПО).

Среди прочих применяются программные продукты, предоставляющие пользователю формальный язык описания математических выражений и его интерпретатор, выполняющий необходимые вычисления на машине пользователя. К таким системам относятся, например, Mathcad. Также стоит отметить системы специализирующиеся на символьной алгебре, такие, как Maple[1] и Wolfram Mathematica. В настоящее время данные программные комплексы поддерживают решение задач из различных областей математики, включающих в себя теорию графов, теорию множеств и т.д, предоставляют инструменты визуализации и анализа результатов. Все они позволяют выполнять математическое моделирование, в том числе, сложных технических объектов. При всех их преимуществах необходимость формулировать математические постановки решаемых задач (т.е. формировать математические модели, составлять системы уравнений и т.д.) остаётся за пользователем. Зачастую требуется решать множество задач с схожей постановкой, но с различными входными параметрами. Такая необходимость, например, возникает при решении задач оптимизации, где критерием является некоторая характеристика, получаемая в результате решения задачи анализа. Следовательно, целесообразны автоматизированные средства решения типовых задач анализа и моделирования.

Данные средства относятся к специализированному ПО, а потому при их разработке требуются глубокие познания в предметной области. Кроме того, важно, чтобы создаваемая кодовая база была рассчитана на дальнейшую поддержку, что предъявляет соответствующие требования к

структуре исходного кода и документации. Таким образом целесообразно применение некоторых средств, позволяющих организовать разработку программного обеспечения для решения задач моделирования и анализа и повысить его поддерживаемость.

В наши дни популярность приобретает применение т.н. научных систем управления потоком задач (англ. scientific workflow systems). Они предоставляют средства организации этапов решения вычислительной задачи и управления вычислительными ресурсами. Процесс работы с подобными системами состоит из 4 основных этапов:

- 1) составление описания операций обработки данных и зависимостей между ними;
- 2) распределение процессов обработки данных по вычислительным ресурсам;
- 3) выполнение обработки данных;
- 4) сбор и анализ результатов и статистики [2].

Примерами подобных систем могут служить Pegasus[3], Kepler[4] и pSeven[5]. Помимо инструментов загрузки пользовательских реализаций этапов решения задачи они, как правило, представляют библиотеку типовых действий и преобразований, таких, как считывание данных и их сохранение в файлы одного из поддерживаемых форматов, операции со строками, работы с базами данных, и т.д. Кроме того, некоторые из них имеют средства интеграции с другими системами моделирования и анализа, что позволяет задействовать их при расчётах. На рисунке В.1 изображён пример описания некоторого процесса в системе Kepler.

Кроме того, для облегчения процесса разработки трудоёмкого ПО существуют т.н. платформы малокодовой разработки (англ. low-code development platforms, LCPD)[6]. В них, подобно системам управления потоком задач, логика разрабатываемого программного продукта описывается при помощи некоторого формального языка или с использованием графического редактора. От системы к системе подход к описаниям варьируется. Может применяться структурный подход, описывающий шаги алгоритма, или предметно-ориентированный, при

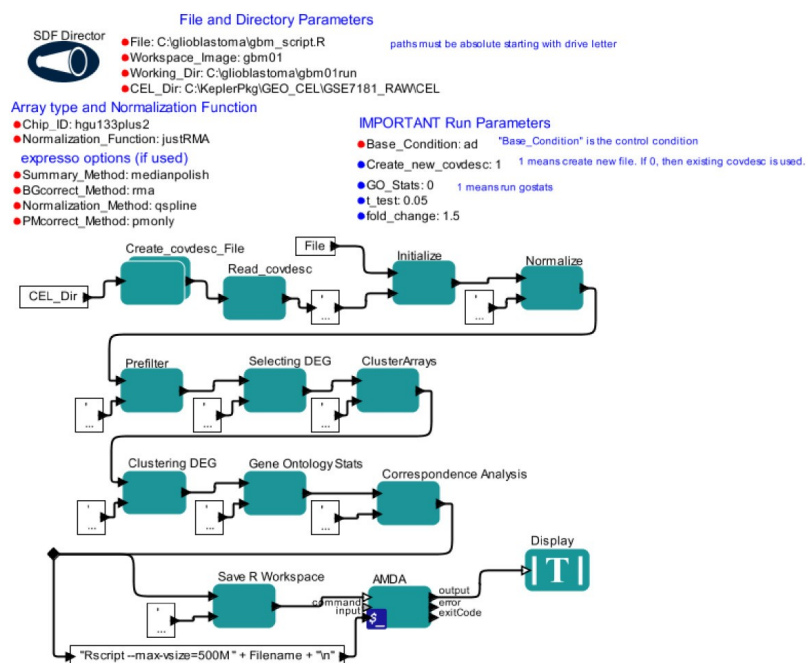


Рисунок В.1. Описание процесса обработки данных в системе Kepler

котором описываются взаимодействующие сущности. Некоторые системы позволяют по созданному описанию генерировать готовые компоненты будущего программного продукта. Так платформа Codebots реализует предметно-ориентированный подход и по составленным UML-диаграммам взаимодействующих сущностей позволяет генерировать API, JSON-схемы данных и документацию[6]. Тем не менее, при реализации сложных вычислительных методов целесообразнее использовать структурный подход.

Одной из ключевых особенностей описанных технологических решений является выделение операций обработки данных в отдельные программные модули (функции, подпрограммы, скрипты). Как правило, при создании описаний алгоритмов в них используется следующий подход. Поскольку известно, что выходные данные одного программного модуля могут являться входными для одного или нескольких других модулей, можно сказать, что между ними формируются зависимости по входным и выходным данным. Тогда возможно составить такой ориентированный граф, описывающий общую логику алгоритма, в котором узлами являются операции обработки данных, а рёбрами – пути данных. Такой подход получил название “диаграммы потоков данных” (англ. Dataflow Diagram, DFD). На рисунке В.2 приведён пример такого ориентированного графа, описывающего

процесс вычисления среднего арифметического и среднего геометрического двух массивов вещественных чисел.

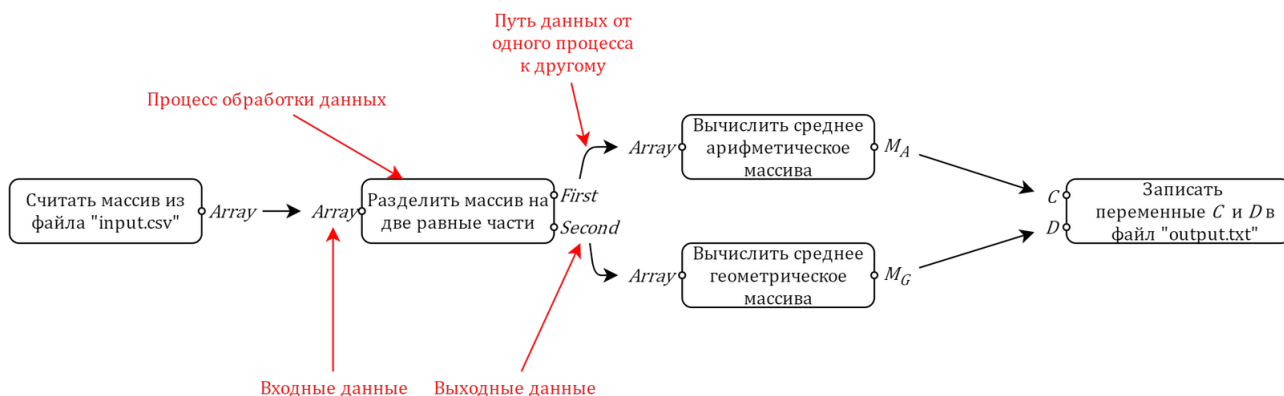


Рисунок В.2. Пример диаграммы потоков данных

При известных входных и выходных данных каждого модуля они могут создаваться независимо друг от друга[7]. Возникает возможность распределения задач создания отдельных программных модулей между разработчиками. Таким образом, уменьшается объём работы по написанию исходных кодов, приходящийся на одного исследователя. Это в свою очередь облегчает отладку и написание документации, что положительно сказывается на общем качестве реализуемого ПО.

В приведённом выше подходе существует необходимость явно описывать входные и выходные данные каждого процесса обработки. При этом на начальных этапах проектирования не всегда данные требуемые данные могут быть полностью определены в силу недостатка представлений о программной реализации тех или иных этапов. Таким образом, в некоторых случаях может быть целесообразен такой подход к построению описания логики реализуемого решения, что в нём не указываются конкретные обрабатываемые данные. Последовательность выполнения отдельных этапов в таком случае должна задаваться явно. В предпринимательстве и управлении проектами подобный подход широко распространён и реализован в сетевых графиках. Сетевой график представляет собой ориентированный граф, в котором вершины – это события или состояния проекта, а рёбра – это работы. В работе [8] рассматривается применение идеи переходов между состояниями при описании логики вычислительных алгоритмов. Описанный подход получил название graph-based software engineering (GBSE). Кроме

того в указанной работе описана реализация GBSE в библиотеке comsdk для языка C++.

Таким образом, объектом разработки является программный каркас для реализации сложных вычислительных методов comsdk. Целью разработки является создание новых программных средств для описания и представления графовых моделей сложных вычислительных методов и их обхода. Для достижения цели были поставлены следующие задачи:

- 1) провести аналитический обзор источников по теме “Системы для реализации сложных вычислительных методов”;
- 2) провести сравнение рассматриваемой разработки с аналогом;
- 3) сформировать требования к алгоритму интерпретации графовых описаний, составленных по методологии GBSE;
- 4) спроектировать структуры данных для описания и представления графовых моделей и их элементов в программном каркасе comsdk;
- 5) разработать алгоритм обхода графовых моделей с использованием спроектированных структур данных;
- 6) реализовать разработанные алгоритмы и структуры данных на языке C++

1 Аналитический обзор

Был проведён сравнительный анализ программного каркаса comsdk с аналогичным программным комплексом, в котором реализованы диаграммы потоков данных. В качестве такой реализации был рассмотрен программный комплекс pSeven, разработанный отечественной компанией DATADVANCE. Он направлен в первую очередь на решение конструкторских, оптимизационных задач и, помимо этого, задач анализа данных, что в первом приближении делает его аналогом comsdk по предметному назначению.

В терминах pSeven: графовое описание процесса решения задачи называется *расчётной схемой* (англ. workflow); вершинам графового описания поставлены в соответствие процессы обработки данных (используется термин *блоки*), а рёбра определяют *связи* между блоками и направления передачи данных между процессами [5]. При работе с pSeven используются следующие понятия:

- *расчётная схема* – формальное описание процесса решения некоторой задачи в виде ориентированного графа;
- *блок* – программный контейнер для некоторого процесса обработки данных, входные и выходные данные для которого задаются через порты (см. ниже);
- *порт* – переменная конкретного¹ типа, определённая в блоке и имеющая уникальное имя в его пределах;
- *связь* – направленное соединение типа “один к одному” между выходным и входным портами разных блоков.

С учётом данных понятий можно описать используемую методологию диаграмм потоков данных следующим образом. Расчётная схема содержит в себе набор процессов обработки данных (блоков), каждый из которых имеет (возможно, пустой) набор именованных входов и выходов (портов). Данные передаются через связи. Для избежания т.н. гонок данных (англ. data races) множественные связи с одним и тем же входным портом не поддерживаются. Для начала выполнения каждому блоку требуются данные на всех входных

¹Динамическая типизация не поддерживается.

портах. Все данные на выходных портах формируются по завершении исполнения блока [5].

Сравнение реализаций двух подходов проводилось по следующим критериям:

- особенности реализуемого подхода,
- особенности программной реализации,
- особенности взаимодействия пользователя с реализованной системой.

С учётом этого были выделены конкретные признаки для сравнения:

- предметное назначение,
- значение вершины графа, описывающего алгоритм,
- значение ребра графа, описывающего алгоритм,
- топология графа, описывающего решение,
- поддержка иерархических графовых описаний, когда одно графовое описание является частью (ребром или вершиной) другого
- принцип передачи данных между отдельными этапами описываемого алгоритма,
- необходимость указывать входные и выходные данные каждого шага алгоритма
- язык программной реализации,
- файловый формат графовых описаний,
- файловая структура проекта реализуемого алгоритма,
- поддерживаемые типы данных,
- принцип ввода входных данных для алгоритма и его параметров,
- принцип вывода результатов работы алгоритма,
- поддержка параллельного выполнения независимых шагов алгоритма,
- поддержка распределённого выполнения отдельных этапов алгоритма на вычислительном кластере,
- наличие графического редактора графовых описаний
- средства визуализации результатов работы алгоритма,

- поддержка алгоритмов, требующих принятия решения от пользователя,
- возможность дополнения набора входных данных во время работы алгоритма.

Результаты проведённого сравнения представлены в таблице 1.

Таблица 1. Сравнительная таблица

№	Признак	pSeven	comsdk
1	Предметное назначение	Задачи оптимизации, анализ данных	Задачи автоматизированного проектирования, алгоритмизация сложных вычислительных методов, анализ данных
2	Значение вершины графа, описывающего алгоритм	Блок (процесс обработки данных)	состояние данных
3	Значение ребра графа, описывающего алгоритм	Связь (направление передачи данных)	переход меду состояниями с указанием функций, осуществляющих переход

4	Топология графа, описывающего решение	<p>По умолчанию поддерживаются только ациклические графы. Поддерживаемая топология расширяется за счёт специальных управляющих блоков, которые отслеживают выполнение условий: для условного ветвления используется блок "Условие" (англ. condition), который перенаправляет данные на один из выходных портов в зависимости от выполнения описанного условия (подробнее см. [9]); Для реализации циклов в общем случае используются блоки "Цикл" (англ. loop) [10], но для некоторых задач существуют специализированные блоки, организующие логику работы цикла (например, блок "Оптимизатор" (англ. optimizer))</p>	Любая
5	Поддержка иерархических графовых описаний	Присутствует	

6	Принцип передачи данных между отдельными этапами описываемого алгоритма	Данные между узлами передаются согласно определенным связям, которые на уровне выполнения создают пространство в памяти для ввода и вывода данных для выполняемых в отдельных процессах блоков. Транзитная передача данных, которые не изменяются в данном блоке, на выход невозможна.	Поскольку узлами графа являются состояния данных, существует возможность задействовать в расчётах только часть данных, оставляя их другую часть неизменной. Фактической передачи данных не производится.
7	Необходимость указывать входные и выходные данные каждого шага алгоритма	Присутствует	Отсутствует
8	Язык программной реализации	C++, Python	

9	Файловый формат графовых описаний	Расчетная схема (в форме орграфа) сохраняется в двоичный файле закрытого формата с расширением .p7wf.	Графовая модель (определяет алгоритм проведения комплексных вычислений в форме орграфа) сохраняется в текстовом файле открытого формата, подготовленного на языке aDOT[11], являющегося “сужением” (частным случаем) известного формата DOT (Graphviz).
10	Файловая структура проекта реализуемого алгоритма	Проект состоит из непосредственно файла проекта, в котором хранятся ссылки на созданные расчётные схемы и локальную базу данных, сами расчётные схемы, файлы с их входными данными, файлы отчётов, где сохраняются выходные данные последних расчётов и результаты их анализа.	Проект состоит из .aDOT файла с описанием графа, .aINI-файлов с описанием форматов входных данных, библиотек функций-обработчиков, функций-предикатов и функций-селекторов, файлов, куда записываются выходные данные.
11	Поддерживаемые типы данных	Целые числа, числа с плавающей точкой, строки, логические переменные, логические, целочисленные и вещественные векторы и матрицы	Целые и вещественные числа, строки, целочисленные и вещественные векторы

12	Принцип ввода входных данных для алгоритма и его параметров	Входные данные должны быть указаны при настройках внешних входных портов расчётной схемы.	Входные данные хранятся в файле в формате aINI[12], откуда считываются при запуске обхода графа [13].
13	Принцип вывода результатов работы алгоритма	Данные с выходных портов схемы сохраняются в локальной базе данных. Для их записи в файлы для обработки/анализа вне pSeven необходимо воспользоваться специально предназначенными для этого блоками.	Для записи выходных/промежуточных данных в файлы или базы данных необходимо добавить соответствующие функции-обработчики. Формат выходных данных не регламентирован.
14	Поддержка параллельного выполнения независимых шагов алгоритма	Присутствует. Блоки, входящие в состав различных ветвлений схемы могут быть выполнены параллельно, поскольку они не зависят друг от друга по используемым данным.	Присутствует. Существует возможность обойти различные ветвления графа одновременно.
15	Поддержка распределённого выполнения отдельных этапов алгоритма на вычислительном кластере	Присутствует	В текущей версии отсутствует

16	Наличие графического редактора графовых описаний	Да	Да ²
17	Средства визуализации результатов работы алгоритма	Реализованы как часть системы формирования отчётов (см. выше)	В текущей версии отсутствуют
18	Поддержка алгоритмов, требующих принятия решения от пользователя	По умолчанию отсутствует. Требуется реализация дополнительных скриптов на языке Python, отвечающих за взаимодействие с пользователем	Частично присутствует засчёт средства генерации форм ввода[13]
19	Возможность дополнения набора входных данных во время работы алгоритма	Отсутствует	Частично реализована при помощи функций-обработчиков специального типа, создающих формы ввода

²В виде отдельного веб-приложения

Таким образом, на данный момент comsdk обладает сравнительно меньшим числом функциональных возможностей, чем современные научные системы управления потоком задач, подобные pSeven, но предоставляет потенциально больше средств для взаимодействия реализуемых алгоритмов с пользователем. В условиях существующей на сегодняшний день потребности в отечественном программном обеспечении для реализации сложных численных методов, актуально развитие данного программного каркаса.

2 Постановка задачи

Для элементов графовой модели в методологии GBSE введены следующие понятия.

Определение 1. Множеством элементарных состояний данных сложного вычислительного метода (CBM) будем называть такое множество \mathbb{W} пар “имя параметра – множество допустимых значений параметра” таких, что $\mathbb{W} = Ind(string) \times \bigcup_{i=0}^{N_T} \{Ind(T_i)\}$, где $Ind(X)$ – оператор индукции, определяющий множество всех возможных значений типа X , $string$ – тип строк в заданном языке программирования и T_i – i -тый произвольный тип заданного языка программирования, N_T – число поддерживаемых типов данных в заданном языке программирования (при поддержке языком пользовательских типов N_T бесконечно).

Элементами множества элементарных состояний CBM могут служить следующие пары:

$$\begin{aligned} s_a &= (\langle RealParam \rangle, \mathbb{R}), \\ s_b &= (\langle IntegerParam \rangle, \mathbb{Z}), \\ s_c &= (\langle MethodName \rangle, Ind(string)), \end{aligned}$$

где $s_a, s_b, s_c \in \mathbb{W}$.

Определение 2. Пространством состояний CBM \mathbb{S} будем называть множество таких подмножеств $S_i \subset \mathbb{W}$, что элементы каждого такого подмножества имеют уникальные имена в рамках этого подмножества, т.е. $\forall s_1, s_2 \in S_i : s_1 \neq s_2 \Rightarrow pr_1(s_1) \neq pr_1(s_2)$, где $pr_1(X)$ – проекция декартова произведения на первую координату соответствующей пары.

Определение 3. Состоянием CBM будем называть элемент $S \in \mathbb{S}$

Определение 4. Пусть $S \in \mathbb{S}$ – состояние CBM. Тогда множество D , такое что $\forall s \in S \exists d \in D : d = (n, v), n = pr_1(s), v \in pr_2(s)$, будем называть *данными сложного вычислительного метода* в состоянии S .

Ниже приведены примеры данных в некоторых состояниях:

$$\begin{aligned} D_1 \multimap S_1 &= \{(\langle RealParam \rangle, 1.0), (\langle StringParam \rangle, \langle Hello, world! \rangle)\}, \\ S_1 &= \{(\langle RealParam \rangle, \mathbb{R}), (\langle StringParam \rangle, Ind(string))\}, \\ D_2 \multimap S_2 &= \{(\langle N \rangle, 5), (\langle M \rangle, 3)\}, \end{aligned}$$

$$S_2 = \{(\ll N \gg, \mathbb{Z}), (\ll M \gg, \mathbb{Z})\}$$

Определение 5. Пусть даны два состояния – S_{in} и S_{out} . Тогда функцией-обработчиком f , переводящей данные из S_{in} в S_{out} , будем называть отображение вида:

$$f : pr_2(s_1) \times pr_2(s_2) \times \cdots \times pr_2(s_n) \rightarrow \\ pr_2(\tilde{s}_1) \times pr_2(\tilde{s}_2) \times \cdots \times pr_2(\tilde{s}_m),$$

где $s_1, s_2, \dots, s_n \in S_{in}$, а $\tilde{s}_1, \tilde{s}_2, \dots, \tilde{s}_m \in S_{out}$, $n = |S_{in}|$, $m = |S_{out}|$. При этом состояние S_{in} назовём *входным* для функции обработчика, а состояние S_{out} – *выходным*.

Для случая, когда на вход функции-обработчику f , ожидающей данные $D \multimap S$, подаются именно такие данные, возможно получение непредсказуемых и неверных результатов в силу отсутствия обработки не учтённых при реализации обработчика исключительных ситуаций. Таким образом, требуются некоторые объекты, позволяющие обрабатывать подобные исключительные ситуации.

Определение 6. Пусть S_{in} – входное состояние для функции-обработчика f . Пусть $\mathbb{D} = \{\{D_i\}_{i=1}^n \mid D_i \multimap S_{in}\}$ – семейство всех возможных данных СВМ в состоянии S_{in} , и есть такое разбиение $S_i \Rightarrow \{S_{true}, S_{false}\}$. Тогда функцией-предикатом с входными состоянием S_{in} будем называть такое отображение $p : \mathbb{D} \rightarrow \{0,1\}$, что существуют два подмножества \mathbb{D}_{true} и \mathbb{D}_{false} такие, что $\mathbb{D}_{true} \cup \mathbb{D}_{false} = \mathbb{D}$, и $\forall D_i \in \mathbb{D}_{true} \Rightarrow p^{-1}(1) = D_i$, где $\mathbb{D}_{true} \multimap S_{true}$, и $\forall D_i \in \mathbb{D}_{false} \Rightarrow p^{-1}(0) = D_i$, где $\mathbb{D}_{false} \multimap S_{false}$.

Определение 7. Функцией перехода будем называть объединённую функцию $F = \langle p, f \rangle$ такую, что её значение получается в соответствии с логикой тренера оператора языка C/C++: $F(D) = p(D)?f(D) : D$.

Поскольку для реализации большей части алгоритмов, требуется проверка условий, для их описания требуется некоторый объект, отвечающий за условное ветвление. В методологии GBSE за него отвечают функции-селекторы.

Определение 8. Пусть S – некоторое состояние СВМ. Пусть $\mathbb{D} = \{\{D_i\}_{i=1}^n \mid D_i \multimap S\}$ – семейство всех возможных данных СВМ в состоянии S

и разбиение $S \Rightarrow \{S_j\}_{j=1}^m$. Тогда *функцией-селектором* будем называть такое отображение $h : \mathbb{D} \rightarrow \mathbb{B}^m$, где $\mathbb{B} = \{0, 1\}$, $\mathbb{B}^m = \times_{k=1}^m \mathbb{B}$, что $\forall D_i \in \mathbb{D} : D_i \multimap S \Rightarrow \exists! b \in \mathbb{B}^m : h^{-1}(b) = D_i \multimap S_j$

Пример работы функции-селектора демонстрирует рисунок 3.

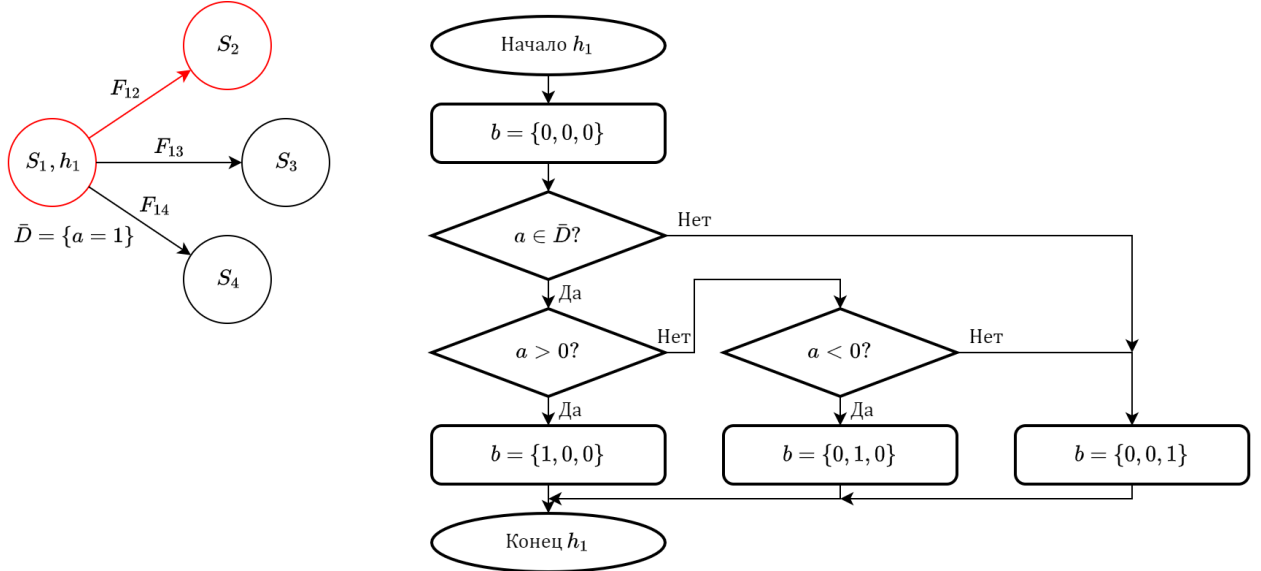


Рисунок 3. Пример фрагмента графовой модели с функцией-селектором

На рисунке 3 красным обозначено ребро, переход по которому будет совершён после вызова функции-селектора.

На основании приведённых выше определений можно ввести понятие графовой модели сложного вычислительного метода.

Определение 9. *Графовой моделью (ГМ) СВМ* Γ будем называть набор объектов $\Gamma = (G, \Sigma, \Phi, H)$, где $G = (N, E)$ – ориентированный граф с множеством вершин V и множество рёбер E , $\Sigma = \{S_j\}_{j=1}^m$ – множество состояний данных алгоритма, $\Phi = \{F_i = \langle p_i, f_i \rangle\}_{i=1}^n$ – множество функций перехода и H – множество функций-селекторов. Каждой вершине ставится в соответствие состояние $S_j \in \Sigma$, в то время, как каждому ребру ставится в соответствие функция перехода $F_i \in \Phi$.

Таким образом, разрабатываемые структуры данных должны реализовать объекты GBSE в соответствии с их определениями.

3 Архитектура программной реализации

Была проведена реорганизация исходного кода программного каркаса, в результате которой было выделено 3 модуля. Текущая структура каталогов проекта представлена на рисунке 4. Были выделены отдельные каталоги

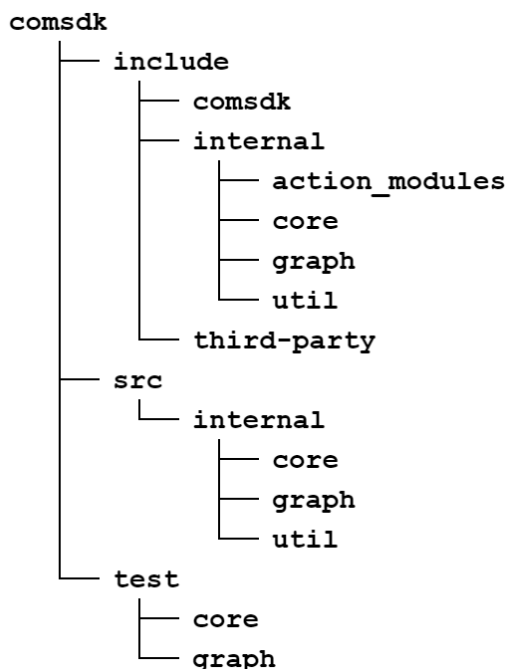


Рисунок 4. Структура каталогов проекта

для заголовочных файлов, где объявлены интерфейсы разработанных структур данных, и для файлов с их реализациями (include/internal и src/internal соответственно). Кроме того, создан отдельный каталог для публичных заголовочных файлов библиотеки include/comsdk, которые будут использоваться конечным пользователем при разработке программных реализаций различных алгоритмов.

При описании спроектированных структур данных использовался язык UML и были учтены возможности стандарта C++-11 языка C++.

3.1 Требования к алгоритму обхода графовых моделей

Методология GBSE подразумевает параллельное независимых ветвей графа. На рисунке 5 после выполнения рёбер F_{12} и F_{13} будет получено два независимых состояния данных S_2 и S_3 соответственно. Далее возникает задача правильным образом перевести данные из этих состояний в общее состояние S_4 .

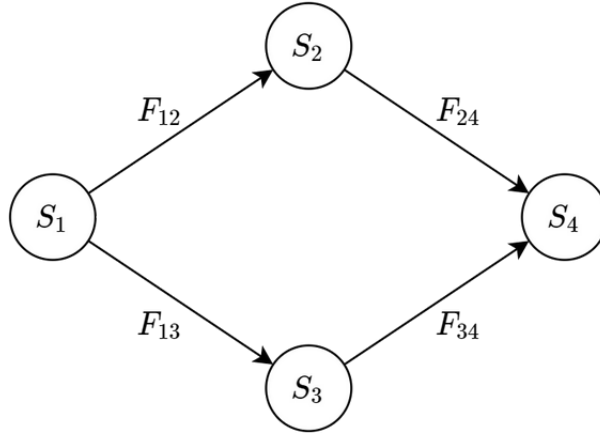


Рисунок 5. Пример графовой модели, требующей параллельного исполнения

Данный подход значительно увеличивает эффективность использования ресурсов вычислительной системы и ускоряет процесс решения, однако добавляет некоторые второстепенные задачи при разработке. Так в примере на рисунке 5 рёбра F_{12} и F_{13} выполнялись параллельно, а значит полученные в результате их выполнения данные существуют в самом общем случае в различных адресных пространствах оперативной памяти (возможно даже на двух разных вычислительных машинах). В момент разветвления графа должно происходить корректное предоставление вычислительным ресурсам (потокам, процессам, узлам кластера и т.п.) доступа к обрабатываемым данным. Помимо этого алгоритм обхода графовой модели должен корректно отрабатывать слияние ветвей графа и в частности при необходимости выполнять сбор данных.

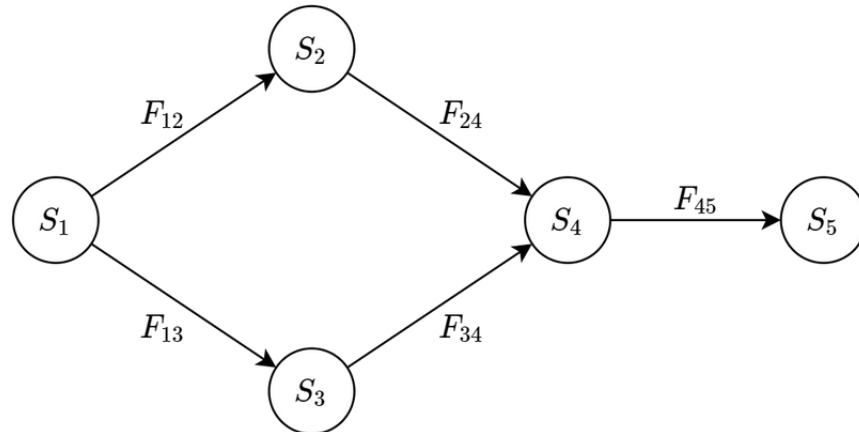


Рисунок 6. Пример графовой модели с совмещением ветвей

На рисунке 6 ветви $S_1 \rightarrow S_2 \rightarrow S_4$ и $S_1 \rightarrow S_3 \rightarrow S_4$ выполняются с использованием различных вычислительных ресурсов, но ребро F_{45} выполняется в пределах одной «общей» ветви графа $S_4 \rightarrow S_5$, и в момент

его выполнения ресурсы, выделенные на выполнение двух параллельных ветвей уже не требуются. Таким образом, целесообразно разработать управляющую структуру, которая бы отвечала за выделение и освобождение вычислительных ресурсов во время работы с несколькими параллельными ветвями графа.

Кроме того, разрабатываемая архитектура должна поддерживать несколько вариантов параллельного исполнения. Среди прочих желательна поддержка:

- поочерёдного выполнения (в первую очередь для отладки) в одном потоке управления;
- выполнения с использованием нескольких процессов операционной системы;
- выполнения с использованием нескольких потоков процессора;
- выполнения на удалённых узлах (через SSH-соединение).

Таким образом целесообразна поддержка единого интерфейса обозначенной управляющей структуры для разных режимов выполнения (последовательный, параллельный, распределённый и пр.).

В случае, когда параллельного выполнения не требуется и предусмотрено условное ветвление, оно должно быть реализовано при помощи специальных функций, привязываемых к узлам графовой модели. В контексте «графоориентированного подхода» такие функции называются *селекторами*. Формально функция-селектор h_i , привязанная к узлу v_i , должна входному набору данных \bar{D} ставить в соответствие множество рёбер E_i , выходящих из v_i , проход по которым нужно совершить.

Алгоритмы обхода графовых моделей, предусматривающие условное ветвление и параллельное выполнение ветвей представлены в разделе 3.1.

3.2 Функциональные структуры данных

Поскольку методология GBSE позволяет во время работы алгоритма загружать функции-предикаты, функции-обработчики и функции-селекторы из стандартных или пользовательских динамических библиотек, требовалась некоторая абстракция вида «функция, загружаемая из динамической

библиотеки». При анализе исходных компонентов библиотеки `comsdk`, было обнаружено, что такая абстракция уже реализована в классе `ActionItem`. Он представляет собой абстракцию над любой функциональной возможностью реализуемой системы. После выполнения `ActionItem` возвращается код, сообщающий об успехе или ошибке. Возможные значения кода были вынесены в отдельный перечислимый тип `ActionRetCode`. Все значения кодов приведены на рисунке 7. Изначально входными данными для `ActionItem` мог быть только объект класса `Anymap` (подробное описание приведено в разделе 4.1), поэтому был разработан шаблон класса для поддержки различных типов входных данных. UML-диаграмма разработанных шаблонов классов представлена на рисунке 7.

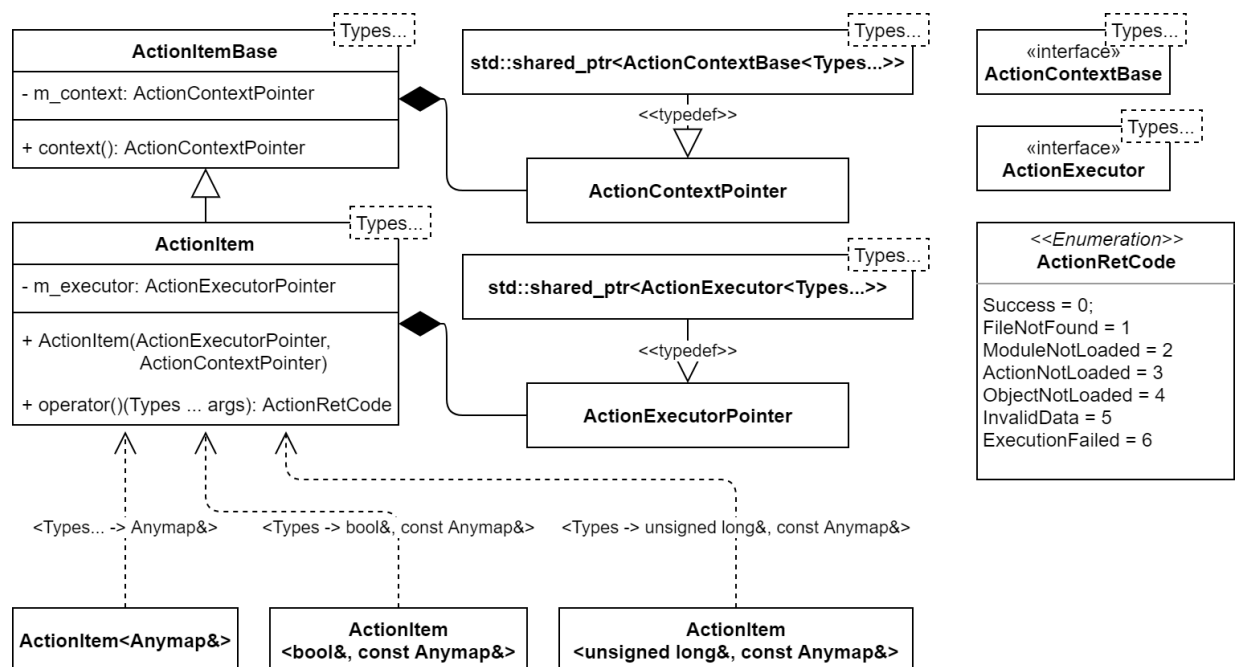


Рисунок 7. Структура классов, описывающих функциональные возможности системы

Запуск функциональной возможность осуществляется посредством перегруженного оператора вызова. Абстрактный класс `ActionContextBase` представляет собой интерфейс для объектов, отвечающих за запуск и хранение мета-информации о выполнении различных `ActionItem`. Базовый класс `ActionExecutor` представляет собой интерфейс для объектов, отвечающих за синхронное выполнение `ActionItem`. В исходной версии `comsdk` была реализация этого интерфейса, позволяющая выполнять загружаемые из динамических библиотек функции – класс `SharedLibFuncExecutor`.

Для реализации обработчиков, предикатов и селекторов были созданы специализации шаблона `ActionItem`, которые были использованы в качестве базовых классов для реализуемых в дальнейшем функциональных структур данных.

Функция-предикат должна принимать на вход данные в начальном состоянии и ставить им в соответствие логические 0 или 1. Поскольку результат выполнения `ActionItem` – это числовой код `ActionRetCode`, то к входным данным помимо ссылки на объект класса `Anymap` была добавлена ссылка на переменную типа `bool`, куда будет сохраняться результат работы предиката. UML-диаграмма класса предиката, представлена на рисунке 8.

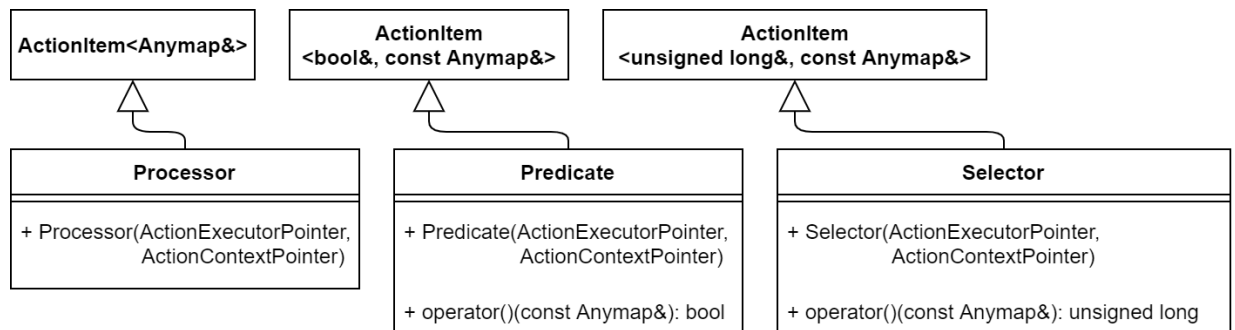


Рисунок 8. UML-диаграмма классов, описывающих функции, выполняемые при обходе графовой модели

Была добавлена обёртка вокруг оператора вызова базового класса, которая возвращает переменную типа `bool`. Это даёт предикату интуитивно понятный интерфейс. В случае возникновения непредвиденной ошибки должно быть возвращено значение `false`.

Функция-обработчик отвечает за преобразование данных. Ей на вход требуется только ссылка на объект класса `Anymap`, содержащий обрабатываемые данные. Возвращаемым значением такой функции является код ошибки. В текущей версии класс `Processor` не расширяет базовый класс `ActionItem<Anymap&>`, но предоставляет возможность расширения в будущем. UML-описание класса `Processor` представлено на рисунке 8.

Класс `Selector` должен реализовать функциональность, описанную в разделе 3.1. Для описания множества рёбер, исходящих из текущей вершины, которые должны быть выполнены, были выбраны битовые маски, хранимые в одном беззнаковом целом числе (`long unsigned int`), где 1 в i -том справа разряде обозначает, что i -тое исходящее из вершины ребро

должно быть выполнено. Это позволяет значительно экономить память, но ограничивает максимально возможное число исходящих рёбер разрядностью беззнаковых целых чисел. Для удобства была создана обёртка над базовым оператором вызова, которая возвращает битовую маску, а не код ошибки. При возникновении исключительной ситуации должна быть возвращена битовая маска, состоящая из нулей. UML-описание класса `Selector` представлено на рисунке 8.

Задачей функции перехода является перевод данных из одного состояния в другое. Перед выполнением перехода должна выполняться валидация данных с помощью функции-предиката. В случае успеха должна вызываться функция-обработчик. Более наглядное представление логики функции перехода представлено на рисунке 9.

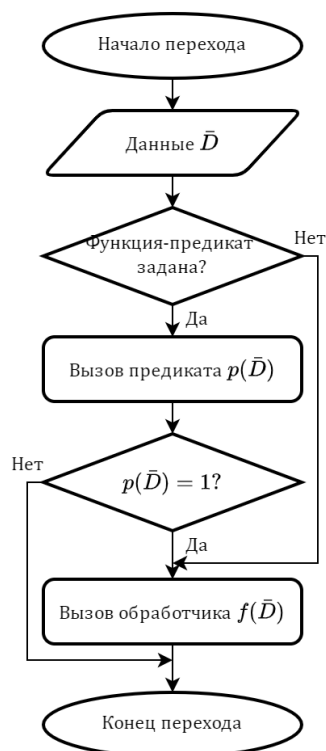


Рисунок 9. Блок-схема работы функции перехода

Таким образом, структура данных, описывающая функцию перехода, должна содержать в себе и функцию-предикат, и функцию-обработчик. UML-диаграмма разработанной структуры данных представлена на рисунке 10.

Класс `Transfer` хранит в себе “умные” указатели на объекты классов `Predicate` и `Processor`. В него встроены методы проверки наличия предиката и

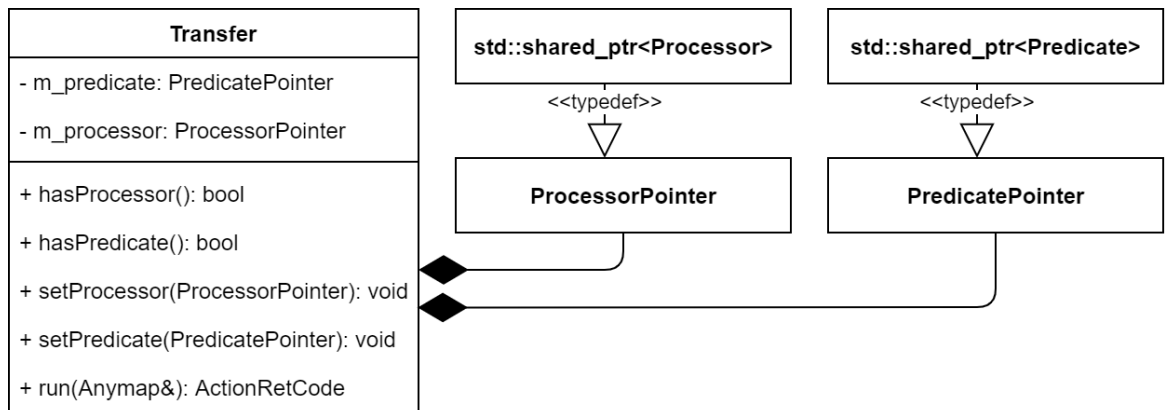


Рисунок 10. UML-описание класса, реализующего функции перехода

обработчика – `hasPredicate()` и `hasProcessor()` соответственно – и их задания – `setPredicate()` и `setProcessor()` соответственно. Предполагается, что переход будет совершён даже в случае, если функция-предикат не задана. Тогда обработка данных будет выполнена в любом случае без предварительной валидации.

Таким образом, разработанные в этом разделе структуры данных необходимы для реализации логики обхода графовых моделей.

3.3 Информационные структуры данных

В данном разделе представлены спроектированные структуры данных, отвечающие за представление графовых моделей и их элементов.

3.3.1 Класс вершины графовой модели

Был разработан класс `Node` для представления узла графовой модели. Его UML-диаграмма приведена на рисунке 11.

Класс `Node` хранит в себе имя вершины, режим параллельного обхода нескольких ветвей, выходящих из данной вершины и “умный” указатель на объект класса `Selector`, описанный в разделе 3.2. Режим параллельного обхода описан при помощи переменной перечислимого типа `ParallelMode`. В случае, когда логика алгоритма предполагает условное ветвление в текущей вершине, а не параллельное выполнение выходящих из неё вершин, поле `m_parmode` должно иметь значение `NoParallel`. Ситуация, когда функция-селектор в вершине с режимом параллельного обхода `NoParallel` возвращает битовую маску, имеющую больше одной единицы, считается исключительной.

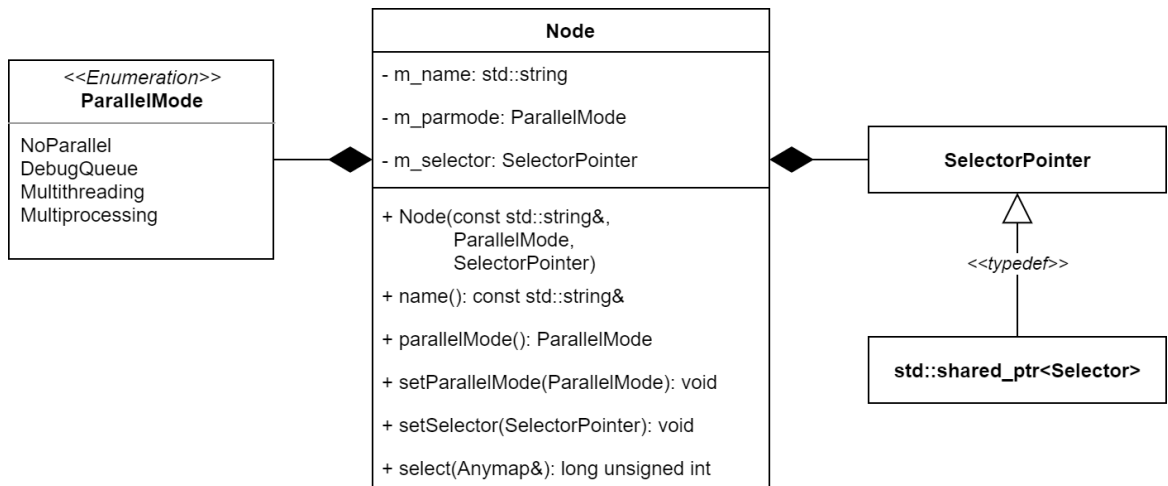


Рисунок 11. UML-описание структуры данных, описывающей вершину графовой модели

Класс **Node** имеет в себе метод для доступа к имени вершины, имеет методы для задания режима параллельного исполнения (**setParallelMode()**) и функции-селектора (**setSelector()**). Кроме того, в нём описан метод вызова привязанной функции-селектора с выбранным набором данных **select()**. В случае, если функция-селектор не задана, метод **select()** должен возвращать битовую маску из всех единиц, обозначая, что все исходящие из вершины рёбра должны быть выполнены.

3.3.2 Класс ребра графовой модели

С рёбрами графовой модели в методологии GBSE связаны функции перехода. При этом должна быть реализована возможность в одном ребре описать три этапа обработки данных:

1. подготовка данных;
2. непосредственная обработка данных;
3. пост-обработка данных.

При этом ребро должно иметь хотя бы этап обработки, когда пре- и пост- процессор являются необязательными. При проектировании структуры данных ребра графовой модели эта возможность была учтена. Описание спроектированной структуры данных приведено на рисунке 12.

Конструктор класса **Edge** принимает на вход три “умных” указателя (**std::shared_ptr**) на объекты класса **Transfer**. Они обозначают обработчик,

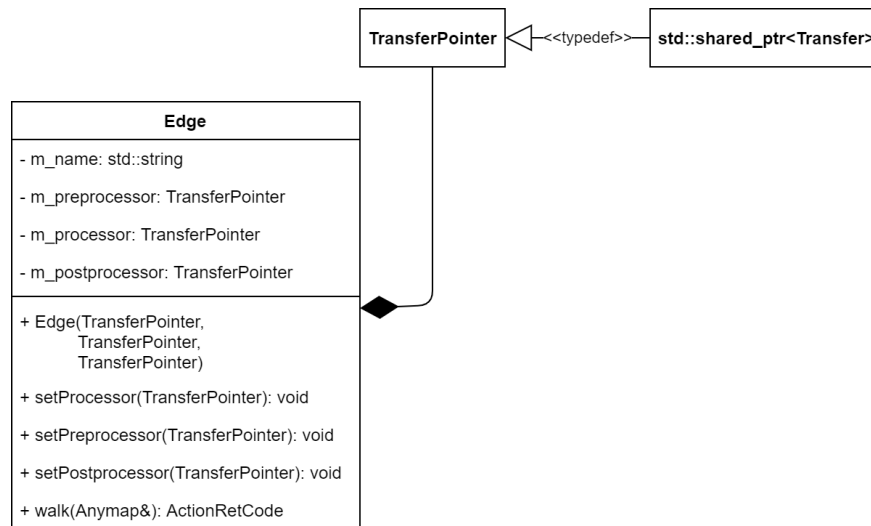


Рисунок 12. UML-описание структуры данных, описывающей ребро графовой модели

препроцессор и постпроцессор. Два последних аргумента имеют значение по умолчанию `nullptr` и являются необязательными. Класс **Edge** имеет методы проверки наличия в нём обработчика, пре- и постпроцессора и методы для их назначений. Проход по ребру и запуск хранимых функций в установленном порядке осуществляется при помощи метода `walk()`.

3.3.3 Класс графовой модели

Согласно определению, данному в разделе 2, графовая модель вычислительного метода должна хранить в себе множество вершин и множество рёбер. Задача их хранения была отведена непосредственно классу графовой модели (**Graph**). Поскольку основной операцией с узлами и рёбрами является обращение к ним по их индексу, в качестве контейнеров для них были выбраны динамические массивы (`std::vector`). Кроме того, данному классу была отведена задача хранить данные о топологии графовой модели в виде матрицы смежности. Данное решение позволяет легко проверять наличие ребра между двумя узлами и, кроме того, даёт возможность сразу узнать индекс ребра при его существовании. Поскольку граф должен формироваться за счёт поочерёдного добавления в него узлов и рёбер, в качестве контейнера для этой матрицы был взят двумерный динамический массив. UML-диаграмма разработанного класса **Graph** представлена на рисунке 13.

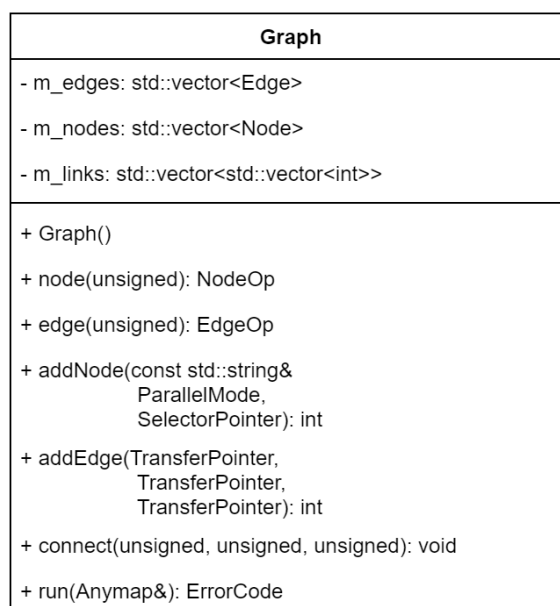


Рисунок 13. UML-диаграмма структуры данных, описывающей графовую модель алгоритма

Класс **Graph** имеет в себе методы для получения вершин и рёбер графовой модели по их индексам – **node()** и **edge()** соответственно. Данные методы возвращают специальные структуры данных, предназначенные в первую очередь для получения данных о топологии графовой модели. Более подробное их описание приведено в разделе 3.3.4. Метод **addNode()** добавляет к графовой модели вершину с заданным именем, режимом параллельного исполнения и селектором и возвращает индекс этой вершины. В случае, если по какой-то причине вершину добавить не удалось, будет возвращено значение -1. Метод **addEdge()** добавляет в графовую модель ребро с заданными обработчиком, пре- и пост- процессором и возвращает индекс добавленного ребра. В случае, если ребро создать не удалось, аналогично методу **addNode()** будет возвращено значение -1. Метод **connect()** отвечает за изменение матрицы смежности и соединение вершин. Его аргументами являются индекс начальной вершины, индекс конечной вершины и индекс соединяющего их ребра. Запуск обхода графовой модели с заданными входными данными в формате **AnyMap** осуществляется при помощи метода **run()**.

3.3.4 Классы операций с вершинами и рёбрами графовой модели

В процессе обхода графовой модели для каждой вершины требуются сведения об исходящих из неё рёбрах. Для каждого ребра требуются сведения о его конечной вершине. Эти сведения могут быть получены при обращении к матрице смежности графа. Кроме того, в процессе обхода требуется обращение к методам классов `Node` и `Edge`. При этом сами эти классы не поддерживают обращение к матрице смежности графа. Поэтому для процедуры обхода были разработаны специальные промежуточные структуры данных являющихся облегчёнными представлениями вершин и рёбер. Их UML-диаграммы представлены на рисунке 14.

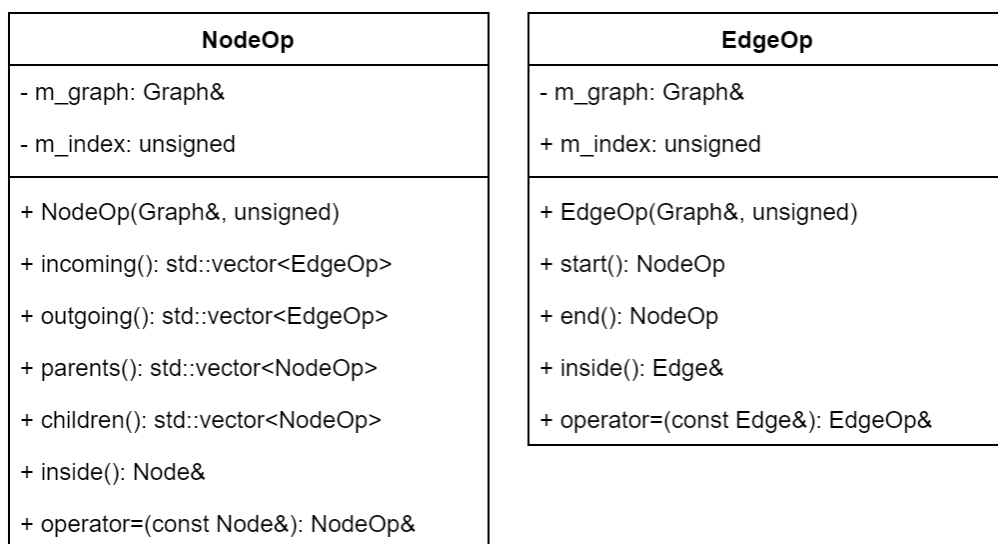


Рисунок 14. UML-диаграммы вспомогательных структур данных, используемых при обходе

Полями классов `NodeOp` и `EdgeOp` являются ссылка на объект графовой модели и индекс вершины и ребра соответственно. Это позволяет избежать копирования информации из вершин и рёбер, но при этом всё ещё позволяет обращаться к их интерфейсам через их индексы. Классы `NodeOp` и `EdgeOp` являются дружественными классу `Graph`, и поэтому могут обращаться к контейнерам рёбер и вершин и матрице смежности напрямую.

В классе `NodeOp` определены методы для получения множества входящих (`incoming()`) и исходящих (`outgoing()`) рёбер из данной вершины. Кроме того, определён метод `parents()`, позволяющий получить множество вершин, соединённых с данной входящими в неё рёбрами. Аналогично метод

`children()` позволяет получить множество вершин, соединённой с данной выходящими из неё рёбрами. Метод `inside()` даёт доступ к интерфейсу вершины графовой модели, индекс которой хранится в поле `m_index`.

В классе `EdgeOp` определены методы для получения начальной (`start()`) и конечной (`end()`) вершин для данного ребра. Аналогично классу `NodeOp` метод `inside()` даёт доступ к интерфейсу ребра графовой модели, индекс которого хранится в поле `m_index`.

Использование в качестве контейнера для множеств динамического массива `std::vector` позволяет легко обращаться к его элементам по индексам и итерироваться по нему. Это полезное свойство было задействовано при реализации алгоритма обхода.

Таким образом, описанные в данном разделе структуры данных позволяют формировать программное представление графовых моделей алгоритмов и взаимодействовать с ними в процессе их обхода.

Общая UML-диаграмма разработанных функциональных и информационных структур данных представлена на рисунке 15.

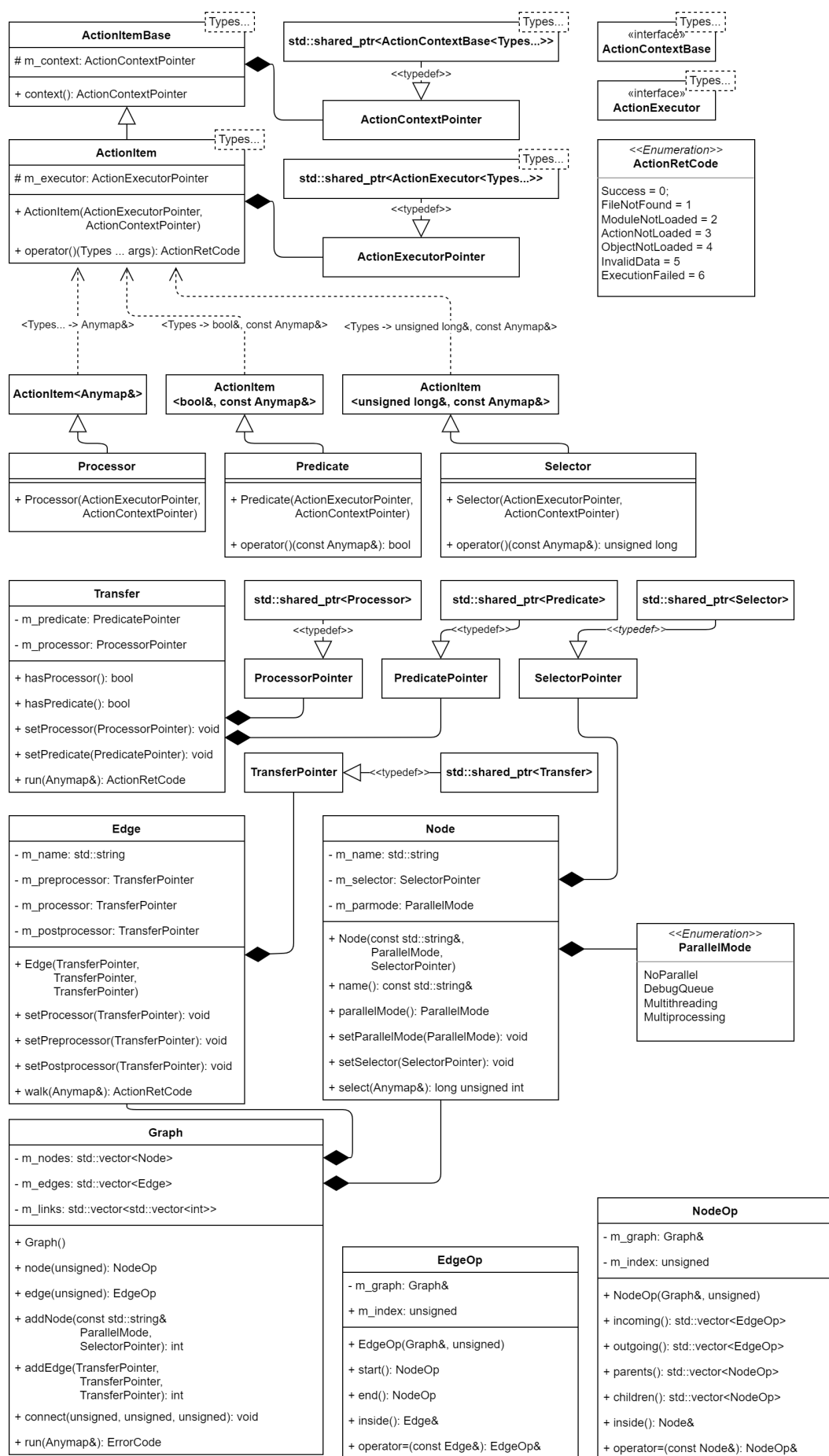


Рисунок 15. Общая UML-диаграмма информационных и функциональных структур данных

3.4 Управляющие структуры данных

В данном разделе описаны структуры данных, использующиеся при обходе графовой модели алгоритма. Основной структурой данных, поддерживающей разработанный алгоритм обхода (описан в разделе 4.2) является структура данных «контейнер выполнения». Задача её внутренней логики – контролировать параллельный обход нескольких ветвей графа и отслеживать их слияние. Помимо прочего это подразумевает выделение и освобождение вычислительных ресурсов для параллельного обхода. Поскольку в общем случае могут применяться различные вычислительные ресурсы (потоки ядер процессора, процессы операционной системы, узлы вычислительного кластера и пр.), было принято решение разработать единый абстрактный интерфейс структуры «контейнер выполнения» без привязки к конкретным ресурсам. Это позволит независимо разрабатывать различные реализации данной структуры для различных режимов параллельного обхода.

Помимо прочего при проектировании интерфейса структуры данных «контейнер выполнения» были задействованы разработанные информационные структуры данных «операция с вершиной» (NodeOp) и «операция с ребром» (EdgeOp).

Интерфейсы спроектированных структур данных были описаны с помощью UML-диаграмм, представленных на рисунке 16.

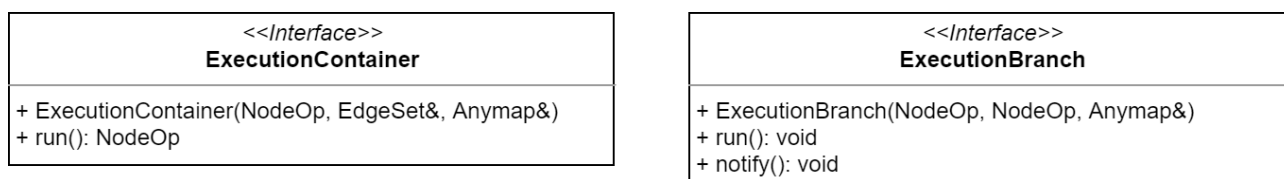


Рисунок 16. UML-диаграммы управляющих структур данных

Конструктор класса ExecutionContainer соответствует алгоритму, описанному в разделе 4.2. Предполагается, что различные реализации метода run() будут реализовать алгоритм, описанный блок-схемой на рисунке 18, для разных вариантов используемых вычислительных ресурсов.

Кроме того, был разработан интерфейс структуры «ветви исполнения» (ExecutionBranch на рисунке 15). Предполагается, что реализации её виртуального метода run() будут выполнять обход одной

конкретно взятой ветви графовой модели с использованием тех или иных вычислительных ресурсов. При этом планируется уведомление «контейнера исполнения» о завершении каждой функции перехода (метод `notify()`), как того требует алгоритм.

Таким образом, разработанные классы реализуют в себе логику параллельного обхода ветвей графовой модели и хранят данные в соответствии с разработанной процедурой обхода.

4 Описание реализованных средств

4.1 Используемые технологии

В процессе разработки компонентов новой версии программного каркаса `comsdk` были использованы некоторые его ранее созданные компоненты. Так, среди прочих, был использован класс для представления данных в виде универсального ассоциативного массива `Anymap`. В данный ассоциативный массив записываются входные данные для алгоритма из файла формата `.aINI`[12]. Данный класс позволяет хранить разнотипные данные в одном объекте с использованием строковых ключей для доступа к ним. Кроме того, в данном классе реализована возможность экспорта данных в файл в формате `aINI`.

Кроме того, был использован уже реализованный инструментарий для загрузки функций из скомпилированных динамических библиотек и интерфейс функциональной возможности `ActionItem`.

Реализация проводилась с использованием стандарта языка `C++` `C++-11` и стандартной библиотеки шаблонов (англ. `Standard Template Library (STL)`).

4.2 Алгоритмы

Для определения общей логики обхода графовой модели без учёта возможности обхода параллельных ветвей был разработан упрощённый алгоритм, поддерживающий только условное ветвление. Его блок-схема представлена на рисунке 17.

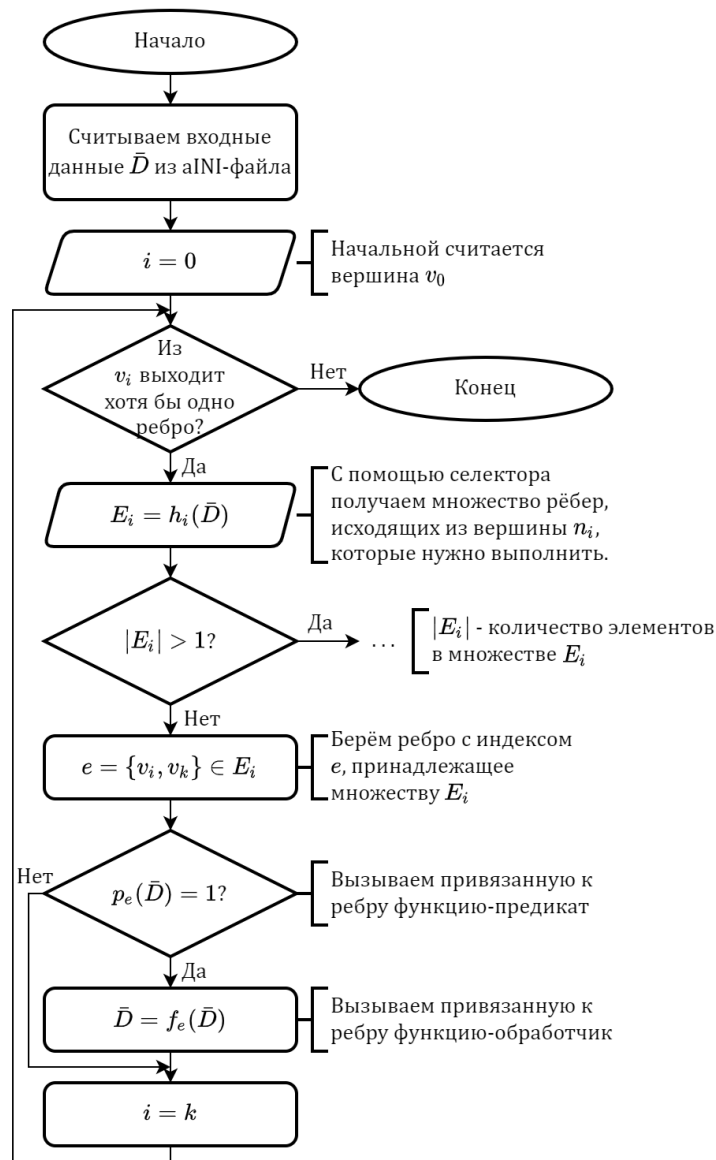


Рисунок 17. Блок-схема алгоритма обхода графовой модели, не предполагающей параллельное исполнение

Был отдельно рассмотрен случай, когда необходимо параллельно обойти несколько ветвей. Для контроля за параллельным исполнением функций перехода и выполнения необходимых операций по распределению и сбору данных с задействованных вычислительных ресурсов была разработана управляющая структура «контейнер выполнения». Более подробно она описана в разделе 3. Логика работы с данной структурой представлена на рисунке 18.

При этом внутри управляющей структуры «контейнер выполнения» подразумевается создание отдельных «ветвей».

Алгоритм обхода одной ветви представлен на рисунке 19.

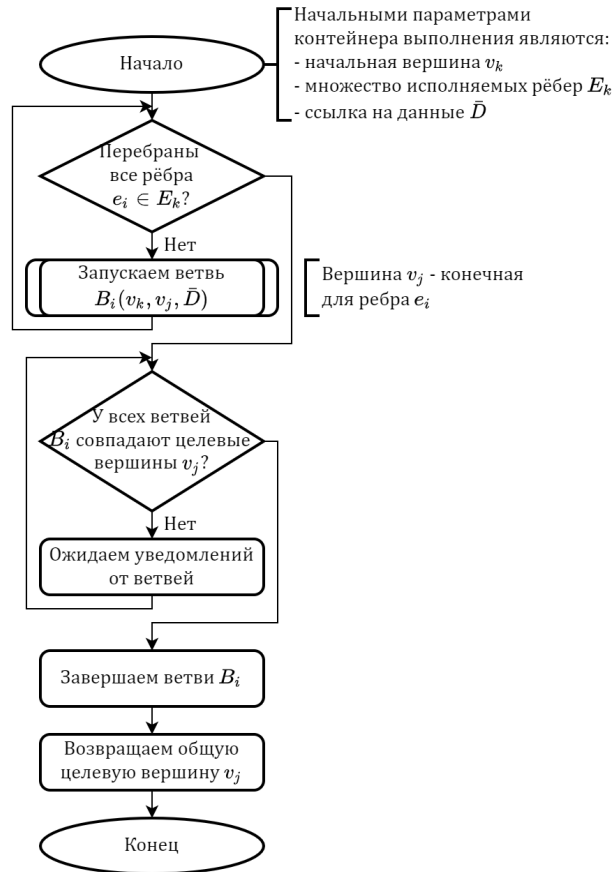


Рисунок 18. Блок-схема алгоритма отслеживания параллельного исполнения ветвей графа

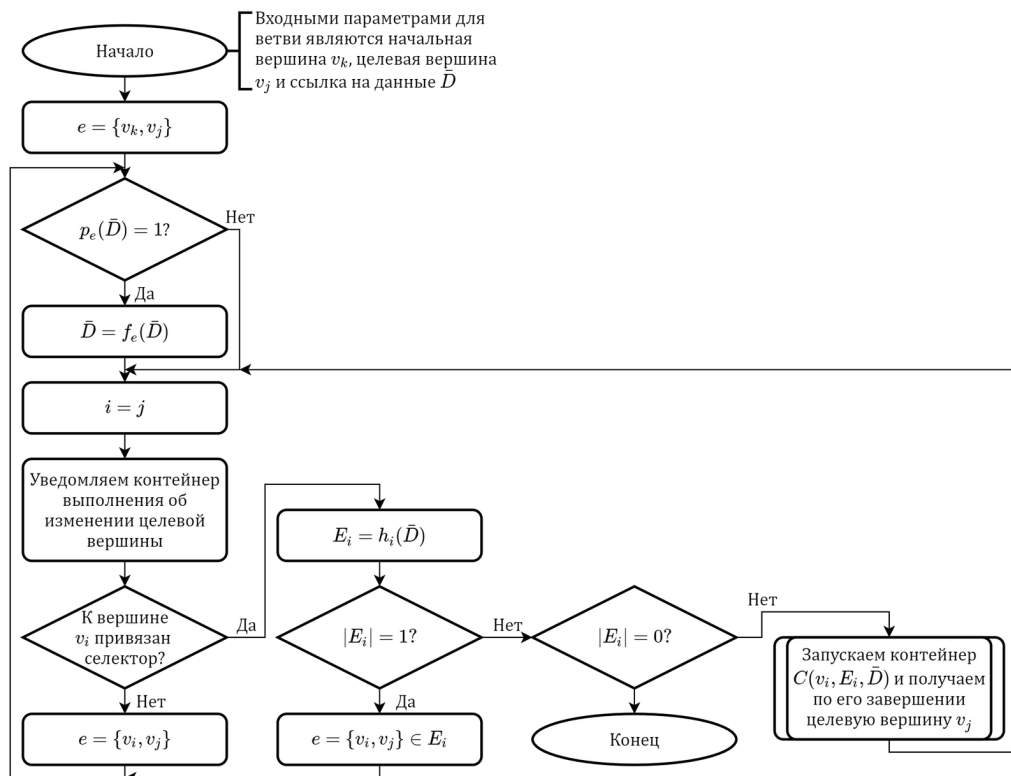


Рисунок 19. Блок-схема алгоритма исполнения одной из параллельных ветвей

Общий алгоритм предполагает завершение выполнения ветви по сигналу от «контейнера выполнения». Ситуация, когда по какой-то причине для текущей вершины v_i и данных \bar{D} не было выбрано ни одного ребра, является исключительной и должна обрабатываться отдельно.

Разработанная логика параллельного обхода ветвей графа была добавлена к исходному алгоритму. Итоговая блок-схема разработанного алгоритма обхода представлена на рисунке 20.

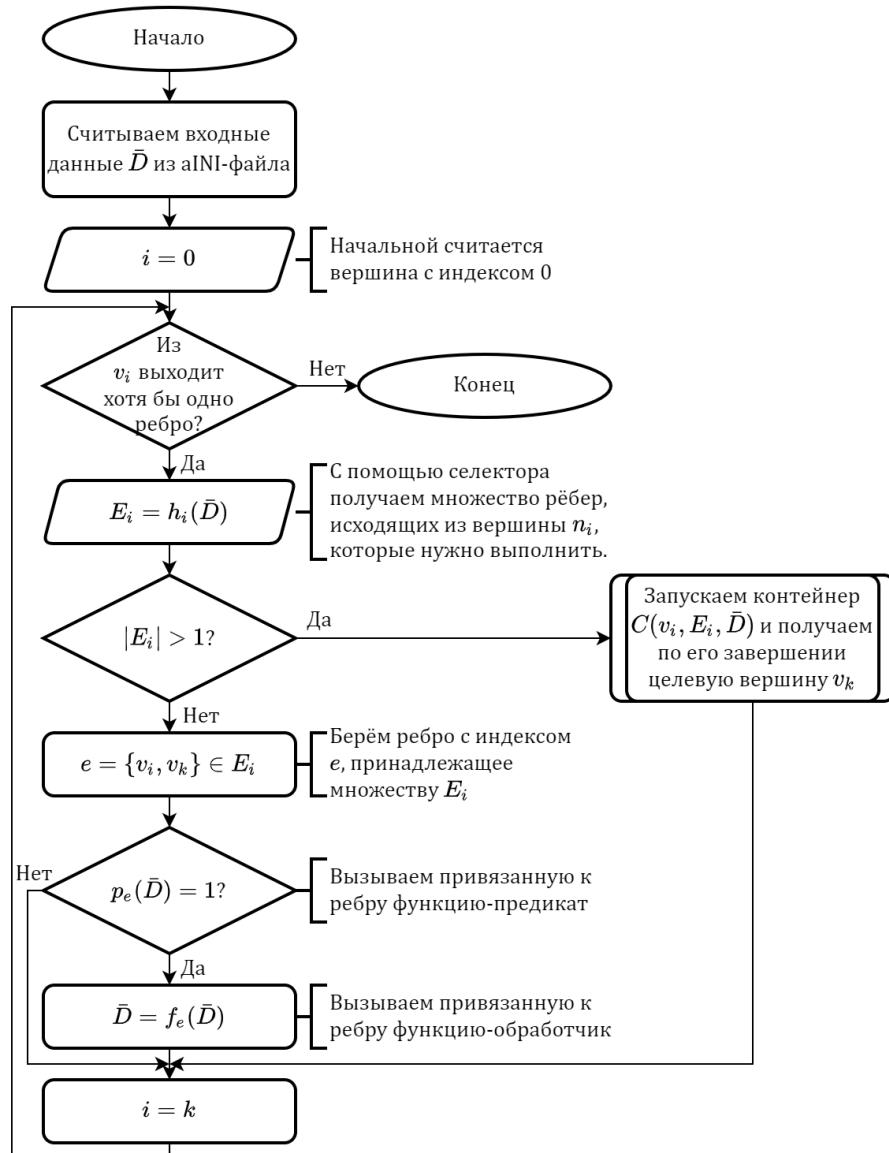


Рисунок 20. Блок-схема алгоритма обхода графовой модели

4.3 Сборка и тестирование

Для сборки разработанных программных средств требуется библиотека Boost, компилятор языка C++, поддерживающий стандарт

C++-11, и система сборки CMake. При разработке и отладке использовался компилятор gcc и библиотека Boost версии 1.78.

Для реализованных компонентов были разработаны unit-тесты для проверки корректности их внутренней логики. Разработанные юнит-тесты были размещены в директории comsdk/test (см. рисунок 4). Для тестирования разработанных средств необходимо в корневой папке проекта создать директорию build, перейти в неё и выполнить в ней следующую последовательность команд:

```
cmake -DCMAKE_BUILD_TYPE=Release ..
```

```
make -j4,
```

где флаг CMAKE_BUILD_TYPE отвечает за тип сборки проекта.

Возможные типы:

- Debug – файлы библиотеки comsdk, библиотеки для тестов и бинарные файлы тестов будут размещены в директории build-debug;
- Release – файлы библиотеки comsdk, библиотеки для тестов и бинарные файлы тестов будут размещены в директории build-release.

После этого необходимо запустить утилиту CTest, которая использовалась для прогона юнит-тестов.

ЗАКЛЮЧЕНИЕ

Таким образом, в ходе выполнения данной работы были выполнены следующие задачи:

- 1) в ходе аналитического обзора литературы и сравнительного анализа разработки с аналогичной обоснована её актуальность;
- 2) обозначены теоретические основы подхода GBSE к построению описаний алгоритмов;
- 3) сформулированы требования к алгоритму интерпретации описаний, построенных по методологии GBSE;
- 4) спроектированы структуры данных, отвечающие за программное представление описаний алгоритмов и их элементов в программном каркасе comsdk;
- 5) спроектированные структуры данных были реализованы на языке C++.

В результате разработки:

- 1) была повышена безопасность существующих элементов программного каркаса comsdk;
- 2) программная реализация графовых моделей была приближена к их концептуальному определению в методологии GBSE;
- 3) были разработаны интерфейсы, позволяющие дальнейшее расширение функциональных возможностей программного каркаса;
- 4) были расширены возможности документирования реализуемых при помощи программного каркаса comsdk алгоритмов.

Перспективы дальнейшей разработки включают в себя помимо прочего следующее:

- повышение надёжности и быстродействия разработанных средств;
- реализация алгоритма параллельного обхода для различных вариантов задействованных вычислительных ресурсов;
- разработка и интеграция средств взаимодействия с пользователем;
- интеграция разработанных программных средств в распределённую вычислительную систему GCD.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 The design of Maple: A compact, portable, and powerful computer algebra system / Bruce W. Char, Keith O. Geddes, W. Morven Gentleman [и др.] // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 1983. T. 162 LNCS. С. 101 – 115.
- 2 Workflows and e-Science: An overview of workflow system features and capabilities / D. E., G. D., S. M. et al. // Future Generation Computer Systems. 2009. Vol. 25, no. 5. P. 528 – 540.
- 3 Pegasus in the cloud: Science automation through workflow technologies / Deelman E., Vahi K., Rynge M. [и др.] // IEEE Internet Computing. 2016. T. 20, № 1. С. 70 – 76.
- 4 Kepler: An extensible system for design and execution of scientific workflows / Altintas I., Berkley C., Jaeger E. [и др.]. T. 16. 2004. С. 423 – 424.
- 5 Alexey M. Nazarenko Alexander A. Prokhorov. Hierarchical Dataflow Model with Automated File Management for Engineering and Scientific Applications // Procedia Computer Science. 2015. T. 66. URL: <https://www.sciencedirect.com/science/article/pii/S1877050915034055?pes=vor>.
- 6 Low-code development and model-driven engineering: Two sides of the same coin? / Davide Di Ruscio, Dimitris Kolovos, Juan de Lara [и др.] // Software and Systems Modeling. 2022. T. 21, № 2. С. 437 – 446.
- 7 Данилов А.М., Лапшин Э.В., Беликов Г.Г., Лебедев В.Б. Методологические принципы организации многопоточной обработки данных с распараллеливанием вычислительных процессов // Известия вузов. Поволжский регион. Технические науки. 2001. № 4. С. 26–34.
- 8 Соколов А.П. Першин А.Ю. Графоориентированный программный каркас для реализации сложных вычислительных методов // Программирование. 2018. № X.
- 9 Condition - pSeven 6.31.1 User Manual [Электронный ресурс] [Офиц. сайт]. 2022. (дата обращения 07.03.2022). URL:

<https://www.datadvance.net/product/pseven/manual/6.31.1/blocks/Condition.html>.

10 Расчётные схемы - Руководство пользователя pSeven 6.27 [Электронный ресурс] [Официальный сайт]. 2021. Дата обращения: 15.11.2021. URL: <https://www.datadvance.net/product/pseven/manual/ru/6.27/workflow.html#workflow-links>.


11 Соколов А.П. Першин А.Ю. Описание формата данных aDOT (advanced DOT). 2020.

12 Соколов А.П. Описание формата данных aINI (advanced INI) [Электронный ресурс]. Облачный сервис SA2 Systems. [Официальный сайт]. 2020. URL: <https://sa2systems.ru/nextcloud/index.php/f/403527>.

13 Соколов А.П. Першин А.Ю. Программный инструментальный для создания подсистем ввода данных при разработке систем инженерного анализа // Программная инженерия. 2017. Т. 8, № 12. С. 543–555.

Выходные данные

Тришин И.В.. Разработка компонентов графоориентированного программного каркаса для реализации сложных вычислительных методов по дисциплине «Модели и методы анализа проектных решений». [Электронный ресурс] — Москва: 2022. — 54 с. URL: <https://sa2systems.ru>: 88 (система контроля версий кафедры РК6)

Постановка:  канд. физ.-мат. наук, Соколов А.П.

Решение и вёрстка:  студент группы РК6-81Б, Тришин И.В.

2022, весенний семестр

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)

АКТ
проверки выпускной квалификационной работы

Студент группы РК6-81Б

Фамилия Имя Отчество
(Фамилия, имя, отчество)

Тема выпускной квалификационной работы: [Тема]

Выпускная квалификационная работа проверена, размещена в ЭБС «Банк ВКР» в полном объеме и соответствует / не соответствует требованиям, изложенным в Положении о порядке подготовки и защиты ВКР.
ненужное зачеркнуть

Объем заимствования составляет % текста, что с учетом корректного заимствования соответствует / не соответствует требованиям к ВКР
ненужное зачеркнуть

← от руки

↑ бакалавра, специалиста, магистра
от руки

Нормоконтролёр

Согласен:

Студент

Дата:

↑ от руки

↑ подписать
С.В. Грошев
(подпись) (ФИО)

↑ подписать
И.О. Фамилия
(подпись) (ФИО)

A