Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Безгузиков Артемий Валерьевич

Диаграммный исполнитель

Выпускная квалификационная работа

Научный руководитель:
к.т.н., доц. Брыксин Т.А.

Рецензент:
ООО “ALM Works”
разработчик ПО Барташев А.Н.

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Artemii Bezguzikov

Diagram Executor

Graduation Thesis

Scientific supervisor:
Candidate of Engineering Sciences Timofey Bryksin

Reviewer:
"ALM Works"
Software Developer Artem Bartashev

Saint-Petersburg
2019

Оглавление

Введение

В современном мире информационные технологии проникли практически во все сферы жизни человека, что повлекло усиленный спрос на IT-специалистов, способных работать в узких, предметных областях — будь то машинное обучение, управление активами или моделирование физических экспериментов. Каждая такая область предоставляет свой набор объектов и свои задачи, которые описываются в терминах данных объектов. Для автоматизации и упрощения разработки в рамках фиксированной сферы все большую и большую популярность набирает подход, при котором сначала разрабатывается специальный язык (Domain-Specific Language, DSL [9]), ориентированный на ее конкретные задачи, а уже затем на нем и разрабатывается решение. В отличие от языков общего назначения, таких как Java или C++, DSL оперирует не с ветвлениями и циклами, а с заранее известными сущностями предметной области, в то время как его конструкции описывают основные процессы взаимодействия этих сущностей.

Приведенный выше подход лежит в основе идеи визуального предметно-ориентированного моделирования (Domain-Specific Modeling, DSM [15]), которое предполагает создание и исполнение программ с помощью заранее разработанного специфичного для данной сферы графического языка. Хорошим примером реализации этой идеи является продукт TRIK Studio[1], который позволяет описывать поведение роботов с помощью построения диаграмм. Помимо редактора TRIK Studio включает в себя среду исполнения, репозиторий для хранения данных и модуль генерации исходного кода. В совокупности перечисленные программные средства, как инструменты автоматизации написания исходного кода, называют DSM-решением [22].

Использование DSM-решений позволяет делать программы более наглядными и доступными для понимания и поддержки, уменьшить количество ошибок, и привлекать специалистов данной, конкретной области не имеющих навыков написания кода [10]. Кроме того, опублико-

[1]Среда программирования роботов — URL: http://blog.trikset.com/p/trik-studio.html

ваны статьи, где описывается успешный опыт внедрения DSM-подхода, повлекший прирост производительности разработчиков в несколько раз [16].

В силу того, что различных сфер, где применим диаграммный подход, довольно много, логичным решением является создание системы для конструирования как подобных предметно-ориентированных языков, так и соответствующих DSM-решений. Такие продукты называются MetaCASE-системами или DSM-платформами. Хорошим примером реализации этой идеи является проект QReal[2], разрабатываемый на кафедре Системного программирования СПбГУ. Он позволяет по ряду формальных описаний проектируемого языка (сущности, связи между ними, их свойства) автоматически получать среду визуальной разработки, имеющую в своей основе данный язык [19]. В частности, с помощью QReal, был создан упомянутый выше проект TRIK Studio.

Одним из наиболее популярных языков программирования на сегодняшний день является Python [3]. Чаще всего его применяют в сферах анализа данных, машинного обучения и научных исследованиях [17]. Python имеет небольшой порог вхождения и обладает обилием удобных библиотек для решения разного рода задач. Существует несколько сред разработки на языке Python, наиболее востребованными из них являются PyCharm[3] и Jupiter Notebook[4]. Последний примечателен тем, что он не является типичной средой разработки, где упор делается на написание монолитных блоков кода. Jupiter в одном файле позволяет последовательно создавать микропрограммы, которые в дальнейшем исполняются, а результат их вычислений кэшируется. Все микропрограммы имеют общую память и способны использовать переменные друг друга. Кроме того, данная среда разработки позволяет удобно отображать картинки и графики, что в дополнение к вышеперечисленным особенностям, делают ее прекрасным выбором для проведения небольших исследований.

[2]QReal - metaCASE-система — URL: https://github.com/qreal/qreal

[3]Среда разработки для языка Python — URL: https://www.jetbrains.com/pycharm/

[4]Веб-среда разработки для языка Python. URL: https://jupyter.org/

Структура программ, реализуемых с помощью Jupiter Notebook, описывается шаблоном проектирования “Каналы и фильтры” (Pipes and Filters) [5]. В качестве канала выступает общая память, а в качестве фильтров — микропрограммы. Шаблон Pipes and Filters широко применяется для задач, решения которых можно представить в виде нескольких независимых шагов, выполняемых отдельными обработчиками. Как пример, к описанному классу задач относится проблема классификации объекта по его признакам [1]. Ниже приводится вероятный алгоритм действий для ее решения.

Сначала необходимо прочитать данные, нормализовать, произвести их анализ. Следующим шагом является выбор необходимой модели для обучения. Здесь обычно исследователь пробует разные типы моделей, будь то решающие деревья, линейная регрессия или нейронная сеть, и смотрит, насколько быстро и качественно они обучаются. Выбрав самый подходящий тип модели, исследователь пытается подобрать к ней наилучшие параметры и в конце концов получает приемлемый результат. Таким образом, конечную программу можно представить в виде последовательности блоков, результат исполнения которых можно наблюдать, и, основываясь на его анализе, принимать решение о том, какой блок необходимо добавить следующим.

Несмотря на широкий функционал существующих MetaCASE-систем, ни одна из них не способна создать удобного DSM-решения, с интегрированной средой исполнения для диаграммных языков, программы которых реализуют шаблон проектирования Pipes and Filters, а блоки интерпретируются на языке Python.

Постановка задачи

Целью данной работы является создание DSM-платформы для диаграммных языков, программы которых реализуют паттерн “Каналы и фильтры” на языке Python.

Для достижения этой цели были поставлены следующие задачи.

1. Спроектировать основные компоненты DSM-платформы и их взаимодействие.
2. Разработать язык метамоделирования и сформулировать его ограничения.
3. Реализовать описанную DSM-платформу.
4. Провести апробацию работы.

1. Обзор

В начале настоящей главы подробно описывается шаблон проектирования “Каналы и фильтры” и приводятся несколько примеров его реализации. Затем раскрывается связь данного паттерна с парадигмой программирования потоков данных (Dataflow программирование), рассматриваются ее особенности, перечисляются основные преимущества.

Далее по тексту приводятся примеры диаграммных Dataflow языков для различных предметных областей. Особое внимание уделяется теме машинного обучения и, конкретно, продуктам Microsoft Machine Learning Studio [7] и Machine Flow [18]. Производится анализ языков, предоставляемых этими DSM-решениями, перечисляются их преимущества и недостатки.

В заключительной части обзора формулируются критерии к подходящей DSM-платформе. Они основываются на рассмотренных ранее успешных диаграммных решениях и на возможностях среды разработки Jupiter Notebook. Затем приводятся самые зрелые аналоги среди существующих DSM-платформ, каждый из которых проверяется на соответствие приведенным требованиям. В результате делается обоснованный вывод о необходимости создания нового решения.

1.1. Паттерн “Каналы и фильтры”

Шаблон проектирования “Каналы и фильтры” уже давно активно применяется в индустрии. Наиболее известной его реализаций является алгоритм обработки инструкций командной оболочки UNIX (Shell) [12, 32], первые версии которой начали выпускать еще в 1970-ых годах. Shell предоставляет пользователю набор команд-инструкций, которые способна исполнять операционная система, и интерфейс командной строки для взаимодействия с ней. Кроме того, с помощью специального символа “|”, Shell позволяет комбинировать команды: консольный вывод исполнения первой команды перенаправляется второй команде и используется в качестве входных данных для нее. Две или более команды, соединенные таким образом, образуют канал (Pipe). По отдель-

ности они называются фильтрами (Filters).

Паттерн “Каналы и фильтры” широко применяется в задачах, где предполагается последовательная трансформация данных, выполняемая разными обработчиками. В таком случае работа каждого конкретного фильтра описывается следующим сценарием.

1. Ожидание, пока не придут все данные.

2. Исполнение внутренней логики.

3. Распространение результата по всем доступным каналам.

Данный подход наглядно представляется с помощью диаграмм. Обработчики являются блоками, связи между ними — стрелками. Каждый блок может иметь несколько входных портов и максимум один выходной. Вышесказанное иллюстрирует рис. 1.

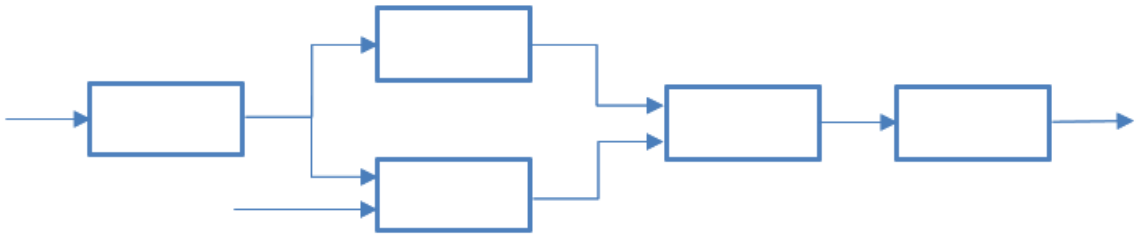


Рис. 1: Графическое представление паттерна “Каналы и фильтры”

Шаблон проектирования “Каналы и фильтры” лег в основу целой парадигмы программирования под названием “программирование потоков данных” (Dataflow programming). В отличие от традиционного подхода, где программа есть последовательный набор инструкций, в Dataflow подходе она представляется в виде графа вычислений. Как результат, происходит существенное упрощение параллельного программирования, повышается модульность (за счет разделения логики между обработчиками) и наглядность процесса исполнения. В настоящее время Dataflow идеи стремительно набирают популярность в виде реактивного программирования (Reactive programming) [13] и обработки

потоков (Stream processing) [6], что выражается как и внутренней языковой поддержкой, так и обилием сторонних библиотек [31].

1.2. Графические Dataflow языки

Одним из важнейших преимуществ, которые дают Dataflow методы, является наглядность. Программы, написанные в таком стиле, очень просто и изящно переносятся на диаграммный подход, что позволяет создавать простые и удобные языки для различных предметных областей. Так, например, для создания и обработки музыки можно использовать среду визуального программирования Audiomulch [25]. Для работы в сфере "интернета вещей" подойдет продукт Node-Red [30], а для создания шаблонных веб-приложений — DGLogic [26].

Особое внимание стоит уделить теме машинного обучения. Еще в 1997 году искусственный интеллект компании IBM DeepBlue[5] обыграл чемпиона мира по шахматам Гари Каспарова и навсегда закрыл вопрос о превосходстве человека в этой дисциплине. В современном мире возможности вычислительных машин выходят далеко за рамки игры в шахматы: роботы, способные на уровне человека играть в теннис, самоуправляемые автомобили, переводчики с китайского на английский в реальном времени. Задачи машинного обучения возникают во многих коммерческих сферах, они укладываются в Dataflow парадигму и многие из них способны представляться в виде диаграмм.

1.2.1. Microsoft Azure ML

Одним из популярных визуальных решений для работы в области искусственного интеллекта является Microsoft Machine Learning Studio. Данный продукт хорошо задокументирован, имеет простые и интуитивные интерфейсы и предоставляет богатый функционал, способный покрыть довольно большой объем задач. В частности, ML Studio позволяет считывать как собственные данные, так и известные коллекции, имеет солидный набор функций для обработки этих данных и

[5]Шахматный суперкомпьютер, разработанный компанией IBM

предоставляет различные типы моделей для обучения на них. Кроме того, продукт поддерживает методы машинного обучения в обработке и анализе изображений. Все перечисленное в совокупности позволяет эффективно решать типовые задачи предметной области даже людям, далеким от программирования.

Основным недостатком ML Studio, на наш взгляд, является слабая расширяемость: существуют серьезные ограничения на возможности добавления новых блоков [34]. Несмотря на то, что ML Studio все-таки позволяет добавлять свои функции на языке Python, им на вход смогут поступать данные только определенного заданного типа. Таким образом, если пользователь столкнется с необходимостью добавления функций, выходящих за рамки стандартного пакета, ему скорее всего придется выбрать другой инструмент для работы.

1.2.2. Machine Flow

Существуют продукты, где, в отличие от ML Studio, пользователь постепенно сам добавляет нужные конкретно ему функции. Таким продуктом является Machine Flow, который, как и предшественник, позволяет применять диаграммный подход для задач машинного обучения. С точки зрения архитектуры он состоит из сервера, где описываются и запускаются заданные функции на языке Python, и клиента, предоставляющего пользовательский интерфейс для составления графических программ. Каждая такая программа — это граф вычислений, где вершинами (блоками) и являются эти функции. Исполнение графа сводится к последовательному исполнению его вершин в некотором порядке, определяемым его ребрами.

При исполнении блока происходит REST-вызов на сервер, там этот вызов обрабатывается и запускает соответствующий данному блоку исходный код. Поскольку Dataflow подход подразумевает передачу данных между вершинами, необходим способ, который позволит использовать вывод одной функции в качестве входных данных следующей. В Machine Flow это реализуется с помощью операций записи в файл и чтения из файла, что накладывает существенные ограничения на пере-

даваемые данные — они должны быть сериализуемы [6].

Еще одним существенным недостатком этого инструмента является сама концепция REST-вызовов. Операции, производимые на сервере, могут занимать часы или даже дни. Примером такого сценария является обучение какой-нибудь глубокой нейронной сети. В результате пользователь увидит лишь программу, зависшую в состоянии “исполняю”.

Тем не менее, подход к построению диаграммных языков, который был реализован в Machine Flow, обладает важным преимуществом: он позволяет настраивать DSL в строгом соответствии с нуждами пользователя. В результате получается гибкий и расширяемый язык, который в теории способен покрыть даже самые специфические потребности.

1.3. DSM-платформы

Рассмотренные в предыдущих разделах продукты включают в себя графический редактор, репозиторий для хранения данных, средства многопользовательской работы и прочие инструменты для автоматизации и упрощения программирования. В совокупности они образуют DSM-решение. Кроме того, вне зависимости от предметной области, Dataflow языки практически идентично выражаются с помощью диаграмм, а соответствующие DSM-решения предоставляют очень схожую функциональность. Как следствие, возникает необходимость создания DSM-платформы для генерации такого рода решений.

Подробный анализ рынка актуальных MetaCASE-систем приведен в работе [35]. В ней были сформулированы требования, которым должна соответствовать зрелая DSM-платформа, выделены основные продукты данной отрасли и сделаны выводы касательно каждого из них. В настоящем обзоре будут сформулированы дополнительные критерии, специфичные для данной конкретной задачи. По ним будут оцениваться уже зрелые, состоявшиеся DSM-платформы.

[6] Сериализация — процесс перевода какой-либо структуры данных в последовательность битов.

1.3.1. Критерии

В силу ограниченности предметной области порожденный язык должен содержать небольшое, конечное число конструкций, в противном случае он бы считался языком общего назначения. Однако в момент создания DSM-решения не всегда можно заранее предугадать, какие в точности объекты и сущности языка будут использоваться. Отсюда возникает требование расширяемости: необходим простой и быстрый способ обогатить диаграммный язык новыми блоками. С этой задачей, в частности, справляется продукт Machine Flow.

В силу того, что именно среда разработки Jupiter Notebook наиболее подходит для текстового программирования рассматриваемого класса задач, следующие критерии к DSM-решению будут формулироваться исходя из тех возможностей, которые предоставляет этот инструмент [8, 27].

Микропрограммы в Jupiter Notebook могут содержать произвольный код с возможностью подключения произвольных библиотек языка Python. Таким образом, искомое DSM-решение также должно включать в себя инструмент для работы с ними. Кроме того, микропрограммы могут порождать сложные и ресурсоемкие вычисления, результаты которых следует кэшировать для дальнейшего переиспользования. Вышесказанное выполняется как для Microsoft ML Studio, так и для Machine Flow.

Многие библиотеки в целях отображения важной информации журналируют события, происходящие в процессе исполнения программы. Этот вывод, как и возникающие ошибки, необходимо перехватывать и возвращать пользователю в режиме реального времени. Данный подход позволяет наблюдать статус текущего, исполняемого блока. В некотором виде эта идея реализована в Microsoft ML Studio.

Еще одним важным условием является отображение графиков и произвольных изображений, которые порождает исполняемый код на Python, поскольку именно на них часто приходится ориентироваться исследователю в процессе разработки программы. Оба рассмотренных

DSM-решения поддерживают такой функционал.

Jupiter Notebook имеет клиент-серверную архитектуру, что позволяет переносить вычисления на удаленную машину. Таким образом, пользователю не требуется устанавливать программу на свой компьютер, вместо этого ему достаточно открыть ссылку в браузере. На наш взгляд, это условие так же необходимо для комфортной работы в DSM-решении.

Ниже приведен итоговый перечень критериев, которые будут оцениваться в существующих DSM-платформах.

1. Простота и скорость создания искомых диаграммных языков, удобство работы в получаемом DSM-решении.
2. Возможность динамического расширения диаграммного языка.
3. Возможность использования функций языка Python.
4. Наличие интегрированной среды исполнения.
5. Возможность переиспользования вычислений.
6. Отображение консольного вывода и ошибок фактически исполняемого исходного кода.
7. Отображение графиков и произвольных изображений.
8. Возможность работы в браузере.

1.3.2. Существующие продукты

Основываясь на работах [23, 24, 35], в каждой из которых проводился довольно подробный обзор существующих MetaCASE-систем, к рассмотрению были выделены следующие DSM-платформы: MetaEdit+ [28], Microsoft Modeling SDK [29] и QReal. Перечисленные продукты зарекомендовали себя как современные, зрелые и стабильные решения. Отдельно будет рассмотрен Eclipse Modeling Project, как совокупность проектов, комбинируя которые можно задать DSM-платформу.

MetaEdit+

MetaEdit+ — это программное решение, которое существует на рынке уже довольно давно. Первые его версии стали доступны еще в 1996 году, и с тех пор он продолжает непрерывно развиваться. На сегодняшний день MetaEdit+ является одним из самых востребованных в области предметно-ориентированного моделирования [14]. Проект имеет богатую документацию, а его интерфейсы просты и понятны. К сожалению, MetaEdit+ является закрытым коммерческим продуктом, и его исходный код не доступен для программных расширений.

Данная DSM-платформа может быть установлена на все популярные операционные системы, но, тем не менее, не имеет web-версии. Функционально она предоставляет графическую среду как для создания различных DSL, так и для их использования, набор генераторов, способных порождать исходный код, в том числе и на языке Python, репозиторий для хранения проектов и др. Как подробно отражено в статье [4], с помощью MetaEdit+ возможно описать предметно-ориентированные языки, удовлетворяющие паттерну Pipes and Filters. В упомянутом источнике это иллюстрируется на примере предметно-ориентированного решения для программирования микроконтроллеров.

Основным недостатком MetaEdit+ является отсутствие интегрированной среды исполнения исходного кода. Программы, написанные на порожденных MetaCASE-системой языках, в качестве результата возвращают сгенерированный исходный код, который может быть запущен только внешними средствами. Отсюда, не выполняются требования 4-7, из чего следует, что MetaEdit+ невозможно использовать в качестве искомого инструмента.

Microsoft Modeling SDK

Microsoft Modeling SDK представляет собой интересный, но постепенно устаревающий продукт. Первые его версии начали разрабатываться еще в 2003 году, однако, уже с 2007 Microsoft Modeling SDK практически перестал развиваться. Тем не менее, он по-прежнему хорошо подходит для программистов, работающих с технологиями компа-

нии Microsoft. Продукт интегрирован в среду разработки Visual Studio[7], что автоматически предоставляет возможность исполнения и отладки сгенерированного года.

Рассматриваемая DSM-платформа обладает обильным набором инструментов для создания сложных метамоделей, позволяет описывать внешний вид редакторов получаемых языков и исполнять сгенерированный код. Однако существует и ряд недостатков: Microsoft Modeling SDK доступен только для операционной системы Windows, требует написания кода на языке C#, и является неотделимым от Visual Studio. Получаемое DSM-решение не рассчитано на тесную интеграцию с языком Python и не может быть использована для комфортной работы с ним.

QReal

QReal — это DSM-платформа, разрабатываемая студентами и преподавателями кафедры системного программирования СПбГУ под руководством профессора А.Н.Терехова. QReal, так же как и рассмотренные выше продукты, предоставляет удобные графические инструменты для создания метамоделей. Получаемое DSM-решение включает в себя диаграммный редактор, репозиторий для хранения моделей, средства обеспечения многопользовательской работы, генераторы программного кода, интерпретаторы, отладчики и механизм проверки ограничений. QReal распространяется под лицензией Apache License v2.0 и имеет открытый исходный код.

В качестве DSM-платформы QReal также позволяет гибко настраивать особенности редакторов порождаемых решений. Здесь имеется в виду покомпонентная настройка. Для большинства языков, реализуемых с помощью QReal, характерны одни и те же элементы редактора: область для размещения диаграмм (сцена), палитра элементов, которые можно перетаскивать на сцену, конфигуратор, где можно задавать значения свойств объектов и др. Сами же порожденные DSM-решения подключаются к основной платформе в качестве плагинов и уже отдельно

[7]Среда разработки на платформе .Net — URL: https://visualstudio.microsoft.com/

реализуют дополнительный функционал, необходимый конкретно им.

Одним из таких плагинов является среда разработки QReal:Robots [20, 21], которая позволяет программировать поведения роботов с помощью построения диаграмм и лежит в основе продукта TRIK Studio. Графическая программа, написанная в QReal:Robots может быть исполнена двумя способами:

1. генерация исходного кода по всей диаграмме целиком и последующая отправка его на робота;

2. последовательное исполнение исходных кодов каждого конкретного блока.

Второй подход потенциально допускает сохранение результатов исполнения каждого конкретного блока и возможное их переиспользование, что является одним из требований к искомой DSM-платформе. Отображение консольного вывода исполняемого кода также реализуемо в случае добавления соответствующей функциональности интерпретатору порождаемого кода на языке Python.

Инструменты QReal позволяют достаточно быстро описывать примитивные языки, программы которых реализуют шаблон проектирования “Каналы и фильтры”, однако существует и ряд проблем связанных и с отсутствием важных графических компонент, и изменением логики исполнения диаграмм, и с применением Python в качестве целевого языка исполнения блоков. Кроме того, QReal является приложением для ПК и не позволяет работать с создаваемыми моделями непосредственно из браузера.

Eclipse Modeling Project

Eclipse Modeling Project — это открытый продукт, разрабатываемый на основе платформы Eclipse. EMP включает в себя множество быстро развивающихся самостоятельных проектов, каждый из которых способен решать свои определенные задачи. Ниже приводятся несколько примеров.

- Eclipse Modeling Framework[8] — средство для генерации основных инструментов исполнения и редактирования метамоделей.
- EcoreTools[9] — средство графического создания метамоделей.
- Sirius[10]— конструктор диаграммных редакторов для работы с метамоделью.
- Acceleo[11] — средства кодогенерации для моделей формата EMF по некоторым шаблонам.

Для получения полноценного DSM-решения часто приходится комбинировать различные проекты EMP. Этот факт хорошо иллюстрируется на примере создания DSM-решения для управления роботами [11]. Ниже приводится рисунок, описывающий используемые EMP продукты.

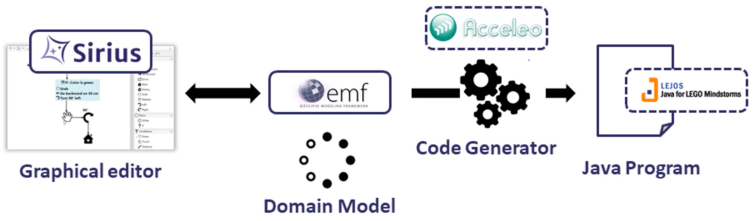


Рис. 2: Продукты EMP для среды управления роботами

Задачи, реализуемые каждым конкретным продуктом, часто могут пересекаться. Интеграция между программными решениями, в свою очередь, может требовать использования отдельного проекта EMP. Эти проекты могут иметь разные темпы развития и разную степень задокументированности. Все перечисленное создает серьезные препятствия для вхождения нового разработчика в предметную область. Учитывая также, что готовых решений для работы с библиотеками языка Python

[8]Eclipse Modeling Framework — URL: https://www.eclipse.org/modeling/emf/
[9]EcoreTools — URL: https://www.eclipse.org/ecoretools/
[10]Sirius — URL: https://www.eclipse.org/sirius/
[11]Acceleo — URL: https://www.eclipse.org/acceleo/

найдено не было, считаем интеграцию с Eclipse Modeling Project нецелесообразной для достижения поставленной цели.

Ввиду перечисленных недостатков Eclipse Modeling Project, мы не считаем оправданным приводить какую-либо комбинацию его проектов в качестве самостоятельного MetaCase-решения. Он не будет учитываться в следующем анализе.

1.3.3. Сравнительный анализ

Каждая из рассмотренных DSM-платформ имеет подходящие инструменты для описания искомого класса предметно-ориентированных языков. Однако ввиду существенных ограничений на метамодели таких языков, во всех перечисленных случаях эти платформы оказываются сильно избыточными и в результате предоставляют пользователю сильно больше инструментов, чем ему необходимо.

Еще один критерий, который в том или ином виде выполняется для всех перечисленных DSM-платформ, это поддержка изменения и расширения порожденного диаграммного языка. Однако ни одно из решений не позволяет обогащать язык в режиме реального времени, в том числе в процессе исполнения диаграммы.

Также все вышеупомянутые DSM-платформы в том или ином виде способны генерировать исходный код на языке Python. Тем не менее, интегрированную среду исполнения для программ, которые в него транслируются, способен предоставить только QReal.

Пункты 5-8 из перечисленных ранее требований не выполняются нигде. Однако при значительном программном расширении платформы QReal, она потенциально сможет удовлетворить пунктам 5-7. Учитывая вероятные сложности с интеграцией новой функциональности в уже существующую большую систему, и невозможности легкого переноса QReal в web пространство, было принято решение о написании новой, заточенной под конкретный, искомый класс задач DSM-платформы.

2. Архитектура

В рамках настоящей работы была спроектирована и реализована DSM-платформа под названием “Диаграммный исполнитель”. Она удовлетворяет всем критериям, перечисленным в соответствующем разделе обзора, и позволяет создавать DSM-решения, имеющие клиент-серверную архитектуру. В совокупности со средой метамоделирования эти решения образуют единую платформу, которая позволяет обогащать предметно-ориентированный язык новыми блоками непосредственно во время работы с ним.

2.1. Компоненты системы

Схема Диаграммного исполнителя изображена на рис. 3. На ней перечислены следующие компоненты:

- Среда метамоделирования: отвечает за создание предметно-ориентированных языков.
- Среда исполнения. Отдельный сервис, исполняющий функции на языке Python.
- GUI-редактор. Позволяет конструировать, настраивать и запускать диаграммные программы для данного DSL.
- Система хранения данных. Предоставляет возможность добавлять и загружать как пользовательские файлы, так и сами диаграммы.
- Связующее ядро. Является центральной компонентой системы, объединяет остальные модули в единый продукт.

Далее эти компоненты будут рассмотрены более подробно.

Основным языком программирования в проекте является Kotlin[12]. В настоящие дни он стремительно набирает популярность и как язык программирования под платформу Android, и как язык общего назначения, хотя первые его версии были выпущены только в 2011 году. Kotlin способен транслироваться как в JVM байт код, так и в Javascript[13]. Кроме того, сама структура языка позволяет просто и элегантно описывать различные внутренние DSL, самым известным примером которого является KotlinHtml[14]. Он позволяет задавать HTML разметку с помощью вложенных функций.

Существует также проект Kotlin Common[15]. Он позволяет описывать классы, которые транслируются одновременно и в байт код и в Javascript. Данная технология позволяет описывать общие сущности, которые могут использоваться как на серверной, так и на клиентской стороне. Таким образом, Kotlin, как язык программирования, способен покрыть практически весь проект, что является весомым преимуществом. В "Диаграммном исполнителе" все компоненты кроме среды исполнения выполнены на языке Kotlin.

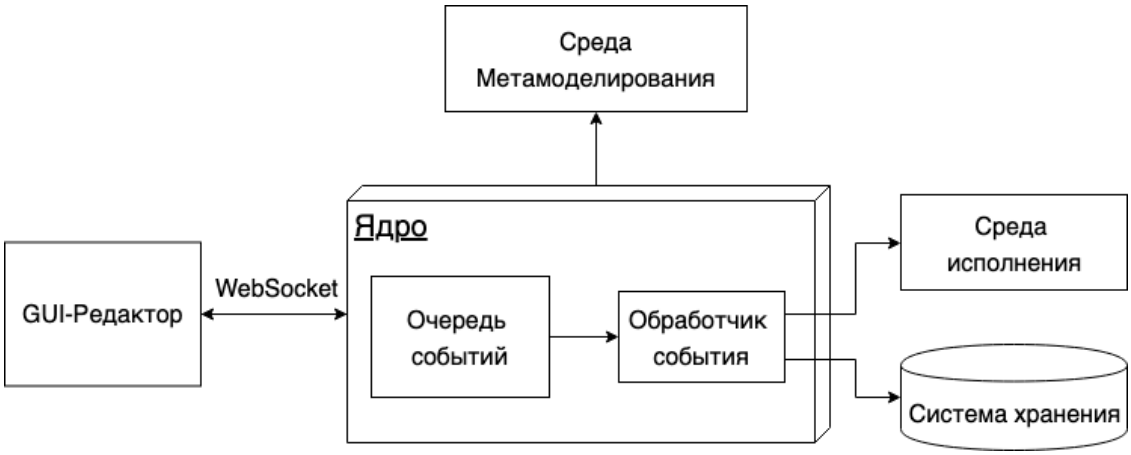


Рис. 3: Архитектура "Диаграммного исполнителя"

[12]Kotlin — URL: https://kotlinlang.org/

[13]KotlinJs — URL: https://kotlinlang.org/docs/tutorials/javascript/kotlin-to-javascript/kotlin-to-javascript.html

[14]KotlinHtml — URL: https://github.com/Kotlin/kotlinx.html

[15]Multiplatform project — URL: https://kotlinlang.org/docs/reference/multiplatform.html

2.1.1. Среда метамоделирования

Среда метамоделирования представляет собой git-репозиторий с некоторой заданной иерархией папок. Первый уровень содержит три папки: pure, render и resource. Они названы по трем типам возможных сущностей-блоков диаграммного языка. Подробнее они, как и сам метаязык, будут отражены в разделе 1 главы 3. Следующий уровень иерархии также состоит из папок, названия которых описывают различные категории блоков. На третьем уровне вложенности находятся файлы с расширением .py[16]. Те из них, что специальным образом помечены (с помощью средств метаязыка), будем называть дескрипторами.

Внутри дескрипторов описываются блоки диаграммного языка. Это описание состоит из функции на языке Python и специальных правил, сформулированных на метаязыке в комментариях к исходному коду. Эти правила задают различные свойства блоков, в том числе и их способность быть соединенными с другими блоками. Подробнее они раскрываются в разделе 1 главы 3. Таким образом, дескрипторы содержат абсолютно корректный программный код. В нем можно подключать как сторонние библиотеки, так и функции из других файлов. Приведенная выше иерархия папок является значимой только для создания дескрипторов — помимо них репозиторий может содержать произвольное наполнение.

Использование git-репозитория позволяет работать над созданием и модификацией языка сразу нескольким пользователям, поддерживая единственную истинную версию. Это становится особенно актуально в силу возможности обогащения языка в режиме реального времени. Чтобы избежать ситуаций, когда диаграмма содержит блок, который был модифицирован уже после добавления, было введено версионирование, которое и разрешает конфликты.

[16] *.py — расширение файлов для языка Python

2.1.2. Среда исполнения

Среда исполнения представляет из себя сервер, написанный на языке Python. Он имеет доступ как к локальной копии репозитория среды метамоделирования, так и к системе хранения пользовательских файлов. Информация об их физическом расположении приходит в виде RPC-вызова во время начала работы системы. В общей сложности среда исполнения поддерживает четыре вида операций.

1. Сохранить информацию о хранилищах данных.
2. Обновить функции некоторого дескриптора.
3. Удалить значение по ссылке.
4. Исполнить некоторую функцию.

Также при инициализации системы поступают запросы на обновление функций всех дескрипторов среды метамоделирования. На каждый такой запрос среда исполнения находит соответствующий файл в локальной копии git-репозитория, импортирует его и записывает все функции, которые в нем заданы, в специальное хранилище по ключу “категория-дескриптор”. В результате, чтобы получить исходный код, требуется указать категорию, дескриптор и название функции.

Эти исходные коды впоследствии исполняются, а их возвращаемый результат сохраняется в некоторой таблице. Для доступа к ней генерируется ссылка, которая и отправляется внешнему клиенту. Этот подход позволяет передавать данные между блоками диаграммы без их непосредственной сериализации: достаточно передать ссылку на объект, что является обыкновенной строкой.

Ядро системы может пометить ссылку как устаревшую и послать соответствующий запрос среде исполнения. Та, в свою очередь, очистит значение по указанной ссылке, и таким образом избавится от “мусора”.

На рис. 4 представлена схема исполнения последней из перечисленных операций.

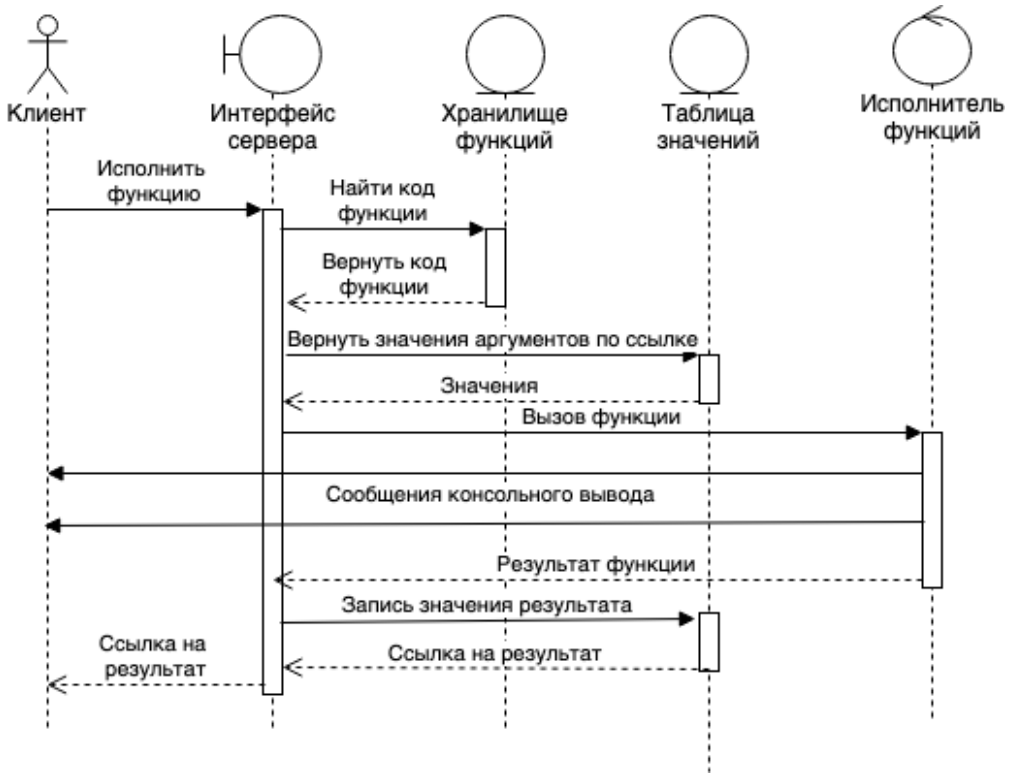


Рис. 4: Схема исполнения функции

Приведенной схеме соответствует следующая последовательность действий.

1. Из тела запроса извлекается информация о категории, дескрипторе и названии функции. По этим данным из хранилища, описанного выше, извлекается соответствующий исходный код.

2. Из тела запроса извлекаются аргументы, от которых должна исполниться функция. Эти аргументы представляют из себя ссылки на хранимые в некоторой таблице объекты.

3. Выполняется вызов функции от полученных аргументов. Во время ее исполнения перехватывается стандартный вывод в консоль, и в режиме реального времени возвращается клиенту среды исполнения (ядру системы).

4. По завершении шага 3 возможны два варианта в зависимости от типа функции.

   (a) Сохранение результата в таблицу из шага 2 и возвращение клиенту ссылки на это значение.

   (b) Возвращение клиенту ссылки на некоторый сохраненный ресурс (картинка или HTML-файл). Этот ресурс впоследствии может быть показан пользователю в GUI-редакторе.

При обновлении git-репозитория среды метамоделирования ядро диаграммного исполнителя формирует специальное событие (Обновление), которое в конечном итоге приводит к вызову соответствующей операции среды исполнения. Таким образом, поддерживается хранение только актуальных исходных кодов.

2.1.3. GUI-редактор

GUI-редактор написан на языке Kotlin с использованием KotlinReact DSL [2]. Он позволяет пользоваться каркасом библиотеки React из исходного кода на Kotlin. Как и в случае с TypeScript [33], у него существует проблема взаимодействия с уже написанными Javascript библиотеками, и Kotlin решает ее аналогично: написание классов-оберток с явным указанием типов для Javascript функций. В настоящей работе указанная операция в полной мере или частично была проделана для целого ряда библиотек таких как react-diagrams [17], react-bootrap[18], react-tippy[19] и др. Этот результат является самостоятельным, он размещен на GitHub[20] и может быть использован в других проектах.

Реализованный редактор состоит из следующих компонент.

1. Сцена. Это центральная область редактора, на ней размещаются диаграммы.

[17]Библиотека для работы с диаграммами — URL: https://github.com/projectstorm/react-diagrams

[18]Бибилиотека, содержащая различные пользовательские компоненты — URL: https://react-bootstrap.github.io/

[19]Библиотека для вывода всплывающих сообщений — URL: https://github.com/tvkhoa/react-tippy

[20]Репозиторий Диаграммного исполнителя — URL: https://github.com/artbez/DataFlowGram

2. Палитра элементов. Находится справа от сцены и содержит все доступные функции выбранного типа: pure, render или resource. Они сгруппированы по категориям и по дескриптору. Тип функции можно указать в области над палитрой.

3. Конфигуратор. Находится слева от сцены и используются для присваивания значений свойствам выделенного элемента. Так же он позволяет пользователю посмотреть, какой код соответствует какому блоку.

4. Верхнее меню. Предоставляет возможность сохранять и открывать диаграммы.

5. Панель исполнения. Заменяет конфигуратор во время исполнения диаграммы. Отражает текущий статус исполняемых блоков и предоставляет либо консольный вывод соответствующего исходного кода, либо объект, который требуется показать пользователю.

6. Хранилище данных. Предоставляет интерфейс для манипуляции с пользовательскими файлами. Позволяет загружать новые файлы в систему, и выгружать и удалять существующие.

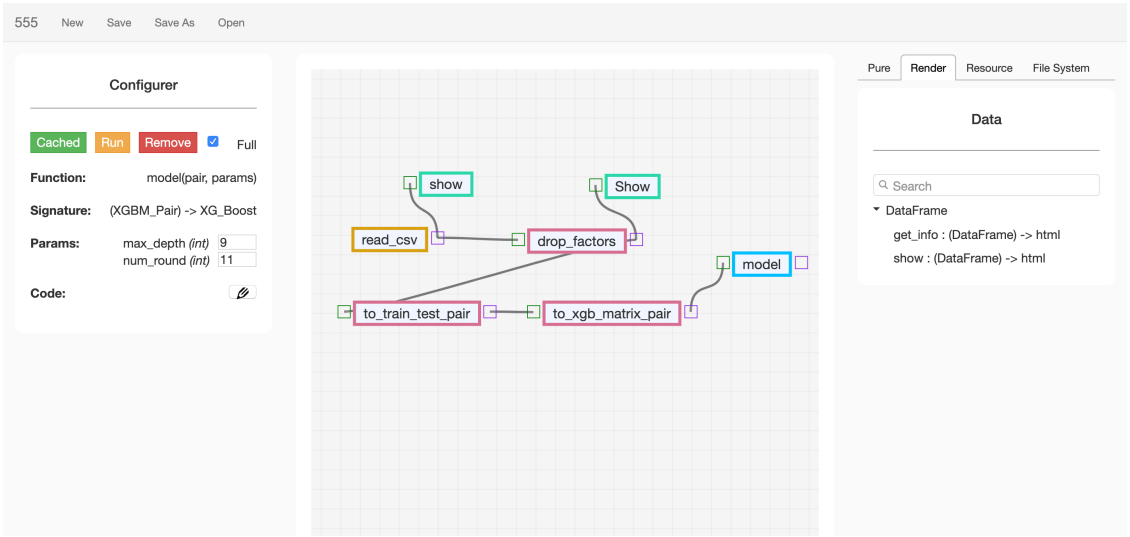


Рис. 5: GUI-редактор

На рисунке 5 представлен реализованный GUI-редактор. На нем отражены компоненты под номерами 1-4.

Ранее было упомянуто, что блоки могут быть трех разных типов: pure, render или resource. Каждому из них соответствует свой цвет — красный, зеленый и желтый. В него окрашивается граница прямоугольника, задающего блок. Pure функции честно исполняют свой исходный код и распространяют результат по всем доступным каналам. Функции render служат для отображения графиков и картинок в панели исполнения. Они не имеют выходных портов. Resource функции используются для связи с файловой системой. Они позволяют читать из файла и записывать в файл.

После того, как диаграмма была составлена, ее можно запустить. Для этого достаточно сначала кликнуть по элементу диаграммы, а затем по кнопке Run в конфигураторе. Реализованный редактор позволяет кэшировать вычисления: если кликнуть по кнопке Cached, блоки, которые уже исполнялись, повторно вычисляться не будут. Их значения автоматически распространятся дальше. Ниже представлен рисунок, отражающий панель исполнения (слева).

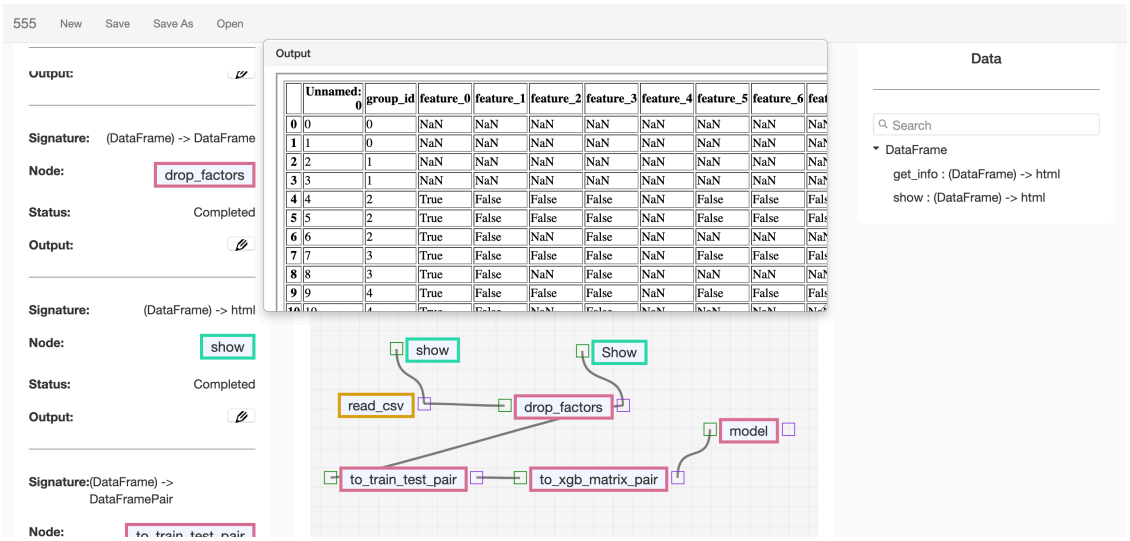


Рис. 6: Исполнение диаграмммы

2.1.4. Система хранения данных

Система хранения данных состоит из трех модулей.

- Репозиторий диаграмм.
- Репозиторий пользовательский файлов.
- Хранилище временных файлов.

Репозиторий диаграмм обеспечивает все CRUD[21] операции с графическими программами, составленными в GUI-редакторе. В качестве физического хранилища была выбрана NoSQL база данных MongoDB. Сами диаграммы помещаются в нее в формате JSON.

Репозиторий пользовательских файлов позволяет читать, сохранять и удалять произвольные файлы. Физически они располагаются локально в некоторой папке по специальному пути. Операции над этим репозиторием могут быть вызваны как прямыми командами пользователя через интерфейс GUI-редактора, так и из системы исполнения, во время выполнения функции.

Третий модуль отвечает за хранение временных файлов, которые создаются для последующего отображения пользователю в момент исполнения диаграммы. Они удаляются автоматически, в момент когда блоки диаграммы перестают на них ссылаться.

2.1.5. Связующее ядро

Связующее ядро объединяет все перечисленные компоненты в единую систему. Оно состоит из обработчиков внешних событий, очереди внутренних событий, и исполнителя, который последовательно на них реагирует. Он может обращаться как к среде исполнения, чтобы обновить ее состояние или выполнить команду, так и к хранилищу данных, если событие направлено на работу с пользовательскими файлами. На рис. 7 представлена подробная схема решения. Блоки на ней отражают перечисленные модули ядра, а стрелки характеризуют взаимодействие

[21] CRUD — Create, Read, Update and Delete.

между ними: вызов из одного модуля (начало стрелки) методов другого (конец стрелки). Словом “интерфейс” помечены блоки, предоставляющие интерфейсы для обращения к глобальным компонентам системы.

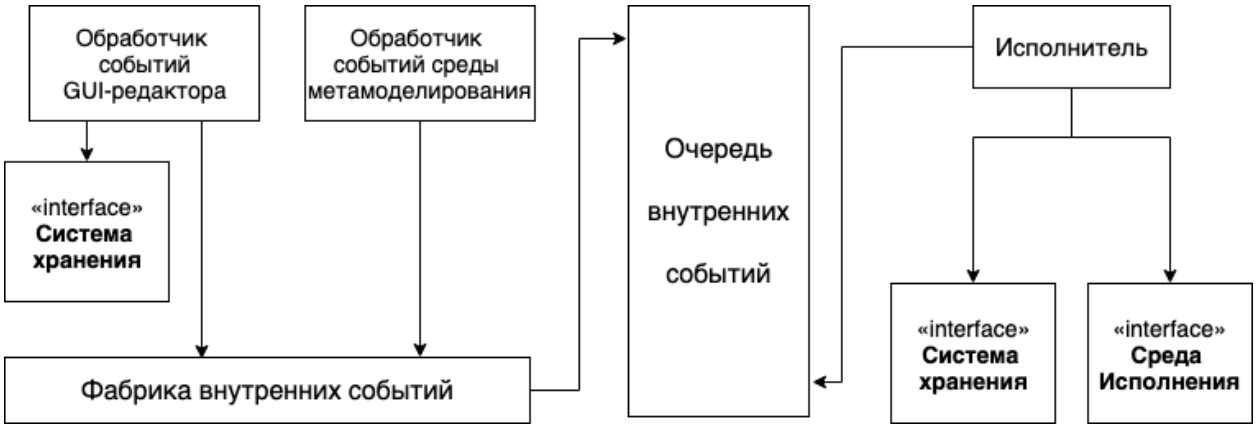


Рис. 7: Архитектура ядра

В качестве примера рассмотрим внешнее событие “исполнить диаграммный блок”. К нему будет применена следующая последовательность действий.

1. Обработчик событий GUI-редактора анализирует внешнее событие и решает, что нужно создать внутреннее.
2. Фабрика создает это событие и кладет в специальную очередь.
3. Ожидание, пока исполнитель не возьмет его на обработку.
4. Исполнитель анализирует событие и решает, что его должна обработать среда исполнения, и перенаправляет его.

Данная схема гарантирует обработку событий в порядке их поступления, если это необходимо. Иные операции исполняются на уровне обработчика GUI-редактора. Таким образом, например, если git-репозиторий среды метамоделирования обновился, среда исполнения будет по прежнему работать с устаревшим исходным кодом, пока до соответствующего события не дойдет очередь.

2.2. Протоколы взаимодействия

Между GUI-редактором и ядром системы взаимодействие происходит как посредством WebSocket соединения, так и с помощью REST-вызовов. Последние применяются только для операций с пользовательскими файлами. Использование технологии WebSocket обуславливается двумя факторами:

1. Операции могут продолжаться длительное время. Например, обучение нейронной сети.

2. Необходим механизм, который бы позволил перенаправлять консольный вывод исполняемого кода напрямую клиенту в режиме реального времени.

Ядро системы написано на языке Kotlin, а среда исполнения на языке Python. В качестве интрумента, обеспечивающего взаимодействие между ними, был выбран gRPC[22]. Он позволяет по текстовому описанию интерфейсов к RPC-вызовам генерировать исходные коды на различных языках программирования для клиентской и серверной сторон. Более того, эта библиотека поддерживает не только интерфейсы функций, которые в результате возвращают значения примитивного типа, она позволяет асинхронно получать промежуточные значения. Описанный механизм используется для передачи ядру системы консольного вывода.

Диаграммный исполнитель как целостная DSM-платформа имеет модульную архитектуру, с четко выраженными компонентами. Ядро системы обладает очередью событий, которая упорядочивает вызовы исходного кода. Взаимодействие между модулями зачастую происходит с помощью асинхронного обмена сообщениями, благодаря чему появляется возможность наблюдать состояние системы в режиме реального времени. Все перечисленное позволяет параллельно воспроизводить различные диаграммы, наблюдать консольный вывод каждой конкретной функции и при этом редактировать сам язык.

[22]URL: https://grpc.io/

3. Особенности реализации

В начале настоящей главы подробно описывается реализованный язык метамоделирования. Перечисляются его правила, отражается соответствие с исходными кодами на языке Python, задаются ограничения. Данный раздел также содержит примеры определения диаграммных блоков средствами разработанного языка.

Далее по тексту перечисляются графические компоненты GUI-редактора, которые отвечают за создание и исполнение блоков. Также указывается отличие в их поведении для всех заданных типов функций.

В заключении данной главы подробно раскрываются возможности реализованной платформы относительно работы с диаграммами. Перечисляются способы их запуска: частично и целиком. Описывается механизм кэширования, который позволяет последовательно наращивать диаграмму, не теряя результатов предыдущих вычислений.

3.1. Язык метамоделирования

Среда метамоделирования предоставляет специальные файлы — дескрипторы. Они имеют расширение .py и содержат корректный код на языке Python. Чтобы отличать их от остальных файлов репозитория было введено следующее правило: первая строка каждого такого файла должна начинаться с префикса “# exec”.

В дескприпторах перечисляются функции на языке Python (в дальнейшем — исходные функции), над которыми в комментариях и встраиваются конструкции метамоделирования. В настоящий момент реализовано четыре вида этих конструкций.

- $\#\ function = [Name]$ — Определяет отображаемое в GUI-редакторе имя блока, его задает параметр [Name]. Если этот вид конструкции отсутствует, в качестве имени блока будет использоваться название исходной функции.

- $\#\ signature = ([CType]...) \rightarrow [CType|RType|Unit]$ — Определяет правила, по которым данный блок может быть соединен с други-

ми. Состоит из двух частей, которые определяют типы входных портов (их может быть несколько) и тип выходного порта соответственно. Причем последнего может не быть, в этом случае во второй части *signature* ставится Unit. Два блока могут быть соединены стрелкой через свои порты, если типы этих портов совпадают.

- $\#\ description = [Text]$ — Позволяет добавить описание к данному блоку в компоненте “Конфигуратор” GUI-редактора.

- $\#\ param@[Name] : [PType]$ — Позволяет добавлять дополнительные свойства блока. [Name] задает имя свойства, [PType] — его тип.

CType расшифровывается как Custom Type и обозначает любой пользовательский тип. Его не требуется нигде описывать отдельным образом, достаточно указать в *signature* в виде строки. Будем говорить, что два пользовательских типа совпадают, если совпадают соответствующие им строки.

RType расшифровывается как Render Type, используется в случаях, когда результат исполнения функции есть ссылка на некоторый ресурс. RType может принимать одно из следующих значений: img и html. Первое служит для отображения графиков и изображений, второе для визуализации произвольной HTML разметки.

PType расшифровывается как Primitive Type и может принимать одно из следующих значений: int, float, string и bool. Используется только для добавления свойств блоку.

Код 1 отражает пример дескриптора, внутри которого описывается один диаграммный блок. Функция fit по данным для обучения и данным для валидации результата обучает некоторый классификатор из библиотеки xgboost. С помощью средств метамоделирования создается соответствующий ей диаграммный блок с именем “Train model”. Он имеет два входящих порта типа Data и один исходящий порт типа Model. Конфигуратор блока будет содержать описание “Train XGBoost

models.” и два настраиваемых параметра max_depth и num_round, оба имеющих тип int.

```
# exec
import xgboost as xgb

# function=Train model
# signature=(Data, Data)->Model
# description=Train XGBoost models.
# param@max_depth:int
# param@num_round:int
def fit(train_data, validate_data, params):
    param = {'silent': 1,
             'max_depth': int(params['max_depth']),
             'min_child_weight': 1,
             'objective': 'binary:logistic',
             'eval_metric': 'auc'}

    evallist = [(validate_data, 'eval'), (train_data, 'train')]
    num_round = int(params['num_round'])
    return xgb.train(param, xtr, num_round, evallist)
```

Код 1: Дескриптор

Существуют ограничения на вид исходной функции при фиксированных правилах метамоделирования, которые к ней применяются. Рассмотрим произвольный диаграммный блок. Пусть он имеет

$$signature = (Type_1, Type_2, ..., Type_N) \rightarrow Out.$$

Тогда количество аргументов исходной функции должно быть равно N + 1. Причем k-ый аргумент должен иметь тип $Type_k$ для любого k. Последним аргументом всегда идет $params$, который предоставляет доступ к значениям свойств по их именам.

Расположение дескриптора в файловой системе тоже влияет на блоки, которые он содержит. Директория первого уровня относительно корня определяет тип блока: pure, render или resource. Директория вто-

рого уровня определяет его категорию, а название самого дескриптора — подкатегорию.

Блоки pure

Для pure блоков описание *signature* должно иметь следующий вид

$$signature = ([CType]...) \rightarrow [CType|Unit].$$

Результат их исполнения либо Unit, либо значение пользовательского типа. Оно (значение) представляет из себя ссылку на объект, который должна вернуть исходная функция с помощью оператора return. Код 1, отображенный выше, содержит пример определения pure блока.

Блоки render

Для render блоков описание *signature* должно иметь следующий вид

$$signature = ([CType]...) \rightarrow [RType].$$

Аргумент *params* в таком случае автоматически будет содержать дополнительное свойство “file”, значение которого есть путь (включая имя файла), по которому требуется сохранить ресурс. Ссылка на него сгенерируется в среде исполнения. При этом из исходной функции никакое значение возвращать не требуется. Ниже приводится пример описания render блока, построенного по функции, которая отображает пользователю объект типа DataFrame[23].

```
# signature=(DataFrame)->html
# param@limit:int
def show(df, params):
    df.head(int(params['limit'])).to_html(params['file'])
```

Код 2: Render блок

[23]DataFrame — класс библиотеки Pandas, реализующий табличную структуру данных. URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

Блоки resource

Для resource блоков описание *signature* должно иметь следующий вид

$$signature = () \to [CType|Unit].$$

Кроме того, требуется, чтобы они содержали хотя бы одно из перечисленных ниже свойств:

$$param@file\_in : string \quad \text{или} \quad param@file\_out : string.$$

Первое позволяет выбрать файл из пользовательского репозитория для последующего его прочтения из исходного кода. Второе отражает путь, по которому исходная функция может осуществить запись в этот же репозиторий. Ниже приводится пример описания resource блока, в котором считываются данные из csv-файла в объект типа DataFrame, и возвращается ссылка на него.

```
# signature=()->DataFrame
# param@file_in:string
def read_csv(params):
    return pd.read_csv(params['file_in'])
```

Код 3: Resource блок

3.2. Отображение блоков

Расположение дескрипторов в файловой системе напрямую влияет на отображение блоков в палитре GUI-редактора. Категории отражаются отдельными UI-компонентами, подкатегории и дескрипторы выраждаются в древовидную структуру с одним уровнем вложенности, где листовыми элементами являются названия блоков. Если кликнуть по такому элементу, конфигуратор предоставит подробную информацию о нем. На рисунке 8 приведено соответствие между описанием блока и его представлением.

```
# function=CalculateData
# signature=(Data, Data)->Data
# param@columns:string
# param@value:int
# param@coefficient:float
def simple_function(x, y, params):
    ...
    return x
```

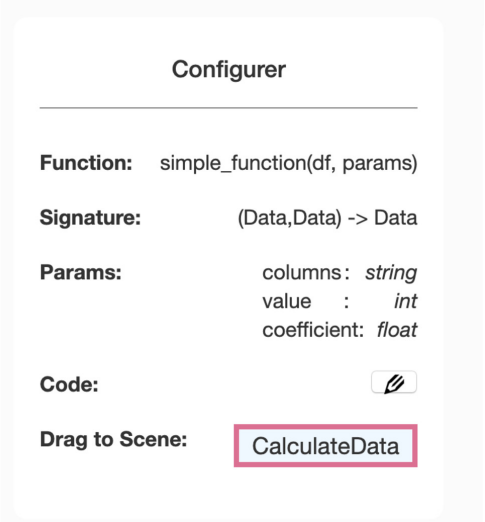


Рис. 8: UI-блока

Если переместить этот блок на сцену и кликнуть по нему, его свойства можно будет редактировать. Эти значения будут передаваться на сервер в момент исполнения блока.

В зависимости от своего типа блоки по-разному отображаются на панели исполнения. Так, pure и resource позволяют наблюдать консольный вывод исходных функций, а render предоставляет пользователю изображение, которое предварительно сохраняется в процессе исполнения блока. Рисунки, иллюстрирующие данные случаи представлены в главе 4.

3.3. Исполнение диаграммы

У каждой вершины присутствует метод “исполниться”. В результате его вызова формируется запрос на сервер, который содержит идентификатор текущего блока, значения его свойств и значения, лежащие во всех входящих портах. Выполняемый блок добавляется на панель исполнения со статусом “Executing...”, с которым он продолжит оставаться до получения от сервера результата данной операции. Затем он переходит в состояние со статусом “Complete”, сохраняет полученный результат и передает его через свой исходящий порт всем доступным вершинам. Кроме того, настоящее DSM-решение поддерживает вывод ошибок, если такие возникают в процессе исполнения исходного кода. В

этом случае пользователю будет показано соответствующее сообщение, а самому блоку будет присвоен статус “Rejected”.

Исполнение диаграммы происходит по следующим двум правилам.

- Если у вершины отсутствуют входящие порты, она исполняется автоматически в момент запуска диаграммы.
- Как только входящие порты некоторого блока становятся заполнены, он немедленно начинает исполняться, после чего распространяет полученный с сервера результат по всем своим исходящим портам.

Разработанное DSM-решение поддерживает как исполнение диаграммы целиком, так и частичное: запускаются только те блоки, которые лежат на пути к заданному. Это позволяет производить вычисления последовательно, шаг за шагом.

Для переиспользования и сохранения результатов было добавлено кэширование на стороне клиента. Пользователь сам решает, использовать его или нет. Если запустить диаграмму в этом режиме, то для каждого уже подсчитанного блока, запрос на сервер будет опущен, вместо этого распространится предыдущее значение. В частности, это позволяет выполнить один раз ресурсоемкую операцию, а затем экспериментировать с применением ее результатов к различным другим блокам. В случае, когда подходящего блока не нашлось, благодаря возможности динамического расширения языка, можно добавить новый.

4. Апробация

Апробация данной работы проводилась в двух разных предметных областях: машинное обучение и томография. Для каждой из них рассматривался некоторый эксперимент, разрабатывался под него соответствующий предметно-ориентированный язык и составлялась графическая программа. В результате получалось решение, способное наглядно представить ход эксперимента. Кроме того, благодаря кэшированию и параметризации отдельных блоков, эти программы можно легко модифицировать и исполнять, и, как следствие, получать новые результаты.

4.1. Решение задачи классификации

В главе 1 настоящей работы были рассмотрены Microsoft ML Studio и MachineFlow в качестве средств визуального программирования для работы в области машинного обучения.

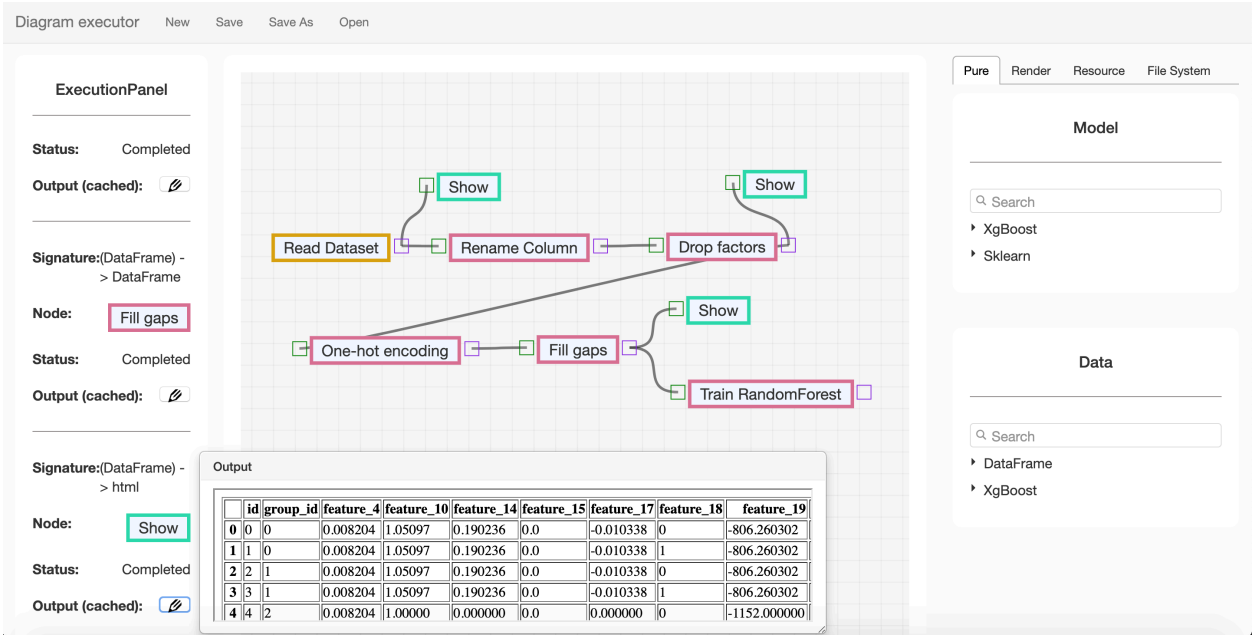
Продукт компании Microsoft содержит широкий набор функций-блоков, имеет удобный интерфейс и позволяет проводить сложные эксперименты. Однако, с его помощью невозможно создавать отдельные, пользовательские блоки на языке Python, аргументы или результат исполнения которых, отличны от типа DataFrame. Это накладывает очень серьезные ограничения на расширяемость языка под конкретные нужды пользователя.

MachineFlow лучше справляется с проблемами расширения: пользовательские блоки принимают в качестве входных данных некоторый указанный файл, и в качестве выходных порождают новый. Тем не менее, не все объекты языка Python просто поддаются сериализации, которая, к тому же, может занимать довольно много времени.

В рамках настоящей работы был создан небольшой предметно-ориентированный диаграммный язык для решения задачи классификации объекта по его признакам. Он лишен перечисленных недостатков конкурентов.

Рассматриваемая задача звучит следующим образом: требуется получить классификатор, который по некоторым признакам объекта предсказывает, к какому классу он относится. Для решения этой проблемы пользователь обычно располагает данными для обучения. В них присутствуют признаки объекта, как правило, требующие дополнительной обработки, и ответы — номера признаков, к которым эти объекты относятся.

Таким образом, формируется последовательность действий: прочитать данные, обработать их, обучиться на них. Причем обработка данных может состоять как из композиции стандартных функций, так и требовать индивидуального подхода, в этом случае пользователю придется создавать новый блок с необходимым исходным кодом. На рис. 9 представлен пример решения поставленной задачи с помощью Диаграммного исполнителя.



	id	group_id	feature_4	feature_10	feature_14	feature_15	feature_17	feature_18	feature_19
0	0	0	0.008204	1.05097	0.190236	0.0	-0.010338	0	-806.260302
1	1	0	0.008204	1.05097	0.190236	0.0	-0.010338	1	-806.260302
2	2	1	0.008204	1.05097	0.190236	0.0	-0.010338	0	-806.260302
3	3	1	0.008204	1.05097	0.190236	0.0	-0.010338	1	-806.260302
4	4	2	0.008204	1.00000	0.000000	0.0	0.000000	0	-1152.000000

Рис. 9: Обучение классификатора

Ниже представлено подробное объяснение каждого блока приведенного решения.

Read Dataset имеет тип resource. Считывает данные из указанного в параметрах к блоку файла и создает по ним объект класса DataFrame.

Show имеет тип render. Отображает DataFrame пользователю в виде таблицы. С помощью этого отображения можно оценить исходные данные и сделать выводы относительно дальнейшей работы с ними.

Rename Column имеет тип pure. Меняет название колонки в объекте класса DataFrame. Старое и новое значения задаются в параметрах к блоку.

Drop factors имеет тип pure. Удаляет указанные столбцы из объекта класса DataFrame. Имена этих столбцов задаются в параметрах.

One-hot encoding имеет тип pure. Заменяет категориальные факторы (задаваемые строковыми константами) на бинарные.

Fill gaps имеет тип pure. Заполняет пропуски в данных по указанной стратегии. В случае приведенного примера, в таких местах проставляются медианные значения признака по всем объектам.

Train RandomForest имеет тип pure. Создает объект класса RandomForest[24] и обучает его на пришедших данных. Позволяет задавать число деревьев и максимальную глубину в качестве параметров.

Результатом выполнения приведенной диаграммы будет обученный классификатор, который можно впоследствии использовать. В зависимости от числа деревьев обучение может занимать разное количество времени, и благодаря пренаправлению в GUI-редактор консольного вывода, пользователь сможет следить за этим процессом. Microsoft ML Studio поддерживает такую функциональность, MachineFlow нет.

[24]RandomForest — класс библиотеки Sklearn, реализующий классификацию на основе деревьев решений. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Однако, существует вероятность, что пользователю понадобится классификатор, который отсутствует в стандартном наборе функций. Например, необходим классификатор XGBoost.

В Диаграммном исполнителе это реализуется путем добавления нескольких функций в среде метамоделирования, причем уже посчитанные вычисления не теряются. Ниже приведен пример модифицированной диаграммы из рис. 9. На нем также можно наблюдать консольный вывод обучения XGBoost классификатора.

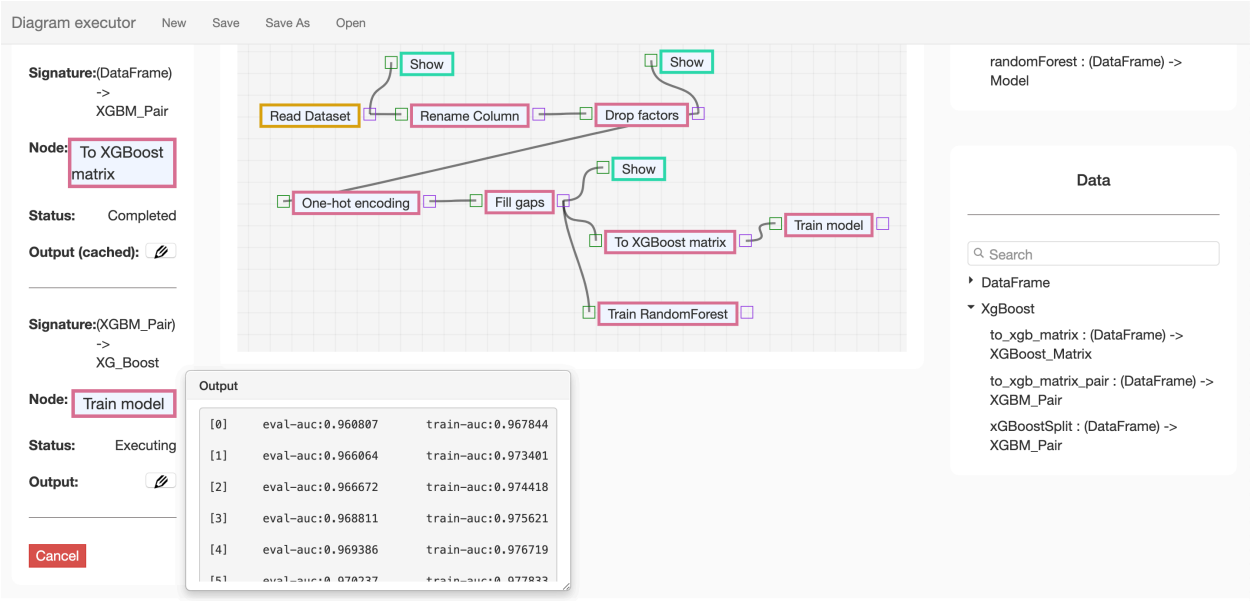


Рис. 10: Добавление блоков

Были добавлены следующие блоки:

To XGBoost matrix преобразует DataFrame в объект класса “xgb.DMatrix” для последующей его подачи на вход классификатору.

Train model создает и обучает классификатор. Этот блок предоставляет широкий набор параметров, которые можно задать в конфигураторе.

Microsoft ML Studio не способен справится с приведенной задачей, потому что этот продукт не предполагает добавление пользователь-

ских функции с аргументами, отличными от DataFrame. С помощью MachineFlow ее реализовать возможно: “xgb.DMatrix” сериализуем, хоть и с дополнительными ресурсными затратами. Однако, если мы захотим объединить описанные два классификатора в MachineFlow, то придется сериализовать уже непосредственно их, что может повлечь большие трудности. В Диаграммной исполнителе таких проблем нет и задача объединения классификаторов сведется к нескольким строкам на языке Python.

4.2. Томографическое исследование

Диаграммный исполнитель использовался в исследовании “Технология восстановления особых областей на основе данных акустической томографии”, выполненном Гонта К.А. Целью данной работы являлась разработка и реализация эффективного алгоритма для выявления патологий организма человека с помощью ультразвука.

В рамках приведенного исследования были проведены эксперименты для проверки ряда гипотез, необходимых для построения алгоритма. Для их осуществления разрабатывались программы на языке Python, которые в последствии запускались на собранных данных. Результатами, как правило, являлись графики, которые впоследствии анализировались. В большинстве своем эксперименты были однотипными и отличались только настраиваемыми параметрами.

В ходе своей работы Гонта К.А. столкнулась с проблемой визуализации проведенных экспериментов. Помимо самих результатов требовалось графически показать процесс их получения, выражаясь терминами предметной области. В частности, требовалось отобразить алгоритм обнаружения объекта в пространстве. Сам процесс происходил следующим образом: специальный прибор, имеющий форму кольца, по периметру которого располагались датчики, перемещался вдоль некоторой оси и сканировал область, заключенную внутри получившегося цилиндра. Итоговый результат предполагал объединение результатов сканирований отдельных сечений, параллельных основанию этого ци-

линдра. На рисунке 11 изображен пример диаграммного представления алгоритма по обнаружению объекта в конкретном, фиксированном сечении.

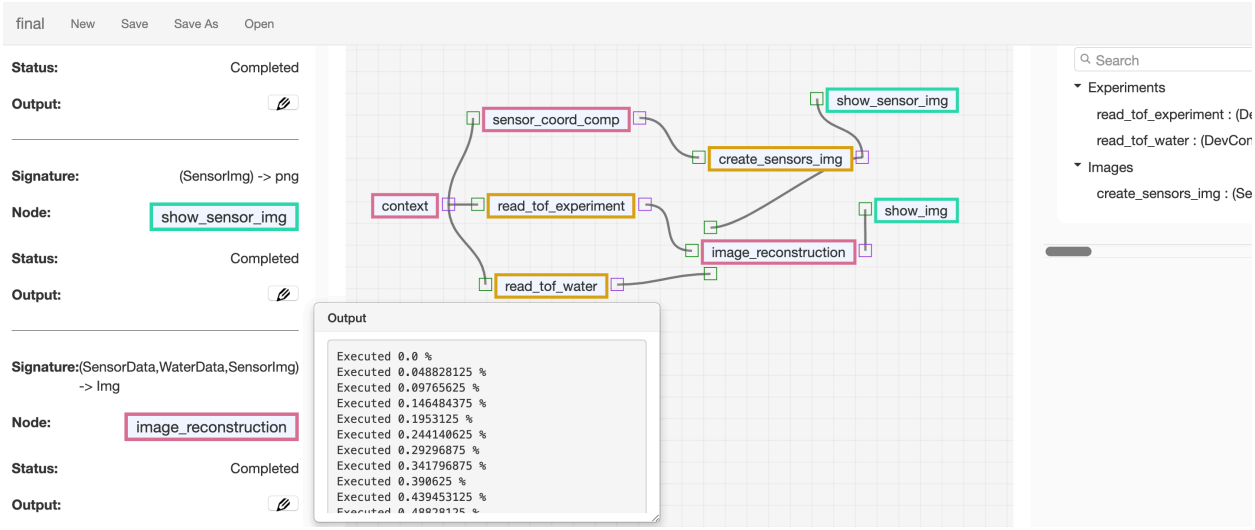


Рис. 11: Исполнение алгоритма

context формирует основные параметры среды. Инициализируются переменные, которые используются в следующих блоках.

sensor_coord_comp извлекает информацию о физическом расположении всех датчиков.

read_tof_experiment считывает фактические данные о временных промежутках между тем, когда один датчик испускает сигнал, а другой его принимает.

read_tof_water считывает данные о временных промежутках между тем, когда один датчик испускает сигнал, а другой его принимает, в случае, когда гарантированно объектов в зоне сканирования нет.

create_sensors_img создает изображение с указанным расположением датчиков. Оно имеет тип SensorImg.

image_reconstruction запускает алгоритм, который по фактическому расположению датчиков, реальным данным и данным в случае, когда объект отсутствует, рассчитывает область, где он вероятно находится. На рис. 11 в панели исполнения отражен консольный вывод этой функции.

show_sensor_img показывает пользователю изображение типа SensorImg.

show_img показывает пользователю произвольное изображение. Результат его исполнения для данного эксперимента показана на рис. 12.

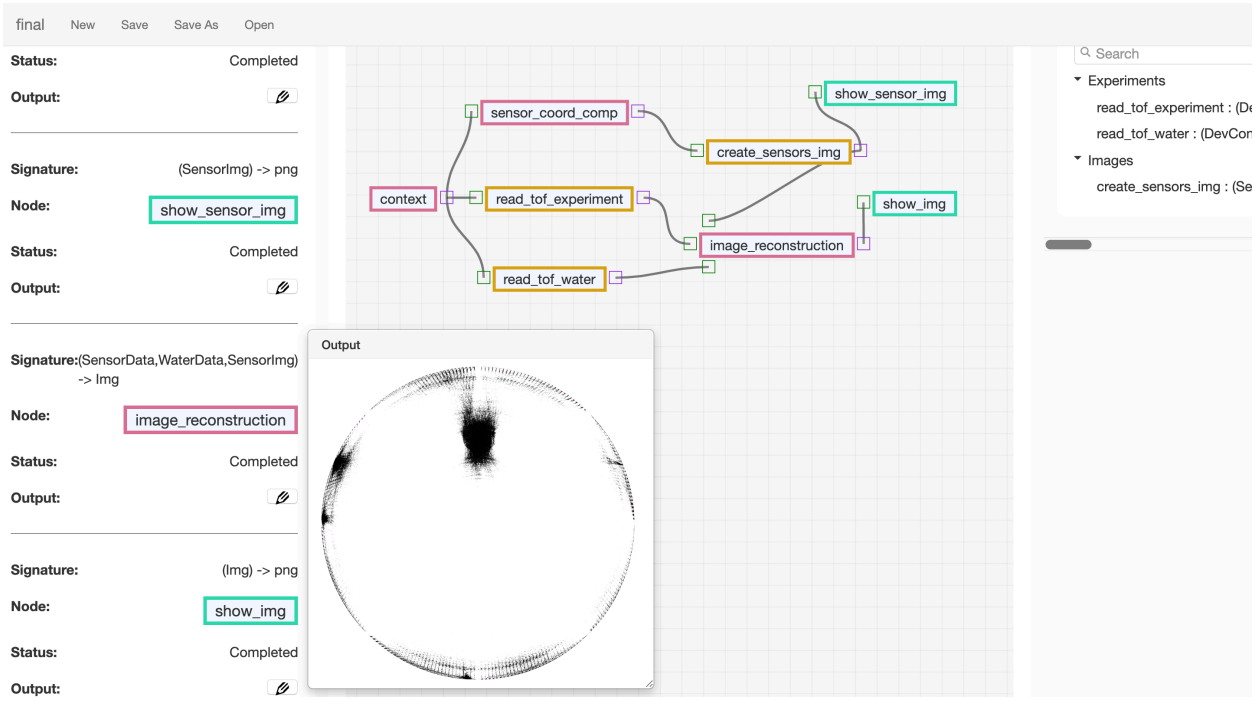


Рис. 12: Результат эксперимента

Диаграммный исполнитель использовался для графического отображения процесса проведения эксперимента. Он был выбран как лучший среди аналогов, позволяющих настраивать и запускать подобные вычисления. После переноса исходного кода эксперимента в Dataflow архитектуру, потребовалось около получаса для составления соответствующего диаграммного языка.

5. Заключение

В ходе данной работы были получены следующие результаты.

1. Разработана архитектура решения, позволяющая удаленно работать с системой. В том числе реализованы следующие компоненты: среда метамоделирования, среда исполнения, GUI-редактор, система хранения данных, связующее ядро.

2. Созданы средства метамоделирования, которые позволяют описывать графические предметно-ориентированные языки, удовлетворяющие паттерну “Каналы и фильтры”. Они включают в себя git-репозиторий с заданной иерархией папок и метаязык, способный встраиваться в файлы исходного кода на языке Python с помощью комментариев.

3. Реализована DSM-платформа на языке программирования Kotlin, позволяющая исполнять и динамически расширять созданные ею языки непосредственно в процессе исполнения программы. Порожденные DSM-решения позволяют переиспользовать результаты произведенных вычислений, предоставляют консольный вывод и ошибки фактически исполняемого кода, и позволяют отображать графики.

4. Произведена апробация работы с применением разработанной DSM-платформы.

   (a) Для решения задачи классификации в области машинного обучения.

   (b) Для исследования “Технология восстановления особых областей на основе данных акустической томографии”, выполненного Гонта К.А.

Диаграммный исполнитель позволяет представить программный код на языке Python с помощью диаграмм, блоки которых выражаются в терминах предметной области. Кроме того, возможности, которые он

предоставляет, соответствуют возможностям, предоставляемым средой разработки Jupiter Notebook, что делает реализованный инструмент удобным для проведения различных исследований.

Список литературы

[1] B. Giuseppe. Machine Learning Algorithms. — Packt Publishing, 2017.

[2] D. Jemerov. Use Kotlin with npm, webpack and react. — URL: https://blog.jetbrains.com/kotlin/2017/04/use-kotlin-with-npm-webpack-and-react/ (дата обращения: 02.09.2018).

[3] D. Robinson. The Incredible Growth of Python. — 2017. — URL: https://stackoverflow.blog/2017/09/06/incredible-growth-python (дата обращения: 03.10.2018).

[4] Defining Generators with MetaEdit+. — URL: https://www.metacase.com/papers/DefiningGeneratorswithMetaEdit.pdf (дата обращения: 13.10.2018).

[5] G. Hohpe. Enterprice integration patterns / Gregor Hohpe, Bobby Woolf. — Addison-Wesley Professional; 1 edition, 2003. — P. 84–91.

[6] J. Beard. A short intro to stream processing. — URL: http://www.jonathanbeard.io/blog/2015/09/19/streaming-and-dataflow.html (дата обращения: 03.10.2018).

[7] J. Barnes. Azure Machine Learning. — Pearson Education, 2015.

[8] J. Vanderplas. Python Data Science Handbook. — O'Reilly Media, Inc, 2017.

[9] M. Voelter. DSL Engineering. — CreateSpace Independent Publishing Platform, 2013. — P. 25–26.

[10] M. Voelter. DSL Engineering. — CreateSpace Independent Publishing Platform, 2013. — P. 40–43.

[11] Mindstorms Robot Tutorial. — URL: https://wiki.eclipse.org/Sirius/Tutorials/Mindstorms (дата обращения: 18.10.2018).

[12] Pipes and Filters. — URL: http://blog.petersobot.com/pipes-and-filters (дата обращения: 02.09.2018).

[13] S. Blackheath. Functional Reactive Programming / Stephen Blackheath, Anthony Jones.

[14] S. Kelly. MetaEdit+ at the Age of 20. — URL: https://www.metacase.com/papers/MetaEdit+_at_the_Age_of_20.pdf (дата обращения: 22.09.2018).

[15] S. Kelly. Domain-Specific Modeling: Enabling Full Code Generation / Steven Kelly, Juha-Pekka Tolvanen. — Wiley-IEEE Computer Society Press, 2008.

[16] S. Kelly. Visual domain-specific modelling: Benefits and experiences of using metaCASE tools / Steven Kelly, Juha-Pekka Tolvanen. — 2010. — URL: https://pdfs.semanticscholar.org/d1a8/0c5b53bf4882090d1e1c31da134f1b1ffd48.pdf (дата обращения: 04.09.2018).

[17] S. Misirlakis. The 7 Most In-Demand Programming Languages of 2018. — URL: https://www.codingdojo.com/blog/7-most-in-demand-programming-languages-of-2018 (дата обращения: 22.09.2018).

[18] S. McClure. GUI-fying the Machine Learning Workflow: Towards Rapid Discovery of Viable Pipelines. — URL: https://towardsdatascience.com/gui-fying-the-machine-learning-workflow-towards-rapid-discovery- (дата обращения: 02.09.2018).

[19] А.Н Терехов. QReal: платформа визуального предметно-ориентированного моделирования / А. Н. Терехов, Т. А. Брыксин, Ю. В. Литвинов // Программная инженерия. – No 6. — 2013.

[20] А.Н Терехов. Среда визуального программирования роботов QReal:Robots / А. Н. Терехов, Т. А. Брыксин, Ю. В. Литви-

нов // III Всероссийская конференция «Современное технологическое обучение: от компьютера к роботу». — 2013.

[21] А.Н Терехов. Среда для обучения информатике и робототехнике QReal:Robots / А. Н. Терехов, Ю. В. Литвинов, Т. А. Брыксин // Девятая независимая научно-практическая конференция «Разработка ПО 2013». — 2013.

[22] Д.В. Кознов. Основы визуального моделирования. — М: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007.

[23] Д.В. Кознов. Методология и инструментарий предметно-ориентированного моделирования // Диссертация на соискание ученой степени доктора технических наук. — СПбГУ. — 2016.

[24] И.О. Сухов. Сравнение систем разработки визуальных предметно-ориентированных языков // Пермский государственный национальный исследовательский университет. — 2012.

[25] Интрументарий Audiomulch. — URL: `http://www.audiomulch.com/` (дата обращения: 11.10.2018).

[26] Интрументарий DG Logic. — URL: `http://www.dglogik.com/` (дата обращения: 11.10.2018).

[27] Интрументарий Jupiter. — URL: `https://jupyter-notebook.readthedocs.io/en/stable/` (дата обращения: 02.09.2018).

[28] Интрументарий MetaEdit+. — URL: `http://www.metacase.com` (дата обращения: 02.09.2018).

[29] Интрументарий Microsoft SDK. — URL: `https://docs.microsoft.com/en-us/visualstudio/modeling/modeling-sdk-for-visual-studio-domain-specific-languages?view=vs-2019` (дата обращения: 15.09.2018).

[30] Интрументарий Node-Red. — URL: https://nodered.org/ (дата обращения: 11.10.2018).

[31] Интрументарий Project Reactor. — URL: https://projectreactor.io/ (дата обращения: 02.09.2018).

[32] Интрументарий Shell. — URL: https://www.case.edu/php/chet/bash/bashref.html (дата обращения: 02.09.2018).

[33] Интрументарий TypeScript. — URL: https://www.typescriptlang.org/ (дата обращения: 02.09.2018).

[34] Ограничения ML Studio. — URL: https://docs.microsoft.com/en-us/azure/machine-learning/studio/execute-python-scripts (дата обращения: 15.09.2018).

[35] Т.А. Брыксин. Платформа для создания специализированных визуальных сред разработки программного обеспечения // Диссертация на соискание ученой степени кандидата технических наук. — СПбГУ. — 2016.