

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

Направление подготовки: 09.03.01 «Информатика и вычислительная техника»
Профиль подготовки: «Вычислительные машины, комплексы, системы и сети»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
БАКАЛАВРА

на тему

**ИНТЕРПРЕТАТОР ДИАГРАММ
ГРАФИЧЕСКИХ ЗАПРОСОВ К RDF-
ХРАНИЛИЩАМ ДАННЫХ**

Студент _____ 15.06.2023 Кобзев Максим Игоревич
(подпись, дата) (ФИО полностью)

Руководитель _____ 16.06.2023 Дубинин В. Н.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ 16.06.2023 Бычков А. С.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 02.06.23 г. № 14)
Заведующий кафедрой _____ И.А. Митрохин
(подпись) (фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК от
20.06.2023, № 22)
Секретарь ГЭК _____ М. Гурии
(подпись) (фамилия, инициалы)

Пенза 2023

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»
«Утверждаю»
Заведующий кафедрой
М.А. Митрохин
«26» апреля 2023

**ЗАДАНИЕ
НА БАКАЛАВРСКУЮ РАБОТУ**

1. Студент Кобзев Максим Игоревич гр. 19ВВП1 факультета вычислительной техники

2. Тема работы «Интерпретатор диаграмм графических запросов к RDF-хранилищам данных».

Тема утверждена приказом ПГУ № 458/о от "26" апреля 2023г.

3. Руководитель бакалаврской работы д.т.н. профессор кафедры ВТ ПГУ Дубинин В.Н.

4. Задание на работу

Разработать модель выполнения функционально-блочных диаграмм для формирования и обработки запросов к RDF-хранилищам данных на основе принципа потоков данных. Трансляционная семантика базисных функциональных блоков должна основываться на языке JavaScript. Композитный функциональный блок должен выполняться как контейнер. Графическая диаграмма запроса должна транслироваться в текстовый SPARQL-запрос, выполняемый на сервере.

Разработать метод объяснения решений на основе обратного поиска в вычисленной функционально-блочной диаграмме.

Программно реализовать интерпретатор функционально-блочных диаграмм и систему объяснения решений.

Язык программирования – C/C++

Среда программирования – Qt.

5. Перечень подлежащих разработке вопросов

- 1) Обзор и анализ визуальных языков и графических редакторов
- 2) Разработка графического языка функционально-блочных диаграмм для описания и обработки SPARQL-запросов
- 3) Разработка графического подязыка описания SPARQL-запросов
- 4) Разработка и программная реализация графического редактора

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	О бъем работ ы	Срок выполнения	Подпи сь руководите ля
1	Обзор и анализ интерпретируемых графических языков программирования и моделирования	15%	26.04.23- 29.04.23	
2	Разработка модели выполнения функционально-блочных диаграмм	20%	30.04.23- 6.05.23	
3	Разработка способа объяснения результатов вычислений.	20%	7.05.23- 13.05.23	
4	Программная реализация интерпретатора функционально-блочных диаграмм и системы объяснения решений	25%	14.05.23- 20.05.23	
5	Тестирование и отладка	10%	21.05.23- 23.05.23	
6	Оформление выпускной работы.	10%	24.05.23- 30.05.23	

Дата выдачи задания " 26 " 04 2023 г.

Руководитель бакалаврской работы  Дубинин В. Н.
(подпись, дата)

Задание к исполнению принял студент  Кобзев М. И.
(подпись, дата)

Работу к защите допустить
Декан факультет  19.06.2023 Фионова Л. Р.
(подпись, дата)

Реферат

Бакалаврская работа содержит 105 л., 43 рис., 0 табл., 14 ист. и 3 прил.
ИНТЕРПРЕТАТОР ДИАГРАММ ГРАФИЧЕСКИХ ЗАПРОСОВ К RDF-ХРАНИЛИЩАМ ДАННЫХ

Цель контекстного исследования, в рамках которого выполнялась данная работа – разработка математического, лингвистического и программного обеспечения для поддержки проектирования специализированных информационно-вычислительных систем, предназначенных для оценки объектов энергетической инфраструктуры, на основе функционально-блочных диаграмм и *RDF*-хранилищ данных. Цель данной работы – разработка операционной семантики графического языка функционально-блочных запросно-вычислительных диаграмм, а также программная реализация интерпретатора диаграмм данного вида и системы объяснения решений.

Задачи, решаемые в процессе выполнения проекта:

- разработка модели выполнения функционально-блочных запросно-вычислительных диаграмм;
- разработка трансляционной семантики базисных функциональных блоков;
- разработка трансляционной семантики *SPAQRL*-блоков;
- разработка и программная реализация интерпретатора функционально-блочных запросно-вычислительных диаграмм;
- разработка системы объяснения результирующих значений;
- разработка иллюстративных примеров для тестирования.

ПГУ1.090301.09.001 ПЗ				
Изм	Лист	№ докум.	Подпись	Дата
Разраб.		Кобзев М. И.		
Провер.		Дубинин В.Н.		
Н. Контр.		Бычков А.С.		
Утверд.		Митрохин М.А.		

Интерпретатор диаграмм графических запросов к RDF- хранилищам данных Пояснительная записка		
Лит.	Лист	Листов
	1	105
ФВТ, каф. ВТ гр.19ВВП1		

Области применения:

- спецификация (формализованное описание) на языке функционально-блочных запросно-вычислительных диаграмм требований к техническим системам, описанным с использованием *RDF*-онтологий, выраженных в числовом виде;

- проектирование, разработка и реализация специализированных информационно-вычислительных систем на основе функционально-блочных запросно-вычислительных диаграмм. Информация о реализации и требования к ПО Язык программирования графического редактора – *C/C++*. Среда программирования – *Qt*. Операционные системы – *Windows 7/10/11* и *Ubuntu 22.10* (и выше).

					ПГУ1.090301.09.001 ПЗ	Лист
						2
Изм	Лист	№ докум.	Подпись	Дата		

Содержание

Реферат	2
Перечень сокращений и обозначений.....	4
Введение.....	9
1 Интерпретатор диаграмм графических запросов к RDF-хранилищам данных	13
1.1 Обзор и анализ интерпретируемых графических языков программирования и моделирования	13
1.1.1. Графические языки программирования и моделирования и методы их интерпретации.....	13
1.1.2. Языки запросов к RDF-хранилищам данных.....	18
1.1.3. Аналоги разрабатываемого программного средства.....	21
1.1.4. Постановка задачи	26
1.2. Теоретические основы интерпретатора функционально-блочных диаграмм	26
1.2.1. Концепция базовых блоков.....	26
1.2.2. Концепция композитных блоков.....	27
1.2.3 Концепция SPARQL-блоков	28
1.2.4 Модель выполнения функционально-блочных диаграмм.....	29
1.2.5 Метод объяснения решений.....	32
1.3 Разработка программного интерпретатора функционально-блочных диаграмм	34
1.3.1 Цели создания программы, её назначение и функциональные возможности	34
1.3.2 Выбор программно-аппаратной платформы.....	34
1.3.3 Архитектура программы	35
1.3.4 Основные структуры данных.....	38
1.3.5 Алгоритм работы интерпретатора и метода объяснения решений	40
1.3.6 Руководство пользователя	42

1.3.7 Отладка и тестирование.....	51
2 Безопасность жизнедеятельности	60
2.1 Введение.....	60
2.2 Требования к освещению на рабочих местах, оборудованных ПЭВМ.....	62
2.3 Выводы... ..	67
Заключение	69
Список использованных источников	70
Приложение А – UML-диаграммы.....	72
Приложение А.1 - Диаграмма классов.....	73
Приложение А.2 - UML-диаграмма прецедентов.....	74
Приложение А.3 - UML-диаграмма развертывания	75
Приложение А.4 - Диаграмма последовательности	76
Приложение Б – Листинг программы	77
Приложение В – Презентация.....	97

Перечень сокращений и обозначений

В настоящей работе применяют следующие сокращения и обозначения

ВМ – виртуальные машины

ФБ – функциональные блоки

DDA (Data-Driven Architecture) - это архитектурный подход, при котором структура и поведение системы определяются и управляются данными.

DiD (Defense-in-Depth) – принцип глубокоэшелонированной защиты.

FBD (functional blocks diagram)- составная блочная диаграмма

I&C (inspection and control) – контроль за выполнением требований

IEC 61131-3 (International Electrotechnical Commission, standard 61131-3) –

стандарт международной электротехнической комиссии

IEC 61499 (International Electrotechnical Commission, standard 61499) –

стандарт международной электротехнической комиссии

IO (input-output) – концепция входов и выходов для работы с блоками.

NoSQL (not only SQL) - базы данных, предназначенные для работы с целым

рядом шаблонов доступа к данным

OWL (Web Ontology Language) – язык описания онтологий для семантической паутины

RDF (Resource Description Framework) - модель для представления данных

SQL (Structured Query Language) - это язык программирования для хранения и обработки информации в реляционной базе данных

					ПГУ1.090301.09.001 ПЗ	Лист
						5
Изм	Лист	№ докум.	Подпись	Дата		

Введение

Диаграммы графических запросов представляют собой графические инструменты, используемые для визуализации и проектирования запросов к базам данных. Они предоставляют удобный способ графического представления сложных запросов и их взаимосвязей, что делает их более понятными и доступными для широкого круга пользователей.

Вместо традиционного текстового подхода, где запросы формулируются с использованием специализированных языков запросов (например, *SQL*), диаграммы графических запросов позволяют разработчикам и аналитикам визуально создавать и оптимизировать запросы к данным. Они представляют собой набор графических элементов, таких как блоки, стрелки, символы и связи, которые отражают различные аспекты запроса и его структуру.

Диаграммы графических запросов упрощают процесс разработки, анализа и коммуникации запросов между различными заинтересованными сторонами, включая разработчиков, аналитиков и конечных пользователей. Они помогают визуализировать логику запросов, отображая связи между таблицами, условиями фильтрации, агрегацией данных и другими операциями.

Одним из преимуществ диаграмм графических запросов является их интуитивный и понятный интерфейс. Даже люди, не имеющие глубокого знания языков программирования или баз данных, могут легко разобраться в создании и понимании запросов, представленных в виде диаграммы. Это способствует эффективной коммуникации и сотрудничеству между различными участниками проекта.

Кроме того, диаграммы графических запросов могут быть автоматически интерпретированы и преобразованы в соответствующий текстовый код (например, *SQL*). Это позволяет легко преобразовывать визуальные запросы в исполняемый код и использовать их в реальных приложениях и системах. В целом, диаграммы графических запросов представляют собой мощный инструмент для проектирования, визуализации и оптимизации запросов к данным. Они облегчают процесс работы с базами

					ПГУ1.090301.09.001 ПЗ	Лист
						6
Изм	Лист	№ докум.	Подпись	Дата		

данных, делая его более понятным, эффективным и доступным для различных пользователей.

Актуальность темы "Интерпретатор диаграмм графических запросов" заключается в следующем:

- визуальное представление запросов. Диаграммы графических запросов предоставляют визуальное представление сложных запросов к данным. Это может быть особенно полезно для пользователей, которые не имеют глубокого знания языков программирования или запросов баз данных. Представление запросов в виде диаграмм делает их более понятными и доступными;

- удобство и эффективность разработки. Использование графических диаграмм для создания запросов упрощает и ускоряет процесс разработки. Разработчики могут визуальнo проектировать и оптимизировать запросы, избегая необходимости написания сложного кода. Это снижает вероятность ошибок и позволяет сосредоточиться на более высокоуровневых аспектах разработки;

- улучшение коммуникации и сотрудничества. Диаграммы графических запросов являются языком, понятным как для разработчиков, так и для обычных пользователей. Они могут служить средством коммуникации между разработчиками, аналитиками и заказчиками, позволяя лучше понять требования и ожидания в отношении запросов к данным.

Автоматизация и оптимизация запросов: Интерпретатор диаграмм графических запросов может предоставлять возможности автоматической генерации кода на основе диаграммы. Это упрощает и автоматизирует процесс создания запросов, ускоряет разработку и снижает вероятность ошибок. Кроме того, такие инструменты могут предлагать оптимизацию запросов, что повышает производительность и эффективность выполнения запросов к данным.

В целом, использование интерпретатора диаграмм графических запросов имеет потенциал упростить и улучшить процесс разработки и выполнения запросов к данным.

					ПГУ1.090301.09.001 ПЗ	Лист
						7
Изм	Лист	№ докум.	Подпись	Дата		

Эта тема актуальна, поскольку визуальное представление запросов и автоматизация их создания могут повысить эффективность работы разработчиков и обеспечить более понятное взаимодействие с данными для пользователей.

Безопасное проектирование критической киберфизической системы предполагает как правильную логику работы, так и соответствие рабочих компонентов их нефункциональным требованиям. В критически важных для безопасности технологических областях последнее также означает выполнение принципа *Defense-in-Depth (DiD)*.

Для решения этой проблемы предлагается метод разработки сложных систем требований (или принципов проектирования) с использованием функциональных блок-схем (*FBD*) для описания общей архитектуры, хранящейся в базе знаний. Имея принцип проектирования, реализованный в виде *FBD*, мы можем не только освободить аналитика от ручного труда, но и предоставить механизмы рассуждений о результате оценки, которые проливают свет на полученное число.

Существует два подхода к реализации такого метода с использованием двух различных технологий. Первая - логическое программирование с использованием *Prolog* с его встроенными возможностями рассуждений и высокой выразительной силой. Здесь информация о нашей архитектуре I&C хранится в базе знаний *Prolog*, а отдельные запросы технологии и классы безопасности. Этот уровень хранит информацию о структуре архитектуры и физических интерфейсах между ее компонентами вместе с их физическим расположением. Функциональный уровень определяет информационные потоки, то есть функции, которые реализует оборудование, необходимые действия и измерения. В настоящее время такое разнообразие знаний об архитектуре и тот факт, что они часто разбросаны по разным источникам, создают препятствия для ручной оценки соответствия принципу *DiD*.

					ПГУ1.090301.09.001 ПЗ	Лист
						8
Изм	Лист	№ докум.	Подпись	Дата		

Работа положила начало формальной проверке нефункциональных требований общей архитектуры ядерных систем КИПиА. Здесь она представлена с помощью онтологии *OWL*, а проверки выполняются с помощью запросов *SPARQL*. Онтология объединяет оба уровня информации, поэтому можно проверить свойства, относящиеся к физическим и функциональным компонентам. Хотя можно легко найти объекты, удовлетворяющие определенному запросу, недостаток этого подхода заключается в рассуждениях о более сложных свойствах вместе с зависимостями между ними формулируются в виде предложений Хорна.

Вторая реализация представляет собой графический инструмент, созданный с использованием объектно-ориентированного языка программирования. Этот инструмент позволяет визуально строить *FBD* требования, выполнять их и объяснять результаты. В этом инструменте архитектура *I&C* представлена с помощью онтологии, а отдельные запросы формулируются с помощью графического языка *SPARQL*. Такие формулировки заключены в специальный вид функциональных блоков (ФБ) - *SPARQL FB*, которые могут быть добавлены в *FBD* принцип проектирования.

Таким образом, в данной работе был создан интерпретатор графических запросов, предназначенный для обработки и выполнения запросов, связанных с графическими данными. Интерпретатор был реализован с использованием современных технологий и алгоритмов, что позволило достичь высокой производительности и эффективности в обработке запросов.

В процессе разработки интерпретатора была проведена анализ и исследование существующих методов работы с графическими данными, что позволило определить оптимальные подходы к обработке запросов.

1 Интерпретатор диаграмм графических запросов к RDF-хранилищам данных

1.1 Обзор и анализ интерпретируемых графических языков программирования и моделирования

1.1.1 Графические языки программирования и моделирования и методы их интерпретации

Графические языки программирования — это языки программирования, которые позволяют разрабатывать программы с использованием графических элементов, таких как иконки, кнопки, графы и т.д., вместо традиционного текстового кода. Они обеспечивают визуальное представление программы, что делает процесс разработки более интуитивным и доступным для новичков в программировании.

Графические языки программирования обычно используются в сочетании с интерпретаторами или компиляторами, которые позволяют выполнить программу, написанную на графическом языке. Существуют различные методы интерпретации графических языков, вот некоторые из них:

- визуальный интерпретатор. Визуальный интерпретатор выполняет программу, представленную в графическом виде, с использованием визуального представления графических элементов. Он может интерпретировать и выполнять блоки кода или графические элементы, связанные линиями или соединителями;

- блочный интерпретатор. Блочный интерпретатор обрабатывает программу, представленную в виде блоков кода, соединенных между собой. Каждый блок выполняет определенную операцию или функцию, и блоки могут быть связаны линиями для определения потока управления программы;

- интерпретация с использованием виртуальных машин. В некоторых случаях графический язык программирования может быть интерпретирован с использованием виртуальных машин (ВМ). Виртуальная машина анализирует

графические элементы программы и выполняет соответствующие действия. Это может включать в себя преобразование графического представления во внутреннее представление, понятное виртуальной машине;

- генерация текстового кода. Некоторые графические языки программирования могут быть преобразованы в эквивалентный текстовый код, который затем интерпретируется или компилируется стандартным способом. Это позволяет использовать существующие среды разработки и инфраструктуру для выполнения программ. Конкретный метод интерпретации зависит от конкретного графического языка программирования и его реализации. Каждый метод имеет свои преимущества и ограничения, и выбор метода интерпретации зависит от требований и целей разработчиков языка программирования.

Диаграммы потоков данных (*Data Flow Diagrams, DFD*) [10] - это графический метод представления системы, который используется для визуализации потоков данных и процессов в системе. *DFD* представляет собой блок-схему, состоящую из различных символов и стрелок, которые отображают потоки данных и процессы,

выполняющиеся над этими данными. Основные элементы диаграмм потоков данных. Процесс представляет собой действие или функцию, выполняемую над данными. Он может быть представлен в виде окружности или прямоугольника с названием процесса внутри.

Внешняя сущность представляет внешний источник данных или назначение. Это может быть другая система, пользователь или организация. Внешние сущности обычно представляются прямоугольниками.

Поток данных отображает передачу данных от одного места к другому. Он представляется стрелкой, указывающей направление потока данных. Потоки данных могут иметь названия, чтобы указать тип данных, передаваемых по этому потоку.

Хранилище данных представляет собой место, где данные сохраняются

или получаются. Это может быть база данных, файловая система или другой источник данных. Хранилища данных обычно представляются двумя параллельными линиями.

Основная цель диаграмм потоков данных - визуализировать потоки данных в системе и понять, как данные перемещаются между процессами и внешними сущностями. Они часто используются для моделирования бизнес-процессов, проектирования информационных систем или анализа потоков данных в системах. *DFD* имеют несколько уровней детализации, от общего представления системы до более подробных диаграмм, где процессы могут быть разбиты на более мелкие подпроцессы. Это позволяет разрабатывать систему на разных уровнях абстракции и уточнять детали по мере необходимости. *DFD* являются полезным инструментом для анализа, проектирования и документирования систем, особенно с точки зрения потоков данных и их обработки в системе.

Dataflow computing (вычисления на основе потока данных) - это концепция и методология компьютерных вычислений, в которой выполнение задач определяется потоками данных и их зависимостями, а не последовательностью команд. В системе на основе потока данных, задачи выполняются только тогда, когда все необходимые данные доступны, и результаты передаются дальше в поток для обработки следующими задачами.

Основные принципы и характеристики вычислений на основе потока данных:

- параллелизм – это вычисления на основе потока данных активно используют параллелизм для ускорения выполнения задач. Задачи могут выполняться независимо друг от друга, если все необходимые данные доступны, что позволяет эффективно использовать множество вычислительных ресурсов;

- асинхронность – задачи в системе на основе потока данных могут выполняться асинхронно и независимо друг от друга, без жесткой последовательности выполнения. Это позволяет более эффективно

использовать вычислительные ресурсы и обеспечивает гибкость в управлении и распределении задач;

- динамическое планирование – вычисления на основе потока данных используют динамическое планирование для определения порядка выполнения задач. Планировщик анализирует доступность данных и определяет, какие задачи могут быть выполнены в данный момент;

- автоматическая управляемая память – в системах на основе потока данных используется автоматическая управляемая память, где данные автоматически перемещаются между задачами, когда они становятся доступными. Это освобождает разработчика от необходимости явно управлять передачей данных между задачами;

- модель данных потока – вычисления на основе потока данных данные передаются между задачами с использованием потоков данных. Каждый поток данных представляет собой последовательность значений, которые передаются от одной задачи к другой;

- гибкость и модульность – вычисления на основе потока данных предоставляют гибкую и модульную архитектуру, которая позволяет легко добавлять, изменять или удалять задачи и данных в системе. Это упрощает разработку и поддержку сложных вычислительных систем.

Dataflow computing находит применение в различных областях, таких как обработка сигналов, машинное обучение, параллельные и распределенные вычисления, графический рендеринг и многие другие, где параллелизм и эффективное использование вычислительных ресурсов являются ключевыми факторами.

Data-Driven Architecture (DDA) [11] - это архитектурный подход, при котором структура и поведение системы определяются и управляются данными. В рамках DDA данные играют центральную роль и определяют, как компоненты системы взаимодействуют друг с другом и как они обрабатывают данные.

Основные принципы и характеристики *Data-Driven Architecture*:

- централизованное хранение данных: *DDA* предполагает наличие централизованного хранилища данных, которое содержит все необходимые данные для функционирования системы. Централизация данных облегчает доступ и управление информацией для всех компонентов системы;
- распределение данных: Данные распределяются между компонентами системы с использованием определенных правил и схем. Компоненты получают доступ к данным из централизованного хранилища или других источников данных, чтобы выполнять необходимые операции;
- разделение ответственности: В *DDA* каждый компонент системы отвечает за определенный аспект обработки данных. Компоненты могут быть специализированы для выполнения определенных операций, таких как обработка, анализ, визуализация или хранение данных;
- поток данных и событий: Компоненты системы обмениваются данными и событиями, передавая информацию и уведомления друг другу. Поток данных и событий определяют взаимодействие между компонентами и управляют последовательностью операций;
- гибкость и масштабируемость: *DDA* обладает гибкой и масштабируемой архитектурой, которая позволяет легко добавлять новые компоненты, изменять правила обработки данных и адаптироваться к изменениям требований и условий.

Преимущества *Data-Driven Architecture*:

- гибкость: *DDA* обеспечивает гибкость в обработке данных, поскольку правила и схемы обработки могут быть легко изменены или модифицированы без необходимости изменения всей системы;
- масштабируемость: Архитектура на основе данных позволяет легко масштабировать систему, добавлять новые компоненты или расширять ее возможности обработки данных;
- централизованное управление. Централизованное хранение данных

					ПГУ1.090301.09.001 ПЗ	Лист
						14
Изм	Лист	№ докум.	Подпись	Дата		

облегчает управление информацией и обеспечивает единое представление данных в системе;

- удобство в разработке: *DDA* упрощает разработку системы, поскольку компоненты могут быть разработаны и тестированы независимо, сосредоточившись на обработке конкретных данных.

Data-Driven Architecture находит применение во многих областях, включая информационные системы, аналитику данных, машинное обучение, рекомендательные системы и другие, где эффективная обработка и управление данными играют важную роль.

IEC 61131-3 [8] и *IEC 61499* [9] - это международные стандарты, разработанные Международной электротехнической комиссией, которые определяют языки программирования и методы проектирования для автоматизации и управления системами.

IEC 61131-3 - это стандарт, который определяет программные языки для программирования автоматических систем управления (АСУ) и систем промышленной автоматизации. Он включает пять основных языков программирования:

- *Ladder Diagram (LD)* - язык, основанный на графической нотации, в котором элементы программы представляются в виде контактов и катушек, подобно электрическим схемам;

- *Function Block Diagram (FBD)* - язык, основанный на блок-схемах, где блоки представляют функции или операции, а соединения между блоками определяют поток данных;

- *Structured Text (ST)* - текстовый язык программирования, основанный на структурированном программировании, подобно языку программирования Pascal;

- *Instruction List (IL)* - язык программирования, основанный на наборе инструкций, которые выполняются последовательно;

					ПГУ1.090301.09.001 ПЗ	Лист
						15
Изм	Лист	№ докум.	Подпись	Дата		

- *Sequential Function Chart (SFC)* - язык, используемый для описания последовательностей действий и состояний в программе.

Эти языки могут быть комбинированы и использованы вместе для разработки программных решений для систем автоматизации и управления.

IEC 61499 - это стандарт, который расширяет и дополняет *IEC 61131-3*, предоставляя модель функционально-блочного программирования для распределенных систем управления. В *IEC 61499* вводится понятие функционального блока (*Function Block, FB*) - независимой и повторно используемой программной единицы, которая инкапсулирует функциональность и данные.

В *IEC 61499* используются следующие элементы:

- *Function Block (FB)* - блок, представляющий функцию или операцию, аналогично *IEC 61131-3*;

- *Resource Block (RB)* - блок, представляющий аппаратные или программные ресурсы, такие как датчики, приводы или виртуальные устройства;

- *Composite Function Block (CFB)* - составной блок, который объединяет несколько функциональных блоков и определяет их взаимодействие;

- *Event* - событие, которое может инициировать выполнение операций в блоках;

- *Data Connections* - соединения для передачи данных между функциональными блоками.

IEC 61499 позволяет создавать более гибкие и модульные программные решения для распределенных систем управления, обеспечивая легкую интеграцию различных компонентов и поддерживая асинхронное и событийно-ориентированное программирование.

Оба стандарта, *IEC 61131-3* и *IEC 61499*, являются важными руководствами для разработчиков систем автоматизации и управления,

предоставляя им средства и методы программирования, чтобы создавать надежные и эффективные системы.

Диаграммы «сущность-связь» (*Entity-Relationship, ER*) - это графическое представление структуры данных и их взаимосвязей в информационных системах. Они широко используются для моделирования и проектирования баз данных. В ER-диаграммах используются следующие основные элементы:

- сущность (*Entity*) - представляет реальный объект, понятие или событие, для которого хранятся данные. Сущность обычно представляется в виде прямоугольника с названием сущности внутри;
- атрибут (*Attribute*) - это характеристика или свойство сущности. Атрибуты представляются внутри сущности, часто в виде овала или эллипса, с их именами рядом;
- связь (*Relationship*) - определяет отношение между двумя или более сущностями. Связи обычно обозначаются линиями, их типы указываются с помощью различных символов (например, прямая линия, стрелка, ромб и т. д.);
- кардинальность (*Cardinality*) - определяет количество связанных объектов в отношении между сущностями. Кардинальность может быть один-к-одному (*one-to-one*), один-ко-многим (*one-to-many*) или многие-ко-многим (*many-to-many*);
- ключ (*Key*) - это атрибут или комбинация атрибутов, которые однозначно идентифицируют каждую запись в сущности. Ключи могут быть первичными (*Primary Key*), уникальными идентификаторами сущности, или внешними (*Foreign Key*), связывающими записи между разными сущностями.

ER-диаграммы помогают визуализировать и организовать структуру данных в информационных системах, обнаружить связи между сущностями, определить их атрибуты и взаимодействия. Они служат основой для разработки схем баз данных и моделей, которые используются в процессе проектирования и разработки программных решений.

1.1.2 Языки запросов к *RDF*-хранилищам данных

До сих пор наиболее распространенной моделью хранения данных была реляционная, которая с конца 70-х годов и поныне является стандартом де-факто на хранение структурированных данных, а язык *SQL* — стандартом на их обработку. Однако доля структурированных данных становится все меньше, и реляционная модель испытывает все больше проблем при работе со значительными объемами данных — давно, например, была замечена деградация производительности реляционных СУБД при решении задач аналитической обработки, характерными чертами которых являются «длинные» *SQL*-запросы, работающие с несколькими таблицами при наличии большого числа агрегатов.

Сегодня задачи, выходящие за рамки реляционной модели, принято относить к классу *NoSQL*, каждый подкласс которого решает ту или иную проблему, плохо реализуемую с помощью *SQL*, — например, базы данных с колонным хранением, документо-ориентированные, графовые, базы данных ключ-значение и объектные. Скажем, базы данных ключ-значение применяются для задач, характеризующихся чрезвычайно большими объемами, отсутствием операций *join* и ограниченными требованиями к обновлению данных (например, только добавление). В силу своего объема такие базы заведомо распределенные, что, в свою очередь, означает полный отказ от транзакций — такая упрощенная модель данных дает новые возможности повышения производительности за счет широкого использования параллельных архитектур.

Еще один класс задач, трудно решаемых на реляционной модели, — это задачи на сильно связанных данных или графовые задачи. Попытки решения таких задач на реляционной модели приводят к непредсказуемому количеству соединений в запросах, поэтому для решения графовых задач сегодня наибольшее распространение получили *RDF*-хранилища, главным достоинством которых является наличие хорошо проработанных стандартов комитета *W3C* на язык описания графов (*Resource Description Framework*,

RDF) и на обработку графовых данных (*SPARQL* — рекурсивный акроним: *SPARQL Protocol And Rdf Query Language*).

Модель *RDF* [14] возникла в конце 90-х годов, а в 2001 году в журнале *Scientific American* была опубликована знаменитая статья Тима Бернерса Ли, провозглашающая приход эры семантической паутины (*Semantic Web*). С той поры в сетевом сообществе стал лавинообразно нарастать интерес ко всему, что связано с семантической обработкой, в том числе и к *RDF*, который в 2004 году был принят как стандарт комитета *W3C*.

Основа *RDF* — это хорошо известное специалистам по искусственному интеллекту представление данных в виде утверждений (троек, *triples*) субъект-предикат-объект, описывающих направленную связь от субъекта к объекту. Для идентификации субъектов, объектов и предикатов используется идентификатор *Uniform Resource Identifier (URI)*, являющийся обобщением понятия *URL*.

Модель *RDF* служит для описания данных, но не описывает методов их обработки. Существует целый ряд языков запросов к *RDF*-данным: *DQL*, *N3QL*, *R-DEVICE*, *RDFQ*, *RDQ*, *RDQL*, *SeRQL* и т. д., но самым популярным стал *SPARQL*, принятый в качестве стандарта *W3C*. Язык *SPARQL*, в отличие от *SQL* (который критикуют, в частности, за отсутствие кросс-платформности, проблемы с обработкой отсутствующих данных, неоднозначную грамматику и семантику), обладает более стройной структурой и мощностью.

Язык *SPARQL* [2] прост в освоении для человека, знакомого с *SQL*, — многое в *SPARQL* ему покажется известным. Например, в языке присутствуют такие конструкции, как *UNION*, *ORDER BY*, *GROUP BY*, *DISTINCT*, *OFFSET* и *LIMIT*. На сегодняшний день *SPARQL* является одним из самых выразительных языков обработки данных. Кроме языка запросов, стандарт *SPARQL* регламентирует протокол взаимодействия с базой данных и формат результата, что является большим шагом вперед по сравнению с *SQL*.

Достоинства модели *RDF* и языка *SPARQL*:

- гибкость. Изменения архитектуры информационной системы, построенной на модели *RDF*, происходят легче, чем для системы, построенной на реляционной модели, и, как правило, даже не требуют реинжиниринга базы;

- современная архитектура. Запросы к хранилищу *RDF* обычно совершаются с помощью протокола *HTTP*, благодаря чему они легко встраиваются в сервисные архитектуры без построения промежуточных слоев, потери надежности и производительности. *RDF* и *SPARQL* лучше работают с интернациональным контентом, чем базы *SQL*;

- стандартизация. Уровень стандартизации *RDF* и *SPARQL* гораздо выше, чем в *SQL*, — усилиями комитета *W3C* определены стандарты не только на модель *RDF* и язык *SPARQL*, но и на идентификацию ресурсов (*URI*), протокол взаимодействия компонентов (*HTTP*), точку доступа *SPARQL* и т. д. Благодаря стандартизации, данные, выгруженные из любого *RDF*-хранилища, можно загружать в *RDF*-хранилища различных производителей. Запросы на *SPARQL* одинаково выполняются на разных хранилищах, что высоко ценят разработчики, сталкивающиеся с проблемами переноса данных и запросов из одной базы в другую;

- метаданные. *SPARQL* позволяет легко отследить происхождение любых единиц данных. В *RDF* легко хранить самые разные метаданные. На основе метаданных можно делать сложные запросы, выбирая, скажем, данные из конкретных источников, в конкретном временном диапазоне и т. д.

Языки запросов к *RDF*-хранилищам играют важную роль в обработке и извлечении данных, хранящихся в формате *RDF* (расширяемый язык описания ресурсов).

1.1.3 Аналоги разрабатываемого программного средства

MATLAB Simulink [12] - это графическое средство моделирования, разработки и симуляции динамических систем. Оно является частью пакета программного обеспечения *MATLAB*, который широко используется в инженерии и науке.

					ПГУ1.090301.09.001 ПЗ	Лист
						20
Изм	Лист	№ докум.	Подпись	Дата		

Simulink позволяет создавать блок-схемы моделей систем, в которых блоки представляют различные компоненты системы, а провода определяют поток сигналов и данных между этими компонентами. Блоки могут быть как предопределенными из библиотек *Simulink*, так и созданными самостоятельно пользователем. С помощью *Simulink* можно моделировать и анализировать широкий спектр систем, включая электрические, механические, гидравлические, тепловые системы и другие. Он также поддерживает различные методы численного решения дифференциальных уравнений, такие как метод конечных разностей и метод Рунге-Кутты, для симуляции и анализа динамики систем. *Simulink* также предлагает возможности для визуализации результатов симуляции, анализа системы, оптимизации параметров и генерации кода для встраиваемых систем или для запуска на аппаратном обеспечении в реальном времени.

Некоторые из недостатков *Matlab Simulink*:

- стоимость. *Matlab Simulink* является коммерческим продуктом, и его лицензия может быть довольно дорогой для индивидуальных разработчиков и небольших компаний. Это может ограничить доступность и применимость *Simulink* в некоторых случаях;
- высокие системные требования. *Simulink* требует значительных вычислительных ресурсов, особенно при работе с большими и сложными моделями. Для эффективной работы с *Simulink* необходимо обеспечить мощное аппаратное обеспечение и достаточный объем оперативной памяти;
- ограниченная гибкость программирования. *Simulink* предоставляет графический подход к моделированию и разработке, что может быть ограничивающим для разработчиков, привыкших к текстовому программированию. В некоторых случаях может быть сложно реализовать сложную логику или алгоритмы, которые легче было бы описать в виде кода.

Пример проекта в *MATLAB Simulink* – рисунок 1.

					ПГУ1.090301.09.001 ПЗ	Лист
						21
Изм	Лист	№ докум.	Подпись	Дата		

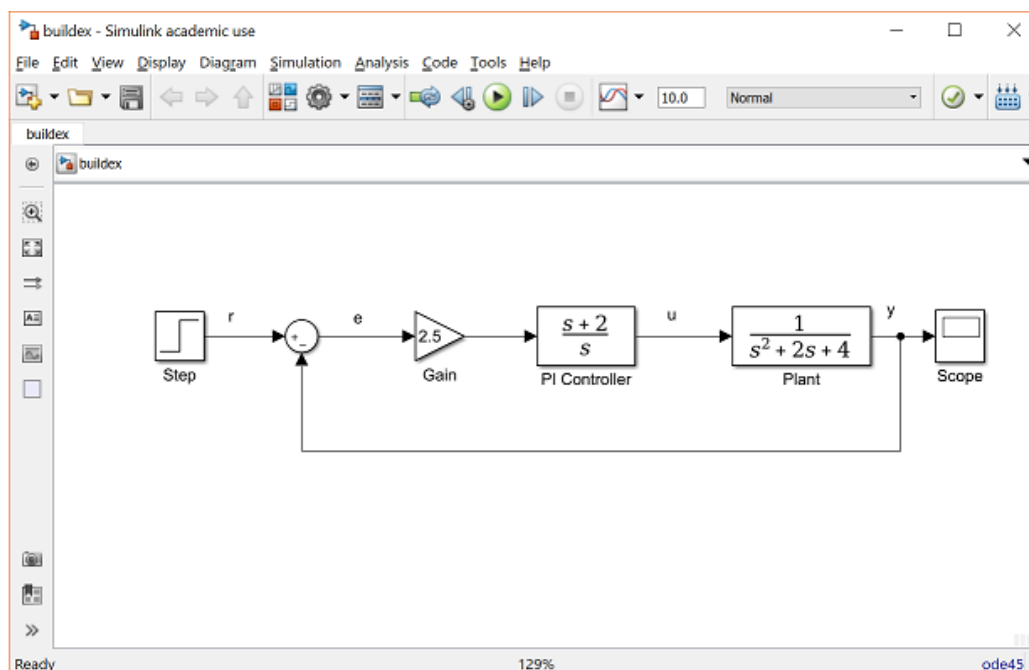


Рисунок 1 – Пример созданного проекта в *MATLAB Simulink*

NXT Studio — это программная платформа, используемая для программирования роботов *LEGO MINDSTORMS NXT*. Он предоставляет визуальный интерфейс программирования, который позволяет пользователям создавать программы для управления и эксплуатации робота.

В *NXT Studio* программирование осуществляется с помощью функциональных блоков, представляющих собой графические элементы, представляющие различные функции программирования. Эти функциональные блоки можно перетаскивать на холст для программирования и соединять вместе для создания последовательности инструкций для робота.

Блоки двигателей: эти блоки управляют двигателями робота, позволяя вам устанавливать скорость, направление и продолжительность движений двигателей.

Блоки датчиков: эти блоки считывают данные с датчиков робота, таких как датчики прикосновения, датчики света, ультразвуковые датчики и т. д. Они предоставляют информацию об окружающей среде робота.

Логические блоки: эти блоки позволяют реализовывать логические операции, такие как условия (операторы *if-else*), циклы (циклы *for* и *while*) и логические операции (И, ИЛИ, НЕ).

Математические блоки: эти блоки выполняют математические операции, такие как сложение, вычитание, умножение, деление и т. д. Они используются для вычислений или обработки данных датчика.

Блоки переменных позволяют создавать переменные и управлять ими, которые используются для хранения и извлечения данных во время выполнения программы.

Звуковые блоки контролируют звуковые возможности робота, позволяя воспроизводить различные мелодии или звуковые файлы.

Блоки отображения: эти блоки взаимодействуют с дисплеем робота, позволяя отображать текст, изображения или анимацию на экране.

Это всего лишь несколько примеров функциональных блоков, с которыми вы можете столкнуться в *NXT Studio*. Фактически доступные блоки могут различаться в зависимости от конкретной версии программного обеспечения или установленных расширений/надстроек.

Однако в *nxtStudio* есть несколько недостатков:

- ограниченная платформа. *NxtStudio* разработан для платформы *Nxt*, что может ограничить возможности разработчиков, если они хотят создать приложения для других платформ или совместимых устройств. Если вы планируете разрабатывать приложения только для платформы *Nxt*, это не является проблемой;

- не очень широкое сообщество. По сравнению с некоторыми другими *IDE* и платформами разработки, сообщество разработчиков *NxtStudio* может быть не таким широким. Это может означать ограниченную поддержку и меньшее количество доступных ресурсов, таких как обучающие материалы и форумы поддержки;

- сложность для новичков. Если вы новичок в разработке *IoT* или платформе *Nxt*, *NxtStudio* может показаться сложным в освоении. Вам может

потребоваться время и усилия, чтобы изучить инструменты и функциональность среды разработки.

Некоторые общие функциональные блоки, которые вы можете найти в *NXT Studio* на рисунке 2.

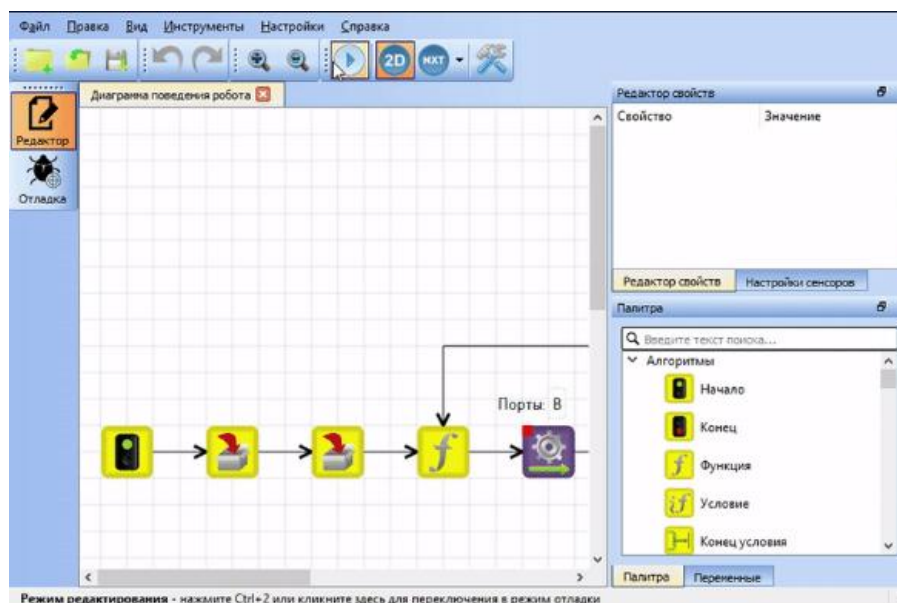


Рисунок 2 – Пример созданного проекта в *NXT Studio*

Owen Logic [13] – среда программирования для создания алгоритмов работы программируемых реле и информационной программируемой панели ОВЕН ИПП120.

Среда *Owen Logic* позволяет создавать собственные функциональные блоки – макросы. Созданные макросы можно использовать в других проектах как готовые функциональные блоки. Это существенно сокращает время на разработку алгоритмов и рабочее место на холсте.

Хранить макросы можно в собственной оффлайн-базе макросов в среде *Owen Logic*. Кроме собственных макросов, в среде *Owen Logic* есть онлайн-база готовых макросов. Это готовые счетчики, аналоговые преобразования, регуляторы, макросы для вентиляции и управления насосами. Программирование практически сводится к конфигурированию – соединению функциональных блоков между собой.

Алгоритм создается на языке функциональных блоков *FBD* с помощью готовых компонентов:

- логических функций;
- арифметических действий;
- функциональных блоков.

Программируемое реле ПР200 и ИПП120 могут быть Мастером сети – управлять приборами по сети *RS-485*. Для этого в *Owen Logic* добавлены шаблоны приборов ОВЕН с готовыми настройками и параметрами. Это значительно сокращает время на настройку по *RS-485*.

Главным недостатком данного продукта – является его узкая сфера использования.

Разрабатываемая программа *FBQL Editor* покрывает большую часть недостатков своих аналогов.

На рисунке 3 можно увидеть пример созданного проекта в *Owen Logic*.

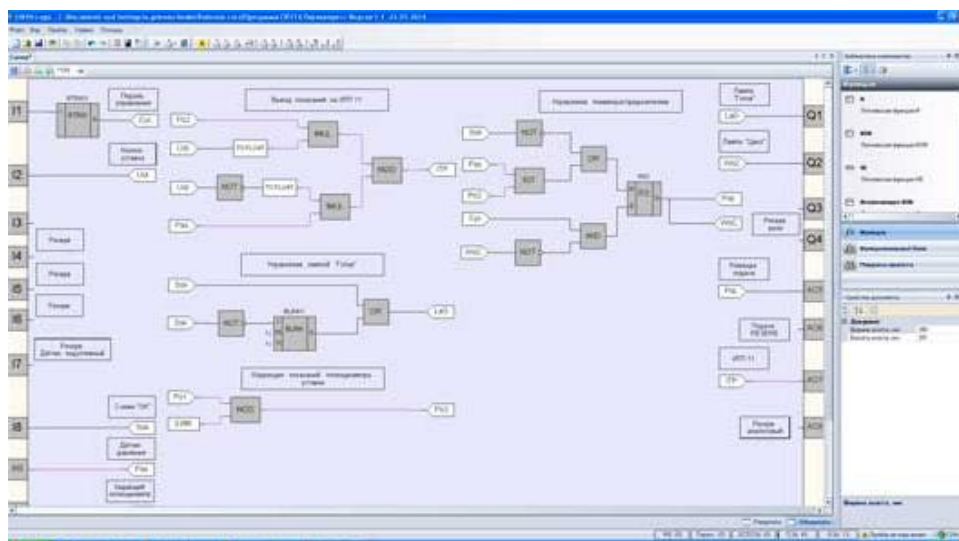


Рисунок 3 – Пример созданного проекта в *Owen Logic*

1.1.4. Постановка задачи

Разработать модель выполнения функционально-блочных диаграмм для формирования и обработки запросов к *RDF*-хранилищам данных на основе принципа потоков данных. Трансляционная семантика базисных функциональных блоков должна основываться на языке

JavaScript. Композитный функциональный блок должен выполняться как контейнер. Графическая диаграмма запроса должна транслироваться в текстовый *SPARQL*-запрос, выполняемый на сервере.

Разработать метод объяснения решений на основе обратного поиска в вычисленной функционально-блочной диаграмме.

Программно реализовать интерпретатор функционально-блочных диаграмм и систему объяснения решений.

1.2 Теоретические основы интерпретатора функционально-блочных диаграмм

1.2.1 Концепция базовых блоков

Базовый блок - это самостоятельная функция или задача, которую выполняет система или ее компонент. Они представляют основные функции и операции системы, которые могут быть объединены в более крупные функциональные блоки. Каждый базовый блок выполняет конкретную функцию и может быть использован повторно или комбинирован с другими блоками для создания более сложных функциональных блоков. Это позволяет упростить анализ и проектирование системы, а также обеспечить ее модульность и масштабируемость.

В создаваемой программе, базовые блоки можно разделить на 2 категории: арифметические и логические блоки.

К группе арифметических базовых блоков можно отнести блоки умножения, деления, сложения и вычитания. С помощью этих блоков реализуется возможность выполнять простейшие арифметические операции с введенными значениями.

К группе логических базовых блоков относятся: блок логического умножения, блок логического сложения, блок инверсии. Данные блоки отвечают за реализацию логических операций в программе.

Кроме уже реализованных базовых блоков имеется возможность создавать пользовательские блоки на языке *JavaScript*. Скрипт для базового

					ПГУ1.090301.09.001 ПЗ	Лист
						26
Изм	Лист	№ докум.	Подпись	Дата		

блока не имеет определённой структуры, таким образом, пользователь вправе сам выбирать наполнение блока для своих нужд.

1.2.2 Концепция композитных блоков

Композитные блоки в формате функциональных блоков представляют собой комбинацию нескольких базовых, *SPARQL* или других композитных блоков, объединенных в более крупный функциональный блок. Они позволяют описывать сложные функции и операции системы, которые могут быть разделены на более мелкие и понятные компоненты.

Композитные блоки имеют входы и выходы, которые определяются их составляющими базовыми блоками или другими композитными блоками. Входы представляют собой данные, которые поступают в блок для обработки, а выходы представляют результат работы блока. Это позволяет создавать цепочки композитных блоков, где выход одного блока становится входом для другого.

Причины использования композитных блоков:

- модульность и повторное использование. Композитные блоки позволяют создавать модули, которые могут быть использованы многократно в разных частях системы. Внутри композитного блока можно определить сложную логику и функциональность, а затем использовать его как единый блок в других системах или проектах. Это способствует повышению эффективности разработки и обеспечивает легкость поддержки и изменений системы;

- упрощение сложных систем. Композитные блоки позволяют упростить сложные системы, разбивая их на более мелкие, логически связанные блоки. Это улучшает читаемость и понимание системы, а также облегчает отладку и тестирование. Композитные блоки помогают снизить сложность системы, разделяя ее на более управляемые и понятные компоненты;

- абстракция и скрывание деталей. Композитные блоки позволяют создавать абстракции, скрывая сложные внутренние детали и реализацию блоков. Внешний интерфейс композитного блока определяет, какие данные и

					ПГУ1.090301.09.001 ПЗ	Лист
						27
Изм	Лист	№ докум.	Подпись	Дата		

управление можно передавать внутрь блока и какие результаты можно получить снаружи. Это упрощает взаимодействие с блоком, так как пользователь может использовать его, не заботясь о сложности его внутренней структуры;

- расширяемость и гибкость. Композитные блоки обладают гибкостью и расширяемостью, так как их внутреннюю структуру можно изменять и модифицировать, добавляя новые внутренние блоки или изменяя существующие. Это позволяет легко вносить изменения в систему без необходимости полностью переписывать код или перестраивать всю систему.

1.2.3 Концепция SPARQL-блоков

SPARQL-блоки нужны для создания запросов к *RDF*-хранилищам данных. SPARQL-блоки обращается к онтологии которая загружена на *Jena Fuseki Server* и обрабатывает результат запроса, отображая количество найденных записей, соответствующих запросу. SPARQL-блоки могут быть использованы в различных контекстах, таких как разработка семантических веб-приложений, анализ данных, интеллектуальный поиск информации и других сценариях, где требуется работа с данными *RDF*.

В границах созданной программы, SPARQL-блок создаётся при помощи вспомогательных *Atom*-блоков. *Atom*-блоки включают в себя 3 типа: «*AREA*», «*VAR*», «*VALUE*». Блок «*AREA*» используется для задания областей, которые потом соединяются с помощью связей «*MINUS*», «*UNION*» и тд. Блок «*VAR*» нужен для обозначения переменных в SPARQL-запросе, «*VALUE*» же для константных значений. Формирование запроса происходит по принципу триплетов, стрелкой соединяются 2 блока типа «*VAR*» или «*VALUE*». При конвертации субъектом является блок из которого идёт стрелка, объектом тот блок в который она идёт, а отношением будет надпись на стрелке. Также SPARQL-блоки можно задать с помощью текстового поля SPARQL-запроса.

1.2.4 Модель выполнения функционально-блочных диаграмм

Семантика выполнения ФБД (Функционально-блочных диаграмм) является синхронной, так что выходные значения каждого блока являются

функциями его входных значений, а сигналы распространяются через соединения мгновенно. Каждый цикл выполнения ФБД независим от предыдущих, что означает, что любому выполнению предшествует удаление ранее полученных назначений (за исключением назначений постоянных значений). Если диаграмма не была изменена между двумя последующими циклами выполнения, то назначения ФБД, полученные после каждого выполнения, будут одинаковыми.

Порядок выполнения ФБ определяется структурой ФБД, к которому они относятся. Основное правило заключается в том, что ФБ может быть выполнен только после того, как все его входы будут вычислены в течение текущего цикла выполнения. Таким образом, первыми выполняются ФБ без входов или те, все входы которых имеют постоянные значения.

Следуя этому правилу выполнения, ФБ ФБД можно разделить на *слои выполнения* - кортежи (i, V) , где $i \in N$ - порядок слоя, а V - набор ФБД, которые могут выполняться параллельно. Например, на рисунке 4 показано распределение слоев выполнения ФБ, составляющих составной ФБ с рисунка 5. Мы показали распределение для начальной внутренней диаграммы ФБ. Внутренняя диаграмма также содержит составные ФБ, которые могут быть представлены своими внутренними сетками. Поскольку в программе запрещена бесконечная вложенность составных ФБ, каждый составной ФБ и ФБД, в конечном итоге, может быть развернут в сеть базовых ФБ. Это разворачивание является первым шагом нашего алгоритма выполнения. За ним следует построение слоев выполнения над результатом разворачивания и их последующее упорядоченное выполнение.

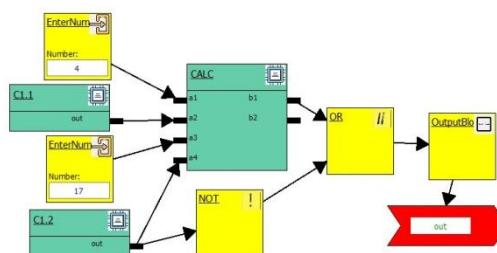


Рисунок 4 – Пример внутренней диаграммы составного блока в программе
FBQL Editor

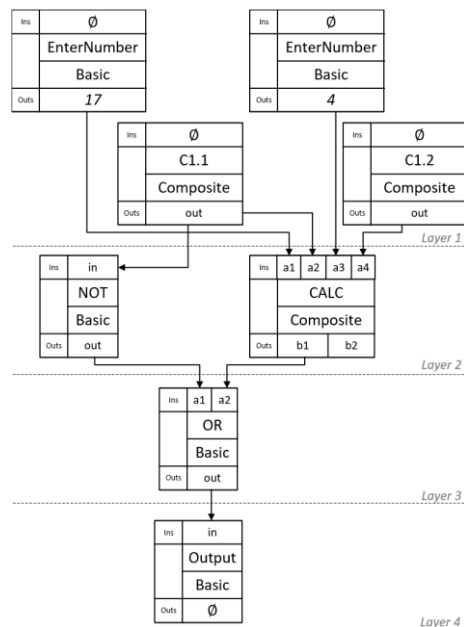


Рисунок 5 – Диаграмма составного ФБ

Введем необходимые объекты, функции и предикаты, а также соответствующие им обозначения.

Предикат, определяющий разрешенность функционального блока (ФБ) fb для выполнения, может быть записан следующим образом.

$$enabled(fb) \equiv \forall in \in IN^{fb} [readyData(in)],$$

где IN^{fb} – множество входов ФБ fb ;

$existData: IN \rightarrow \{true, false\}$ – функция (предикат), определяющая готовность данных на входах ФБ. Здесь IN – множество входов всех функциональных блоков системы.

Предикат $ready(fb)$ является истинным, когда готовы данные на всех входах блока fb .

Предикат, указывающий на наличие хотя бы одного разрешенного ФБ в системе, может быть определен так, как показано ниже.

$$isEnabledFB \equiv \exists fb \in FB [enabled(fb)],$$

где FB – множество функциональных блоков в системе.

Алгоритм активизации (запуска) функциональных блоков в системе:

while isEnabledFB

forall fb ∈ FB with enabled(fb) do

Execute(fb)

end

end

Согласно приведенному алгоритму выполняются все ФБ, которые являются разрешенными. Работа алгоритма заканчивается, когда в системе не останется ни одного разрешенного ФБ. При выполнении ФБ данные с его входов удаляются, а на выходы добавляются. Значения добавляемых на выходы данных зависят от функции, выполняемой блоком.

Данная модель выполнения используется как при интерпретации функционально-блочной диаграммы верхнего уровня, так и композитных функциональных блоков.

1.2.5 Метод объяснения решений

В основе лежит концепция зависимости выходов ФБ от входов. Предположим, что каждый ФБ имеет только один выход и несколько входов. В этом случае его можно представить в виде функции $y = f(x_1, x_2, \dots, x_n)$.

Каждый из аргументов x_i влияет на результат y (другими словами, значение функции зависит от каждого из аргументов). Концептуально, если некоторые аргументы не влияют на результат функции, их можно смело исключить из набора аргументов. Такая зависимость выхода y от входов $x_1, x_2, \dots, x_n$ следует из определения функции.

Теперь рассмотрим концепцию зависимости выходного значения y от назначения входных переменных. Пусть $w = f(d_1, d_2, \dots, d_n)$, где $d_1, d_2, \dots, d_n$ - конкретные назначения входов, а w - конкретное назначение выхода. Будем говорить, что входное значение d_i не влияет на результат w , если любое изменение, примененное исключительно (т.е. сохраняя тот же контекст) к d_i , не приведет к изменению результата.

Назовем входное значение d_i *индивидуально доминирующим* среди входных параметров $d_1, d_2, \dots, d_n$, если оно полностью определяет выходное значение w . В этом случае остальные значения могут быть изменены, но результат w останется. Мы определяем такую зависимость выходного значения w от входного значения d_i как *индивидуальную зависимость доминирования* (контекстно-независимую).

Следуя той же логике, если существует группа входных значений $d_k, d_m, \dots, d_s$, которая полностью определяет выходное значение w , то есть остальные значения могут быть изменены, но результат w останется, то такая зависимость выходного значения w от группы входных значений $d_k, d_m, \dots, d_s$ является *зависимостью доминирования группы* (также не зависящей от контекста). Как и выше, здесь контекст также относится к входным значениям, которые не входят в группу. Для одного выходного значения может существовать несколько групп доминирования. Если в группу входят все входные значения, то такая зависимость является *полногрупповой зависимостью доминирования* (или *полногрупповой зависимостью*). Поскольку в этом случае контекст как таковой отсутствует, зависимость напрямую определяет результат.

В целях пояснения мы рассматриваем только отношения доминирования (индивидуальные и групповые) между выходом и входами. Назначение, значение которого влияет на выход, мы называем *активным назначением*. Назначение входа является активным, если существует изменение в значении этого назначения, которое приводит к изменению выходного значения или значение этого назначения напрямую определяет выходное значение.

Для каждого типа ФБ необходимо сформулировать определенные правила определения активных назначений.

В программе используется идея обратного хода, т.е., начиная с цели объяснения (интересующая нас переменная величина), двигаясь в направлении, противоположном потоку информации, мы определяем назначения, влияющие на цель (активные назначения). Сущность алгоритма

здесь заключается в том, что набор базовых ФБ может быть создан вручную, что означает, что их индивидуальные функции объяснения определяются пользователем.

Функция объяснения явно кодирует, какие присваивания активны в зависимости от текущих значений выхода и входа. Так, например, для базового ФБ, который кодирует дизъюнкцию $y = x_1 \vee x_2$, функция объяснения включает следующие правила. Если $y = T$, то активными являются те присваивания, значения которых равны T , а если $y = F$, то оба входных присваивания полностью определяют выходное значение.

На рисунке 6 показан результат объяснения для выхода $c1$ составного блока SYS. Внутренняя диаграмма содержит два составных ФБ и четыре базовых ФБ; как видно, результат $fb2$ не повлиял на цель.

Связи между назначениями, влияющими на $c1$, показаны жирным красным шрифтом. Блок имеет два составных блока ($fb3$ и $fb4$), которые также показаны в развернутом виде. Красные числа на желтом прямоугольном и круглом фоне показывают оценки переменных и предопределенных констант соответственно – рисунок 6.

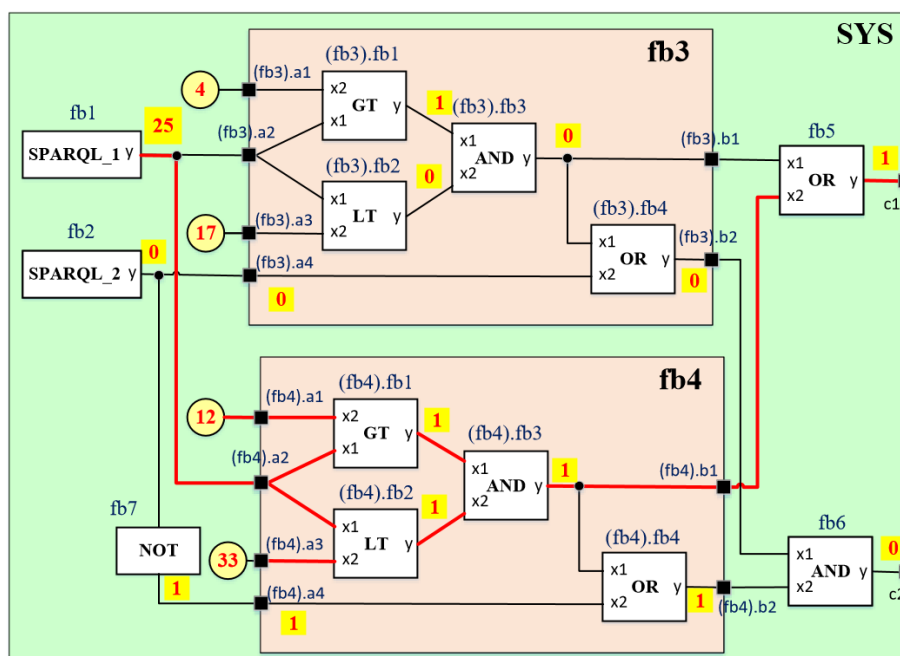


Рисунок 6 – Пример объяснения ФБД

1.3 Разработка программного интерпретатора функционально-блочных диаграмм

1.3.1 Цели создания программы, её назначение и функциональные возможности

Данная программа была создана для пользователей, не знакомых с технологией *SPAQRL* запросов, поскольку изучение данной технологии может занять огромное количество времени, да и написание самих запросов – не такая уж и простая задача. Пользоваться данной программой можно не только в пределах темы контрольно-измерительных систем, а также для создания *SPAQRL* запросов для любых других целей, например, парсинг сайтов. Для упрощения, был разработан подход с использованием графического редактора с удобным интерфейсом, реализующего в себе алгоритмы интерпретирования функционально-блочных диаграмм. Путём соединения блоков между собой, пользователь может создавать триплеты запроса, что куда проще, чем писать его вручную. Получившиеся запросы отправляются на сервер “*Apache Jena Fuseki*” [3], который возвращает результат. Также, в программе предусмотрена система сохранения уже имеющихся запросов, что позволяет повторно производить редактирование созданных запросов. Кроме того, программа имеет потенциал к развитию, поскольку есть возможность разрабатывать пользовательские блоки для своих нужд.

1.3.2 Выбор программно-аппаратной платформы

Для написания данной программы использован язык программирования *C/C++*, язык разметки веб-страниц *HTML/CSS*. Язык программирования *C/C++* - универсальный язык программирования, который завоевал особую популярность у программистов, благодаря сочетанию возможностей языков программирования высокого и низкого уровней. *HTML* — стандартизированный язык разметки веб-страниц во Всемирной паутине. Код *HTML* интерпретируется браузерами; полученная в результате интерпретации страница отображается на экране монитора компьютера или мобильного устройства.

					ПГУ1.090301.09.001 ПЗ	Лист
Изм	Лист	№ докум.	Подпись	Дата		34

Проект был создан в виде графического приложения *Win64* на языке C++/Qt в кроссплатформенной *IDE* для разработки программного обеспечения *Qt Creator 4.15.2*. [1].

Qt позволяет запускать написанное с его помощью программное обеспечение в большинстве современных операционных систем путём простой компиляции программы для каждой системы без изменения исходного кода. Включает в себя все основные классы, которые могут потребоваться при разработке прикладного программного обеспечения, начиная от элементов графического интерфейса и заканчивая классами для работы с сетью, базами данных и *XML*. Является полностью объектно-ориентированным, расширяемым и поддерживающим технику компонентного программирования.

1.3.3 Архитектура программы

В разработанной программе присутствует много различных модулей, которые используются для различных задач. На рисунке 7 представлены следующие модули программы:

- основной модуль - модуль отвечающий за создание главного окна и установке в него стилей;
- модуль главного окна - модуль для отрисовки главного окна и взаимодействия с пользователем. Данный модуль подгружает настройки, вызывает загрузки блоков из сохранения, создаёт внешний вид приложения (*GUI*);
- модуль библиотеки блоков занимается поиском блоков и проектов для загрузки. Также предоставляет главному окну доступ к настройкам блоков;
- модуль графического редактора отвечает за создания окон с графическими редакторами для создания и редактирования блоков и проекта. Использует Библиотеку *Graphviz* для создания изображения проекта при запуске.

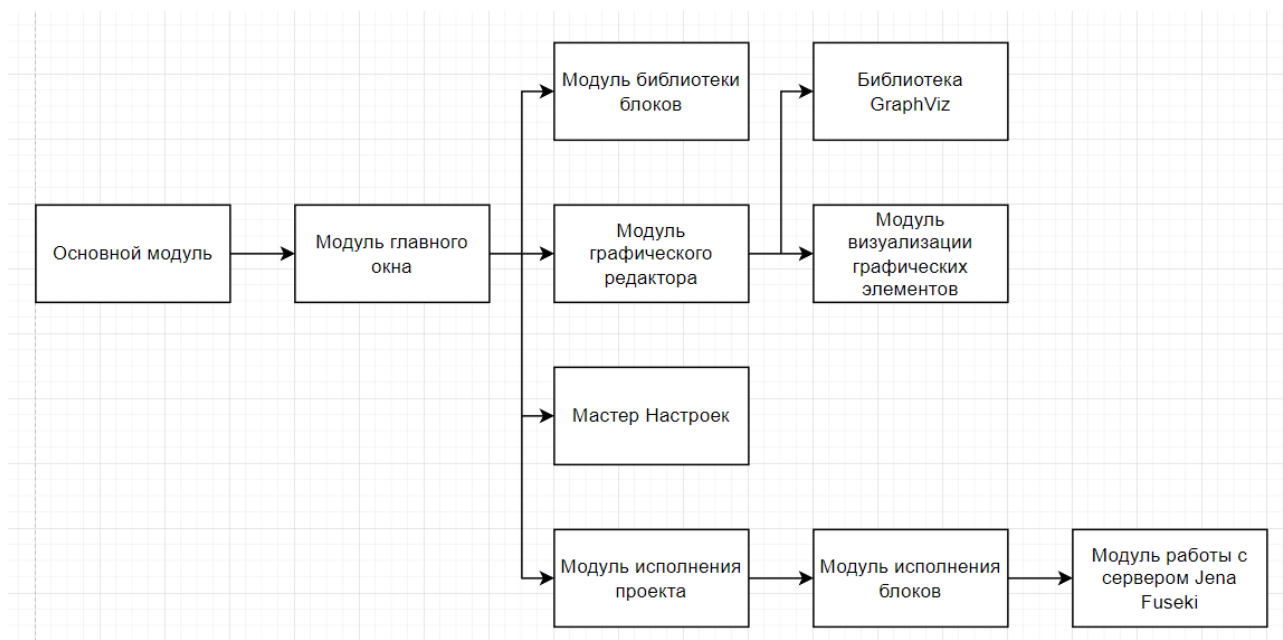


Рисунок 7 - Схема взаимодействия модулей программы

- модуль визуализации графических элементов - отвечает за отображение блоков на графической сцене и их поведение при взаимодействии с пользователем;

- мастер настроек - позволяет получать доступ к глобальным настройкам программы. Данный модуль использует файл для хранения настроек, так что они не теряются при закрытии программы;

- модуль исполнения проекта - модуль отвечающий за создание схемы выполнения блоков и их последующий запуск. Модуль исполнения блоков, отвечает за непосредственное исполнение скриптов блоков и передачу данных от одного блока к другому;

- модуль работы с сервером *Jena Fuseki* отвечает за *API* взаимодействия с сервером для проведения *SPARQL*-запросов.

На Рисунке 8 можно увидеть диаграмму вариантов использования для разработанной программы. На ней представлено взаимодействие основных блоков программы в упрощённом виде, более понятном для пользователя.

- *SMenuBar* - класс для формирования меню;
- *SToolBar* - класс для формирования панели инструментов;
- *SToolBox* - класс боковой панели блоков;
- *STabWidget* - класс отвечающий за виджет отображения окон настроек и редакторов наследованных от *SWidget*.

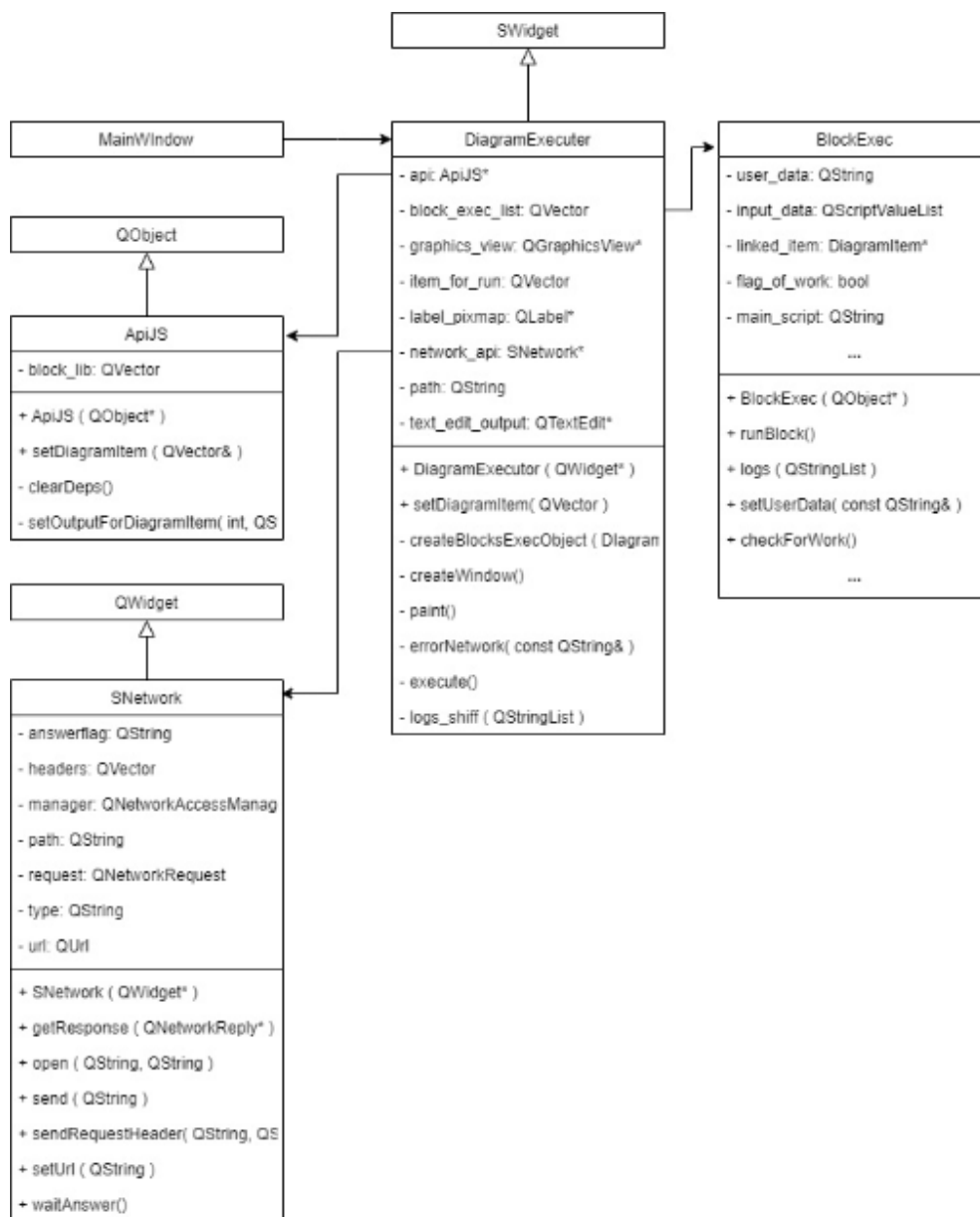


Рисунок 9 – Диаграмма класса исполнения блоков

Диаграмма класса *MainWindow* – рисунок 10.

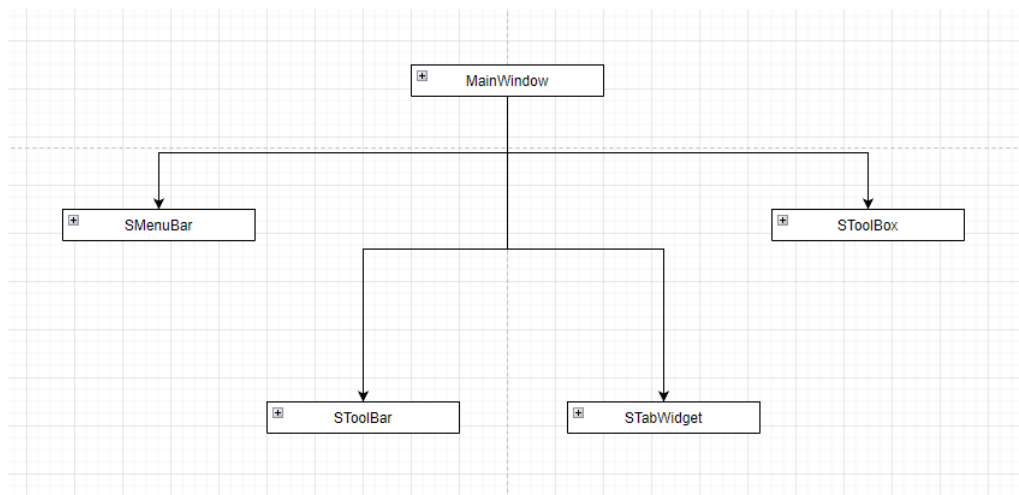


Рисунок 10 - Диаграмма классов для главного окна

1.3.5 Алгоритм работы интерпретатора и метода объяснения решений

Для начала, очищается буфер ввода пользовательских значений, далее запускается цикл для обработки информации блоков. Для начала, запускается проверка работоспособности блока (Если все входные данные получены), далее начинается построения модели выполнения блоков. Для базовых блоков условием работоспособности блока – является наличие информации на всех его входах. Перед работой *SPARQL*-блоков, необходимо убедиться в работоспособности сервера *Apache Jena Fuseki*, поскольку работа блока напрямую связана с онтологией данных. Первый шаг работы интерпретатора композитных блоков начинается с их разворачивания и замена их на блоки, находящиеся внутри данных блоков. За ним следует построение слоев выполнения, выполняющихся последовательно.

Суть метода объяснения решений заключается в заранее введенных входных зависимостях, которые будут использоваться при работе метода объяснения решений. Эти зависимости вводятся пользователем при создании блока. Метод начинает функционировать после работы функциональной диаграммы при наведении на блок, зависимости которого мы хотим узнать. Начало работы данного метода начинается с проверки всех зависимостей. Если зависимостей у текущего блока не существует, то по стандарту в его зависимости запишутся все входные значения с других блоков, иначе запишутся зависимости, введенные пользователем. После извлечения

зависимостей, они будут отражены на диаграмме. На рисунке 11 изображен алгоритм интерпретации функционально-блочных диаграмм.

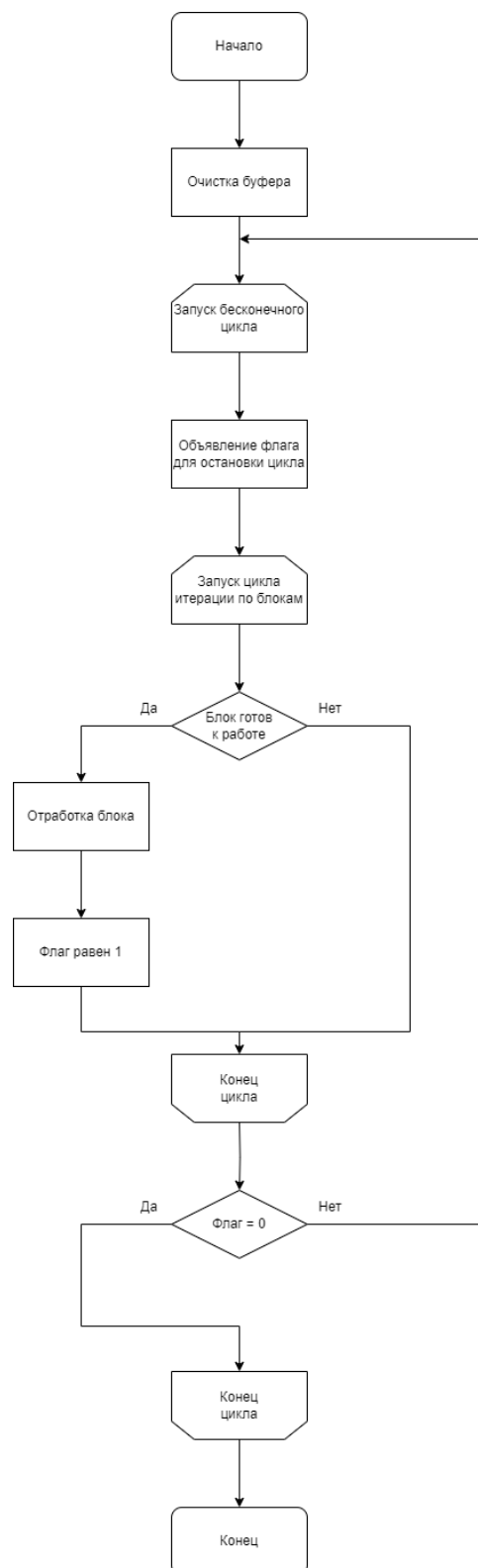


Рисунок 11 – Блок-схема работы интерпретатора диаграмм

На рисунке 12 приведен алгоритм метода объяснений.

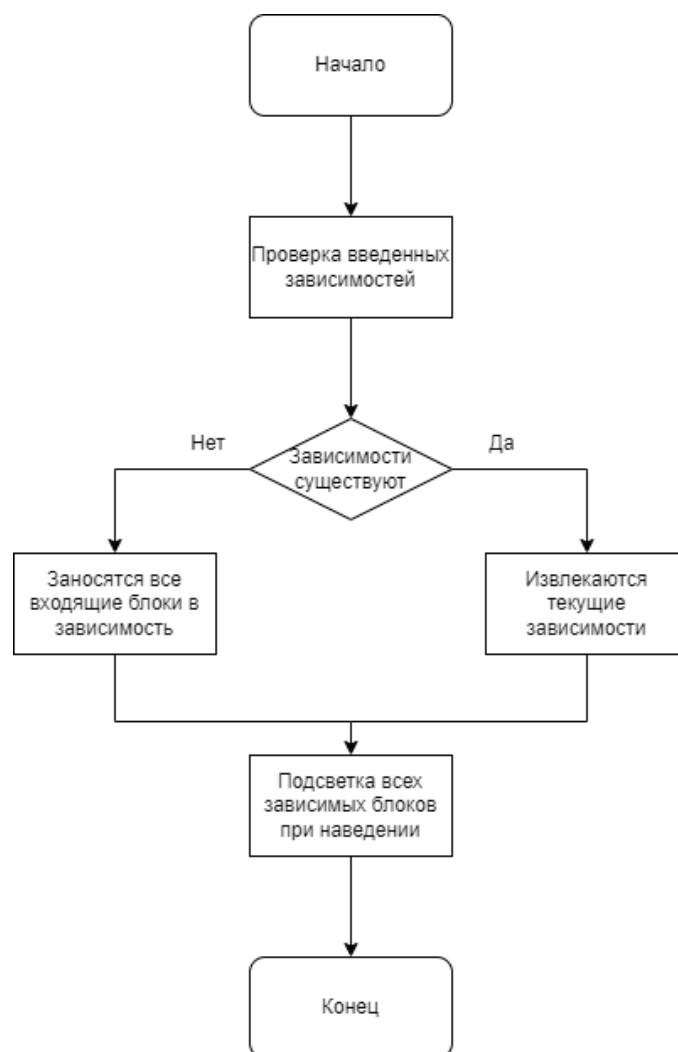


Рисунок 12 – Блок-схема работы метода объяснения решений

1.3.6 Руководство пользователя

При разработке программного продукта необходимо составить руководство пользователя для того, что у пользователя не возникало никаких проблем при работе с программой. Модель интерпретатора и концепция объяснения решений были внедрены в графический редактор *FBQL Editor*. Объяснение работы навигационной панели показано на рисунке 13. Основное окно программы – это окно создания проекта – рисунок 14.



Рисунок 13 – Приветственный интерфейс программы

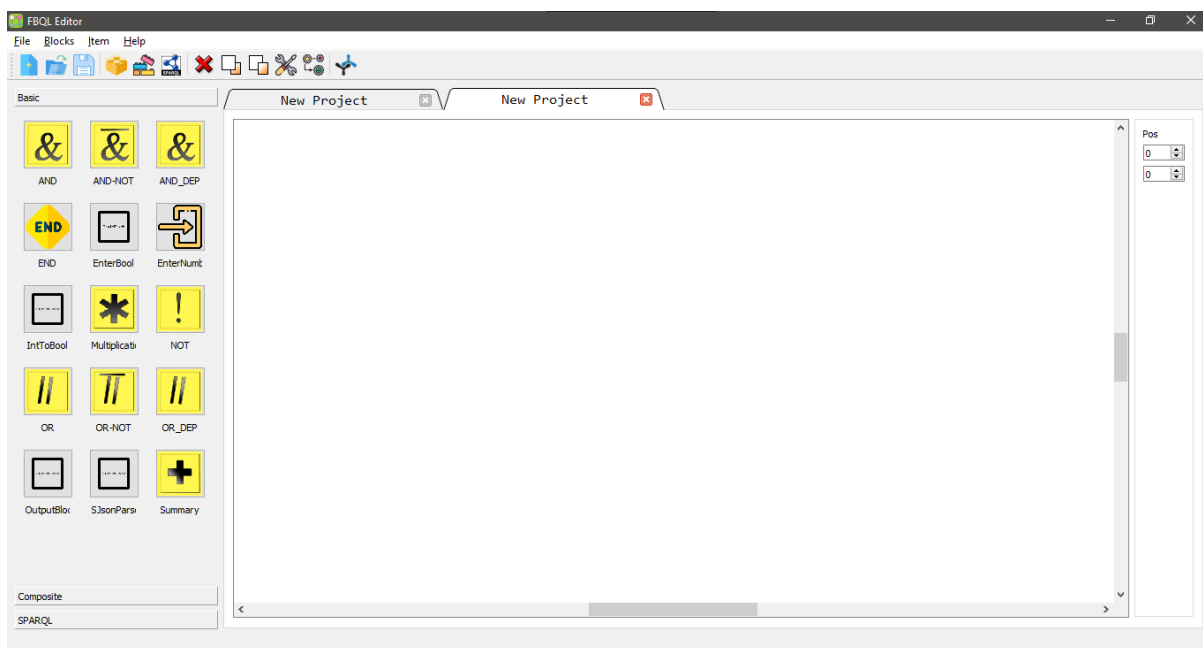


Рисунок 14 – Окно создания проекта

Все блоки поделены на 3 вкладки: базовые, композитные и *SPARQL*. Для перемещения между вкладок, необходимо кликнуть на название типа блока – рисунок 15.

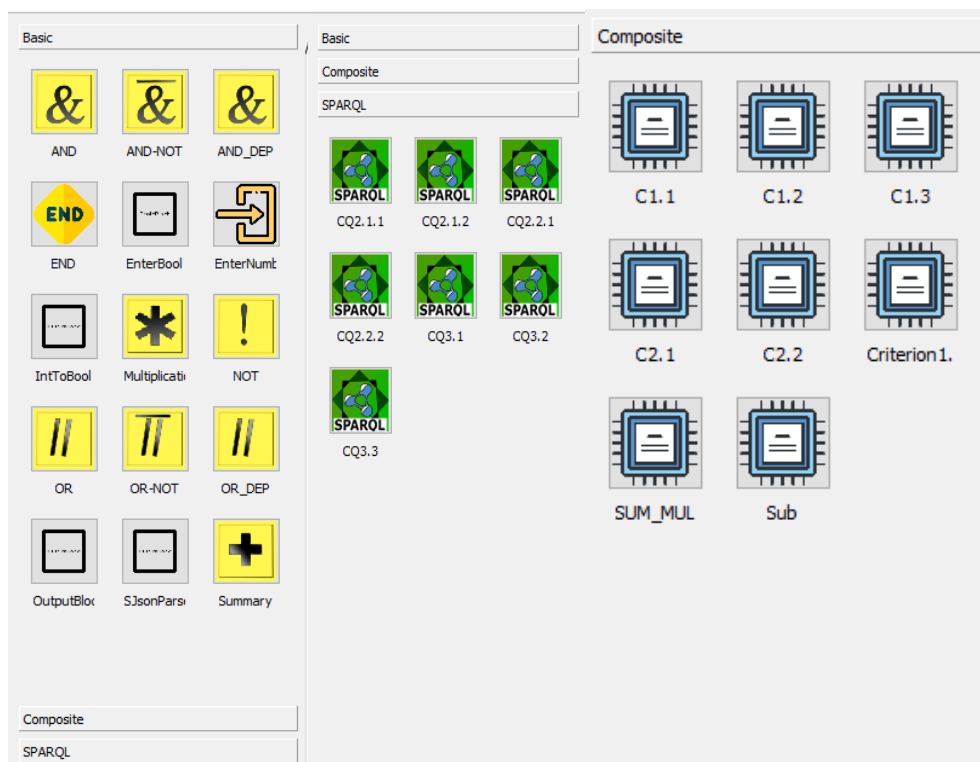


Рисунок 15 – Внешний вид созданных блоков

Создание базисного функционального блока на языке *JavaScript*. Чтобы попасть в окно создания базовых блоков, необходимо нажать на кнопку в навигационной панели, показанную на рисунке 16.

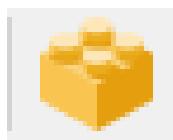


Рисунок 16 – Кнопка для отображения окна создания базисных блоков

Далее, вас перенесет в окно создания базового блока, где в текстовом поле “*Script*” вы сможете написать логику работы создаваемого блока – рисунок 17.

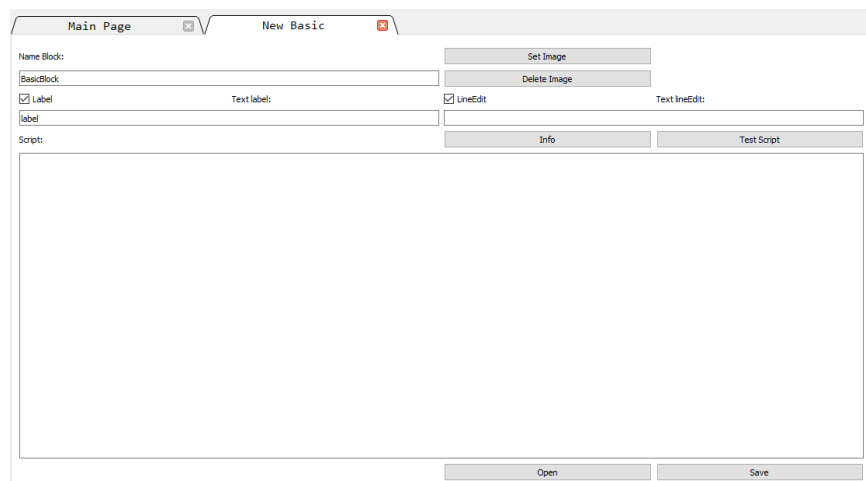


Рисунок 17 – Окно создания базисного блока

Для примера, рассмотрим реализацию блока “логического ИЛИ”.

Код на *JavaScript*, используемый для реализации данного блока. (см. рисунок – 18).

```
Script:
output = "false";
for(var i = 0; i < inputs.length; i++){
  if(inputs[i] == "true"){
    output = "true";
    break;
  }
}
```

Рисунок 18 – Реализация скрипта для блока логического ИЛИ

Входы данного блока являются массивом Булевых значений – “*inputs*”. Результат будет записан в переменную “*output*”. Логика работы заключается в прогоне всех значений входного массива, если хотя бы одно значение является истинным, то результат работы блока будет истинным, иначе ложным.

Для того, чтобы выставить зависимости блока с другими блоками, а также для работы метода объяснения решений, используется секция кода, показанная на рисунке 19.

```
// depends
for(var i = 0; i < inputs.length; i++){
  if(inputs[i] == output){
    causes.push(i);
  }
}
```

Рисунок 19 – Реализация блока кода для создания зависимостей.

Чтобы вход блока был записан в массив зависимостей – “*causes*”, значение входа должно быть равным выходному значению, поскольку именно эти входы влияют на работу блока. Массивы *causes* и *inputs*, а также переменная *output* – являются зарезервированными переменными, то есть они используются при работе интерпретатора, а также метода объяснения решений.

Создание *SPARQL*-блока.

Чтобы попасть в окно создания *SPARQL*-блоков, необходимо нажать на кнопку в навигационной панели, показанную на рисунке 20, тогда откроется окно – рисунок 21.



Рисунок 20 – Кнопка для отображения окна создания *SPARQL*-блоков

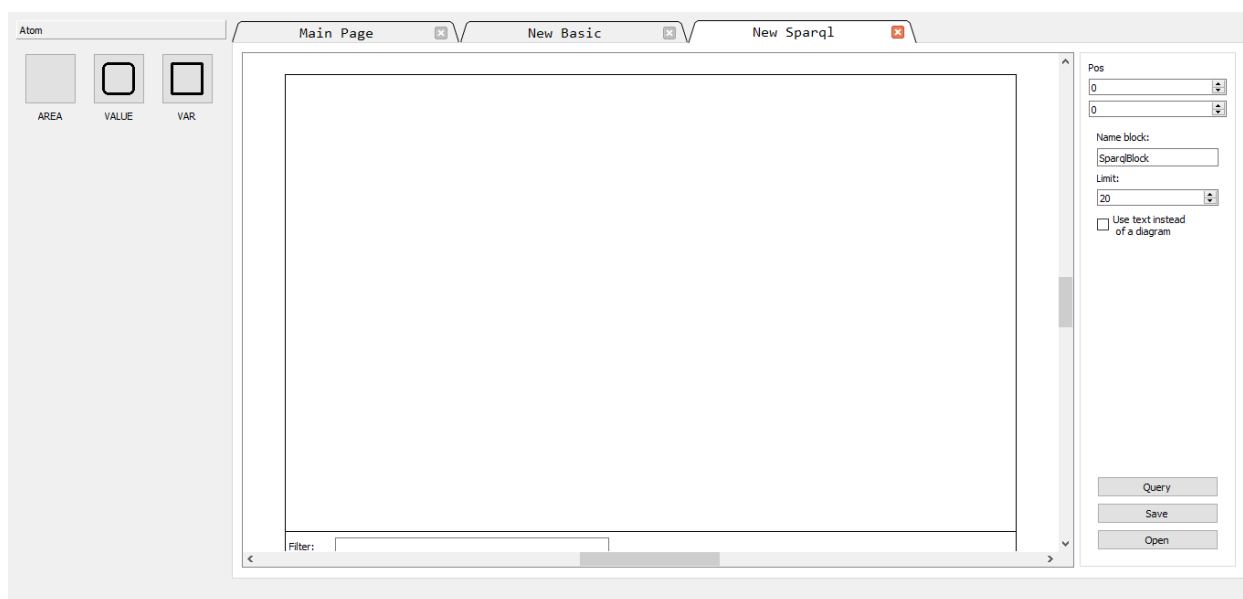


Рисунок 21 – Окно создания *SPARQL*-блока

Блок “*AREA*” – это место, где можно создавать конструкции из триплетов. Блоки “*AREA*” можно соединять между собой, с помощью связи соответствующего *SPARQL* метода. (Например – *MINUS* или *UNION*). Внутри этого блока есть поле *Filter* для создания конструкции *FILTER* в *SPARQL*

запросе. Всё, что находится внутри данного поля, попадает в конструкцию *FILTER* в получившемся запросе.

Блок “*VALUE*” необходим для создания постоянного значения в триплете. Блок “*VAR*” необходим для создания переменной в триплете. Кнопка “*Query*” позволяет написать запрос самому в текстовом виде, после её нажатия, откроется текстовое поле для ввода запроса. Для работы данного вида запроса, необходимо нажать галочку в поле “*Use text instead of a diagram*”. Кнопка “*Save*” нужна для сохранения созданного блока для дальнейшего использования. Кнопка “*Open*” позволяет открыть уже сохраненный блок.

SPARQL-блоки имеют только один выход и возвращают только одно значение – число результирующих кортежей рисунок 22.

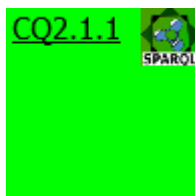


Рисунок 22 – Внешний вид *SPARQL*-блока после его создания

Настройка и подключение к *SPARQL*-серверу

Для работы со *SPARQL* сервером “*Apache Jena Fuseki*” существует вкладка “*Fuseki*”, рисунок 23. Текстовое поле “*IP-adress Fuseki Server*” нужно для введения IP-адреса работы сервера. Текстовое поле “*Port Fuseki Server*” нужно для установления порта работы сервера. В текстовое поле “*Dataset Name*” устанавливается имя онтологии, к которой идёт обращение.

Поле *Prefixes* служит для добавления префиксов в запрос. Для того, чтобы добавить префикс, нужно нажать на кнопку “*Add row*”. Далее, в появившейся строке, необходимо записать префикс в угловых скобках. Чтобы удалить строку с префиксом, используется кнопка “*Remove row*”. Префиксы нужны для сокращения написания предикатов для запроса к онтологии. Веб-интерфейс *Apache Jena Fuseki* можно увидеть на рисунке 24.

IP-adress Fuseki Server:

Port Fuseki Server:

Dataset Name:

Prefixes:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX : <http://www.semanticweb.org/SEARCH/ontologies/2021/4/OICA/Classes#>
PREFIX USEPR: <http://www.semanticweb.org/SEARCH/ontologies/2021/4/OICA/USEPR#>
```

Add Row Remove Row

Рисунок 23 – Внешний вид окна редактирования информации о запросе к серверу

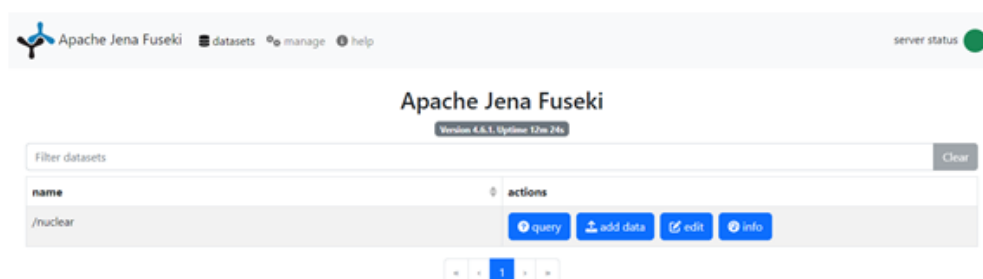


Рисунок 24 – Внешний вид Apache Jena Fuseki

Создание композитного функционального блока

Чтобы попасть в окно создания композитных блоков, необходимо нажать на кнопку в навигационной панели, показанную на рисунке 25.



Рисунок 25 – Кнопка для отображения окна создания композитных блоков

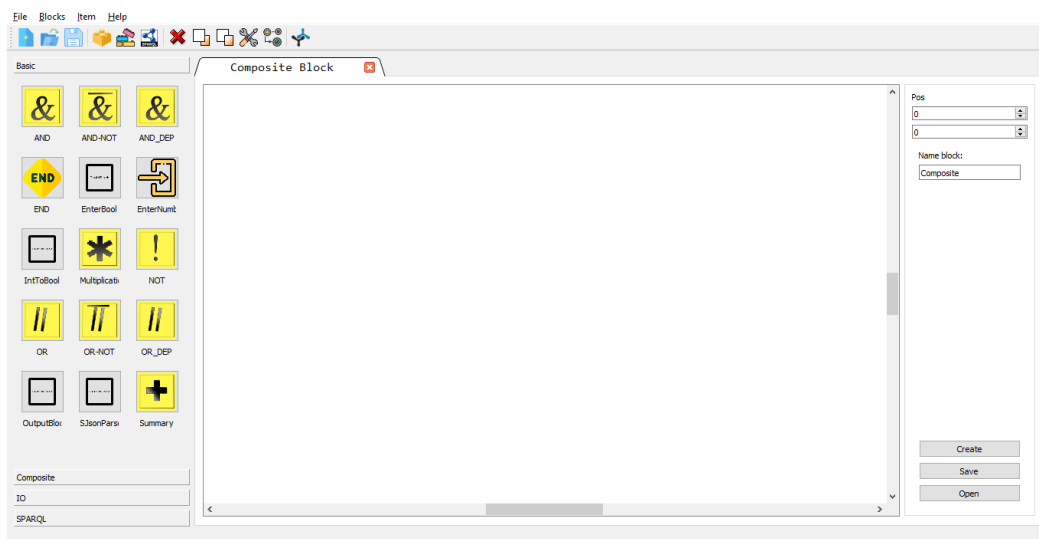


Рисунок 26 – Окно для работы с композитными блоками

В левой части окна находятся все доступные блоки, с помощью которых можно создать новый композитный блок.

Чтобы выставить блок на сцену, необходимо нажать на выбранный блок левой кнопкой мыши, далее нажать левой кнопкой в любом месте на сцене, чтобы отобразить блок.

У композитного блока нет ограничений по входам и выходам. Для того, чтобы сделать вход или вывод у композитного блока, необходимо нажать на вкладку “IO” рисунок 27.

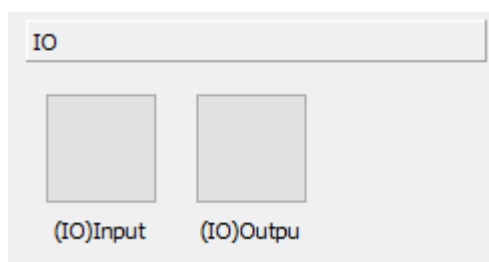


Рисунок 27 – Входы и выходы композитного блока

Соединение блоков между собой происходит по аналогии со *SPAQRL* блоками – рисунок 28.

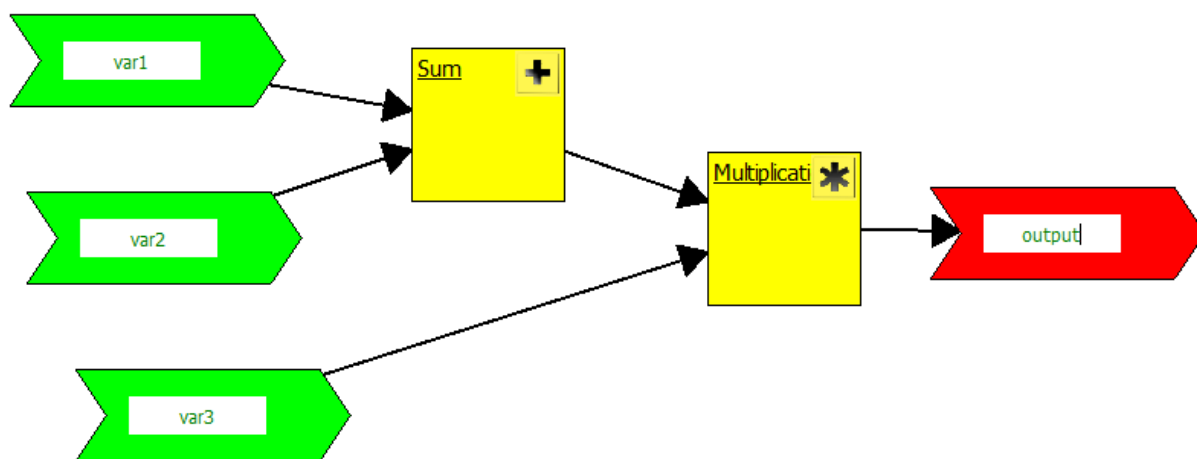


Рисунок 28 – Пример композитного блока, который складывает два значения, а затем умножает их на третье значение

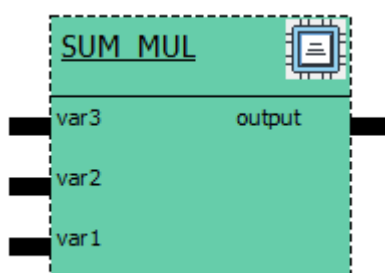


Рисунок 29 – Внешний вид композитного блока на сцене

Использование *Graphviz* для визуализации работы программы.

В программе есть встроенная библиотека – *Graphviz* [4]. С её помощью можно наглядно отследить работу блоков и их связи. После запуска проекта, программа построит схему взаимосвязи блоков. В состав блока входят: его тип, название, количество входов/выходов, значения на входе/выходе и пользовательские данные на рисунке 30.

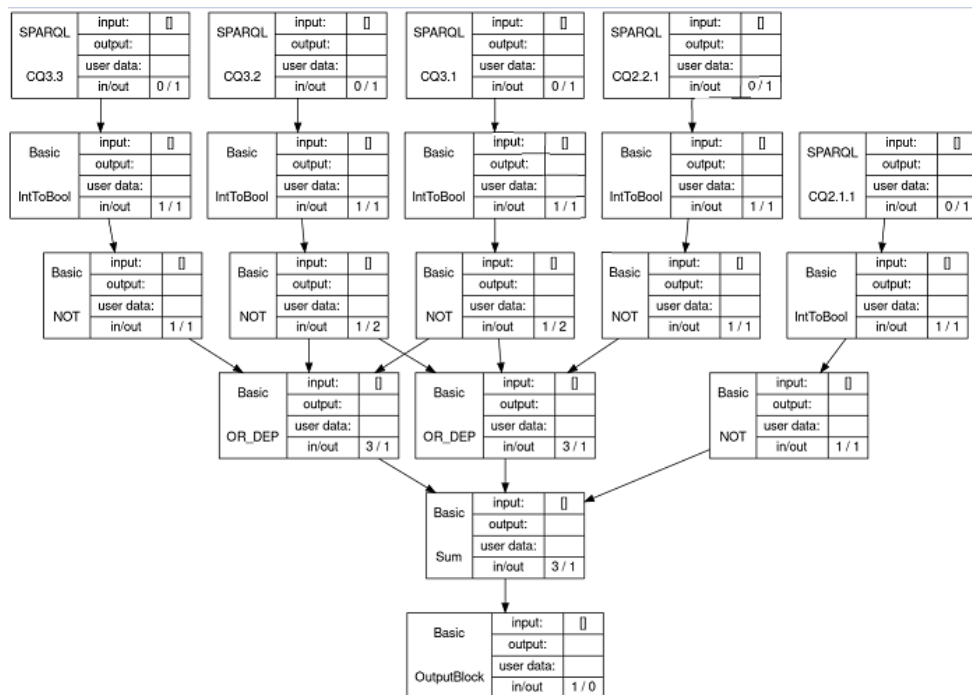


Рисунок 30 – Визуальный пример использования библиотеки *Graphviz* из пункта 1.3.7

1.3.7 Отладка и тестирование

Тестирование работоспособности интерпретатора

Для тестирования разработанного приложения, будет использована онтология для оценки свойств глубокоэшелонированной защиты общих ядерных архитектур I&C [7].

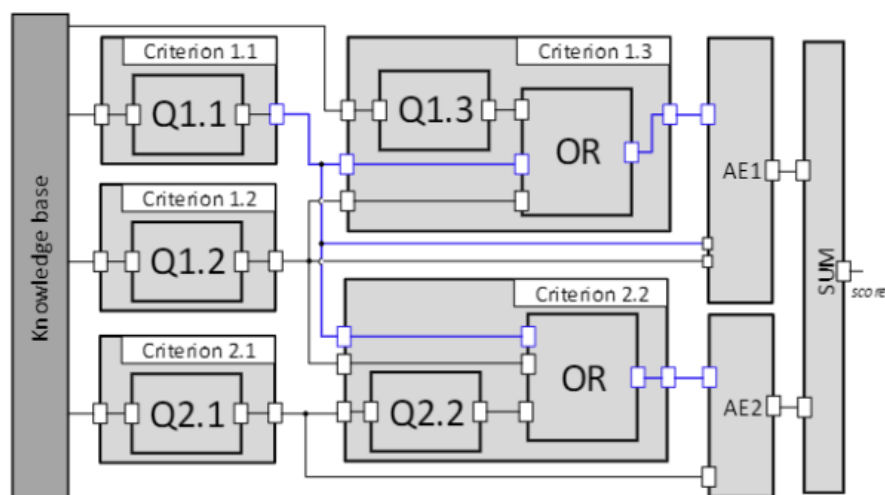


Рисунок 31 – Диаграмма выполняемого теста

Каждый критерий Q внутри будет представлять собой *SPARQL*-запрос, блок конвертации из целочисленного значения в булево, а также блок инверсии.

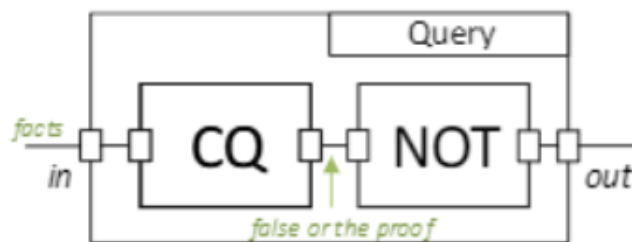


Рисунок 32 – Графическое представление запроса

Запросы, используемые в тесте:

SPARQL-представление для запроса Q1.1 – “Существует ли интерфейс между системами в разных классах безопасности, так что информация течет от более низких к более высоким классам безопасности?” – рисунок 33.

SELECT ?fromSystem ?toSystem ?interface

?fromSC ?toSC

WHERE {

?interface :interfaceTo ?toSystem.

?interface :interfaceFrom ?fromSystem.

?fromSystem :hasSafetyClass ?fromSC.

?toSystem :hasSafetyClass ?toSC.

?toSC :isHigherClassThan ?fromSC

}

ORDER BY ?interface

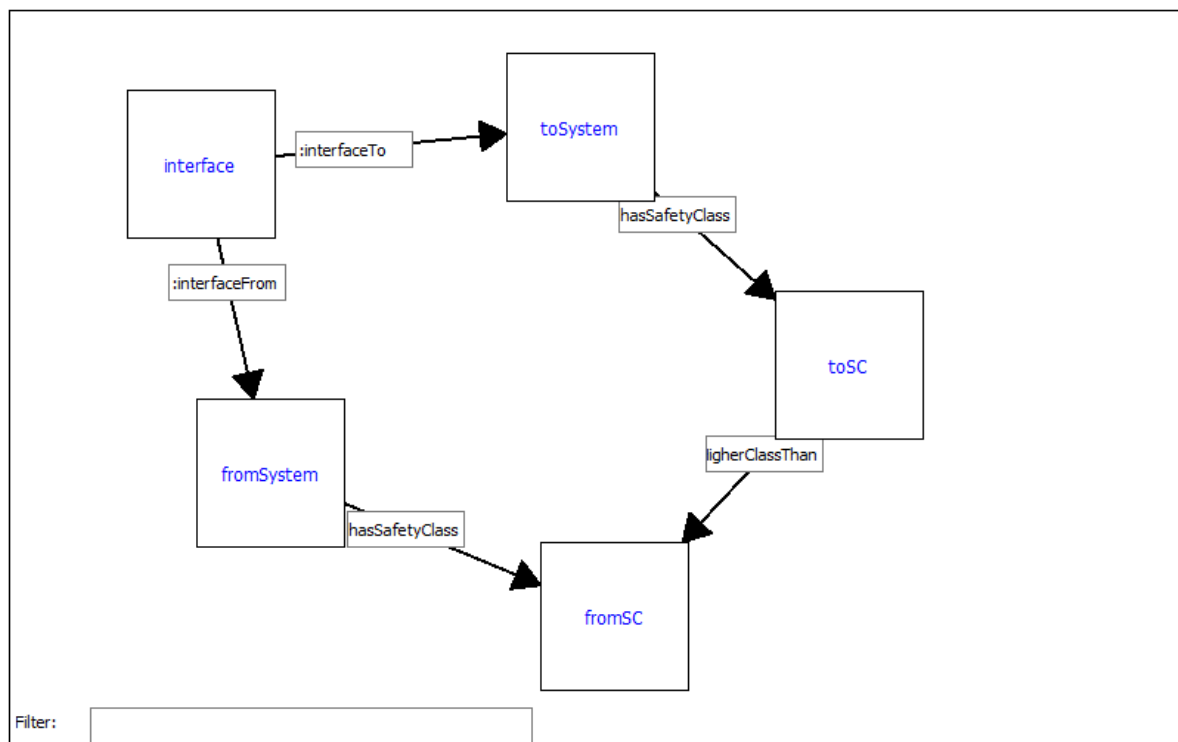


Рисунок 33 – Графическое представление запроса Q1.1

SPARQL - представление для запроса Q1.2 – “Существуют ли интерфейсы между различными системами?” – рисунок 34.

```

SELECT ?fromSystem ?toSystem ?interface
WHERE {
  ?interface :interfaceTo ?toSystem.
  ?interface :interfaceFrom ?fromSystem.
  ?toSystem :diverseSystemTo ?fromSystem
}

```

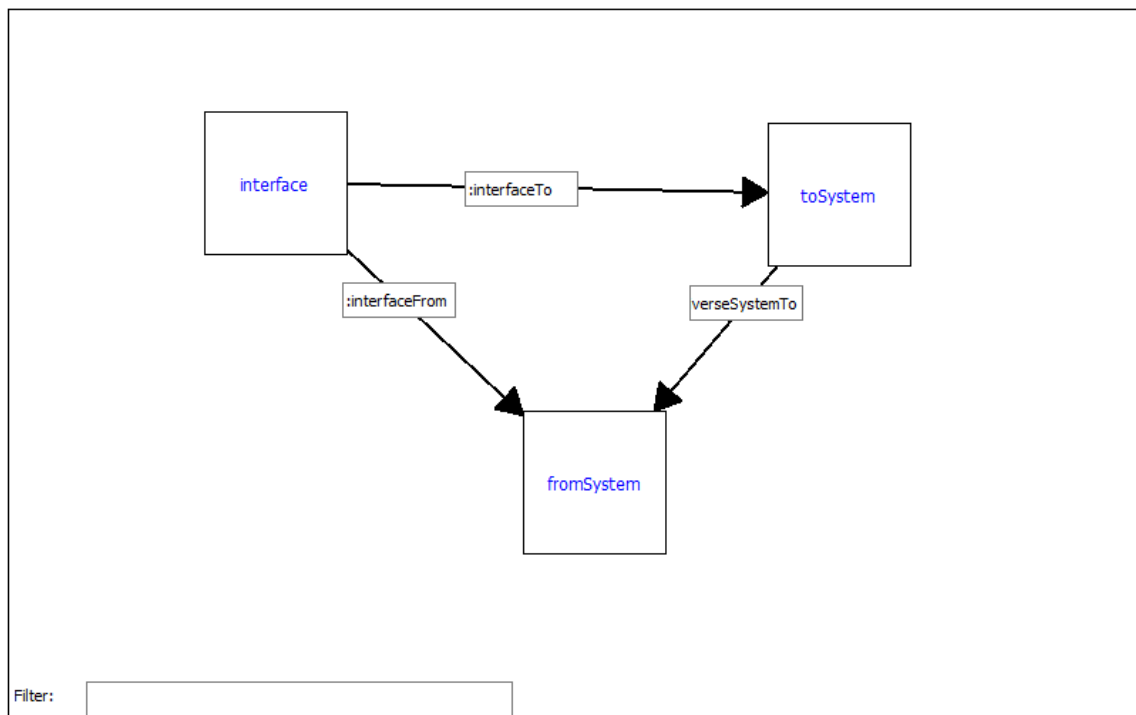


Рисунок 34 – Графическое представление запроса Q1.2

SPARQL представление для запроса Q1.3 – “Существуют ли интерфейсы между линиями глубокоэшелонированной защиты?” – рисунок 35.

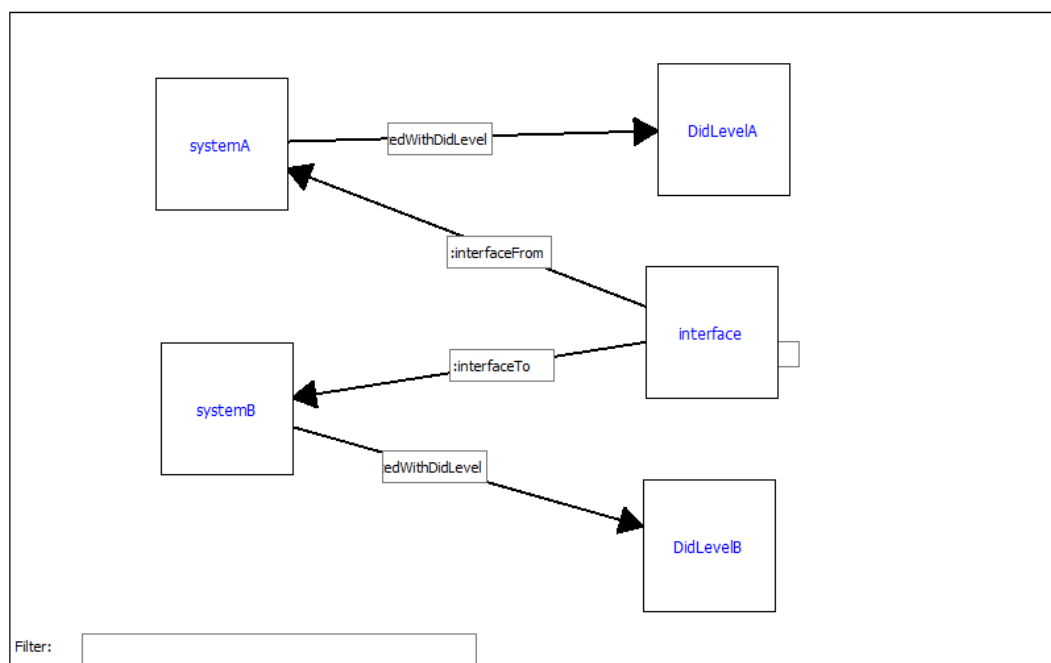


Рисунок 35 – Графическое представление запроса Q1.3

```

SELECT ?systemA ?DidLevelA ?systemB ?DidLevelB ?interface
WHERE {
  ?systemA :associatedWithDidLevel ?DidLevelA.
  ?systemB :associatedWithDidLevel ?DidLevelB.
  ?interface :interfaceFrom ?systemA.
  ?interface :interfaceTo ?systemB.
  FILTER (?DidLevelA != ?DidLevelB)
}
ORDER BY ?interface

```

SPARQL представление для запроса Q2.1 – “ Имеется ли электрическая изоляция в интерфейсах между классифицированными и неклассифицированными по безопасности системами?” – рисунок 36.

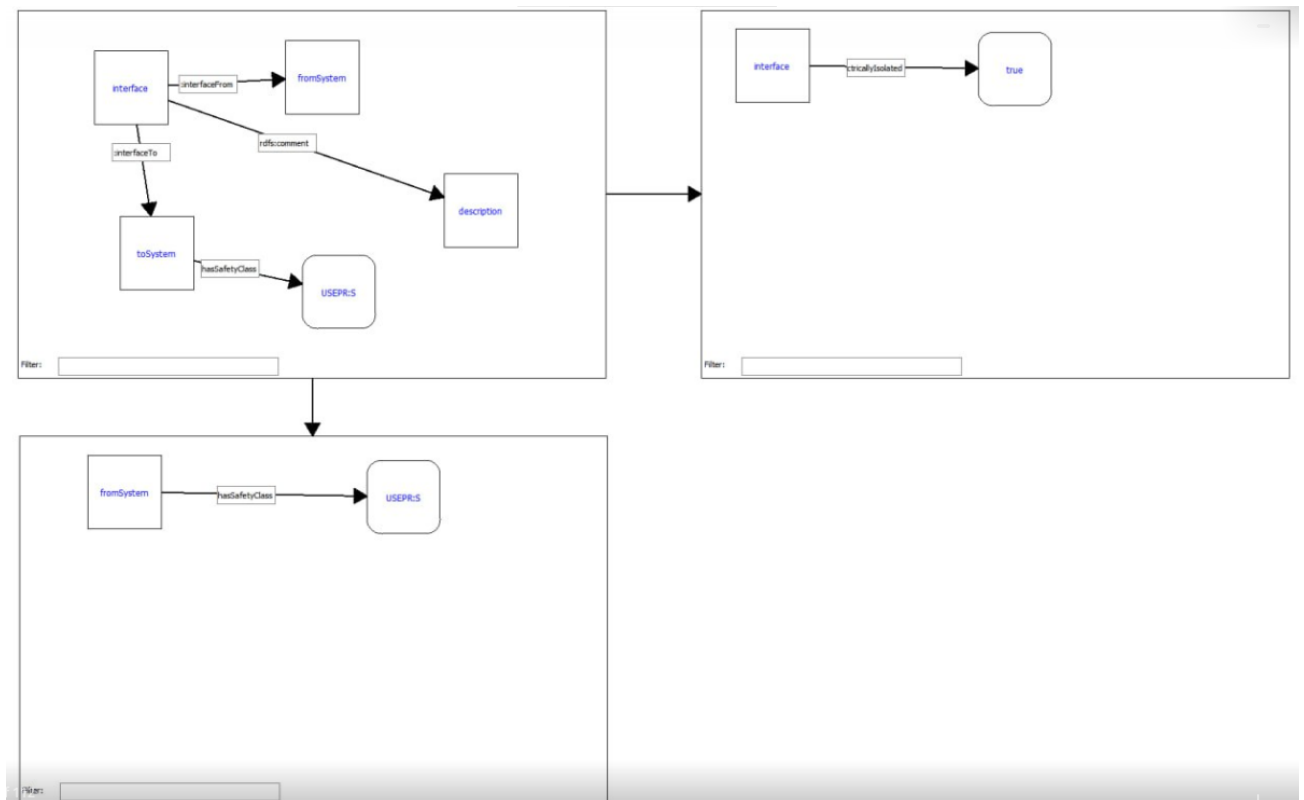


Рисунок 36 – Графическое представление запроса Q2.1

```

SELECT ?fromSystem ?toSystem ?interface ?description
WHERE { ?interface :interfaceFrom ?fromSystem.
  ?interface :interfaceTo ?toSystem.
  ?interface rdfs:comment ?description.

```

?fromSystem :hasSafetyClass USEPR:S.

MINUS {?toSystem :hasSafetyClass USEPR:S}

MINUS {?interface :electricallyIsolated

SPARQL- представление для запроса Q2.2 – “ Имеется ли оборудование (включая вспомогательные системы), которое используется двумя системами, которые должны быть электрически изолированы?”. – рисунок 37.

SELECT ?equipment ?systemA ?systemB

WHERE { ?systemA rdf:type :ICSystem.

?systemA :hasSafetyClass USEPR:S.

?systemB rdf:type :ICSystem.

MINUS {?systemB :hasSafetyClass USEPR:S}.

?equipment :partOf+ ?systemA.

?equipment :partOf+ ?systemB }

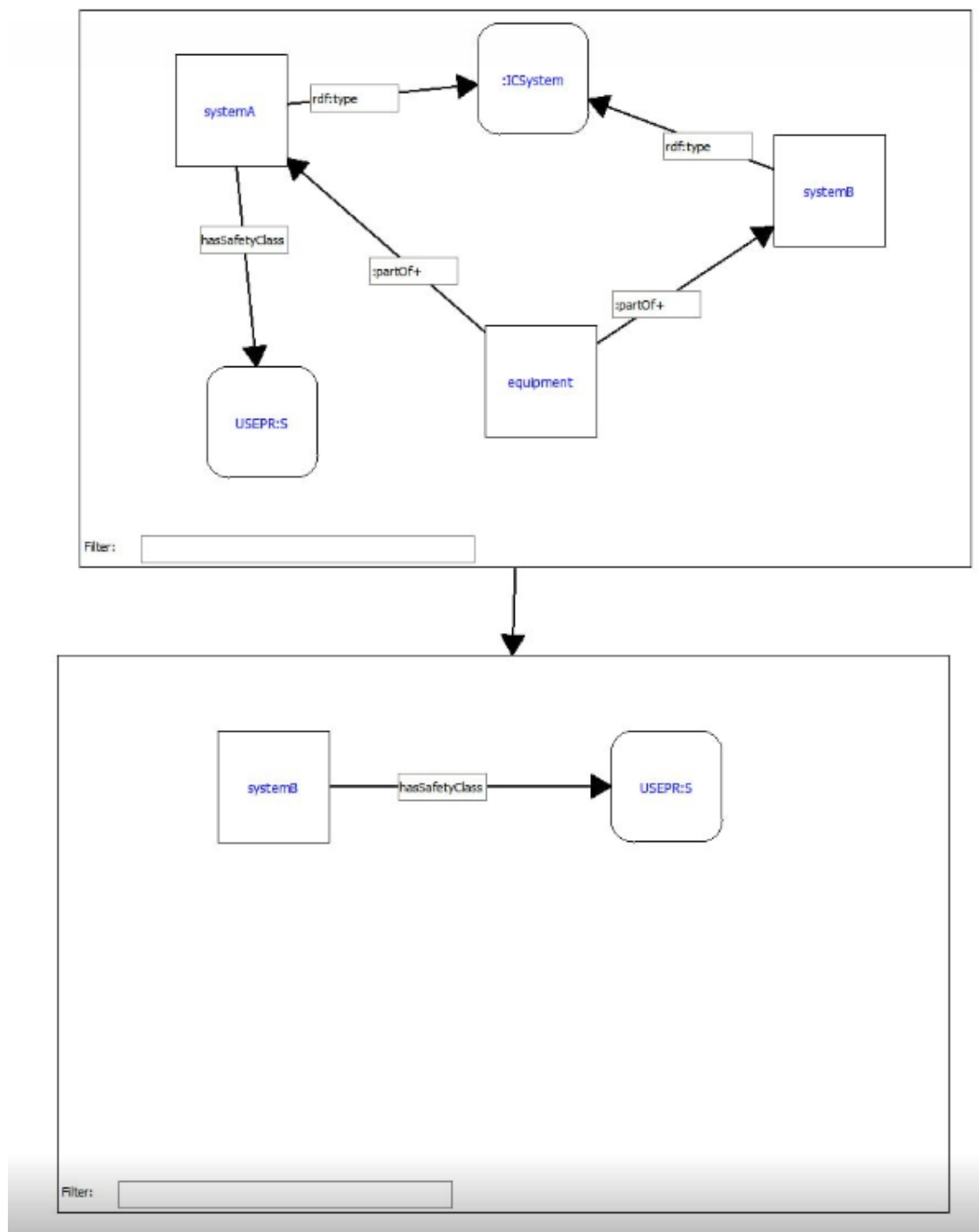


Рисунок 37 – Графическое представление запроса Q2.2

Для используемых критериев используется схожая архитектура. Рассмотрим внешний вид получившегося запроса на примере критерия Q1.1 – рисунок 38.

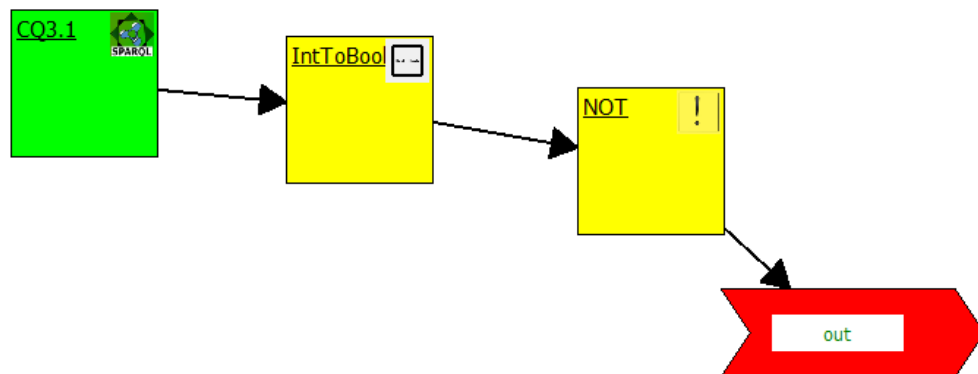


Рисунок 38 – Визуализация критерия *Q1.1* в программе *FBQL Editor*

Внешний вид остальных критериев идентичный, кроме внутреннего наполнения *SPARQL*-блоков – рисунок 39.

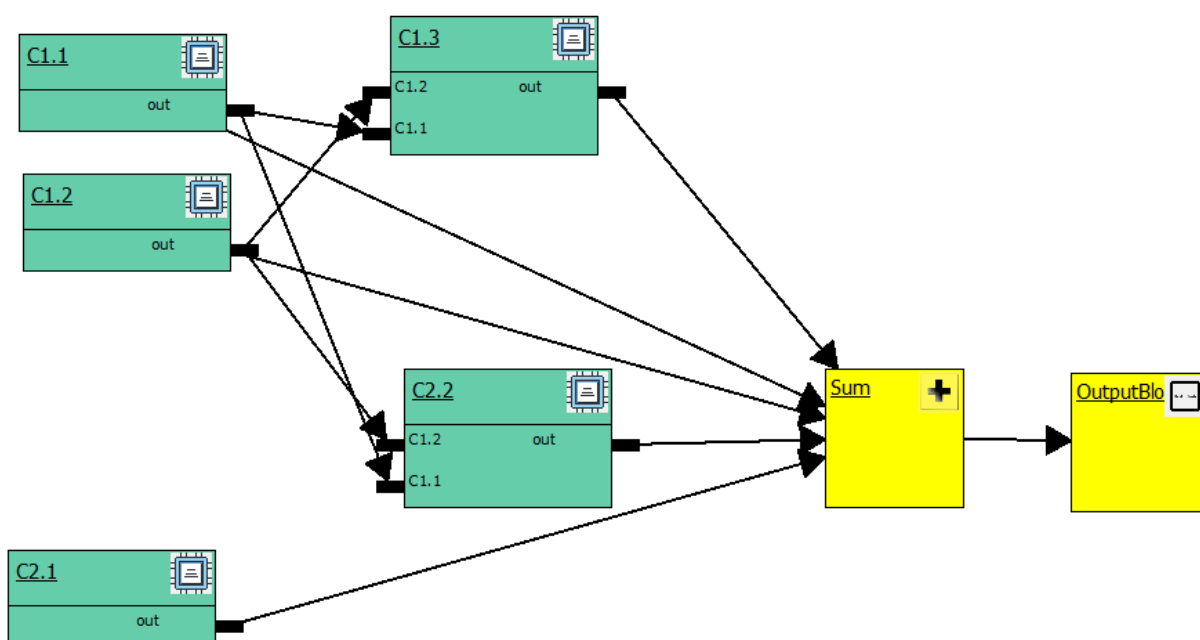


Рисунок 39 – Получившийся проект на основе композитных, *SPARQL* и базовых блоков

Работа с онтологиями данных происходит с помощью сервера *Apache Jena Fuseki*. После запуска проекта, программа выводит следующий результат – рисунок 40.

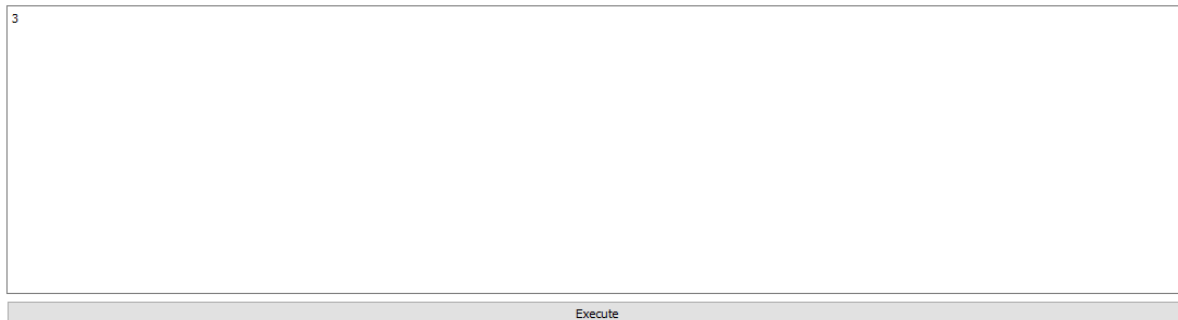


Рисунок 40 – Результат работы тестового примера

Тестирование работы метода объяснения решений

Чтобы понять результат работы теста, целесообразно воспользоваться методом объяснения решений. Для того чтобы это сделать, после работы программы, необходимо вернуться на окно создания проекта. На выходах стрелок на рисунке 41 будет виден их результат работы блоков.

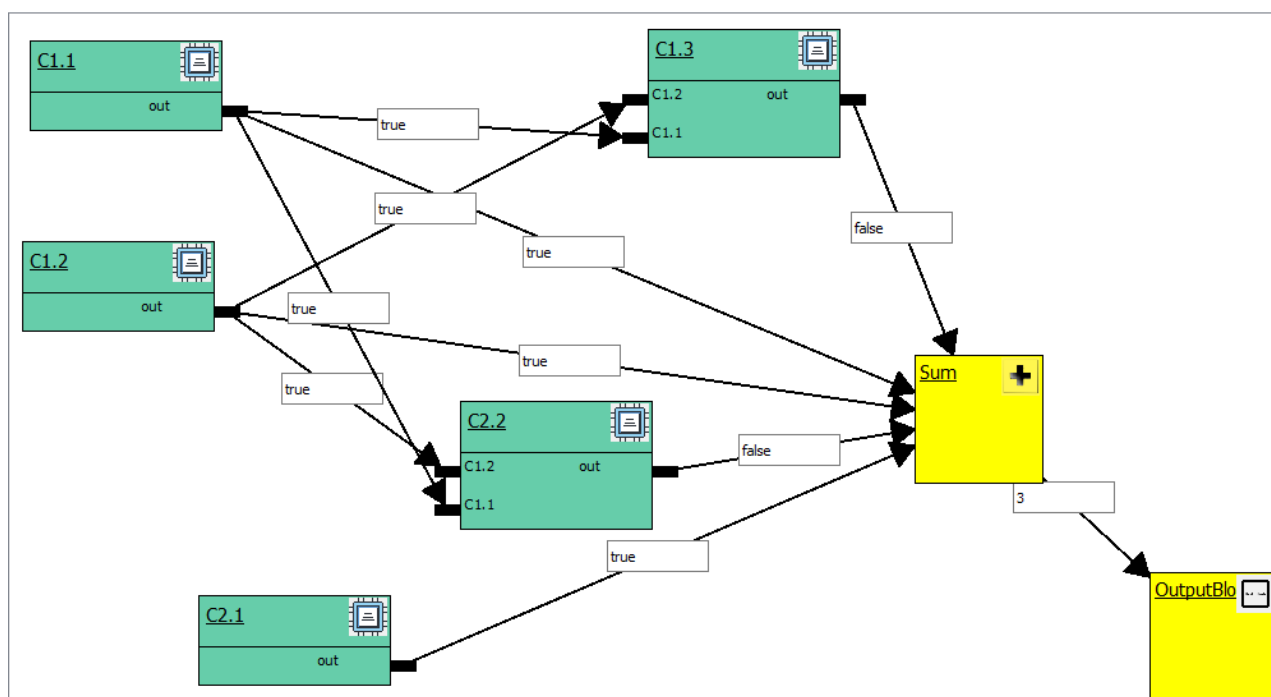


Рисунок 41 – Внешний вид окна создания проекта после работы программы

Как видно из диаграммы, на блок суммы *Sum* пришло пять значений, два из которых являются ложными, то есть 0, и три значения равных 1. Из этого следует, что ответом на данную диаграмму будет являться число – 3, количество истинных значений, пришедших в блок суммы.

Также, существует возможность в интерактивном режиме посмотреть на выходы блоков, которые привели к текущему значению. Пример, после наведения мышью на блок суммы.

Из рисунка 42 видно, что на работу блока суммы будут влиять те блоки, выходы которых отмечены красной стрелочкой – это блоки: C1.1, C1.2, C2.1. Так как у блоков C1.3 и C2.2 выходные значения являются ложными, можно сказать, что работа данных блоков не повлияет на результат работы блока суммы.

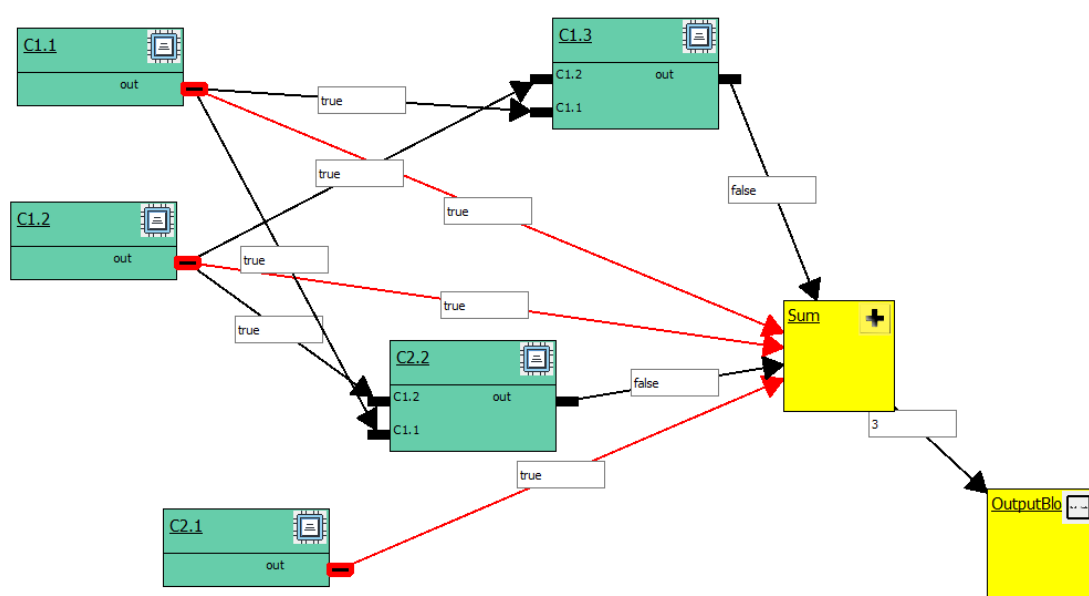


Рисунок 42 – Визуализация метода объяснения решений

Кроме того, в программе реализована работа с библиотекой *Graphviz*. С её помощью можно схематично увидеть соединения блоков, а также их входы и выходы, рисунок 43.

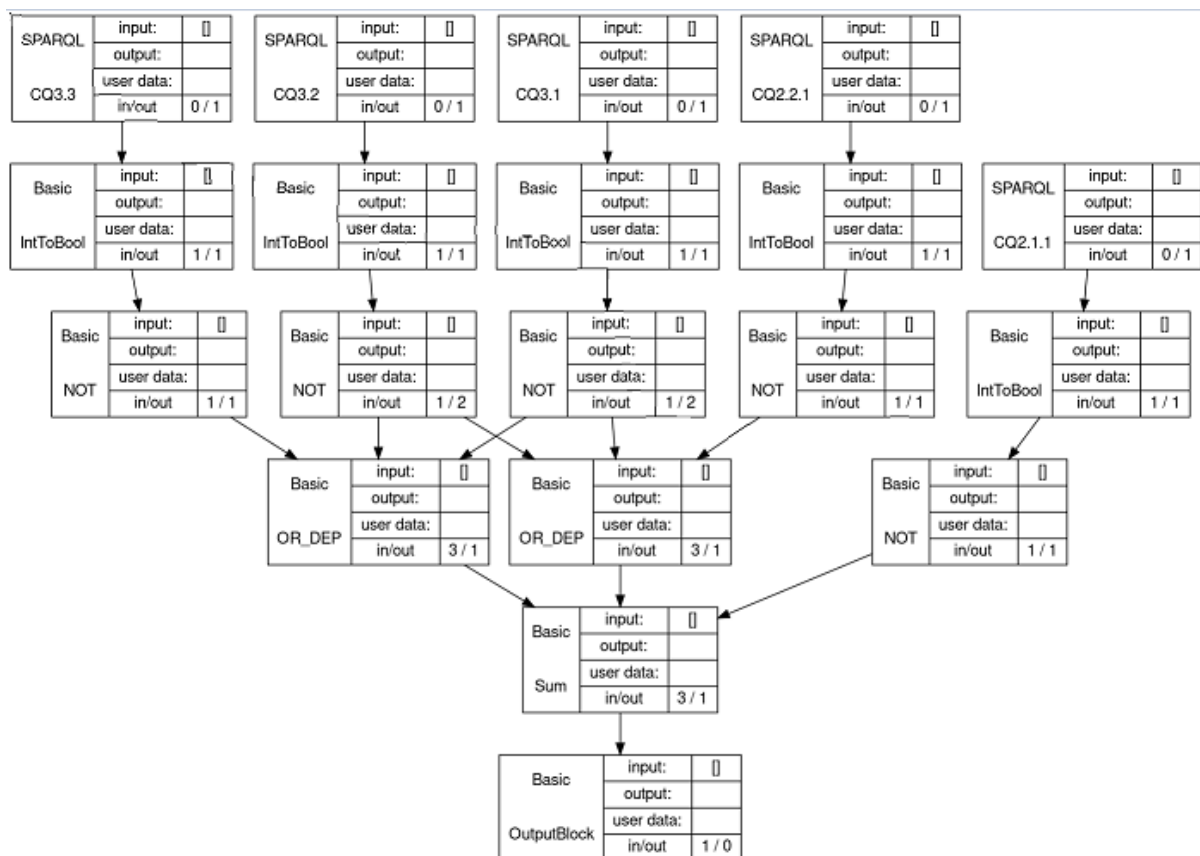


Рисунок 43 – Представление диаграммы с помощью библиотеки *Graphviz*

2 Безопасность жизнедеятельности

2.1 Общие положения

Эффективность труда специалистов всех категорий независимо от характера условий их деятельности зависит от того, как устроено и оснащено рабочее место. Организация рабочего места включает в себя оснащение всем необходимым в соответствии с характером работы, рациональное расположение этого необходимого оснащения, создание удобств, комфортных условий работы, предотвращение вредного воздействия на человека неблагоприятных факторов внешней среды. Работа оператора ЭВМ утомительна, так как связана с постоянным статическим напряжением мышц и многими другими вредными для организма явлениями, возникающими из-за относительной неподвижности рабочей позы. К этому добавляется также постоянное нервное напряжение. Такой труд нуждается в облегчении и, прежде всего в рациональном устройстве рабочего места.

Важнейшие положения в области охраны труда закреплены в 10 разделе Трудового Кодекса РФ от 30 декабря 2001 г. N 197-ФЗ. Государственными нормативными требованиями охраны труда, содержащимися в федеральных законах и иных нормативных правовых актах Российской Федерации и законах и иных нормативных правовых актах субъектов Российской Федерации об охране труда, устанавливаются правила, процедуры и критерии, направленные на сохранение жизни и здоровья работников в процессе трудовой деятельности. Требования охраны труда обязательны для исполнения юридическими и физическими лицами при осуществлении ими любых видов деятельности, в том числе при проектировании, строительстве (реконструкции) и эксплуатации объектов, конструировании машин, механизмов и другого оборудования, разработке технологических процессов, организации производства и труда.

Порядок разработки и утверждения подзаконных нормативных правовых актов об охране труда, а также сроки их пересмотра устанавливаются Правительством Российской Федерации.

Нормы охраны труда содержатся в правилах внутреннего трудового распорядка предприятий (учреждений).

Обеспечение здоровых и безопасных условий труда возлагается на администрацию предприятий (учреждений). Администрация обязана внедрить современные средства техники безопасности, предупреждающие производственный травматизм, и обеспечивать санитарно-гигиенические условия, предотвращающие возникновение профессиональных заболеваний рабочих и служащих.

Производственные здания, сооружения, оборудование, технологические процессы должны отвечать требованиям, обеспечивающим здоровые и безопасные условия труда. Эти требования включают: рациональное использование территории и производственных помещений, правильную эксплуатацию оборудования и организацию технологических процессов, защиту работающих от воздействия вредных условий труда, содержание производственных помещений и рабочих мест в соответствии с санитарно-гигиеническими нормами и правилами, устройство санитарно-бытовых

					ПГУ1.090301.09.001 ПЗ	Лист
Изм	Лист	№ докум.	Подпись	Дата		61

помещений. При проектировании, строительстве и эксплуатации производственных зданий и сооружений должны соблюдаться правила и нормы по охране труда.

Администрация предприятий (учреждений) обязана осуществлять организационную работу по обеспечению безопасных и здоровых условий труда (планирование и финансирование различных мероприятий по охране труда, проведение инструктажа рабочих и служащих по технике безопасности и производственной санитарии и т.п.).

2.2 Требования к освещению на рабочих местах, оборудованных ПЭВМ

Основная задача освещения внутри помещения — создание наилучших условий для зрения. Требования к освещению на рабочих местах, оборудованных ПЭВМ подробно освещены в шестом разделе СанПиН 2.2.2./2.4.1340-03. Согласно данному документу, освещение должно соответствовать следующим требованиям:

2.2.1 Освещенность на рабочем месте должна соответствовать характеру зрительной работы, который определяется следующими тремя параметрами: объект различения — наименьший размер рассматриваемого предмета, отдельная его часть или дефект, который необходимо различить в процессе работы (например, при работе с приборами — толщина линии градуировки шкалы; при чертежных работах — толщина самой тонкой линии на чертеже); фон — поверхность, прилегающая непосредственно к объекту различения, на которой он рассматривается; характеризуется коэффициентом отражения, зависящим от цвета и фактуры поверхности, значения которого находятся в пределах 0.02— 0.95; при коэффициенте отражения поверхности более 0,4 фон считается светлым; 0,2 — 0,4 - средним и менее 0,2 — темным; контраст объекта с фоном К характеризуется соотношением яркостей рассматриваемого объекта (точка, линия, знак, пятно, трещина, риска, раковина или другие элементы, которые требуется различить в процессе работы) и фона.

					ПГУ1.090301.09.001 ПЗ	Лист
						62
Изм	Лист	№ докум.	Подпись	Дата		

Увеличение освещенности рабочей поверхности улучшает видимость объектов за счет повышения их яркости, увеличивает скорость различения деталей, что сказывается на росте производительности труда.

2.2.2 Необходимо обеспечить достаточно равномерное распределение яркости на рабочей поверхности, а также в пределах окружающего пространства. Если в поле зрения находятся поверхности, значительно отличающиеся между собой по яркости, то при переводе взгляда с ярко освещенной на слабо освещенную поверхность глаз вынужден переадаптироваться, что ведет к повышенному утомлению зрения.

2.2.3 На рабочей поверхности должны отсутствовать резкие тени. Наличие резких теней создает неравномерное распределение поверхностей с различной яркостью в поле зрения, искажает размеры и формы объектов различения, в результате повышается утомляемость, снижается производительность труда. Особенно вредны движущиеся тени, которые могут привести к травмам. Тени необходимо смягчать, применяя, например, светильники со светорассеивающими молочными стеклами.

В отделах, оборудованных рабочими местами с ПЭВМ, необходимо предусматривать на окнах солнцезащитные устройства (жалюзи, козырьки, светорассеивающие стеклопластики), предотвращающие проникновение прямых солнечных лучей, которые создают на рабочих местах резкие тени.

2.2.4 В поле зрения должна отсутствовать прямая и отраженная блескость. Блескость — повышенная яркость светящихся поверхностей, вызывающая нарушение зрительных функций (ослепленность), т. е. ухудшение видимости объектов. Видимость V характеризует способность глаза воспринимать объект; зависит от освещенности, размера объекта, его яркости, контраста объекта с фоном, длительности экспозиции.

Прямая блескость связана с источниками света, отраженная возникает на поверхности с большим коэффициентом отражения или отражением по направлению к глазу. Ослепленность приводит к быстрому утомлению и

					ПГУ1.090301.09.001 ПЗ	Лист
						63
Изм	Лист	№ докум.	Подпись	Дата		

снижению работоспособности. Критерием оценки слепящего действия, создаваемого осветительной установкой, является показатель ослепленности P_o значение которого определяют по формуле:

$$P_o = (V_1/V_2 - 1) * 1000,$$

где V_1 — видимость объекта различения при экранировании;

V_2 — видимость объекта различения при наличии ярких источников света в поле зрения.

Экранирование источников света осуществляют с помощью щитков, козырьков и т. п. Прямую блескость ограничивают уменьшением яркости источников света, правильным выбором защитного угла светильника, увеличением высоты подвеса светильников. Отраженную блескость ослабляют правильным выбором направления светового потока на рабочую поверхность, а также изменением угла наклона рабочей поверхности. Там, где это возможно, следует заменять блестящие поверхности матовыми.

2.2.5 Величина освещенности должна быть постоянной во времени. Колебания освещенности, вызванные резким изменением напряжения в сети, имеют большую амплитуду, каждый раз вызывая переадаптацию глаза, приводят к значительному утомлению. Пульсация освещенности связана также с особенностью работы газоразрядных ламп.

Коэффициент пульсации освещенности K_p — критерий оценки относительной глубины колебаний освещенности в результате изменения во времени светового потока газоразрядных ламп при питании их переменным током.

Постоянство освещенности во времени достигается стабилизацией питающего напряжения, жестким креплением светильников, применением специальных схем включения газоразрядных ламп. Например, снижение коэффициента пульсации освещенности люминесцентных ламп с 55 до 5% (при трехфазном включении) приводит к уменьшению утомления и повышению производительности труда на 15% для работ высокой точности.

2.2.6 Следует выбирать оптимальную направленность светового потока, что позволяет в одних случаях рассмотреть внутренние поверхности деталей, в других — различить рельефность элементов рабочей поверхности.

2.2.7 Следует выбирать необходимый спектральный состав света. Это требование особенно существенно для обеспечения правильной цветопередачи, а в отдельных случаях для усиления цветовых контрастов. Правильную цветопередачу обеспечивают естественное освещение и искусственные источники света со спектральной характеристикой, близкой к солнечной. Для создания цветовых контрастов применяют монохроматический свет, усиливающий одни цвета и ослабляющий другие.

2.2.8 Все элементы осветительных установок — светильники, групповые щитки, понижающие трансформаторы, осветительные сети — должны быть достаточно долговечными, электробезопасными, а также не должны быть причиной возникновения пожара или взрыва. Обеспечение указанных условий достигается применением зануления или заземления, ограничением напряжения для питания местных и переносных светильников до 42 В и ниже (36, 24, 12 В), выбором оборудования, соответствующего условиям среды в помещениях, и защитой элементов осветительных сетей от механических повреждений при эксплуатации. Кроме того, необходимо уменьшить до минимума теплоту, выделяемую осветительной установкой, и шум.

2.3 Выводы

В ходе выполнения раздела дипломного проекта по безопасности жизнедеятельности, был рассмотрен ряд законодательных документов по охране труда и проанализирована другая литература, связанная с тематикой раздела. Согласно заданию, было рассмотрено освещение и в соответствии с проведенным анализом рассчитано необходимое количество светильников, которое обеспечивает необходимое освещение помещения. Таким образом, выяснено, что для наиболее благоприятной работы оператора необходимо установить 9 светильников типа ОД.

					ПГУ1.090301.09.001 ПЗ	Лист
						65
Изм	Лист	№ докум.	Подпись	Дата		

Заключение

В ходе выполнения выпускной квалификационной работы была разработана программа, включающая в себя интерпретатор диаграмм графических запросов, реализация метода объяснения решений, а также методику выполнения и обработки результатов обращения к хранилищам данных *RDF*. Данная программа используется для оценки свойств систем, представленных в виде онтологии на основе *RDF*, она также обладает функционалом для работы с сервером *Apache Jena Fuseki*, который обрабатывает запросы к хранилищам данных.

В рамках создания программы были разработаны:

- семантика выполнения функционально-блочных диаграмм;
- концепция метода графического объяснения решений;
- графический редактор для работы с блоками;
- возможность настройки работы с *Jena Fuseki Server*;
- блок исполнения схемы с выводом промежуточных и итоговых результатов.

При создании программы также возникли некоторые затруднения:

- не удалось реализовать все возможности *SPARQL*, например выстраивание чёткой последовательности блоков «*MINUS*» и «*UNION*»;
- неполная работа с библиотекой *Graphviz* на *OC Windows*;
- недостаточная оптимизация использованных методов.

В дальнейшем исследовании данной темы есть идеи по улучшению функционала программы, а также исправлению некоторых недоработок. Например, реализовать надстройку над языком *SPARQL*, а именно – реализацию условных операций и циклов.

Список использованных источников

1. Документация Qt. [Электронный ресурс]: Режим доступа: <https://doc.qt.io>
2. Документация SPARQL. [Электронный ресурс]: Режим доступа: <https://www.w3.org/TR/rdf-sparql-query/>
3. Документация Apache Jena Fuseki Server. [Электронный ресурс]: Режим доступа: <https://jena.apache.org/documentation/fuseki2/>
4. Документация Graphviz. [Электронный ресурс]: Режим доступа: <https://graphviz.org/documentation/>
5. Муромский Д.А. Кобзев М.И. Условные операторы для SPARQL запросов к RDF-хранилищам данных // Сб. ст. X Ежегодн. всеросс. межвуз. науч.-практ. конф. «Информационные технологии в науке и образовании. Проблемы и перспективы (ВМНПК – 2023)», Пенза, 2023. – с. 310-313.
6. Ovsianikova P., Pakonen A., Muromsky D., Kobzev M., Dubinin V., Vyatkin V. Formal verification of non-functional requirements of overall instrumentation and control architectures // IEEE Open Journal of the Industrial Electronics Society. 2023 (submitted).
7. Ontology for assessing defence-in-depth properties of nuclear overall I&C architectures. [Электронный ресурс]: Режим доступа: <https://zenodo.org/record/5010644#.Yg96EuhBwuU>
8. Документация IEC 61131-3. [Электронный ресурс]: Режим доступа: https://d1.amobbs.com/bbs_upload782111/files_31/ourdev_569653.pdf
9. AMES H. CHRISTENSEN., THOMAS STRASSER., ANTONIO VALENTINI., VALERIY VYATKIN., NZ ALOIS ZOITL. The IEC 61499 Function Block Standard: Overview of the Second Edition. [Электронный ресурс]: Режим доступа: https://holobloc.com/papers/iec61499/61499_ED2_OVW.pdf
10. Документация Data Flow Diagrams (DFD). [Электронный ресурс]: Режим доступа: <https://www.lucidchart.com/pages/data-flow-diagram>

11. Документация Data-Driven Architecture [Электронный ресурс]:
Режим доступа: <https://habr.com/ru/companies/timeweb/articles/590675/>
12. Документация MATLAB Simulink. [Электронный ресурс]: Режим
доступа: <https://docs.exponenta.ru/simulink/index.html>
13. Документация Owen Logic. [Электронный ресурс]: Режим
доступа:
https://owen.ru/product/programmnoe_obespechenie_owen_logic/documentation
14. Документация RDF. [Электронный ресурс]: Режим доступа:
https://www.w3.org/2007/03/rdf_concepts_ru/Overview.html

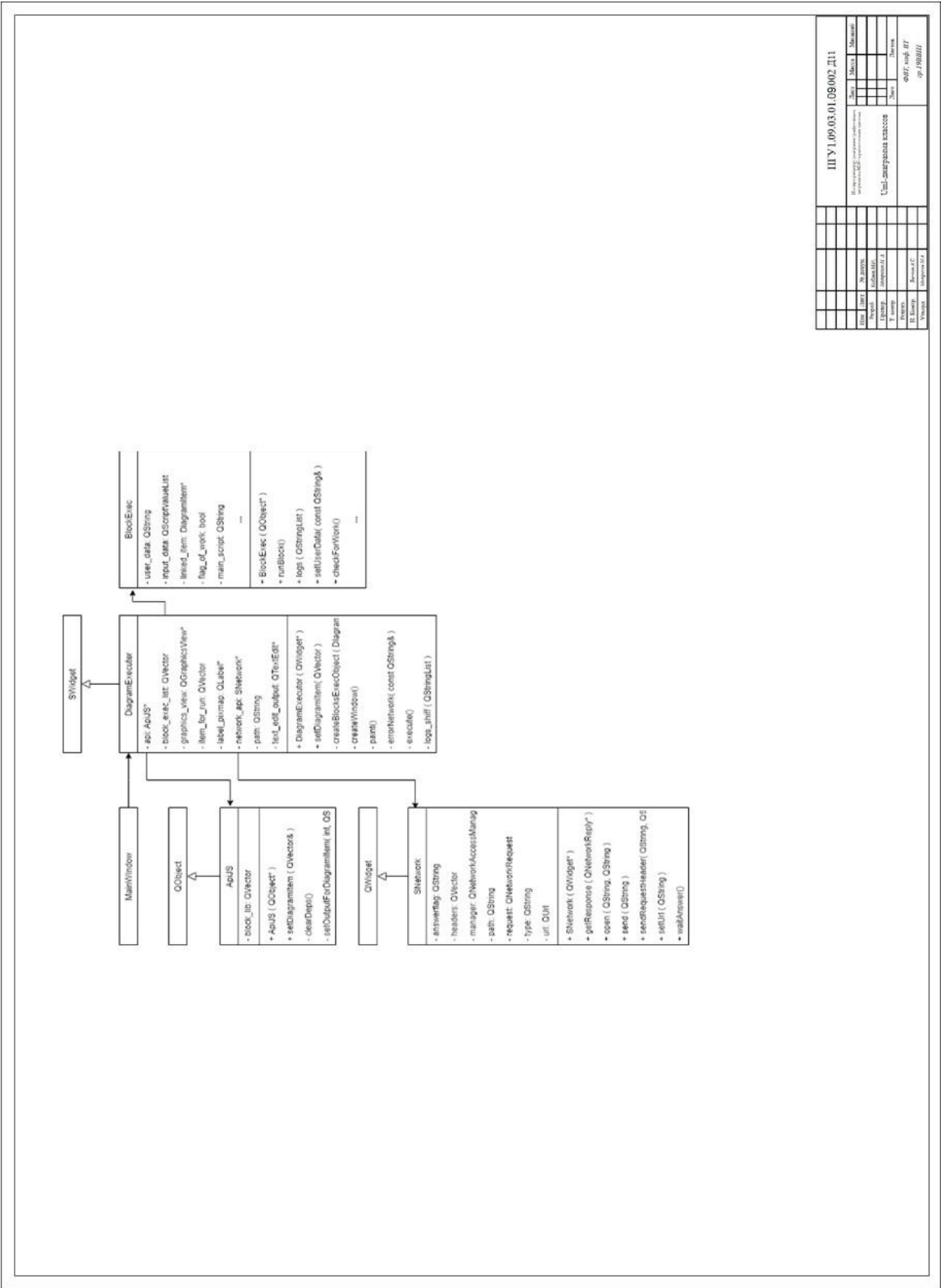
Приложение А

(обязательное)

Интерпретатор диаграмм графических запросов к RDF-хранилищам данных
UML-диаграммы

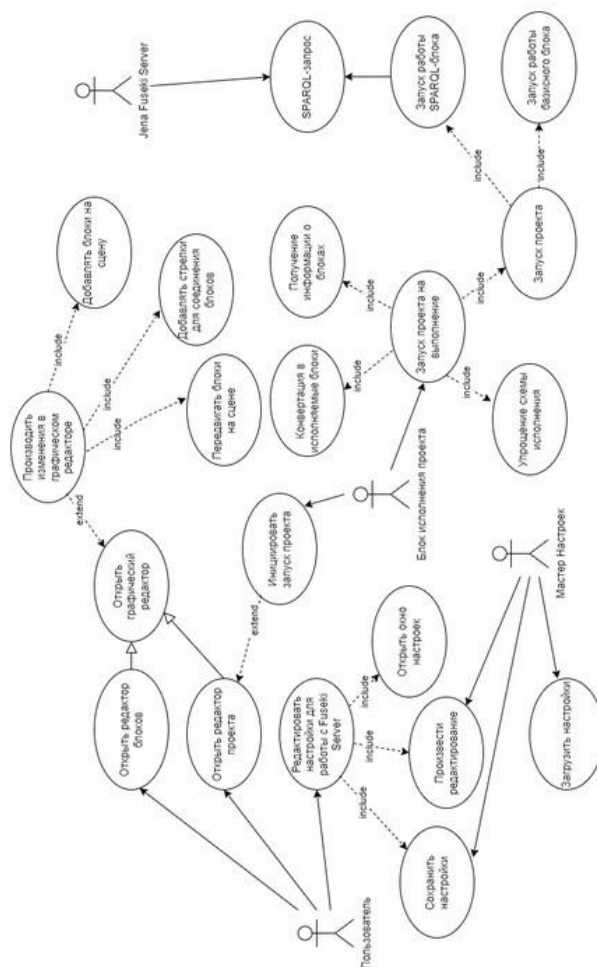
					ПГУ1.090301.09.001 ПЗ	Лист
						69
Изм	Лист	№ докум.	Подпись	Дата		

Приложение А.1 – Диаграмма классов



--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Приложение А.2 – UML-диаграмма прецедентов



ИТУ 1.09.03.01.09.002 Д13									
Имя	Фамилия	Имя	Фамилия	Имя	Фамилия	Имя	Фамилия	Имя	Фамилия
Подпись	Подпись	Подпись	Подпись	Подпись	Подпись	Подпись	Подпись	Подпись	Подпись
Дата	Дата	Дата	Дата	Дата	Дата	Дата	Дата	Дата	Дата
ФЭТ, сф. БТ									
Ф.И.О. БТ									

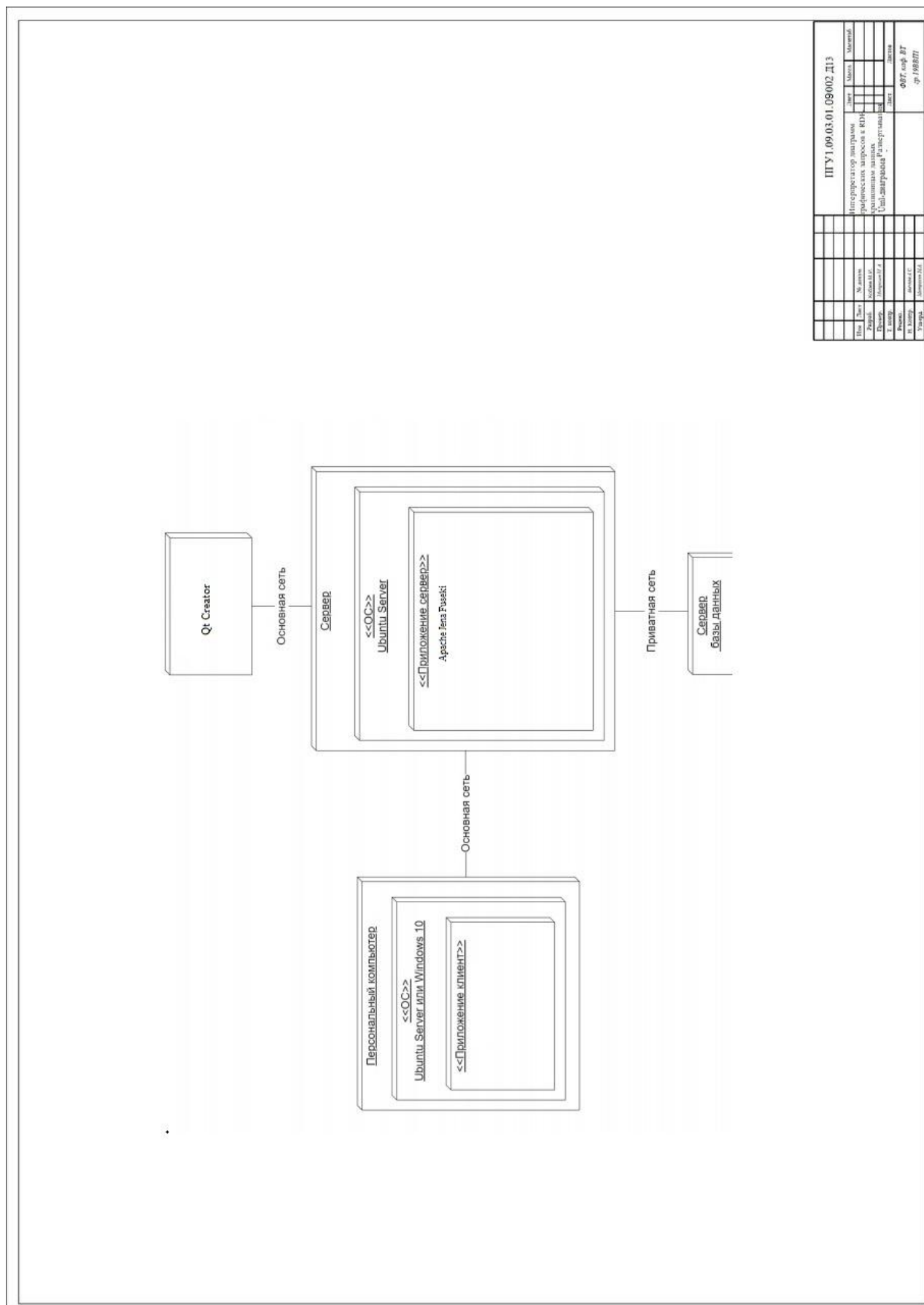
Изм	Лист	№ докум.	Подпись	Дата

ПГУ1.090301.09.001 ПЗ

Лист

71

Приложение А.3 – UML-диаграмма развертывания

[illegible]

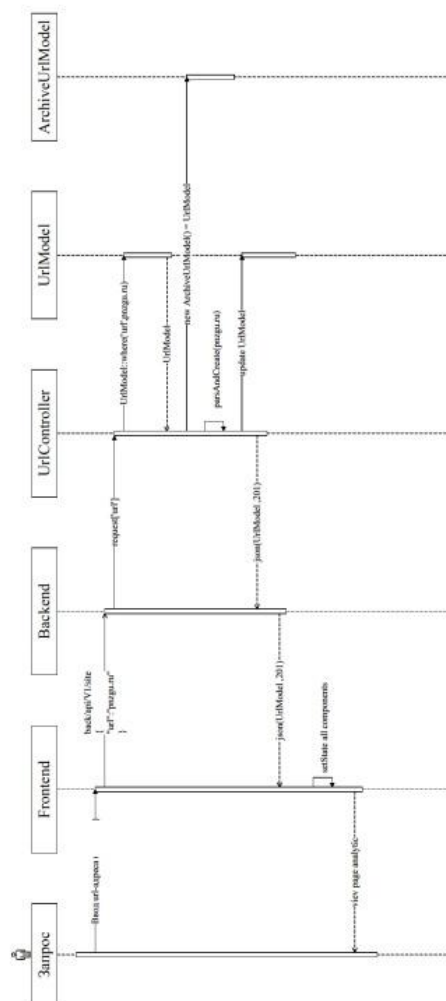
Изм	Лист	№ докум.	Подпись	Дата

ПГУ1.090301.09.001 ПЗ

Лист

72

Приложение А.4 – Диаграмма последовательности

[illegible]

Изм	Лист	№ докум.	Подпись	Дата

ПГУ1.090301.09.001 ПЗ

Лист

73

Приложение Б

(обязательное)

Интерпретатор диаграмм графических запросов к RDF-хранилищам данных

Листинг программы

					ПГУ1.090301.09.001 ПЗ	Лист
Изм	Лист	№ докум.	Подпись	Дата		74

В листинге представлены некоторые основные файлы программы, больше исходников можно найти на сайте: [Электронный ресурс] Режим доступа: <https://github.com/FBQLEditor/FBQLEditor>

Файл snetwork.cpp

```
#include "snetwork.h"

SNetwork::SNetwork( QWidget* parent )
    : QWidget( parent )
{
    manager = new QNetworkAccessManager( this );
    connect( manager, SIGNAL( finished( QNetworkReply* ) ), this,
        SLOT( getResponse( QNetworkReply* ) ) );
}

void SNetwork::setUrl( QString str )
{
    url.setUrl( str );
}

void SNetwork::open( QString type, QString path )
{
    this->type = type;
    this->path = path;
}

void SNetwork::setRequestHeader( QString name, QString value )
{
    QVector<QString> temp = { name, value };
    headers.push_back( temp );
}

void SNetwork::send( QString data )
{
    request.setUrl( url );

    for ( int i = 0; i < headers.size(); i++ )
    {
        request.setRawHeader( headers[i][0].toUtf8(),
headers[i][1].toUtf8() );
    }

    request.setHeader( QNetworkRequest::ContentTypeHeader,
"application/x-www-form-urlencoded" );

    if ( type == "POST" )
    {
        manager->post( request, data.toUtf8() );
    }
    else
    {
        manager->get( request );
    }
}
```

					ПГУ1.090301.09.001 ПЗ	Лист
						75
Изм	Лист	№ докум.	Подпись	Дата		

```

    }
}

namespace
{
void delay( int millisecondsToWait )
{
    QTime dieTime = QTime::currentTime().addMSecs(
millisecondsToWait );
    while ( QTime::currentTime() < dieTime )
    {
        QCoreApplication::processEvents( QEventLoop::AllEvents,
100 );
    }
}
} // namespace

QString SNetwork::waitAnswer()
{
    int waitTime = 3000;
    while ( answerflag == "" && waitTime )
    {
        delay( 100 );
        waitTime -= 100;
    }
    if ( !waitTime )
    {
        emit ERROR( "Server timeout error!" );
    }
    QString result = answerflag;
    answerflag = "";
    return result;
}

void SNetwork::getResponse( QNetworkReply* reply )
{
    answerflag = reply->readAll();
}

```

Файл diagramexecutor.cpp

```

#include "diagramexecutor.h"

#include <QGridLayout>
#include <QPushButton>

#include <compositeblocksettings.h>
#include <diagramarrow.h>
#include <diagramitembasic.h>
#include <diagramitemio.h>
#include <diagramitemspargl.h>

#ifdef Q_OS_LINUX
#define FUSEKI_SERVER_PATH "FusekiServer/fuseki-server"

```

```

#include <graphviz/cgraph.h>
#include <graphviz/gvc.h>
#else
#define FUSEKI_SERVER_PATH "FusekiServer/fuseki-server.bat"
#endif

DiagramExecutor::DiagramExecutor( QWidget* parent )
    : QWidget( parent )
{
    createWindow();

    network_api = new SNetwork( this );
    connect( network_api, SIGNAL( ERROR( QString ) ), this, SLOT(
errorNetwork( QString ) ) );
    api = new ApiJS( this );
}

void DiagramExecutor::errorNetwork( const QString& error_message )
{
    text_edit_output->setText( error_message );
}

BlocksExec* DiagramExecutor::createBlocksExecObject(
DiagramItemSettings* settings )
{
    BlocksExec* block_exec = new BlocksExec( this );
    block_exec->setValueForEngine(
        { { "log_output", text_edit_output },
          { "network", network_api } } );
    block_exec->setScript( settings->getScript() );
    block_exec->setSettings( settings );
    return block_exec;
}

void DiagramExecutor::setDiagramItem( QVector<DiagramItem*>&
item_list )
{
    // Добавление блоков
    for ( auto item : item_list )
    {
        auto block_exec = createBlocksExecObject( item-
>getSettings() );
        block_exec->setDiagramItem( item );
        block_exec->setUserData( item->getInputData() );
        blocks_exec_list.push_back( block_exec );
    }
    paint();

    // Настройка связей
    for ( auto i = 0; i < item_list.size(); ++i )
    {
        auto arrows = item_list[i]->getEndArrows();
        for ( auto arrow : arrows )
        {

```

```

        int index = item_list.indexOf( arrow->startItem() );
        blocks_exec_list[i]->addPrevBlocks(
blocks_exec_list[index] );
        blocks_exec_list[index]->addNextBlocks(
blocks_exec_list[i] );
    }
}
paint();

// Удаление IO блоков
for ( auto i = 0; i < blocks_exec_list.size(); ++i )
{
    auto block_settings = blocks_exec_list[i]->getSettings();
    if ( DiagramItemSettings::IOItemSettingsType ==
block_settings->typeSettings() )
    {
        auto type_block = static_cast<IOBlockSettings*>(
block_settings )->type_block;
        auto text = static_cast<IOBlockSettings*>(
block_settings )->text;
        auto io_block = blocks_exec_list[i];
        auto vec_prev_blocks = io_block->getPrevBlocks();
        auto vec_next_blocks = io_block->getNextBlocks();
        for ( auto prev : vec_prev_blocks )
        {
            for ( auto next : vec_next_blocks )
            {
                prev->addNextBlocks( next );
                next->addPrevBlocks( prev );
                if ( IOBlockSettings::TypeIO::Input ==
type_block
                        &&
DiagramItemSettings::CompositeItemSettingsType == next-
>getSettings()->typeSettings() )
                {
                    next->removeConnectName( io_block );
                    next->addBlockConnectName( text, prev );
                    if (
DiagramItemSettings::CompositeItemSettingsType == prev-
>getSettings()->typeSettings() )
                    {
                        auto name_connect = prev-
>getBlockConnectName( io_block );
                        if ( !name_connect.isEmpty() )
                        {
                            prev->removeConnectName( io_block
);
                            prev->addBlockConnectName(
name_connect, next );
                        }
                    }
                }
            }
        }
    }
}

```

```

        if ( IOBlockSettings::TypeIO::Output ==
type_block
        &&
DiagramItemSettings::CompositeItemSettingsType == prev-
>getSettings()->typeSettings() )
        {
            prev->removeConnectName( io_block );
            prev->addBlockConnectName(
static_cast<IOBlockSettings*>( block_settings )->text, next );
            prev->addLinkedItemForComposite(
static_cast<IOBlockSettings*>( block_settings )->text, io_block-
>getDiagramItem() );
            if (
DiagramItemSettings::CompositeItemSettingsType == next-
>getSettings()->typeSettings() )
            {
                auto name_connect = next-
>getBlockConnectName( io_block );
                if ( !name_connect.isEmpty() )
                {
                    next->removeConnectName( io_block
);
                    next->addBlockConnectName(
name_connect, prev );
                }
            }
        }
        io_block->removeConnections();
        io_block->deleteLater();
        blocks_exec_list.erase( blocks_exec_list.begin() + i
);
        --i;
    }
    paint();
    // Просмотр КОМПОЗИТНЫХ БЛОКОВ
    for ( auto i = 0; i < blocks_exec_list.size(); ++i )
    {
        if ( DiagramItemSettings::CompositeItemSettingsType ==
blocks_exec_list[i]->getSettings()->typeSettings() )
        {
            auto composite = blocks_exec_list[i];
            auto settings = static_cast<CompositeBlockSettings*>(
composite->getSettings() );
            QVector<BlocksExec*> new_blocks;
            for ( auto& block_set : settings->blocks )
            {
                auto block_exec = createBlocksExecObject(
block_set );
                if ( DiagramItemSettings::BasicItemSettingsType ==
block_set->typeSettings() )
                {

```

```

        block_exec->setUserData(
static_cast<BasicBlockSettings*>( block_set )->line_edit_text );
    }

    new_blocks.push_back( block_exec );
    blocks_exec_list.push_back( block_exec );
}

for ( const auto& line : settings->lines )
{
    auto start_block = new_blocks[line.start_block];
    auto end_block = new_blocks[line.end_block];
    start_block->addNextBlocks( end_block );
    end_block->addPrevBlocks( start_block );

    if (
DiagramItemSettings::CompositeItemSettingsType == start_block-
>getSettings()->typeSettings() )
    {
        if ( !line.text_start.isEmpty() )
        {
            start_block->addBlockConnectName(
line.text_start, end_block );
        }
        else
        {
            emit ERROR(
"DiagramExecutor::setDiagramItem() -> Line without text" );
        }
    }
    if (
DiagramItemSettings::CompositeItemSettingsType == end_block-
>getSettings()->typeSettings() )
    {
        if ( !line.text_end.isEmpty() )
        {
            end_block->addBlockConnectName(
line.text_end, start_block );
        }
        else
        {
            emit ERROR(
"DiagramExecutor::setDiagramItem() -> Line without text" );
        }
    }
}

paint();

for ( int k = 0; k < new_blocks.size(); ++k )
{
    auto block = new_blocks[k];
    if ( DiagramItemSettings::IOItemSettingsType ==
block->getSettings()->typeSettings() )

```

```

        {
            if ( IOBlockSettings::TypeIO::Input ==
static_cast<IOBlockSettings*>( block->getSettings() )->type_block
)
                {
                    for ( auto io : composite->getPrevBlocks()
)
                        {
                            if ( composite->getBlockConnectName(
io )
                                == static_cast<IOBlockSettings*>(
block->getSettings() )->text )
                                {
                                    for ( auto next : block-
>getNextBlocks() )
                                        {
                                            io->addNextBlocks( next );
                                            next->addPrevBlocks( io );
                                            if (
DiagramItemSettings::CompositeItemSettingsType == next-
>getSettings()->typeSettings() )
                                                {
                                                    next->addBlockConnectName(
next->getBlockConnectName( block ), io );
                                                }
                                            if (
DiagramItemSettings::CompositeItemSettingsType == io-
>getSettings()->typeSettings() )
                                                {
                                                    io->addBlockConnectName(
io->getBlockConnectName( composite ), next );
                                                }
                                            }
                                        }
                                    }
                                }
                            else
                                {
                                    for ( auto io : composite->getNextBlocks()
)
                                        {
                                            if ( composite->getBlockConnectName(
io )
                                                == static_cast<IOBlockSettings*>(
block->getSettings() )->text )
                                                {
                                                    auto item = composite-
>getLinkedItemForComposite( composite->getBlockConnectName( io )
);
                                                    for ( auto prev : block-
>getPrevBlocks() )
                                                        {
                                                            prev->setDiagramItem( item );
                                                            io->addPrevBlocks( prev );

```

```

prev->addNextBlocks( io );
if (
DiagramItemSettings::CompositeItemSettingsType == prev-
>getSettings()->typeSettings() )
{
prev->addBlockConnectName(
prev->getBlockConnectName( block ), io );
}
if (
DiagramItemSettings::CompositeItemSettingsType == io-
>getSettings()->typeSettings() )
{
io->addBlockConnectName(
io->getBlockConnectName( composite ), prev );
}
}
}
}
block->removeConnections();
blocks_exec_list.erase(
blocks_exec_list.begin() + blocks_exec_list.indexOf( block ) );
block->deleteLater();
new_blocks.erase( new_blocks.begin() + k );
--k;
}
}
blocks_exec_list[i]->removeConnections();
blocks_exec_list[i]->deleteLater();
blocks_exec_list.erase( blocks_exec_list.begin() + i
);
--i;

paint();
}
}

void DiagramExecutor::paint()
{
#ifdef Q_OS_LINUX
QString text_format = "digraph graph_blocks\n"
"{\n"
"\tfontname=\"Helvetica,Arial,sans-
serif\"\n"
"\tnode
[fontname=\"Helvetica,Arial,sans-serif\"]\n"
"\tedge
[fontname=\"Helvetica,Arial,sans-serif\"]\n"
"\tconcentrate=True;\n"
"\trankdir=TB;\n"
"\tnode [shape=record];\n";
for ( int i = 0; i < blocks_exec_list.size(); ++i )
{

```

```

        text_format += "\t" + QString::number( i ) + " "
                        + "[label=\""
                        + blocks_exec_list[i]->getSettings()-
>getNameType() + "\\n\\n\\n\\n"
                        + blocks_exec_list[i]->getSettings()-
>block_name + "\\n|"
                        + "{input:|output:|user
data:|in/out}|{|[]}|{"
                        + ( ( "SPARQL" != blocks_exec_list[i]-
>getSettings()->getNameType() ) ? blocks_exec_list[i]-
>getOutputData().toString() : "" )
                        + "}|{"
                        + blocks_exec_list[i]->getUserData() +
"}|{"
                        + QString::number( blocks_exec_list[i]-
>getPrevBlocks().size() ) + " / "
                        + QString::number( blocks_exec_list[i]-
>getNextBlocks().size() ) + "}}\\n\\n";
    }
    text_format += "\\n\\n";
    for ( int i = 0; i < blocks_exec_list.size(); ++i )
    {
        for ( auto block : blocks_exec_list[i]->getNextBlocks() )
        {
            text_format += "\t" + QString::number( i ) + " -> " +
QString::number( blocks_exec_list.indexOf( block ) ) + "\\n";
        }
    }
    text_format += "}";

    GVC_t* gvc;
    gvc = gvContext();

    Agraph_t* graph = agmemread( text_format.toStdString().c_str()
);

    gvLayout( gvc, graph, "dot" );
    char* png_bytes;
    unsigned int size_png_bytes;

    if ( path.isEmpty() )
    {
        auto time = QString::number(
QDateTime::currentMSecsSinceEpoch() );
        path = "logs/" + time + "/";
        QDir( "logs/" ).mkpath( time );
    }
    QString file_name = QDir( path + QString::number(
QDateTime::currentMSecsSinceEpoch() ) + ".png" ).absolutePath();
    gvRender( gvc, graph, "png", fopen(
file_name.toStdString().c_str(), "w" ) );

```

```

        gvRenderData( gvc, graph, "png", &png_bytes, &size_png_bytes
    );

    gvFreeLayout( gvc, graph );
    agclose( graph );

    gvFreeContext( gvc );

    QByteArray byte_array_png( png_bytes, size_png_bytes );

    QPixmap pixmap;
    pixmap.loadFromData( byte_array_png, "PNG" );
    // pixmap = pixmap.scaled( 800, 600 );
    label_pixmap->setPixmap( pixmap );
#endif
}

void DiagramExecutor::createWindow()
{
    QGridLayout* grid_layout = new QGridLayout( this );
    // text_edit_script = new QTextEdit( this );
    // text_edit_script->setTabStopDistance( 30 );
    text_edit_output = new QTextEdit( this );
    text_edit_output->setTabStopDistance( 30 );
    text_edit_output->setMaximumHeight( 300 );
    QPushButton* button_exec = new QPushButton( "Execute", this );
    connect( button_exec, SIGNAL( clicked() ), this, SLOT(
execute() ) );

    graphics_view = new QGraphicsView( this );
    auto scene = new QGraphicsScene( graphics_view );
    graphics_view->setScene( scene );
    label_pixmap = new QLabel();
    scene->addWidget( label_pixmap );

    // label->setSizePolicy( QSizePolicy::Policy::Ignored,
QSizePolicy::Policy::Ignored );
    // grid_layout->addWidget( new QPushButton( "mem" ), 0, 0 );
    grid_layout->addWidget( graphics_view, 0, 0 );
    // grid_layout->addWidget( text_edit_script, 0, 0 );
    grid_layout->addWidget( text_edit_output, 1, 0 );
    grid_layout->addWidget( button_exec, 2, 0 );
}

void DiagramExecutor::logs_sniff( QStringList str_list )
{
    // text_edit_script->insertPlainText( str_list.join( "\n" ) );
}

void DiagramExecutor::execute()
{
    text_edit_output->clear();
    while ( true )
    {

```

```

        bool flag = false;
        for ( auto block_exec : blocks_exec_list )
        {
            if ( !block_exec->getFlagOfWorking() && block_exec->checkForWork() )
            {
                connect( block_exec, SIGNAL( logs( QStringList )
),
                        this, SLOT( logs_sniff( QStringList ) ) );

                block_exec->runBlock();
                flag = true;
            }
        }
        if ( !flag )
        {
            break;
        }
    }
}

```

Файл blocksexec.cpp

```

#include "blocksexec.h"

#include <QDebug>

BlocksExec::BlocksExec( QObject* parent )
    : QObject{ parent }
{
    flag_of_work = false;
    allow_work = true;
}

QString BlocksExec::runBlock()
{
    logs_exec.clear();
    logs_exec.push_back( "\n---->New Start<----\n" );
    setInputData( checkInputValue() );
    logs_exec.push_back( "INPUT DATA: " );
    for ( auto data : getInputData() )
    {
        logs_exec.push_back( data.toString() );
    }

    try
    {
        auto engine = createEngine();

        execScript( engine, "var inputs = [];" );
        execScript( engine, "var output = \"\";" );
        execScript( engine, "var causes = [];" );

        for ( int i = 0; i < prev_blocks.size(); ++i )
        {

```

					ПГУ1.090301.09.001 ПЗ	Лист
						85
Изм	Лист	№ докум.	Подпись	Дата		

```

        execScript( engine, "inputs.push( " + prev_blocks[i]-
>getOutputData().toString() + " );" );
    }
    engine->globalObject().setProperty( "user_data", user_data
);

    execScript( engine, main_script );

    output_data = engine->globalObject().property( "output" );
    logs_exec.push_back( "\nOUTPUT DATA: " +
output_data.toString() );

    auto deps = engine->globalObject().property( "causes"
).toString();
    if ( deps.isEmpty() )
    {
        for ( auto prev : prev_blocks )
        {
            addDepends( prev->getDependens() );
            addDepend( prev->getDiagramItem(), linked_item );
        }
    }
    else
    {
        for ( auto dep : deps.split( "," ) )
        {
            addDepends( prev_blocks[dep.toInt()]-
>getDependens() );
            addDepend( prev_blocks[dep.toInt()]-
>getDiagramItem(), linked_item );
        }
    }

    if ( nullptr != linked_item )
    {
        linked_item->setOutputText( output_data.toString() );
        linked_item->setDependencies( getDependens() );
    }
}
catch ( QString& err )
{
    qDebug() << "Error run block: " << err;
    logs_exec.push_back( "\n----> ERROR <----\n" );

    flag_of_work = true;
    return err;
}
logs_exec.push_back( "\n----> Success <----\n" );

flag_of_work = true;
emit blockFinished();
emit logs( logs_exec );
return {};
}

```

```

void BlocksExec::loadScript( QScriptEngine* engine, const QString&
path )
{
    QFile f( path );
    if ( f.open( QIODevice::ReadOnly ) )
    {
        QString str = f.readAll();
        execScript( engine, str, false );
    }
    else
    {
        emit error( "Failed load! (" + path + ")" );
    }
}

QScriptEngine* BlocksExec::createEngine()
{
    QScriptEngine* engine = new QScriptEngine( this );
    for ( auto& key : default_value_list.keys() )
    {
        auto script_value = engine->newQObject(
default_value_list[key] );
        engine->globalObject().setProperty( key, script_value );
    }
    loadScript( engine, "://Sources/scripts/script.js" );
    loadScript( engine, "://Sources/scripts/XMLHttpRequest.js" );

    return engine;
}

void BlocksExec::execScript( QScriptEngine* engine, const QString&
script, bool logs_flag, bool exception )
{
    auto result = engine->evaluate( script );
    if ( logs_flag )
        logs_exec.push_back( script );
    if ( result.isError() )
    {
        logs_exec.push_back( "\n---> Fail <---\n" );
        logs_exec.push_back( "Error: " + result.toString() );
        emit error( result.toString() );
        if ( exception )
        {
            throw result.toString();
        }
    }
}

void BlocksExec::addDepend( DiagramItem* start, DiagramItem* end )
{
    if ( nullptr != start && nullptr != end )
    {
        depends_list.push_back( { start, end } );
    }
}

```

```

    }
}

void BlocksExec::addDepends( QVector<QPair<DiagramItem*,
DiagramItem*>> items )
{
    for ( auto item : items )
    {
        addDepend( item.first, item.second );
    }
}

QVector<QPair<DiagramItem*, DiagramItem*>>
BlocksExec::getDependens()
{
    return dependens_list;
}

QStringList BlocksExec::getLogs()
{
    return logs_exec;
}

void BlocksExec::removeConnections()
{
    for ( auto block : prev_blocks )
    {
        block->removeConnect( this );
    }
    for ( auto block : next_blocks )
    {
        block->removeConnect( this );
    }
}

void BlocksExec::addBlockConnectName( const QString& name,
BlocksExec* block )
{
    block_connect_name[name].push_back( block );
}

QString BlocksExec::getBlockConnectName( BlocksExec* block )
{
    for ( auto blocks_connection : block_connect_name )
    {
        for ( auto block_next : blocks_connection )
        {
            if ( block == block_next )
            {
                return block_connect_name.key( blocks_connection
);
            }
        }
    }
}

```

```

        return {};
    }

void BlocksExec::removeConnectName( const QString& tag )
{
    block_connect_name.erase( block_connect_name.find( tag ) );
}

void BlocksExec::removeConnectName( BlocksExec* block )
{
    block_connect_name[getBlockConnectName( block )].removeOne(
block );
}

void BlocksExec::removeConnect( BlocksExec* block )
{
    auto it = prev_blocks.indexOf( block );
    if ( it != -1 )
    {
        prev_blocks.erase( prev_blocks.begin() + it );
        return;
    }
    it = next_blocks.indexOf( block );
    if ( it != -1 )
    {
        next_blocks.erase( next_blocks.begin() + it );
    }
}

bool BlocksExec::checkForWork()
{
    for ( auto block : prev_blocks )
    {
        if ( !block->getFlagOfWorking() )
            return false;
    }
    return true;
}

QScriptValueList BlocksExec::checkInputValue()
{
    QScriptValueList value_list;
    for ( auto block : prev_blocks )
    {
        value_list.push_back( block->getOutputData() );
    }
    return value_list;
}

void BlocksExec::setValueForEngine( const QMap<QString, QObject*>&
value_list )
{
    default_value_list = value_list;
}

```

```

void BlocksExec::setDiagramItem( DiagramItem* item )
{
    linked_item = item;
}

DiagramItem* BlocksExec::getDiagramItem()
{
    return linked_item;
}

void BlocksExec::setTags( const QMap<QString, QString>& tags )
{
    block_tags = tags;
}

void BlocksExec::addTag( const QString& key, const QString& value )
{
    block_tags.insert( key, value );
}

QMap<QString, QString> BlocksExec::getTags()
{
    return block_tags;
}

QString BlocksExec::getTag( const QString& key )
{
    return block_tags[key];
}

void BlocksExec::deleteTags()
{
    block_tags.clear();
}

void BlocksExec::deleteTag( const QString& key )
{
    auto it = block_tags.find( key );
    if ( block_tags.end() != it )
    {
        block_tags.erase( it );
    }
}

void BlocksExec::addLinkedItemForComposite( const QString& str,
DiagramItem* item )
{
    if ( item != nullptr )
    {
        linked_items_for_composite.insert( str, item );
    }
}

```

```

DiagramItem* BlocksExec::getLinkedItemForComposite( const QString&
str )
{
    return linked_items_for_composite[str];
}

void BlocksExec::disconnectDiagramItem()
{
    setDiagramItem( nullptr );
}

void BlocksExec::setSettings( DiagramItemSettings* settings )
{
    block_settings = settings;
}

DiagramItemSettings* BlocksExec::getSettings()
{
    return block_settings;
}

void BlocksExec::setFlagOfWorking( bool flag )
{
    flag_of_work = flag;
}

void BlocksExec::setAllowWork( bool flag )
{
    allow_work = flag;
}

void BlocksExec::setScript( const QString& script )
{
    main_script = script;
}

void BlocksExec::setInputData( const QScriptValueList& input_list
)
{
    input_data = input_list;
}

void BlocksExec::addInputData( const QScriptValue& input )
{
    input_data.push_back( input );
}

void BlocksExec::setUserData( const QString& data )
{
    user_data = data;
}

```

```

void BlocksExec::setNextBlocks( const QVector<BlocksExec*>&
next_list )
{
    next_blocks = next_list;
}

void BlocksExec::addNextBlocks( BlocksExec* next )
{
    next_blocks.push_back( next );
}

void BlocksExec::setPrevBlocks( const QVector<BlocksExec*>&
prev_list )
{
    prev_blocks = prev_list;
}

void BlocksExec::addPrevBlocks( BlocksExec* prev )
{
    prev_blocks.push_back( prev );
}

bool BlocksExec::getFlagOfWorking()
{
    return flag_of_work;
}

bool BlocksExec::getAllowWork()
{
    return allow_work;
}

QString BlocksExec::getScript()
{
    return main_script;
}

QScriptValueList BlocksExec::getInputData()
{
    return input_data;
}

QString BlocksExec::getUserData()
{
    return user_data;
}

QScriptValue BlocksExec::getOutputData()
{
    return output_data;
}

QVector<BlocksExec*> BlocksExec::getNextBlocks()
{

```

```

        return next_blocks;
    }

    QVector<BlocksExec*> BlocksExec::getPrevBlocks ()
    {
        return prev_blocks;
    }

```

Файл apijs.cpp

```

#include "apijs.h"

ApiJS::ApiJS( QObject* parent )
    : QObject( parent )
{
}

void ApiJS::setDiagramItem( QVector<DiagramItem*>& blocks )
{
    blocks_lib = blocks;
}

void ApiJS::setOutputForDiagramItem( int index, QString text )
{
    blocks_lib[index]->setOutputText( text );
}

void ApiJS::clearDeps()
{
    for ( auto& block : blocks_lib )
    {
        block->clearDependencies();
    }
}

```

					ПГУ1.090301.09.001 ПЗ	Лист
						93
Изм	Лист	№ докум.	Подпись	Дата		

Приложение В

(обязательное)

Интерпретатор диаграмм графических запросов к RDF-хранилищам данных

Презентация

					ПГУ1.090301.09.001 ПЗ	Лист
Изм	Лист	№ докум.	Подпись	Дата		94

ИНТЕРПРЕТАТОР ДИАГРАММ ГРАФИЧЕСКИХ ЗАПРОСОВ К RDF- ХРАНИЛИЩАМ ДАННЫХ

Автор работы:

студент гр. 19ВВП1 Кобзев Максим Игоревич

Научный руководитель:

д.т.н., профессор кафедры ВТ Дубинин Виктор Николаевич

Пенза 2023

Цель работы

- Цель контекстного исследования, в рамках которого выполнялась данная работа – разработка математического, лингвистического и программного обеспечения для поддержки проектирования специализированных информационно-вычислительных систем, предназначенных для оценки объектов энергетической инфраструктуры, на основе функционально-блочных диаграмм и RDF-хранилищ данных.
- Цель данной работы – разработка операционной семантики графического языка функционально-блочных запросно-вычислительных диаграмм, а также программная реализация интерпретатора диаграмм данного вида и системы объяснения решений.

Задачи, решаемые в процессе выполнения проекта

1. Разработка модели выполнения функционально-блочных запросно-вычислительных диаграмм;
2. Разработка и программная реализация интерпретатора функционально-блочных запросно-вычислительных диаграмм;
3. Разработка подсистемы объяснения результирующих значений.

Области применения:

- Спецификация (формализованное описание) на языке функционально-блочных запросно-вычислительных диаграмм требований к техническим системам, описанным с использованием RDF-онтологий, выраженных в числовом виде;
- Проектирование, разработка и реализация специализированных информационно-вычислительных систем на основе функционально-блочных запросно-вычислительных диаграмм.

Концепция функциональных блоков

Виды функциональных блоков:

- Базисный блок
- Композитный блок
- SPARQL-блок

Внутреннее наполнение блоков:

JavaScript

```
output = "false";
for(var i = 0; i < inputs.length; i++){
  if(inputs[i] == "true"){
    output = "true";
    break;
  }
}
```

Композитный блок является совокупностью базовых блоков, соединенных между собой

```
SELECT ?systemA ?DidLevelA ?systemB
      ?DidLevelB ?interface
WHERE {
  ?systemA :associatedWithDidLevel ?DidLevelA.
  ?systemB :associatedWithDidLevel ?DidLevelB.
  ?interface :interfaceFrom ?systemA.
  ?interface :interfaceTo ?systemB.
  FILTER (?DidLevelA != ?DidLevelB)
}
```

Модель выполнения системы (сети) функциональных блоков

Предикат, определяющий разрешенность функционального блока (ФБ) fb для выполнения, может быть записан следующим образом.

$enabled(fb) \equiv \forall in \in IN^{fb} [readyData(in)],$

где IN^{fb} – множество входов ФБ fb ;

$existData: IN \rightarrow \{true, false\}$ – функция (предикат), определяющая готовность данных на входах ФБ. Здесь IN – множество входов всех функциональных блоков системы. Предикат $ready(fb)$ является истинным, когда готовы данные на всех входах блока fb .

Предикат, указывающий на наличие хотя бы одного разрешенного ФБ в системе, может быть определен так, как показано ниже.

$isEnabledFB \equiv \exists fb \in FB [enabled(fb)],$

где FB – множество функциональных блоков в системе.

Алгоритм запуска функциональных блоков:

while isEnabledFB

forall $fb \in FB$ **with** $enabled(fb)$ **do**

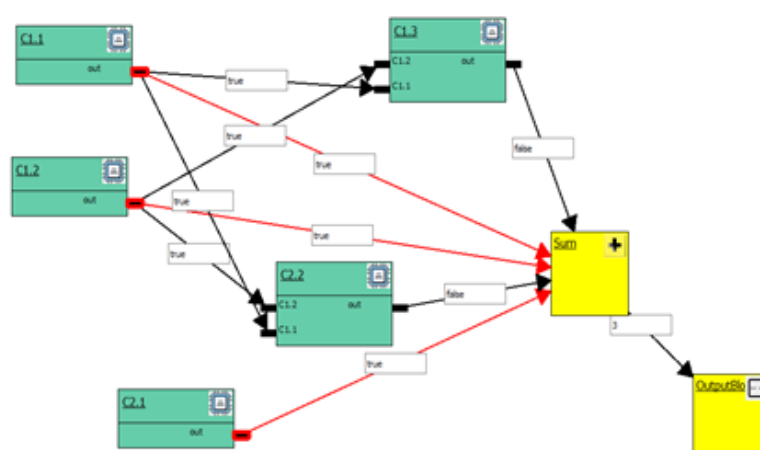
 Execute(fb)

end

end

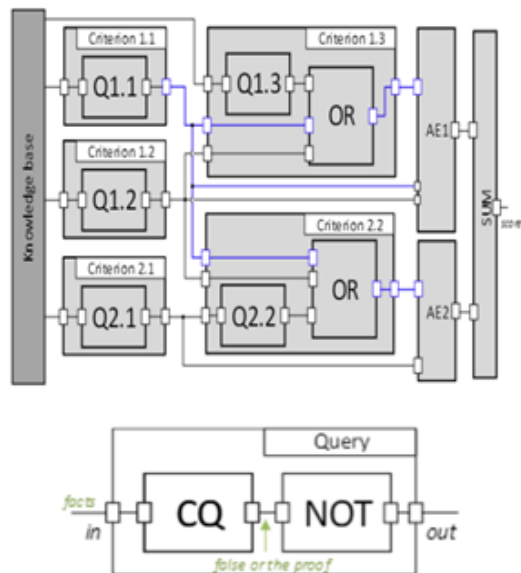
Метод объяснения решений

Графическое представление метода объяснения решений

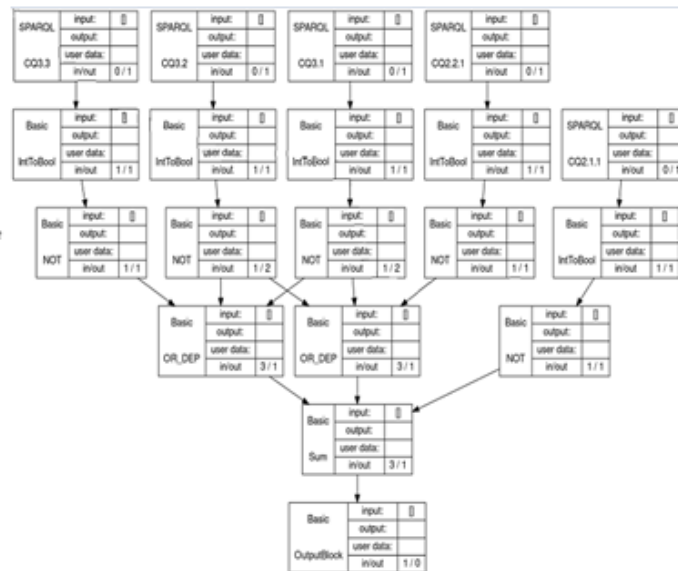


Работа с библиотекой Graphviz

Пример функционально-блочной диаграммы



Пример разворачивания диаграммы с помощью библиотеки Graphviz.



Работа с сервером Apache Jena Fuseki.



Веб-интерфейс Apache Jena Fuseki

Интерфейс для работы с сервером в рамках FBQL Editor

IP-address Fuseki Server:

Port Fuseki Server:

Dataset Name:

Prefixes:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX : <http://www.semanticweb.org/SEARCH/ontologies/2021/4/OICA/Classes#>
PREFIX USEPR: <http://www.semanticweb.org/SEARCH/ontologies/2021/4/OICA/USEPR#>
```

Стек используемых технологий



Архитектура программы FBQL Editor

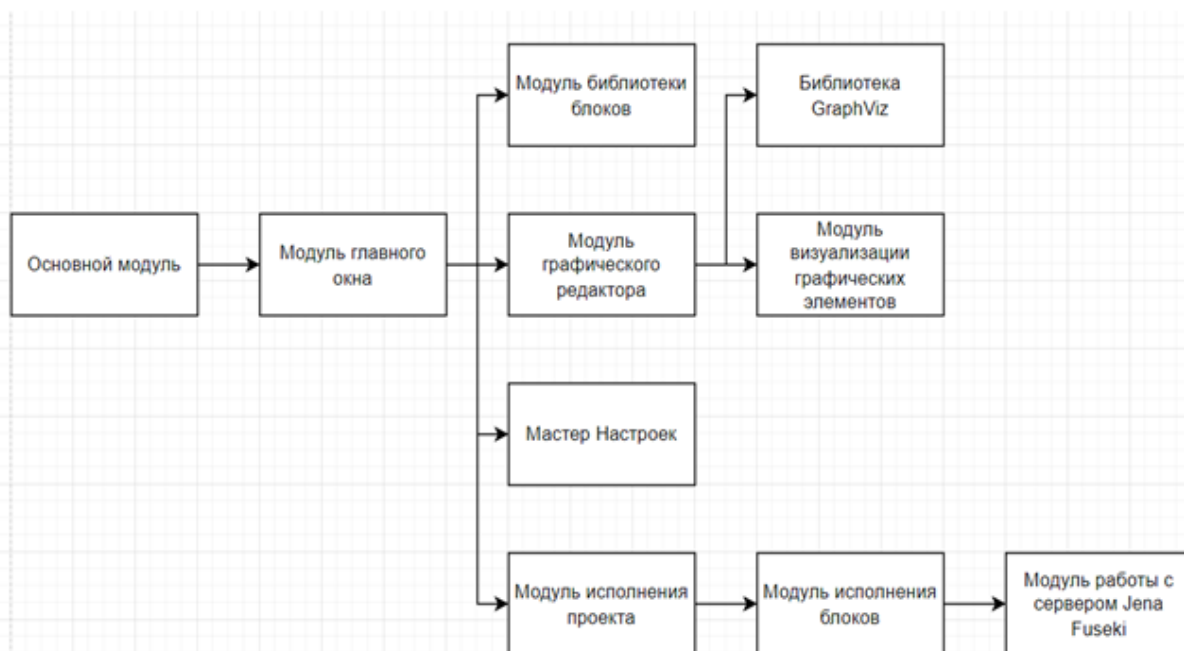


Диаграмма вариантов использования

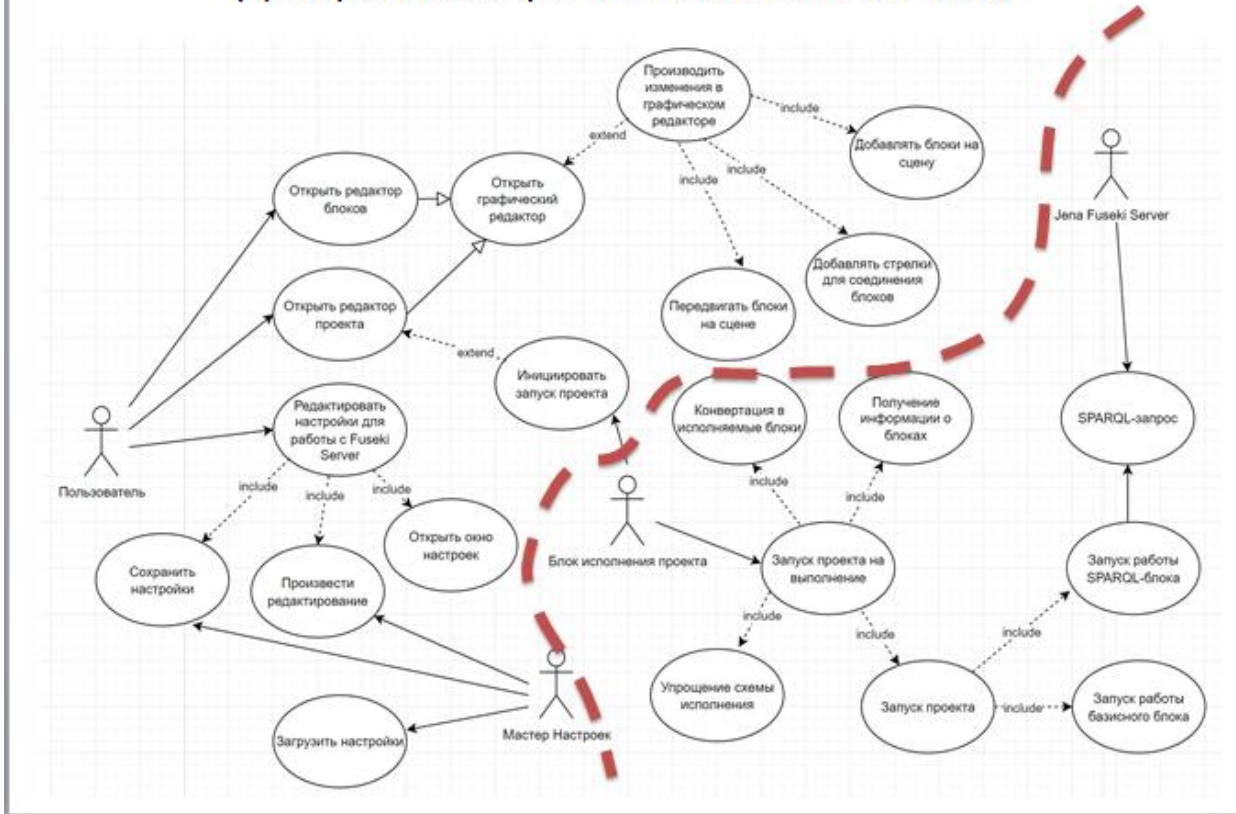
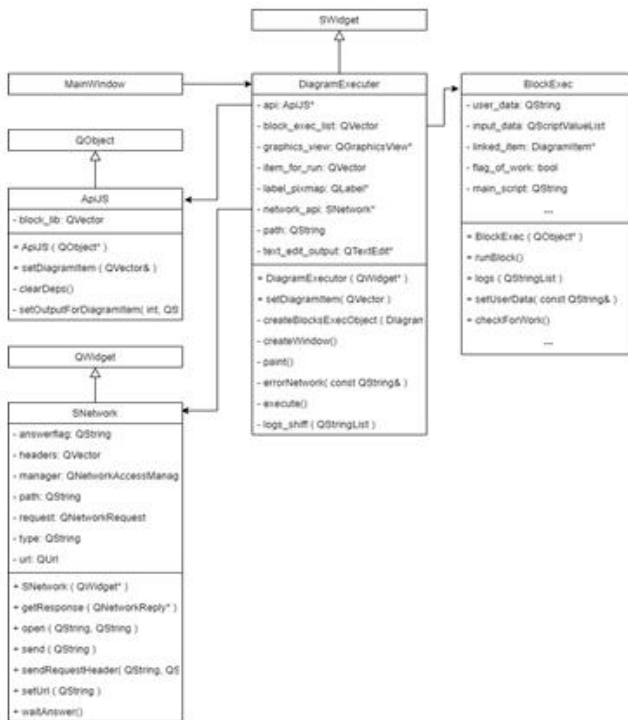


Диаграмма классов



Публикации по теме работы

Муромский Д.А. Кобзев М.И. Условные операторы для SPARQL запросов к RDF-хранилищам данных // Сб. ст. X Ежегодн. всеросс. межвуз. науч.-практ. конф. «Информационные технологии в науке и образовании. Проблемы и перспективы (ВМНПК – 2023)», Пенза, 2023. – с. 310-313.

Ovsiannikova P., Pakonen A., Muromsky D., Kobzev M., Dubinin V., Vyatkin V. Formal verification of non-functional requirements of overall instrumentation and control architectures // IEEE Open Journal of the Industrial Electronics Society. 2023 (submitted).



Заключение

В рамках создания программы были разработаны:

- 1) Семантика выполнения функционально-блочных диаграмм;
- 2) Интерпретатор функционально-блочных диаграмм с возможностью вывода промежуточных и итоговых результатов;
- 3) Подсистема объяснения результатов интерпретации диаграмм;
- 4) Интерфейс для работы с Jena Fuseki Server.

Спасибо за внимание!

					ПГУ1.090301.09.001 ПЗ	Лист
						102
Изм	Лист	№ докум.	Подпись	Дата		