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

ФАКУЛЬТЕТ Робототехники и комплексной автоматизации(РК)

КАФЕДРА Систем автоматизированного проектирования (САПР)

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

Разработка web-ориентированных CASE инструментариев автоматизации построения исходных кодов графоориентированных решателей

Студент РК6-81 _____ **С.А. Неклюдов**
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель ВКР _____ **А.П. Соколов**
(Подпись, дата) (И.О.Фамилия)

Консультант _____ **А.Ю. Першин**
(Подпись, дата) (И.О.Фамилия)

Нормоконтролер _____ **Грошев С.В.**
(Подпись, дата) (И.О.Фамилия)

2019 г.

АННОТАЦИЯ

Работа посвящена проблемам разработки программного обеспечения генерации исходного кода. Был произведен аналитический обзор литературы по теме: “Особенности, технологии и методы генерации исходного кода программ”. Были рассмотрены основные подходы к решению задач генерации кода, найдены существующие разработки и выявлены основные тенденции к развитию технологий в рассматриваемой области. В ходе работ был разработан генератор исходного кода программ, использующий технологию преобразования *Model to text*. Внесены изменения в *web*–клиент *comwpc*, а также в библиотеку *comsdk* Распределенной вычислительной системы *GCD*.

Созданное программное обеспечение стало основой для создания подсистемы генерации исходного кода графоориентированных решателей в рамках существующей программной системы РВС *GCD*.

СОКРАЩЕНИЯ

- РВС *GCD* – распределенная вычислительная система *GCD*;
- *ComSDK* – библиотека для разработки графоориентированных программных реализаций сложных вычислительных методов на основе технологии *graph–based software engeneering*;
- *GUI* – графический пользовательский интерсфейс;
- *UML* – унифицированный язык моделирования;
- *GBSE – graph–base software engeneering*;
- *Comwpc* – веб–клиент РВС *GCD*;
- *M2T*(*model to text*) – преобразование модели в текстовое представление, используется в генераторах кода;
- *M2M*(*model to model*) – преобразования между моделями;
- *MDD* (*model driven development*) - модель–ориентированная разработка;

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Разработка автоматизированных распределенных систем стратегического, корпоративного уровня, создание на их основе прикладных сервисов является сложным процессом, требующим, в том числе, написания повторяющегося программного кода, а также типовых программных конструкций, реализующих шаблоны программирования, характерные для той или иной задачи. Современные средства разработки зачастую накладывают дополнительные идеологические, синтаксические, структурные ограничения [1].

В настоящее время все более актуальным становится создание программных средств, позволяющих автоматизировать труд разработчика прикладного программного обеспечения. При проектировании разработчики с целью облегчения процесса программирования и сопровождения стараются сделать структуру приложения максимально простой и стандартизированной. Поэтому при старте нового проекта разработчику зачастую приходится выполнять однотипные действия по созданию базового шаблона той или иной конструкции. Автоматизировать данный процесс позволяет специальное программное обеспечение – генераторы исходного кода. При правильных входных данных и корректно разработанном генераторе, большая часть кода может быть сгенерирована автоматически.

В результате поиска источников литературы были исследованы несколько работ, описывающих подходы к генерации кода и организации структуры генераторов: 1) генераторы, построенные на основе шаблонов; 2) модель–ориентированная генерация; 3) генерация на основе многоуровневого набора правил;

Общим термином для дальнейшего рассмотрения данных методов является понятие модели. Под моделью понимается условный формально описанный образ объекта исследования, конструируемый так, чтобы отобразить существенные свойства для исследования или разрабатываемой системы. Также важно понимать специфику сферы разработки, для которой генерируется

исходный код. Например, сферой разработки могут быть бизнес – процессы в системах класса *BPM*. Для этих систем моделями предметной области будут являться графические описания бизнес-процессов.

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Общая схема генерации на основе шаблонов включает в себя два этапа. На первом этапе осуществляются процедуры шаблонизации и конфигурирования. На вход шаблонизатора подаются конструкции языка и изменяемые поля. Входными Параметрами процесса конфигурирования являются модели предметной области. В результате выполнения этих двух процедур получается сконфигурированная модель предметной области. Модели предметной области формализуют структурные и функциональные аспекты автоматизируемой предметной области и являются источниками используемых в автоматизируемой системе имен, типов и ограничений. На втором этапе из полученных шаблонов и конфигураций происходит “рендеринг” итогового программного кода[1]. Генераторы кода на основе шаблонов принято разделять в зависимости от типов применяемых шаблонов: а) предварительно определенные (англ. Predefined) или шаблоны, “зашитые” в генерирующую программу; б) основанные на формате генерируемого выходного объекта (англ. *Output base*); в) основанные на правилах (англ. *Rule-based*). На рисунке 1 представлена архитектура генераторов на основе шаблонов.

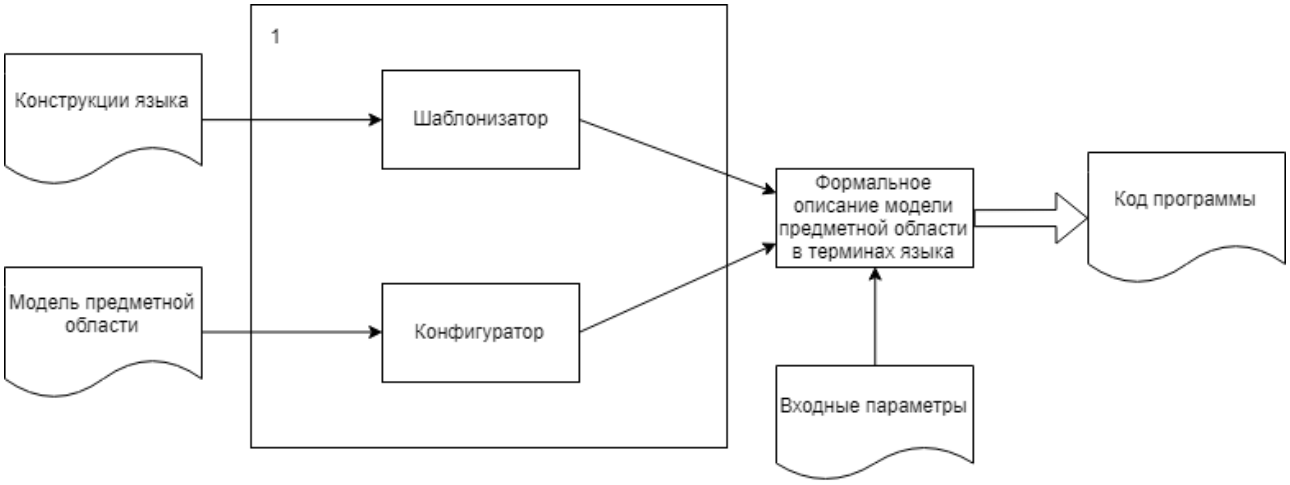

Рис. 1 Архитектура генератора кода на основе шаблонов

В работе [12] автор предлагает методику построения генератора исходного кода программ на основе многоуровневого набора правил. Методика основана на семиотическом подходе к построению информационных систем. В семиотических системах принимаются во внимание синтаксис (набор конструкций языка), прагматика (набор исходных требований к сгенерированному коду) и семантика (отношения между прагматикой и синтаксисом) решаемых задач. Такие системы удобно представлять в виде метаграфов. Отличительная особенность таких графов — наличие метавершин и(или) метаребер, иначе говоря - вложенных структур.

Модель - ориентированная парадигма - подход к разработке программного обеспечения, который может затрагивать практически весь спектр этапов разработки программного обеспечения, начиная от сбора требований до поддержки системы [2]. Модели - основные структурные единицы в данном методе. Модели создаются и используются процессами жизненного цикла программного обеспечения. Применение моделей обеспечивает повышение эффективности путем инкапсуляции логики работы программно-реализуемых объектов информационной системы. Модель части системы *M* представляется в виде документа *D1* на некотором языке моделирования (*UML*, *IDEF*, *ARIS* [4]). Далее осуществляется интерпретация модели *M* и конструирование заготовок исходного кода в документе *D2* на заранее выбранном языке программирования или описания, что также является описанием модели *M*. Такой процесс называют преобразованием моделей.

1.1. ПРЕОБРАЗОВАНИЯ МОДЕЛЕЙ

Преобразования моделей являются частью модель - ориентированного подхода к разработке программного обеспечения.

Преобразование действительно, если при заданных условиях и ограничениях преобразование модели *M1* к *M2* происходит так, что свойства модели *M2* «переносятся» на *M1* [5]. В статье [5] была приведена классификация преобразований по типу задач, для которых они применяются: а) ограничительные запросы; б) конкретизирующие преобразования; в) трансляция и семантика трансляции; г) аналитические преобразования; д) реакционные преобразования. Для демонстрации типов преобразований воспользуемся диаграммами классов *UML*.

Ограничительные запросы обеспечивают выборку отдельных частей из общего множества элементов модели *M1* и перенос их на модель *M2*. Такие преобразования часто используются, когда необходимо “передать” выходную модель *M2* каким-либо другим частям системы. Например, при моделировании конечных автоматов [5]. На рисунке 2 представлена иллюстрация такого преобразования. Модель, полученная в результате преобразования, обладает лишь некоторыми из свойств исходной модели.

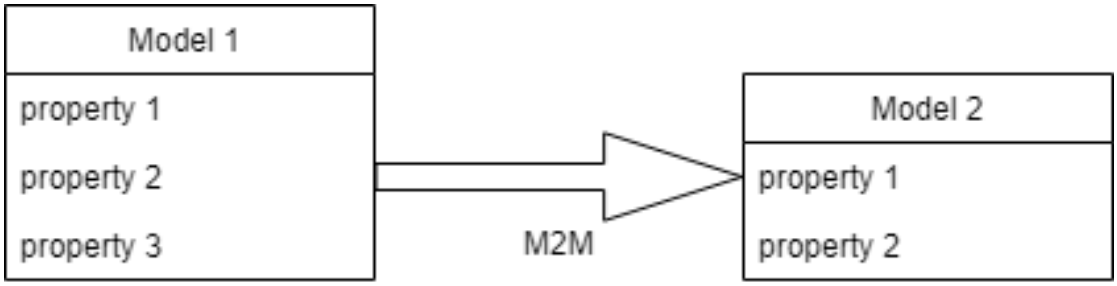

Рис. 2 Ограничительные запросы

Конкретизирующие преобразования (рисунок 3) создают модели, описывающие предметную область на низком уровне абстракции, в то время как исходная модель представляет собой высокоуровневую спецификацию. Например, *XML* схемы конкретизируются документами в формате *XML*;

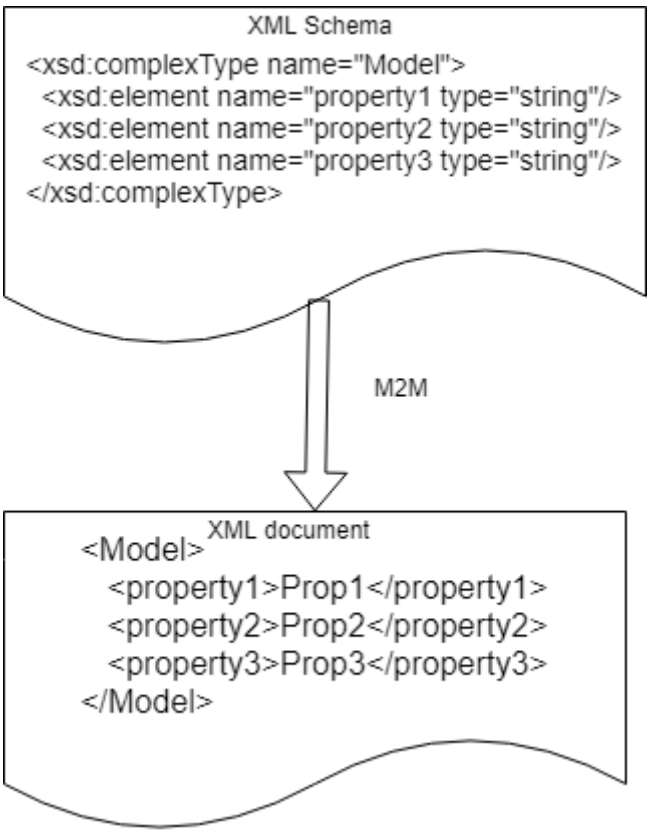

Рис. 3 Конкретизирующие преобразования

Трансляция и семантика трансляции - преобразование, переносящее значения параметров исходной метамодели (модель, описывающая язык моделирования) на целевую метамодель, иными словами, такое преобразование переводит модели с одного языка описания или программирования на другой. Данное преобразование необходимо для обеспечения делегирования функций исходной модели функциям целевой модели. Примером может служить трансляция команд в машинный код, при котором высокоуровневые инструкции делегируют свои функции машинным командам. Важным нюансом является невозможность выполнить какие-либо действия, используя начальную модель (например, нельзя запустить программу на языке программирования, не произведя ее трансляцию в машинные команды или интерпретацию);

Аналитические преобразования реализуют алгоритмы рассмотрения различных свойств моделей и принятия решений на основе полученных данных об этих свойствах. Примером такого алгоритма может быть поиск неиспользуемого кода, применение которого позволит сократить размер,

повысить читаемость и эффективность программы. Схема такого фильтра представлена на рисунке 4. Исходная модель является не оптимальной так как содержит дубликат функции *callFunction1*. Модель *Model 2*, полученная в результате модельного преобразования не содержит дубликатов, но имеет аналогичные *Model 1* функциональные возможности.

Рис. 4 Пример аналитического преобразования

Реакционные преобразования реагируют на внешние возмущения, то есть модель *M2* является полным отображением модели *M1*, за исключением компонент, на которые оказывались внешние воздействия преобразовательного, ограничительного, конкретизирующего или иного характера. На рисунке 5 представлена схема, когда пользователь отправляет инструкцию изменить свойство в модели *Model 2*.

Рис. 5 Реакционные преобразования

1.2. МОДЕЛЬ-ОРИЕНТИРОВАННЫЙ ПОДХОД В ГЕНЕРАЦИИ ИСХОДНОГО КОДА ПРОГРАММ

Стадии разработки приложения, при использовании модель подхода к проектированию включают:

а) разработку модели предметной области;

б) трансформацию модели в платформо-зависимую модель;

в) генерация исходного кода программы на основе платформо–зависимой модели.

Такой подход к генерации кода классифицируют по типу преобразований (трансформации) модели: а) *M2M* (*model to model*); б) *M2T* (*model to text*) [3].

Особенностью *M2M* преобразований является явное представление моделей входной и выходной областей. На рисунке 1 представлена схема генерации с использованием преобразований *M2M*. Исходная модель конкретизируется посредством преобразования *M2M*, а затем из конкретизированной модели генерируются исходные коды программ. Данный подход может применяться в системах, где исходная модель является составной частью не только генератора, но и других узлов информационной системы, в следствии чего исходная модель не может создаваться в правилах, доступных генератору. Например, исходная модель представлена в виде описания в формате *XML*, а генератор работает с моделями, описанными в формате *JSON*. Для этого необходимо произвести уточняющее преобразование *M2M*.

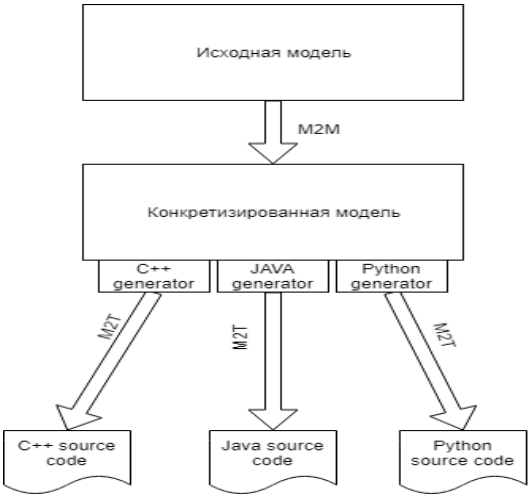

Рис. 1 Генерация на основе подхода M2M

Преобразование *model to text* (*M2T*) - формирование из исходной модели текстовых файлов. *M2T* применяется в реверс-инжиниринге, генерации кода и документации, сериализации модели (обмена моделями), а также для визуализации и исследования моделей [10]. На рисунке 2 представлена схема генераторов, работающих по такому принципу. Примером такой системы может служить разработка авторов статьи [11] *AT2015*: из модели, представленной *UML* диаграммой, генерируется код на языке *C#*.

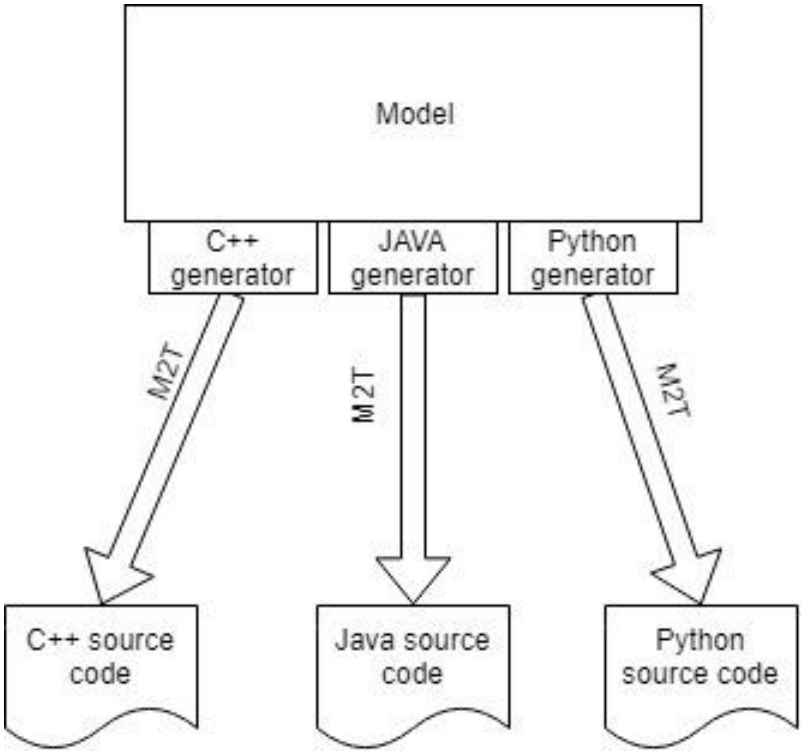

Рис. 2 Схема генерации кода на основе принципа M2T

Преобразования *M2T* являются широко поддерживаемой техникой интеграции платформ в современных инструментах модель-ориентированной разработки (*MDD*). Имеется множество реализаций языков шаблонов (например *Eclipse Xpand*, *Xtend2*, *EGL*, *JET* или *Acceleo*[10]). Для каждого из этих языков генераторы M2T и шаблоны генераторов могут быть реализованы различными способами [17].

В данной разработке был выбран метод генерации, основанный на преобразованиях *M2T*, так как на его основе можно генерировать код

преобразовывая описательное представление модели в текст без промежуточных преобразований, как это было бы в методе *M2M*.

1.3. ИСТОРИЯ РАЗВИТИЯ ПРОГРАММНЫХ СРЕДСТВ ГЕНЕРАЦИИ ИСХОДНОГО КОДА

Генераторы программного кода были важной частью программного обеспечения практически с самых истоков эры вычислительной техники. Стартовой точкой развития генераторов программного кода можно считать появления первых компиляторов. Кодогенерация — важнейшая часть процесса трансляции программы, что является главной задачей компилятора.

С течением времени генераторы стали усложняться. В них стали широко применяться алгоритмы оптимизации, расширяя возможности компиляторов. Например, в статье [6] представлена разработка *CVXGEN* — программного инструмента, генерирующего простой код на языке *C*, практически не содержащий включаемых библиотек. В основе разработки лежит моделирование задач выпуклой оптимизации [7]. Одной из важнейших тенденций в данной сфере является развитие рынка многоядерных и графических процессоров. *CUDA SDK* — инструмент, позволяющий эффективно использовать ресурсы графических процессоров *nVidea* для задач с высокой степенью распараллеливания. В этом инструменте реализованы алгоритмы кластеризации, целью которых является оптимальное распределение параллельных операций между ресурсами процессора. Существует ряд проблем, обусловленных неудобством традиционных подходов к параллельному программированию. Например, недетерменированность характера вычислительного процесса затрудняет выявление ошибок методами отладки [9]. Поэтому, актуально создание средств, позволяющих манипулировать «черными ящиками» - стандартными функциями, реализующими параллельные алгоритмы. Таким образом, конечный пользователь (разработчик) будет писать последовательные программы, не углубляясь в особенности параллельного программирования. В статьях [8, 9] представлены генераторы, призванные скрыть реализацию параллельных алгоритмов.

Еще одной важной ступенью в развитии генераторов кода является принятие на конференции *OMG* в 2001 году стандарта разработки, управляемой моделями (*MDD*) - модель-ориентированного подхода к разработке программного обеспечения и создания модель-ориентированной архитектуры программного обеспечения. Его основной идеей является построение абстрактной метамодели управления и обмена метаданными, а также задание способа преобразования моделей. Дальнейшим шагом было принятие *MOF* - стандарта разработки *MDD* — модели программирования, в которой проект представляется в виде краткого и понятного языка [10]. Причиной появления такого подхода послужила необходимость создания архитектуры метамоделирования для *UML*.

MOF представляет собой четырехуровневую архитектуру. Ядром является модель языка метамоделирования (пример метамодели - описание *UML*). На нижних уровнях находятся сами модели *UML* и данные для построения конкретных экземпляров на основе моделей. *MDD* позволяет проводить преобразования между этими уровнями, обеспечивая переходы от абстракций к конкретным реализациям.

Автор статьи [13] отмечает формирование намечающегося перехода с объектно-ориентированной технологии к язык-ориентированному (предметно-ориентированному) подходу. Идея такого подхода — формирование предметно-ориентированных языков, которые представляют собой некий интерфейс между пользователем и низкоуровневыми конструкциями.

1.4. ПРИМЕРЫ РАЗРАБОТОК В ДАННОЙ ОБЛАСТИ

1) Для разработчиков на платформе *dotNet* компания *Microsoft* разработала генератор кода *Text Template Transformation Toolkit* или *T4* [5] на базе шаблонов. Для использования данного генератора необходимо написать файл-шаблон в формате **.tt*. Существует большое количество готовых шаблонов для генерации стартового проекта под необходимые требования, однако из-за отсутствия единого репозитория поиск этих шаблоном крайне затруднен. Кроме того, *T4* используется исключительно для генерации только *C#* кода [1].

2) Генератор для системы *Graphplus Templet*, реализующий генерацию параллельных алгоритмов. Алгоритм генерации кода реализован в препроцессоре *templet*, совместная работа которого с интегрированной средой разработки *Microsoft Visual Studio* организуется, как описано ниже. Пусть код программы пишется на языке *C++*, что предусмотрено в текущей реализации. Модулем в данном случае является пара файлов: заголовочный файл с расширением h; файл реализации с расширением cpp. Код этих файлов помещается в общее пространство имён. Каждому модулю в проекте интегрированной среды разработки соответствует файл с *XML* спецификацией, который тоже включается в проект (обычно в папку ресурсов приложения). Для файла *XML*-спецификации в проекте указывается способ его обработки. Перед сборкой проекта (*pre-built*) такой файл должен быть обработан *templet* препроцессором, который генерирует фрагменты кода, соответствующие модели программирования в файлах модуля.

3) Десять лет назад группа компаний «ИВС» разработала технологическую программную платформу, которая автоматизировала процесс проектирования информационных систем [1]. Разработчикам удалось значительно снизить временные затраты программистов на реализацию программных инструментов. Идея данной платформы была максимально простой и заключалась в следующем: по аналитическим моделям бизнес - процессов, происходящих в той или иной компании, составляется диаграмма классов в нотации языка *UML*, которая в дальнейшем преобразуется в исходный код (*C#* или *SQL*). На выходе системы разработчик получает готовый скомпилированный прототип системы управления информационными процессами компании со следующей функциональностью: а) наличие графического пользовательского интерфейса (*WinForms* или *WEB*-приложение); б) система авторизации пользователей; в) стандартные операции по работе с данными, такие как чтение, запись, редактирование, удаление, фильтрация и форматирование.

После генерации данный прототип дорабатывается вручную с учетом требований, прописанных в техническом задании. При большом разнообразии

провайдеров баз данных [2] (*Microsoft SQL, MySQL, PostgreSQL, Oracle Database*) прототип системы управления был совместим с любым из перечисленных провайдеров (это дает возможность в будущем без особых усилий менять источник данных). Такая универсальность стала возможна за счет наличия некоторого промежуточного слоя между источником данных и системой управления. Для автоматизации процесса программирования в области проектирования АСУТП было принято решение о разработке специализированной прикладной интегрированной среды разработки, основной задачей которой будет автоматизированная генерация исходного кода и документации на таких языках программирования , как *C#, Java, C* [1].

4) Наиболее известным инструментом для быстрого создания стартового проекта является *Yeoman*. Скаффолдинг - метод метапрограммирования, при котором проект генерируется на базе анализа требований разработчика [4]. *Yeoman* базируется на трех основных компонентах: менеджер зависимостей пакетов, необходимый для загрузки, обновления и удаления дополнительных пакетов (*bower*); инструмент для сборки *JavaScript* проектов с использованием заранее написанных задач (*grunt*); базовое приложение, отвечающее за генерацию базы для нового приложения (*yo*) [1].

5) В [14] статье представлена разработка генератора *Genesys*, основанная на объединении сервис-ориентированного и модель-ориентированного подходов. Автор приводит пример генерации *HTML* кода документации при помощи разработанного инструмента *Genesys*. *Genesys* является частью программной системы поддержки жизненного цикла программного продукта *jABC*. При моделировании с помощью *jABC* системы или приложения собираются на основе (расширяемой) библиотеки четко определенных многократно используемых блоков *SIB*. Из таких моделей можно генерировать код для различных платформ (например, платформ на основе *Java*, мобильных устройств и т. д.). Поскольку генераторы кода для этих моделей также встроены в *jABC*, такие концепции, как начальная загрузка и повторное использование

существующих компонентов, обеспечивают быструю эволюцию библиотеки генерации кода.

6) Исследователи из Университета Малаги представили среду для проверки контрактов преобразований *M2M, M2T, T2M Tracts* [15]. Разработка мотивирована тенденцией к развитию модель-ориентированного подхода, что влечет за собой необходимость тестирования, валидации и верификации преобразований. В *Tracts* для тестирования преобразований применяется *M2M* трансформации, позволяющие сопоставлять структуру моделей. *M2T* разработчики системы применяли для генерации кода, а *T2M* для встраивания текста в модель. Таким образом преобразования *M2T* и *T2M* могут сводиться к *M2M*.

7) *DWE*(*Data Warehouse Evolution* — разработка в сфере *BI*(*Business Intelligence*). Необходимость данного программного инструментария заключается в том, чтобы обеспечивать автоматическое распространение изменений в модели *OLAP* на многомерную модель *DW* [16]. Преобразование *M2M* представляется в виде концепции *QVT* (*Query, View, Template*) и используется вместе с *M2T*, реализованного при помощи шаблонов *Acceleo*.

8) *CVXGEN* - программный инструмент, который на высоком уровне описывает семейство выпуклых задач оптимизации и автоматически генерирует пользовательский код на языке *C*, который компилируется в надежный, высокоскоростной решатель для целого набора задач. *CVXGEN* генерирует простой, плоский, безбиблиотечный код, подходящий для встраивания в приложения реального времени. Сгенерированный код практически не содержит ветвей и поэтому имеет предсказуемое поведение во время выполнения [6].

1.5. ГРАФООРИЕНТИРОВАННЫЙ ПОДХОД

Для решения различных технических и научных задач может применяться графоориентированный подход. Смысл данного подхода состоит в декомпозиции большой задачи на несколько подзадач и представлении алгоритма ее решения в виде связного графа. Узлами графа обозначают состояния данных, используемых программной системой, а ребрами – функции-

обработчики, которые манипулируют этими данными. Перед запуском функций-обработчиков, происходит проверка данных в узле графа при помощи функций-предикатов. Например, алгоритму метода Ньютона будет соответствовать графовая модель, представленная на рисунке 8:

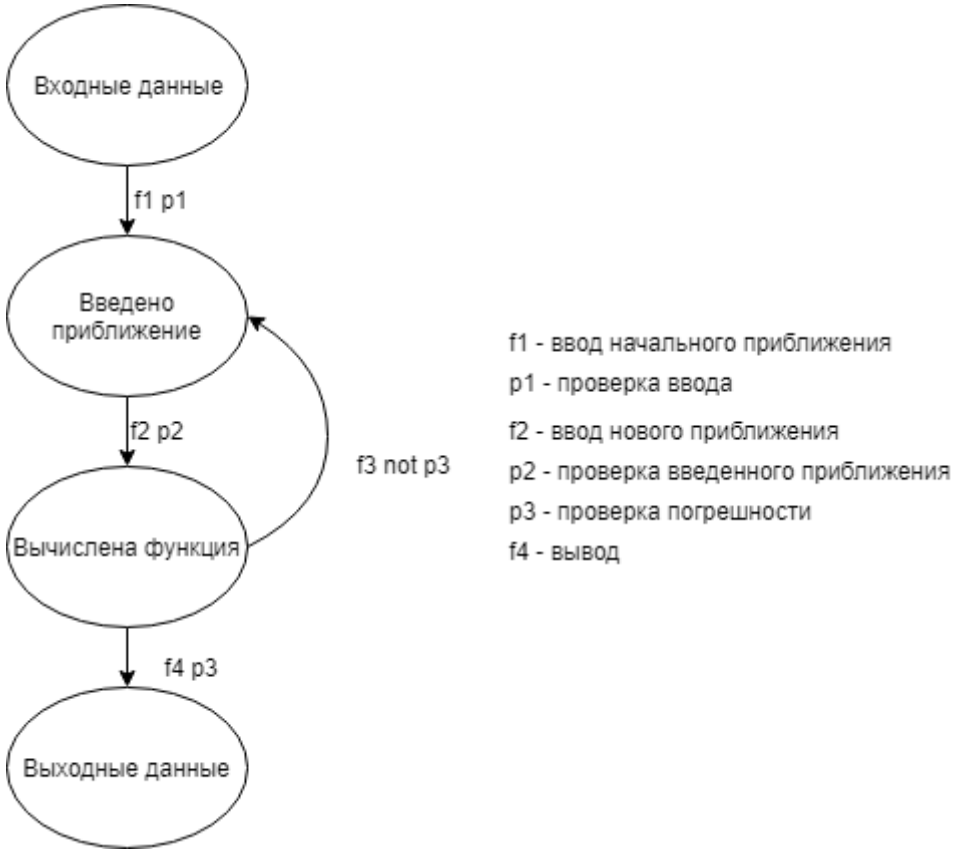

Рис. 8 Графовая модель алгоритма Ньютона

В МГТУ им. Н.Э. Баумана был разработан программный инструментарий для решения задач подобным методом. Суть решения задачи состоит в реализации функций – ребер графа и формировании графовой модели в виде документа в формате *aDot*. После чего, используя библиотеку *ComSDK* возможно совершить обход данной модели и вызов необходимых функций. Задачей данной работы является автоматизация процесса генерации кода, вызывающего функции данной системы.

2. КОНЦЕПТУАЛЬНАЯ ПОСТАНОВКА ЗАДАЧИ

Целью разработки является: создать программное обеспечение генерации исходного кода для Распределенной вычислительной системы *GCD*, а именно:

1. Доработать тестовую функцию системы, формирующую *GUI* форм ввода на основе заранее подготовленного файла aIni. С помощью этой функции должен быть возможен выбор модели графоориентированного решателя, создание настраиваемых конфигураций генератора и файлов входных параметров.
2. Разработать плагин на языке *C++,* обеспечивающий последовательный холостой обход графовой модели выбранного решателя и построение на ее основе исходного кода программы.

Были выделены следующие задачи:

1. Проведение аналитический обзор литературы по теме “Особенности, технологии и методы генерации исходного кода программ”.
2. Доработка плагина web-клиента РВС *GCD* для обеспечения возможности переопределять файлы *aIni* и вызова таблиц базы данных.
3. Доработка библиотеки *comsdk*.
4. Разработка архитектуры программного обеспечения генерации исходного кода программ.
5. Разработка реализации программного обеспечения в соответствии с созданной архитектурой.
6. Тестирование полученных результатов.

2.1. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ГЕНЕРАТОРА GUI

Первой частью данной работы стала доработка подсистемы ввода-вывода Распределенной вычислительной системы *GCD* (РВС *GCD*), реализованной в виде web-приложения *comwpc* (клиентская часть РВС *GCD* или, далее, просто *web*-клиент). Была реализована поддержка параметров типа ссылка на таблицу базы данных для обеспечения возможности выбора графоориентированного решателя на клиентской стороне. Разработка велась при помощи языка *Python* с использованием фреймворка *Django*, позволившего генерировать *html* формы по файлу входных данных в формате *aIni*, а также языка *JavaScript* с использованием фреймворка *JQuery*.

Для понимания архитектуры *WEB*-клиента следует углубиться в особенности *WEB* разработки с использованием данного набора инструментов. Любое приложение, реализованное при помощи *Django*, представляет собой набор так называемых “моделей”, “представлений” и “шаблонов”. Под моделями понимаются классы объектов системы, например: пользователи, таблицы базы данных, администраторы и др. Представления манипулируют моделями, то есть инкапсулируют в себе логику работы с данными. Шаблоны являются визуальным отображением этих данных. На основе шаблонов генерируется *HTML* код, благодаря которому в дальнейшем можно изменять модели.

Для понимания роли разработки в системе следует представить общую архитектуру подсистемы *web*-клиента *comwpc* (рисунок 9) и последовательность выполнения *ajax* запроса на выполнение функции системы (*action item*) построения *GUI* (*wpc_gui_ini_builder*). Логически приложение разделено на клиентскую (фронтенд) и серверную (бэкенд) части.

К серверной части относятся представления, модели, различные сценарии для манипулирования входными данными, установления сетевого соединения и обмена сетевым траффиком с другими узлами РВС *GCD*. В данной работе на серверной части был модифицирован сценарий *iniparser* осуществляющий разбор файла *aIni* и формирование на его основе списка данных для генерации

форм на клиентской части. Была обеспечена поддержка следующих типов данных: а) строки; б) целые числа; в) вещественные числа; г) ссылка на таблицу базы данных; д) диапазон; е) одномерные и двумерные массивы; ж) множество выбора; з) бинарное множество. Поддерживается возможность не отображать данные в форме, но учитывать их при дальнейшей передаче файлов. Для этого необходимо перед переменной указать атрибут “-”. Также поддерживается возможность создавать формы с полями обязательными для заполнения и комментариями.

К клиентской части относятся шаблоны *Django*, из которых конструируется *HTML* страница, а также *JavaScript* обработчики событий, которые реагируют на манипуляции со страницами. Например, обработка нажатий на кнопки, выбор значений из таблиц и другие действия, специфические для конкретной реализации клиента. В данной работе были произведены доработки шаблона *action_item_templates* с целью поддержки возможности вывода различных экранных форм.

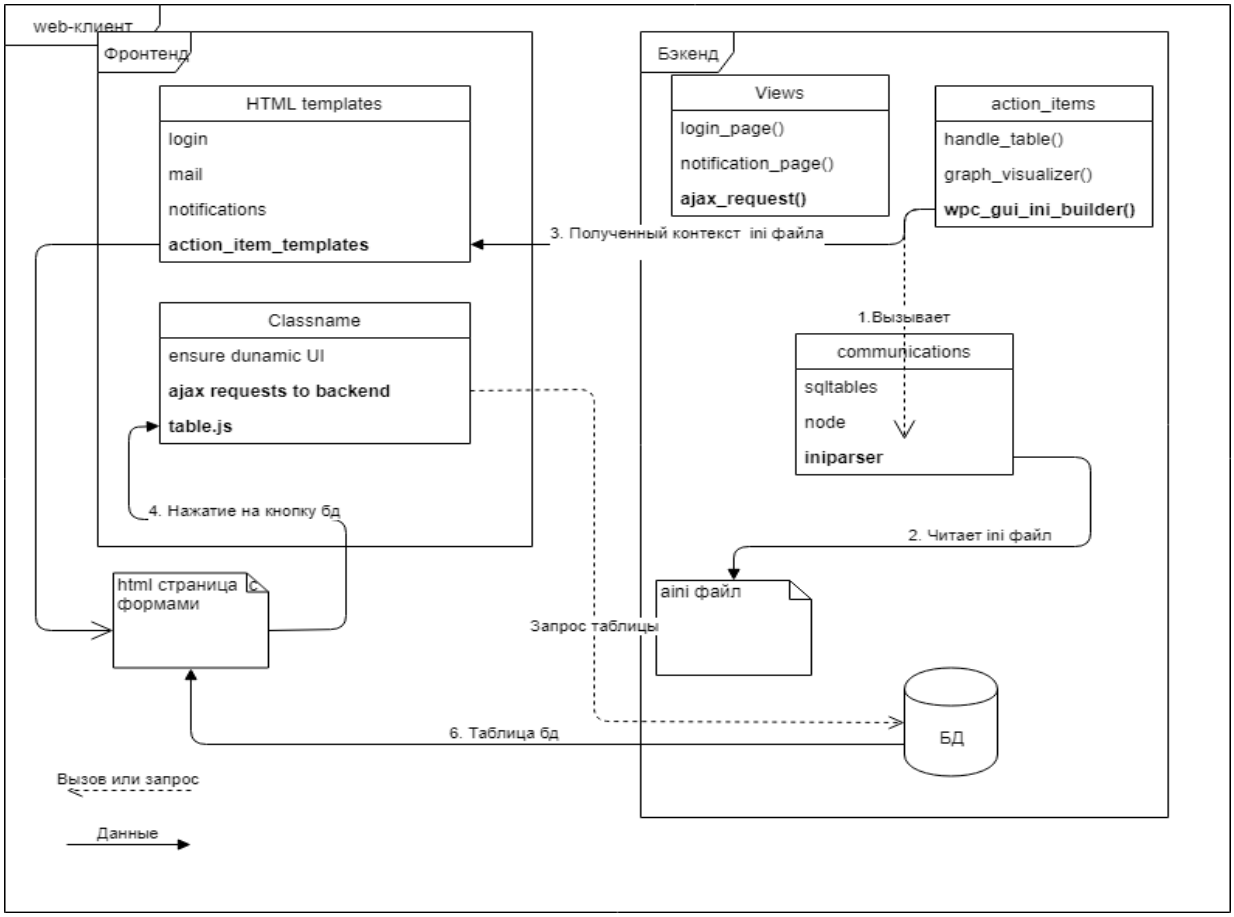

Рис. 9 Архитектура *WEB*-клиента РВС *GCD*

Предполагается, что файл исходных данных в формате *aIni* загружен в директорию *wpc_gui_ini_builder web*-клиента РВС *GCD*. *aIni* представляет собой текстовый формат обмена данными. Типичный файл в формате *aIni* состоит из секций (блоков), содержащих переменные. В листинге 1 представлен пример такого файла.

Листинг 1 Пример файла *aIni*

```
1. [section]
2. variable1 = 1
3. variable2 = 2
```

Процесс генерации формы содержит следующие стадии.

1) Пользователь, находящийся на главной странице, выбирает пункт меню *wpc_gui_ini_test*. В результате формируется *Ajax* запрос, который поступает на вход обработчику *wpc_gui_ini_builder* на серверной части (рисунок 10). На схеме *WEB*-клиента (рисунок 9) этот процесс указан стрелкой под номером 1.

Рис. 10 Меню *WEB*-клиента РВС *GCD*.

2) Обработчик вызывает сценарий *iniparser*, который осуществляет разбор файлов входных параметров в формате *aini* (стрелка 2 на рисунке 9) и формирует контекст для рендеринга *HTML* страницы на основе шаблона. В листинге 2 представлен сформированный контекст для такого файла.

Листинг 2. Пример сформированного контекста

```
Контекст
{
'action_item': 'wpc_gui_ini_builder',
'action_class': 'PYWEBS',
'aini_name': 'WPC_GUI_INI_TEST',
'processing': 'PYTHON_CALLER',
'proc_ai_class': 'PLUGIN',
'all_data':
OrderedDict([('section',
OrderedDict([('variable1', {'value': '1', 'type':
<enu_INIParamType.ptNum: 11>}), ('variable2',
{'value': '2', 'type': <enu_INIParamType.ptNum:
11>})]))])}
```

Парсинг осуществляется построчно, в контекст записывается информация о секциях, атрибутах, переменных, комментариях. На основе регулярных выражений определяется тип значения. Поддерживаются несколько типов данных: целые, вещественные числа, диапазоны, массивы, пары *x-y* для представления функций одной переменной, ссылки на записи таблиц баз данных и др. Разбор этих строк основан на механизме регулярных выражений и реализован с помощью стандартного модуля *re* языка *Python*. Итоговый контекст представляет собой список, содержащий необходимые для генерации формы параметры, такие как тип переменной, информация о секции и атрибутах. Этот список передается в шаблонизатор, где каждому типу данных ставится в соответствие некоторая часть шаблона. Так, на основе переменной логического параметра генерируется чекбокс, а на основе переменной типа целого числа — обычное поле ввода со значением по умолчанию.

3) На основе содержимого контекста генерируется *HTML* страница. Пример сгенерированной на основе контекста из листинга 2 формы представлен на рисунке 11. Схема генерации *HTML* страницы на основе документа в формате *aIni* представлена на рисунке 12.

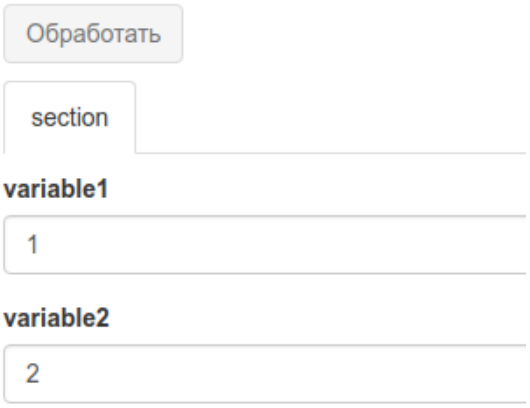

Рис. 11 Форма, сгенерированная на основе файла из листинга

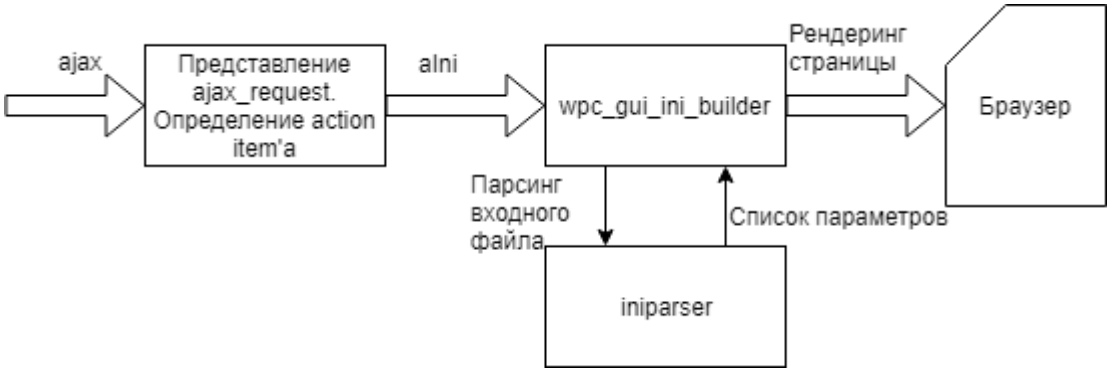

Рис. 12 Схема генерации html страницы на основе документа в формате *aIni*

Таким образом, благодаря произведенным доработкам, стало возможным создавать настраиваемые файлы входных параметров и конфигураций для подсистемы генерации исходного кода на основе графоориентированного подхода.

2.2. РАЗРАБОТКА ГЕНЕРАТОРА ИСХОДНОГО КОДА

Второй частью данной работы является разработка генератора исходного кода графоориентированных решателей. В подходе, используемом в РВС *GCD*, используется графовое представление алгоритмов. Библиотека *comsdk* работает с графами, описанными в документах формата *aDot*. На рисунке 8 приведен пример графовой модели решения задачи нахождения нулей функции методом Ньютона. В листинге 3 представлен документ *aDot*, описывающий такую графовую модель.

Листинг 3. Пример графовой модели алгоритма

```
1. digraph {
2. INPUT [predicate = p1]//входной узел графа
3. STATE1 [predicate = p2]//Состояние - введено приближение
4. STATE2 [predicate = p3]//Состояние - вычислено значение функции
```

```
5. PRED1 [type=boolean, binname=lib, entry_func=p1]//Проверка ввода
6. PRED2 [type=boolean, binname=jugr, entry_func=p2]//Проверка
     введенного приближения
7. PRED3 [type=boolean, binname=jugr, entry_func=p3]
8. FUNC1 [binname=lib, entry_func=f1]//Функция ввода начального
     приближения
9. FUNC2 [binname=lib, entry_func=f2]//вычисление значения функции
10. FUNC3 [binname=lib, entry_func=f3]//Повторный ввод приближения
11. FUNC4 [binname=lib, entry_func=f4]//Вывод значения
12. _BEGIN_ -> INPUT (on_predicate_value = true)
13. INPUT -> STATE_1 [on_predicate_value = true, edge=FUNC1]
14. STATE1 -> STATE_2[on_predicate_value = true, edge=FUNC2]
15. STATE2 -> STATE_1[on_predicate_value = false, edge=FUNC3]
16. STATE2 -> FINILIZE[on_predicate_value = true, edge=FUNC4]
17. FINILIZE -> _END_
18. }
```

Графоориентированный подход предполагает, что перед запуском каждой функции-обработчика осуществляется запуск соответствующей функции-предиката. Функции предиката ставятся в соответствие узлу из которого выходит ребро (ребро в данном контексте эквивалентно функции). Предикат возвращает логический результат. Это позволяет осуществлять проверку состояния данных в узлах графа. Таким образом, на основе возвращаемых значений предикатов можно организовывать разветвляющиеся и циклические конструкции алгоритмов.

Для генерации исходного кода программы на основе графоориентированного подхода необходимо построить очередь вызовов. Очередь представляет собой структуру, содержащую информацию о ребрах графа, такую как название функции, наличие предиката у вершины, значение предиката, по которому будет выполнена функция-обработчик, а также информация о нахождении ребра в цикле. Такая структура будет являться моделью, которая будет преобразована генератором в исходный код при помощи преобразования *M2T*.

Для создания такой модели была модифицирована библиотека *ComSDK*. Была разработана функция *runWithoutExec* (рисунок 13). Функция производит обход графа, не вызывая функций. Результатом выполнения этой функции является модель очереди вызовов, представленная в виде контейнера *std::vector*.

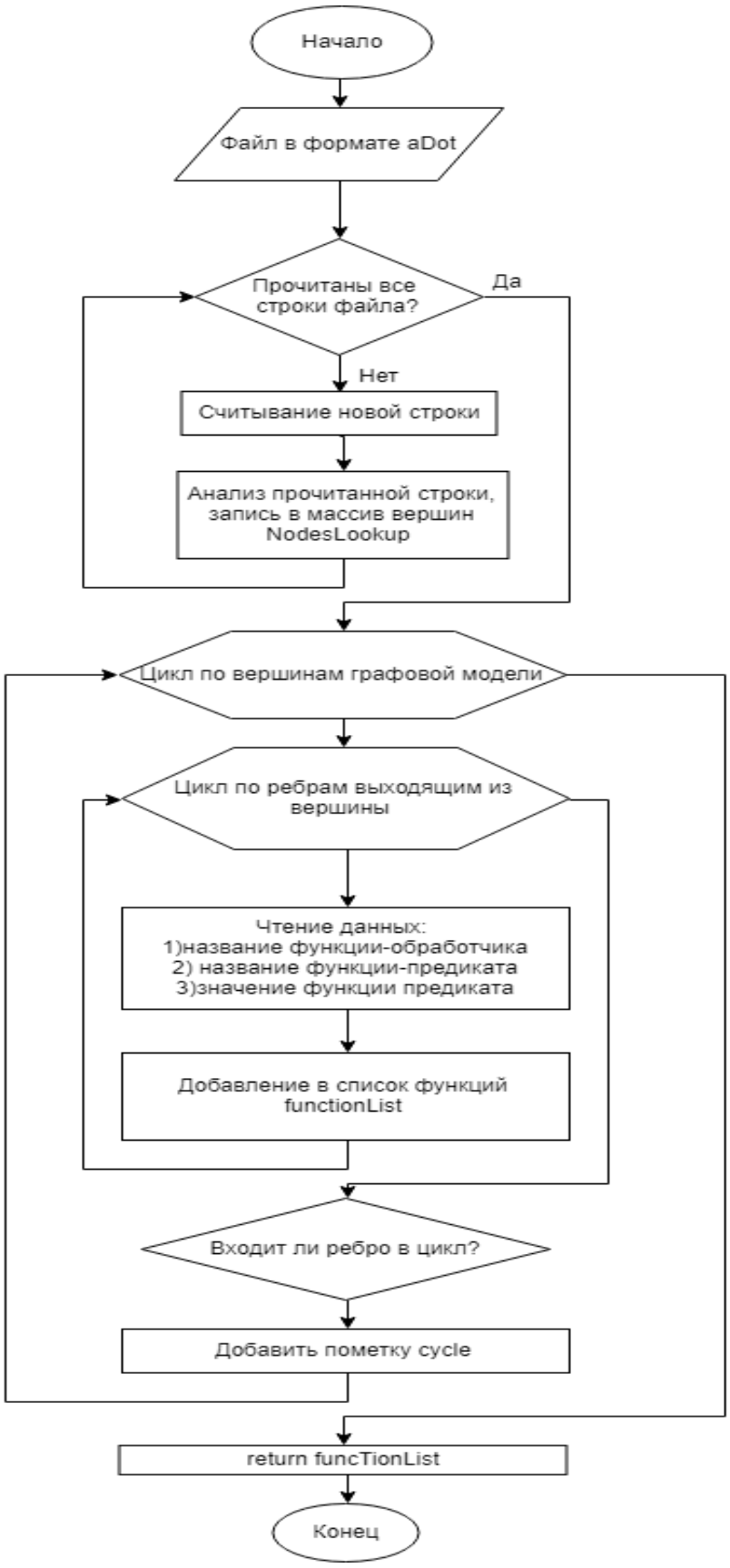

Рис. 13. Блок схема алгоритма составления модели очереди вызовов

Теперь, когда очередь вызовов сформирована, можно приступить к реализации преобразования *M2T*. Была разработана объектно–ориентированная

архитектура приложения (рисунок 14), генерирующего на основе очереди исходный код на языке *C++*. Генерируемый исходный код программы можно логически разделить на 5 составных частей: а) шапка (*header*) – содержит список подключаемых библиотек, прототипы функций и начало функции *main*; б) блок вызова функций поиска динамических библиотек (для *Linux* это функция *dlopen*, для *Windows* – функция *LoadLibrary*); в) блок поиска функций в библиотеке (*dlsym* для *Linux*, *GetProcAddress* для *Windows*); г) блок вызова этих функций; д) *footer* – завершение программы.

Ядром программной реализации является класс *programmGenerator*. В его параметризованном конструкторе на основании данных о конфигурации (операционная система, заданные *header* и *footer*) происходит обход очереди и для каждой функции графовой модели создаются 3 множества объектов соответствующих блокам поиска библиотек, поиска функций в библиотеке и вызова (*loader_set*, *getter_set* и *caller_set* соответственно). В классе присутствуют поля: названия программы, заголовки и строки окончания программы, представленные в виде шаблонного класса *STL string*. Также класс содержит объект класса Anymap, являющийся представлением файла aIni. Класс содержит методы выделения из очереди циклов и сериализации. Метод сериализации предназначен для формирования файла исходного кода на основе данных, содержащихся в полях класса.

Классы *loader*, *getter* и *caller* предназначены для генерации строк программы, соответствующих блокам загрузки библиотеки, поиска функции в библиотеке и вызова этих функций. Класс *cycle* генерирует код цикла, объединенных общим абстрактным классом *iter*. Каждый из классов *loader*, *getter*, *caller*, а также класс *cycle* переопределяет абстрактные методы класса *iter* serialize() и *get_comment*(), генерирующие строку исходного кода на основе полей класса и комментарий соответственно. Для унификации работы с пользовательскими классами был реализован шаблонный класс *collection*, обеспечивающий уникальность сериализованных строк. На рисунке 14 изображена *UML* диаграмма классов данной системы.

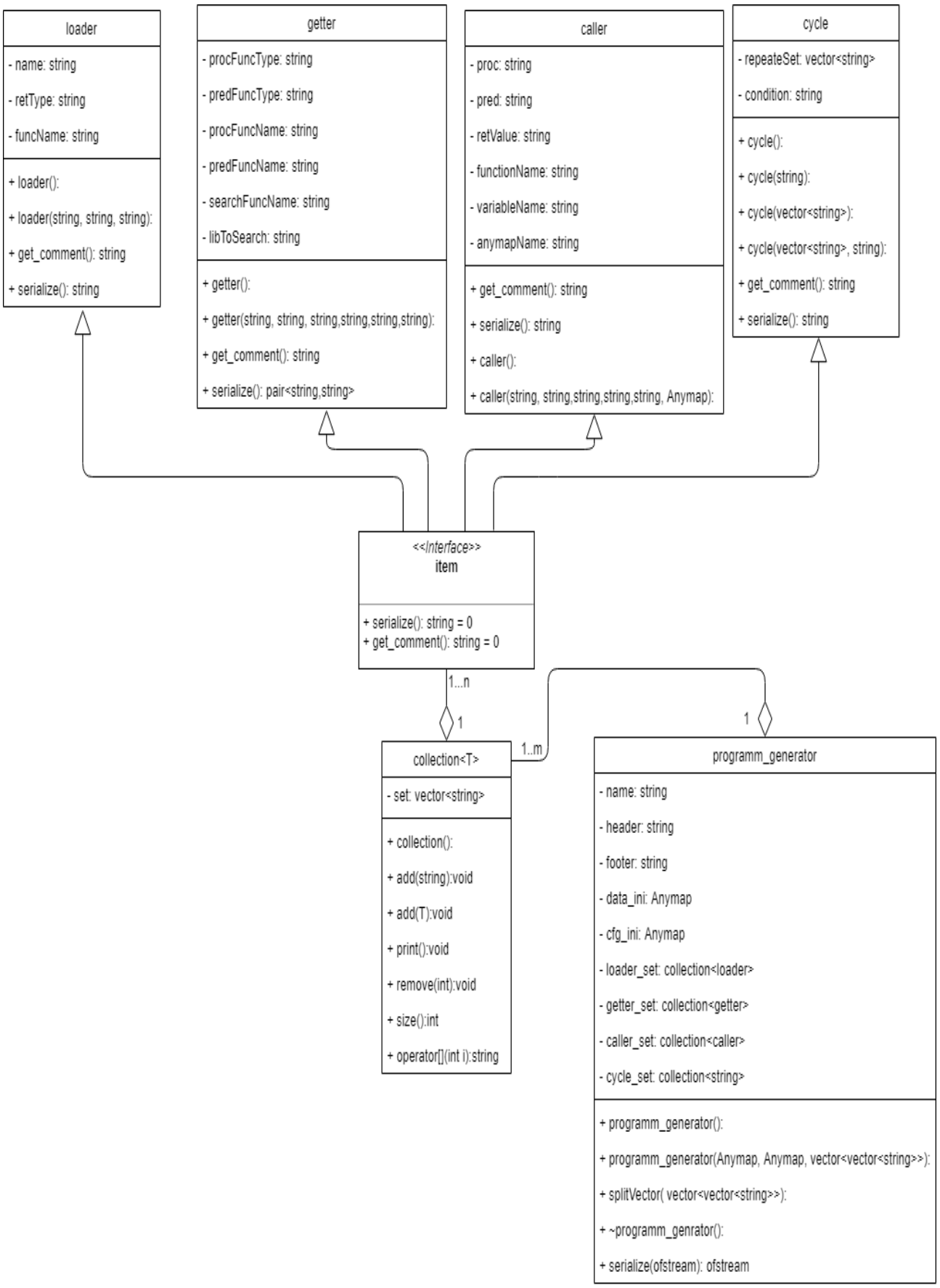

Рис. 14 Программная архитектура генератора исходного кода

Класс *program_generator* содержит следующие поля:

1) *name* – строка имени файла исходного кода

2) *header* – строка, содержащая прототипы функций-обработчиков и функций-предикатов, шаблонную функцию вызова обработчика с соответствующим предикатом, а также начало функции main и строки с подключением библиотек.

3) *footer* – строка окончания программы.

4) *data_ini* – файл входных параметров. Является входным параметром для вызываемых функций.

5) *cfg_ini* – конфигурационный файл, в котором можно задавать header и footer строки, а также выбирать версию системы.

6) *loader_set* - множество объектов, генерирующих строки с вызовом функций поиска динамических библиотек.

7) *getter_set* – множество объектов, генерирующих строки с вызовом функций поиска процедур в библиотеках

8) *caller_set* – множество объектов, генерирующих строки с вызовом функций графовой модели.

Методы класса *programm_generator*:

1) Конструктор с параметром, принимающий на вход конфигурационный файл, файл входных параметров и сформированную очередь вызовов. На основании очереди вызова заполняются множества блоков программы *loader_set*, *getter_set*, *caller_set*. Класс *collection* поддерживает работу со строками, поэтому имеется возможность добавления строк в множество, которые содержат несколько вызовов (используется для сохранения последовательности вызова циклических конструкций). На рисунке 15 представлена блок-схема заполнения множеств

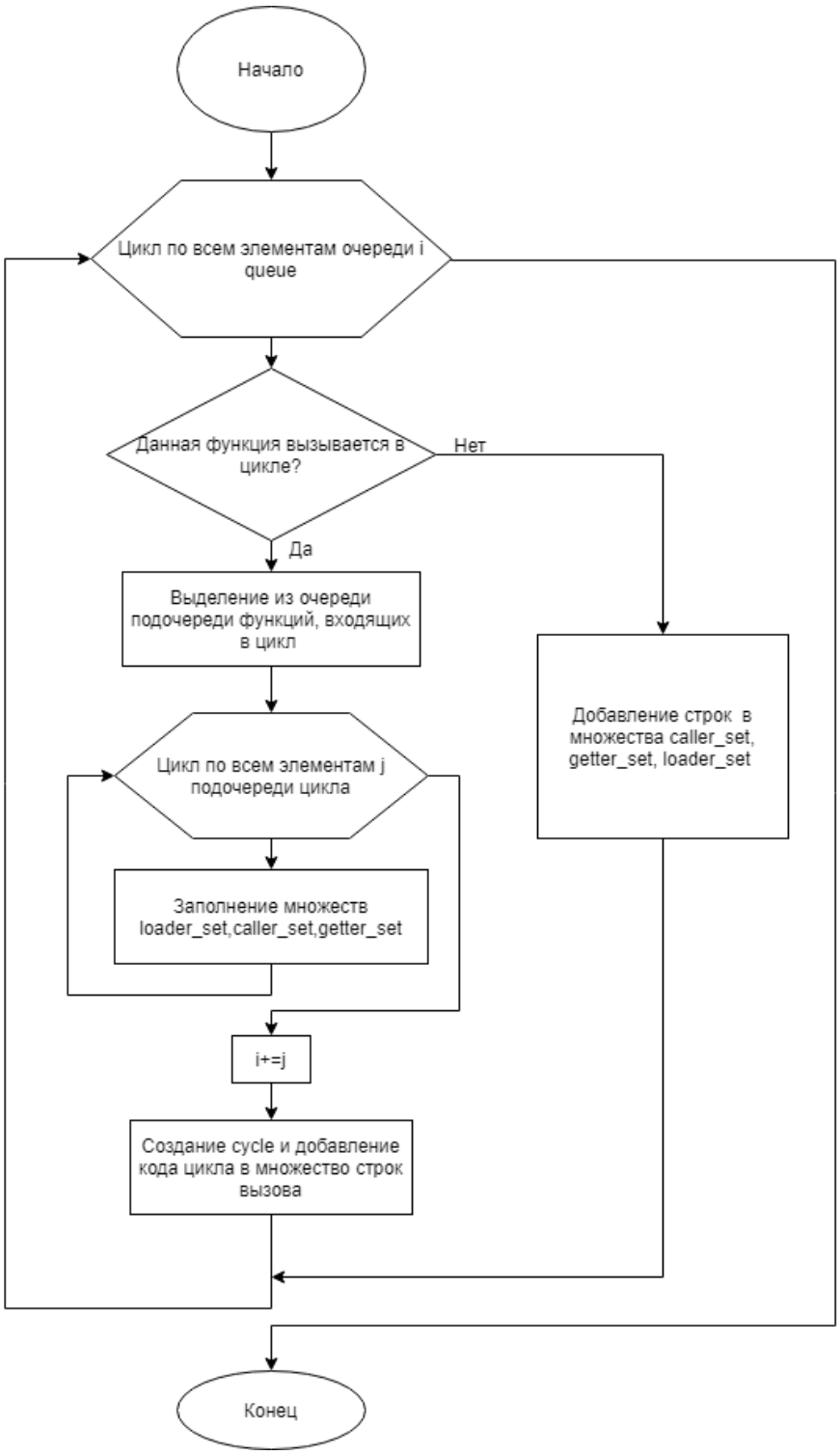

Рис. 15 Алгоритм заполнения множеств загрузки библиотек, поиска функций в библиотеках и вызова функций

2) *splitVector*(*vector<vector<string>> queue*) – метод разбивает очередь на 2 очереди, одна из которых является очередью вызовов для цикла. Метод возвращает структуру *pair<vector<vector<string>>, vector<string>>*. Первое из

возвращаемых значений пары является очередью цикла, вторая содержит те же элементы, что и исходная, за исключением добавленных в цикл.

Класс *loader* предназначен для генерации строки вызова функции поиска динамической библиотеки. Класс следующие поля:

1) *name* – имя искомой библиотеки;

2) *ret_type* – возвращаемое функцией загрузки динамической библиотеки значение;

3) *func* – название функции загрузки динамической библиотеки. Зависит от операционной системы;

А также методы:

1) *get_comment()* – метод, генерирующий комментарий для строки загрузки библиотеки;

2) *serialize()* – метод, возвращающий строку исходного кода;

Класс *getter* предназначен для генерации строки вызова функции поиска процедуры в библиотеке. Класс содержит следующие поля:

1) *procFuncType* – тип функции-обработчика;

2) *predFuncType* – тип функции-предиката;

3) *procFuncType* – название функции-обработчика;

4) *predFuncType* – название функции-предиката;

5) *searchFuncName* – имя функции поиска процедуры;

6) *libToSearch* – библиотека, в которой происходит поиск процедуры;

Также класс содержит методы:

1) *get_comment()* – генерирует комментарий для строки поиска процедуры;

2) *serialize()* – генерирует пару строк для поиска функции-предиката и функции-обработчика;

Класс *caller* предназначен для генерации строки вызова функции. Класс содержит поля:

1) *proc* – название функции-обработчика;

2) *pred* – название функции-предиката;

3) *retValue* – тип переменной;

4) *functionName* – имя шаблонной функции, в которой вызываются обработчики и предикаты;

5) *variableName* – имя переменной, в которую помещается возвращаемое функцией значение;

6) *anymapName* – имя *aini* файла входных параметров;

Также класс содержит методы:

1) *get_comment()* – генерирует комментарий для строки вызова функции;

2) *serialize()* – генерирует строку вызова функции;

Класс *cycle* содержит следующие поля:

1) *repeate_set* – множество строк вызова функций;

2) *cond* – условие выхода из цикла;

Также класс содержит методы:

1) *get_comment()* – генерирует комментарий к циклу;

2) *serialize()* – генерирует код цикла;

Класс является “оберткой” над контейнером *vector<string>* адаптированный для работы с объектами класса *loader*, *caller* и *getter* и поддерживает уникальность элементов в множестве. Класс содержит единственное поле:

1) *set* – поле типа *vector<string>*;

Методы класса *collection*:

1) *add(string)* – добавление строки в множество;

2) *add(T)* – реализует сериализацию объекта класса *T* и добавление этой строки в множество;

3) *remove(int)* – удаление элемента из множества;

4) *size()* – возвращает размер множества;

Также для удобства манипулирования элементами множества перегружен оператор [].

3. ТЕСТИРОВАНИЕ И ОТЛАДКА

3.1. ТЕСТИРОВАНИЕ ГЕНЕРАТОРА GUI

Для тестирования функции генерации форм web–клиента был создан файл входных параметров в формате *aIni*:

Листинг 4. Файл входных параметров *aIni*

```
1. [Author]//Идентификация автора разработки
2. AuthorName=[alsokolo]$sys.user
3. tbl=[GCDDB]$com.strep
4. *AuthorSid=sa//SID автора
5. -OutputFile=@AuthorSid@_@CodeObjectName@.res
6.
7. [Generator Parametrs]//Параметры генерации
8. CopyObjectToRep=[1]{0|1}//Перенести объект генерации в
репозиторий
9. RepPath=H:\gcdrep//Путь к репозиторию
10. TemplatesPath=H:\gcdrep\dev\res\Templates//Путь к каталогу с
шаблонами
11. TemporaryPath=H:\gcdrep\_tmp//Путь к каталогу временного
хранения
12.
13. [Object parameters]//Параметры генерируемого объекта
14. CodeObjectName=CodeGeneration//Наименование
объекта(varchar(25))
15. -Pressure=34[[MPa]]// Давление
16. Description=Прикладное использование программного обеспечения
генерации кода в процессе реализации НИР, ОКР и создании систем
инженерного анализа//*Описание
17. ParametersFile=[EVN_MSU_LEC1_02_10_2018.imp]//Имя файла
дополнительных параметров
18. TemplateSID=[EVN]$gen.tmpls//Тип генерируемого объекта из
19.
20. [Project data]//Идентификация объекта
21. ComplexSID=[rk6]$sys.cmplx//Идентификатор комплекса
22. SolutionSID=[met]$sys.solun//Идентификатор решения
23. ProjectSID=[rvs]$sys.prjct//Идентификатор проекта
24.
25. // Список названий используемых материалов
26. [Material Names]
27. MaterialSIDs={1,2,3}// Идентификаторы материалов в соответствие
с геометрией v.2.2
28. a$name=EGLASS// Наименование материала №1
29. a$anisotropy=[auNO]{auNO|auLCS|auSegmented}// Метод учета
анизотропии
30. a$LCS=((1;0;0);(0;1;0);(0;0;1))// Определение ЛСК для материала
с индексом 1
31. a$anisotropy_data=@file@.ani// Информация по анизотропии
32. a$strength_criterion=SQR
33. a$name=EGLASS// Наименование материала #2
```

```
34. a$anisotropy=[auNO]{auNO|auLCS|auSegmented}// Метод учета
анизотропии
35. a$LCS=((1;0;0);(0;1;0);(0;0;1))// Определение ЛСК для материала
с индексом 2
36. a$anisotropy_data=@file@.ani// Информация по анизотропии
37. a$strength_criterion=SQR
38. arr = (1;2;3; ‘axdxa’)
39. diap = [1;0:3;1]
```

В результате были сформированы 5 секций *GUI*, содержащих различные формы для ввода параметров со значениями из файла. На рисунках 16–19 представлены изображения сгенерированных форм.

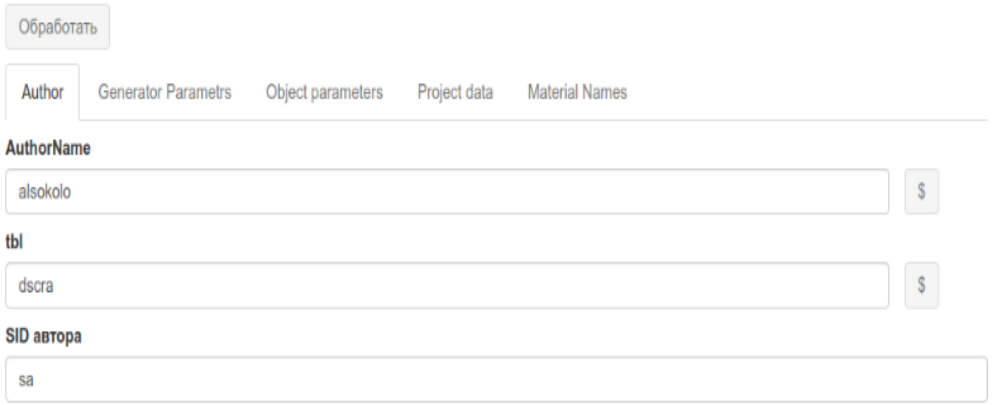

Рис. 16 Секция Author

Первые две записи представляют собой ссылку на таблицу базы данных. Пользователю предоставлена возможность выбрать первичный ключ таблицы. При нажатии на кнопку “$” справа от формы ввода сформируется модальное окно с таблицей. Чтобы строка была классифицирована как сслыка на таблицу базы данных, значение переменной в ней должно иметь следующий формат:

[<первичный ключ>]$<база данных>.<схема>.<id таблицы>

Также в этой форме не отображается скрытый параметр *OutputFile* (начинающийся атрибутом “-”).

Инструменты решения задач (решатели)

Имя таблицы: GCDDB.com.slvrs

Количество столбцов: 11; Количество строк: 68

	slver Идентификатор инструмента решения задачи (решателя)	dscra Описание	mdlid Идентификатор модели	cmsid Идентификатор вычислительного метода	ntmdl Идентификатор сетевой модели	preci Требуемая точность	itlim Лимит по итерациям	dimid Идентифик размерност
☐	ACKLEY_PSO_OPT	Решатель задачи поиска минимума функции Экли	ELASTIC	PSO	PSO_MODEL	1e-07	2500	3D
☐	ANN_MLP	Анализ на основе модели многослойного персептрона	NN_MLP	ANN	MLP_MODEL			1D
☐	ANN_MLP_TRAIN	Инструмент обучения	NN_MLP	ANN	MLP_TRAIN			1D

Рис. 17 Сформированное модальное окно таблицы при нажатии на кнопку “$”

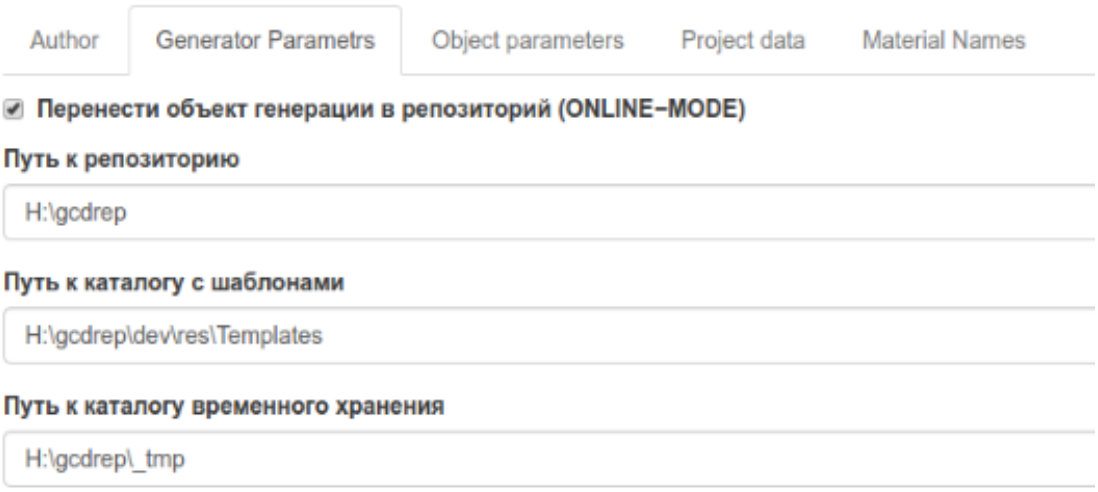

Рис. 18 Секция *Generator Parametrs*. Представлены строчные пути к файлам и каталогам, что может применяться для задания относительного пути файла *aDot*, содержащего графовую модель, а также переменная типа бинарное множество

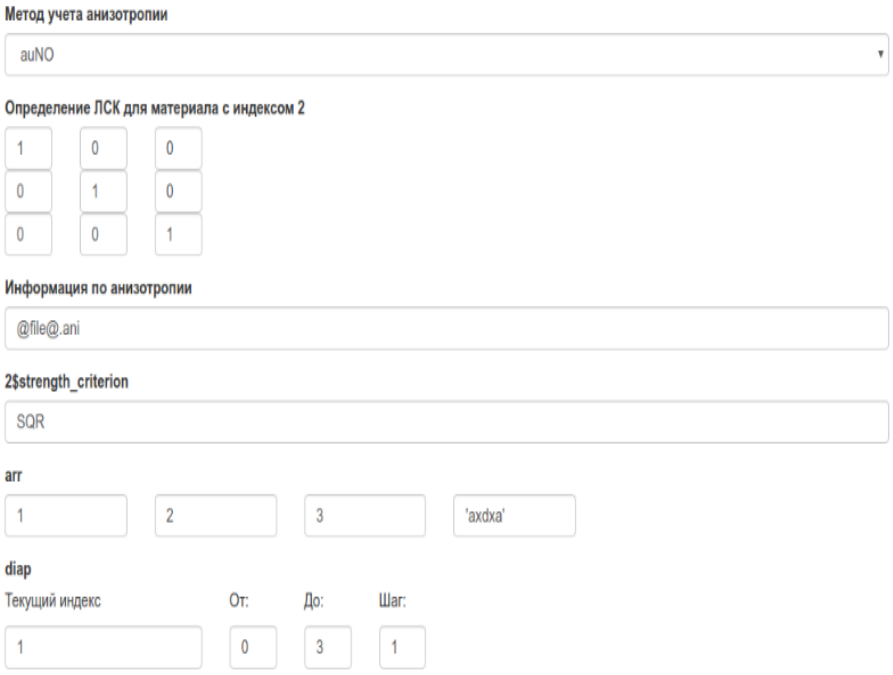

Рис. 19 Сгенерированная форма для ввода парамтеров типа множество выбора, двумерного массива, строк, а также одномерного массива и диапазонов.

3.2. ТЕСТИРОВАНИЕ ГЕНЕРАТОРА ИСХОДНОГО КОДА

На рисунке 20 изображена графовая модель, выбранная для тестирования генератора кода.

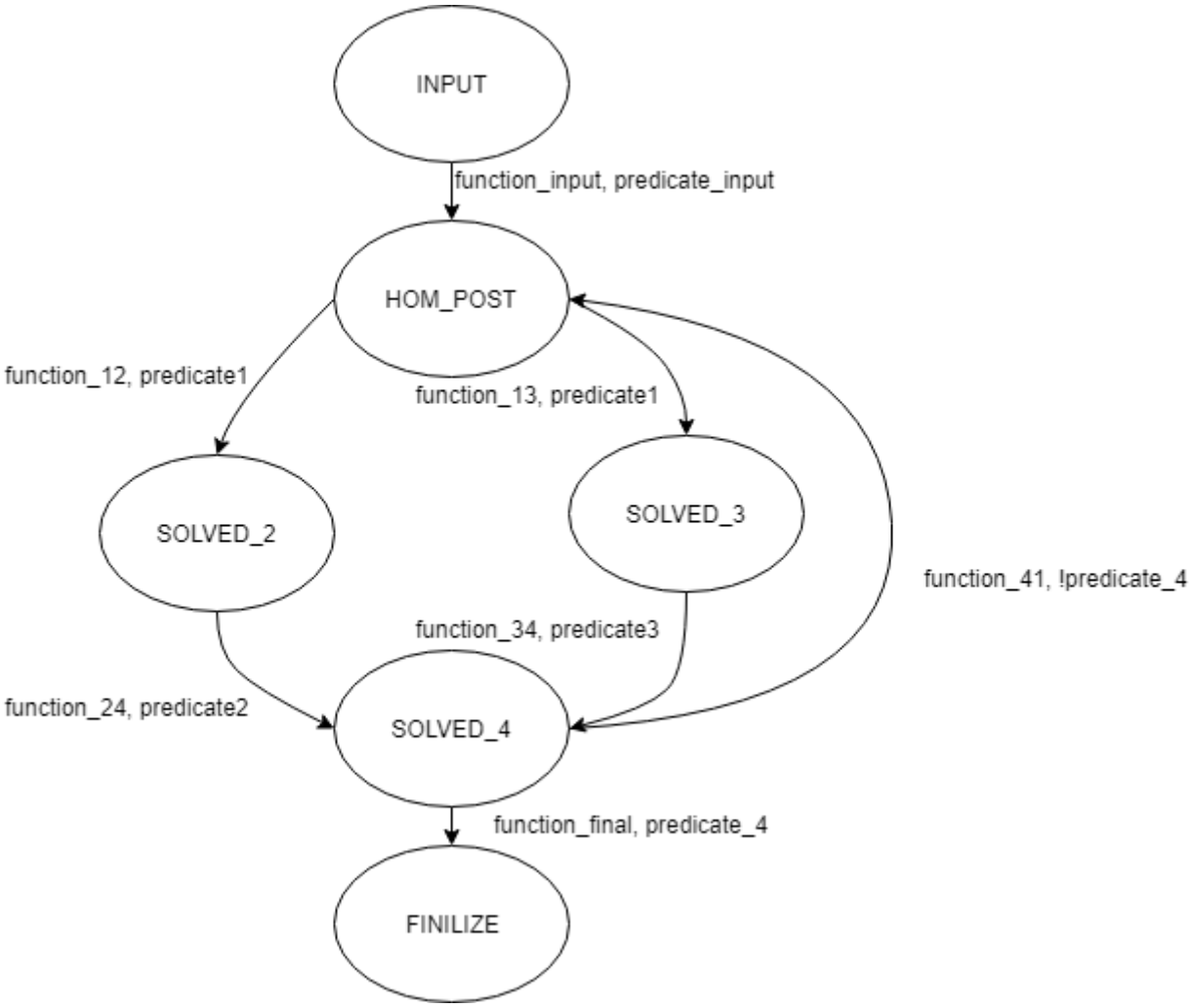

Рис 20. Графовая модель

Такой графовой модели соответствует текстовое описание представленное в листинге 5.

Листинг 5. Описание графовой модели в файле формата *aDot*

```
1. digraph JUGR_GraphModel
2. {
3. INPUT [predicate=INP]
4. HOM_POST [predicate=CHECK_1]
5. SOLVED_2 [predicate=CHECK_2]
6. SOLVED_3 [predicate=CHECK_3]
7. SOLVED_4 [predicate=CHECK_4]
8. INP [type=boolean, binname=jugr, entry_func=predicate_input]
9. CHECK_1 [type=boolean, binname=jugr, entry_func=predicate_1]
10. CHECK_2 [type=boolean, binname=jugr, entry_func=predicate_2]
11. CHECK_3 [type=boolean, binname=jugr, entry_func=predicate_3]
12. CHECK_4 [type=boolean, binname=jugr, entry_func=predicate_4]
```

```
13. FUNC_INP [binnamen=jugr, entry_func=function_input]
14. FUNC_12 [binname=jugr, entry_func=function_12]
15. FUNC_13 [binname=jugr, entry_func=function_13]
16. FUNC_24 [binname=jugr, entry_func=function_24]
17. FUNC_34 [binname=jugr, entry_func=function_34]
18. FUNC_41 [binname=jugr, entry_func=function_41]
19. FINALIZE [binname=jugr, entry_func=function_final]
20. __BEGIN__ -> INPUT
21. INPUT -> HOM_POST [on_predicate_value=true, edge=FUNC_INP]
22. HOM_POST -> SOLVED_2 [on_predicate_value=true, edge=FUNC_12]
23. HOM_POST -> SOLVED_3 [on_predicate_value=true, edge=FUNC_13]
24. SOLVED_2 -> SOLVED_4 [on_predicate_value=true, edge=FUNC_24]
25. SOLVED_3 -> SOLVED_4 [on_predicate_value=true, edge=FUNC_34]
26. SOLVED_4 -> HOM_POST [on_predicate_value=false,edge=FUNC_41]
27. SOLVED_4 -> FINILEZED[on_predicate_value=true,edge=FINALIZE]
28. FINILEZED -> __END__
29. }
```

Структура проекта приложения выглядит следующим образом:

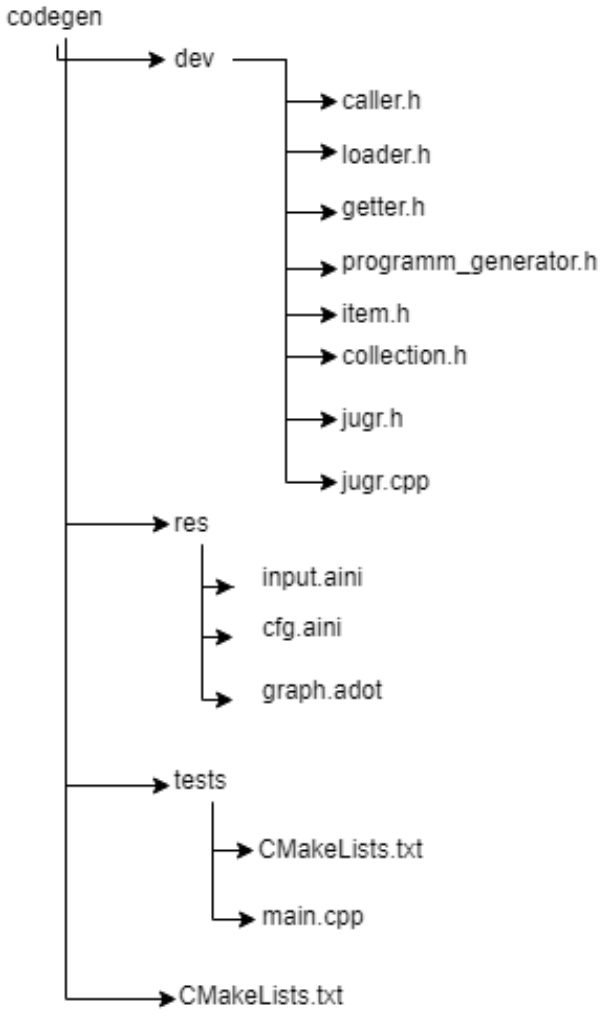

Рис. 21 Структура проекта генератора исходного кода

Был подготовлен bash сценарий для автоматизации сборки проекта.

Листинг 6. Сценарий сборки проекта

```
1. #!/bin/bash
2. rm -r build
3. rm -r rls
4. mkdir build
5. cd build
6. cmake ../ -DCOMSDK_INST=/home/semyon/installed_comsdk;
7. make
```

Для тестирования системы необходимо перейти в директорию *tests*, скомпилировать файл *main.cpp* и запустить получившийся файл. В результате сгенерируется файл исходного кода программы на языке *C++*. Листинг 7 содержит исходный код сформированного файла для данной графовой модели.

Листинг 7. Исходный код сгенерированного файла *tst.cpp*

```
1. #include <anymap.h>
2. typedef int processorFuncType(AnyMap&);
3. typedef bool predicateFuncType(const AnyMap&);
4. template<processorFuncType* tf,predicateFuncType* tp>
5. int F(AnyMap& p_m)
6. {
7. return (tp(p_m))?tf(p_m):tp(p_m);
8. }
9. int main(){
10. Anymap input("input.txt")
11. //Загрузка библиотеки lib_name
12. HMODULE lib_lib_name=LoadLibrary(L"lib_name");
13. //Поиск функции обработчика function_input и функции-предиката predicate_input в
библиотеке lib_name
14. processorFuncType *proc_function_input=(processorFuncType*)GetProcAddress(lib_name,"f
unction_input");
15. predicateFuncType *pred_predicate_input=(predicateFuncType *)GetProcAddress(lib_name,
"predicate_input");
16. //Поиск функции обработчика function_13 и функции-предиката predicate_1 в библиотеке
lib_name
17. processorFuncType *proc_function_13=(processorFuncType *)GetProcAddress(lib_name,"fun
ction_13");
18. predicateFuncType *pred_predicate_1=(predicateFuncType*)GetProcAddress(lib_name,
"predicate_1");
19. //Поиск функции обработчика function_24 и функции-предиката predicate_2 в библиотеке
lib_name
20. processorFuncType *proc_function_24=(processorFuncType*)GetProcAddress(lib_name,"func
tion_24");
21. predicateFuncType *pred_predicate_2=(predicateFuncType *)GetProcAddress(lib_name,
"predicate_2");
22. //Поиск функции обработчика function_34 и функции-предиката predicate_3 в библиотеке
lib_name
23. processorFuncType *proc_function_34=(processorFuncType *)GetProcAddress(lib_name,"fun
ction_34");
24. predicateFuncType *pred_predicate_3=(predicateFuncType*)GetProcAddress(lib_name,
"predicate_3");
25. //Поиск функции обработчика function_41 и функции-предиката predicate_4 в библиотеке
lib_name
```

```
26. processorFuncType *proc_function_41=(processorFuncType*)GetProcAddress(lib_name,"func
tion_41");
27. predicateFuncType *pred_predicate_4=(predicateFuncType*)GetProcAddress(lib_name,
"predicate_4");
28. //Поиск функции обработчика function_final и функции-предиката predicate_4 в библиотеке
lib_name
29. processorFuncType *proc_function_final=(processorFuncType*)GetProcAddress(lib_name,"f
unction_final");
30. //Вызов функции-обработчика function_input с предикатом predicate_input
31. auto res=F<proc_function_input,pred_predicate_input>(input);if (res!=0) return res;
32. do{
33. //Вызов функции-обработчика function_13 с предикатом predicate_1
34. auto res = F<proc_function_13, pred_predicate_1>(input);if (res!=0) return res;
35. //Вызов функции-обработчика function_24 с предикатом predicate_2
36. auto res = F<proc_function_24, pred_predicate_2>(input);if (res!=0) return res;
37. //Вызов функции-обработчика function_34 с предикатом predicate_3
38. auto res = F<proc_function_34, pred_predicate_3>(input);if (res!=0) return res;
39. //Вызов функции-обработчика function_41 с предикатом predicate_4
40. auto res = F<proc_function_41, pred_predicate_4>(input);if (res!=0) return res;
41. } while(!predicate_4);
42. //Вызов функции-обработчика function_final с предикатом predicate_4
43. auto res = F<proc_function_final, pred_predicate_4>(input);if (res!=0) return res;
44. return 0;
45. }
```

Реализовано модульное тестирование классов системы. Листинги тестовых функций описаны в приложении. Результаты тестирования представлены на рисунке 22.

```
Тест 1: Проверка строки загрузки библиотек
//Загрузка библиотеки default_lib
HMODULE lib_default_lib=LoadLibrary(L"default_lib");
Тест 2: Проверка строк поиска функций в библиотеках
//Вызов функции-обработчика default_processor с предикатом default_predicate
auto res = F<proc_default_processor, pred_default_predicate>(default_anymap);if (res!=0) return res;
Тест 3: Проверка строки запуска функции
//Поиск функции обработчика def_proc_name и функции-предиката def_pred_name в библиотеке default_lib
default_processor *proc_def_proc_name = (default_processor *)GetProcAddress(default_lib,"def_proc_name");
default_predicate *pred_def_pred_name = (default_predicate *)GetProcAddress(default_lib, "def_pred_name");
Тест 4: Проверка работы коллекции с loader строками
В коллекцию добавляется строка полученная явным вызовом serialize:
HMODULE lib_default_lib=LoadLibrary(L"default_lib");
В коллекцию добавляется строка полученная loader'ом:
HMODULE lib_default_lib=LoadLibrary(L"default_lib");
Из коллекции удаляется строка:

В коллекцию добавляется строка полученная loader'ом:
//Загрузка библиотеки default_lib HMODULE lib_default_lib=LoadLibrary(L"default_lib");
Тест 5: Цикл
        do {
        a += 1
} while( a < 3 );
```

Рисунок 22. Тестирование классов генератора исходного кода

Тесты дали корректный результат, сгенерированный исходный код пригоден для использования.

ЗАКЛЮЧЕНИЕ

В результате выполнения выпускной квалификационной работы был проведен тщательный анализ подходов к построению программного обеспечения генерации кода, на основании которого разработана программная архитектура генератора исходных кодов графоориентированных решателей. Составленная *UML* диаграмма классов стала основой для реализации приложения генерации исходного кода. Реализованный генератор позволил сформировать циклический код решателя по графоориентированной модели.

В библиотеку *comsdk* добавлены следующие функциональные возможности: а) холостой обход графа; нахождение циклов в графе и формирование очереди вызовов функций-предикатов и фунуций-обработчкиков. Было проведено тестирование разработанных классов и проверена корректность сформированной программы. Также были добавлены новые функциональные возможности *web*-клиента *comwpc*, обеспечившие возможность подготавливать файлы входных параметров и конфигурации для генератора.

Добавлена поддержка следующих типов данных: а) строки; б) целые числа; в) вещественные числа; г) ссылки на таблицы базы данных; д) диапазоны; е) массивы; ж) функциональные зависимости.

В дальнейшем планируется развитие данной разработки. В частности, одной из приоритетных задач будет являться интеграция генератора в Распределенную вычислительную систему РВС *GCD*.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1) Д.В. Жевнерчук, А.С. Захаров Семантическое моделирование генераторов программного кода распределенных автоматизированных систем// Информатика и управление в технических и социальных системах. - 2018. - №1. — С. 23-30.

2) J. Klein, H. Levinson, J. Marchetti Model-Driven Engineering: Automatic Code Generation and Beyond // Carnegie Mellon University Software Engineering Institute. — 2015. — №1. — С. 1-51.

3) Nikiforova, O. Comparison of BrainTool to Other UML Modeling and Model Transformation Tools / O. Nikiforova, K. Gusarov // Applied Computer Systems. — 2013. — №-1 — С. 31-42.

4) V.Y. Rosales-Morales, G. Alor-Hernández, J.L. García-Alcaráz An analysis of tools for automatic software development and automatic code generation.//Revista Facultad de Ingenieria.. — 2015. — №77. — С. 75-87.

5) L. Lúcio, M. Amrani, J. Dingel Model transformation intents and their properties // Springer-Verlag Berlin Heidelberg. — 2014. — №3. — С. 647- 684.

6) J. Mattingley, S. Boyd CVXGEN: a code generator for embedded convex optimization / // Optim Eng. — 2011. — №13. — С. 1-27.

7) Ю.В. Нестеров Методы выпуклой оптимизации / Ю.В. Нестеров; под науч. ред. “Nesterov-final”. — Москва : МЦНМО, 2010. — 281 с.

8) Verdoolaege S., Carlos Juega J. Polyhedral Parallel Code Generation for CUDA// ACM Transactions on Architecture and Code Optimization. — 2013. — №9. — С. 54-77.

9) Востокин, С.В., А.Р. Хайрутдинов Программный комплекс параллельного программирования Graphplus Templet// ACM Transactions on Architecture and Code Optimization. — 2011. — №4. — С. 146-153.

10) L.M. Rose, N. Matragkas A Feature Model for Model-To-Text Transformation Languages// MiSE '12 Proceedings of the 4th International

11. Workshop on Modeling in Software Engineering / Switzerland; Zurich: IEEE Press Piscataway, 2012. — С. 57-63.

12. Э.Н. Самохвалов, Г.И. Ревунков, Ю.Е. Гапанюк Генерация исходного кода программного обеспечения на основе многоуровнего набора правил// Вестник МГТУ им. Н.Э. Баумана. — 2014. — №5. — С. 77-87.

13. А.Е. Александров, В.П. Шильманов Инструментальные средства разработки И сопровождения программного обеспечения на основе генерации кода// БИЗНЕС-ИНФОРМАТИКА. — 2012. — №4. — С. 10-17.

14. S. Jörges, B. Steffen Building Code Generators with Genesys: A Tutorial Introduction// Springer-Verlag Berlin Heidelberg. — 2011. — №6491. — С. 364-385.

15. L. Burgueno, B. Steffen Testing M2M/M2T/T2M Transformations// Springer Science+Business Media. - 2011. — С. 203-219.

16. S. Taktak, J. Feki Model-Driven Approach to Handle Evolutions of OLAP Requirements and Data Source Model // Springer International Publishing AG. — 2019. — №880. — С. 401-425.

17. S. Taktak, J. Feki Higher-Order Rewriting of Model-to-Text Templates for Integrating Domain-specific Modeling Languages // MODELSWARD 2013. — 2013. — №-. С. 1-13

ПРИЛОЖЕНИЕ А

Листинг 8. Модульный тест класса loader

```
1. void loaderDefaultTest(){
2.     loader load_default;
3.     auto comment = load_default.get_comment();
4.     auto str = load_default.serialize();
5.     cout<<"Тест 1: Проверка строки загрузки библиотек"<<endl;
6.     cout<<comment<<endl<<str<<endl;
7. }
```

Листинг 9. Модульный тест класса getter

```
1. void getterDefaultTest(){
2.     getter getter_default;
3.     auto comment = getter_default.get_comment();
4.     auto str = getter_default.serialize();
5.     cout<<"Тест 3: Проверка строки запуска функции"<<endl;
6.     cout<<comment<<endl<<str.first<<endl<<str.second<<endl;
7. }
```

Листинг 10. Модульный тест класса caller

```
1. void callerDefaultTest(){
2.     caller caller_default;
3.     auto comment = caller_default.get_comment();
4.     auto str = caller_default.serialize();
5.     cout<<"Тест 2: Проверка строк поиска функций в библиотеках"<<endl;
6.     cout<<comment<<endl<<str<<endl;
7. }
```

Листинг 11. Модульный тест класса collection

```
1. void collectionTest() {
2.     cout<<"Тест 4: Проверка работы коллекции с loader строками\n";
3.     loader load;
4.     collection<loader> coll;
5.     cout<<"В коллекцию добавляется строка полученная явным вызовом
serialize:\n";
6.     coll.add(load.serialize());
7.     coll.print();
8.     cout<<"В коллекцию добавляется строка полученная loader'ом:\n";
9.     coll.add(load);
10.     coll.print();
11.     cout<<"В коллекции удаляется строка:\n";
12.     coll.remove(0);
13.     coll.print();
14.     cout<<”В коллекцию добавляется строка полученная loader'ом:\n";
15.     coll.add(load);
16.     coll.print();
17. }
```

Листинг 12. Модульный тест класса cycle

```
1. void cycleTest(){
2. std::cout<<"Тест 5: Цикл"<<std::endl;
3. cycle cyc1({ "a += 1" }, " a < 3 ");
4. std::cout<<cyc1.serialize()<<std::endl;
5. }
```

ПРИЛОЖЕНИЕ Б
Графические материалы

Графическая часть выпускной квалификационной работы (5 листов формата А4) состоит из:

1) Модельные преобразования и их роль в генераторах исходного кода (лист 1).

2) Архитектура *web*-клиента РВС *GCD* (лист 2).

3) *UML* диаграмма классов генератора исходного кода графоориентированных решателей на основе преобразования *M2T* (лист 3).

4) Блок-схема алгоритма формирования очереди вызовов (лист 4).

5) Графовая модель, используемая для тестирования программного обеспечения.(лист 5)