МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Левченко Кирилла Константиновича

Тема работы:

ОБЕСПЕЧЕНИЕ СПЕЦИАЛИЗИРОВАННОЙ ПОДДЕРЖКИ КОНСТРУИРОВАНИЯ ФРАГМЕНТИРОВАННЫХ ПРОГРАММ ДЛЯ КЛАССА ЧИСЛЕННЫХ АЛГОРИТМОВ В СИСТЕМЕ LUNA

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э./……..
(ФИО) / (подпись)
«31» мая 2022 г.

Руководитель ВКР
д.т.н., профессор
заведующий каф. ПВ ФИТ НГУ
Малышкин В.Э/……...
(ФИО) / (подпись)
«31» мая 2022 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Перепёлкин В.А. /……..
(ФИО) / (подпись)
«31» май 2022г.

Новосибирск, 2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий
Кафедра параллельных вычислений
Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ
Зав. кафедрой Малышкин В.Э.
(фамилия, И., О.)
…….
(подпись)
«20» января 2022г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА
Студент*у(ке)* Левченко Кирилл Константинович, группы 18209

(фамилия, имя, отчество, номер группы)

Тема: обеспечение специализированной поддержки конструирования фрагментированных программ для класса численных алгоритмов в системе LuNA
(полное название темы выпускной квалификационной работы)
утверждена распоряжением проректора по учебной работе от 20 января 2022 № 0012

Срок сдачи студентом готовой работы 20 мая 2022г.

Исходные данные (или цель работы): разработать модуль, обеспечивающий вариативную сборку LuNA-программы из специализированных решений/системных алгоритмов.

Структурные части работы:

Работа состоит из введения, обзора существующих решений, теоретической части, практической части и заключения.

Руководитель ВКР
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э./……..
(ФИ О) / (подпись)
«20» января 2022г.

Задание принял к исполнению
Левченко К.К./……..
(ФИО студента) / (подпись)
«20» января 2022г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ,
Перепёлкин В.А./……..
(ФИ О) / (подпись)
«20» января 2022г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Проблема эффективной параллельной реализации численных алгоритмов на суперкомпьютерах остаётся актуальной с момента появления суперкомпьютеров. Разработка больших численных моделей для суперкомпьютеров является сложной задачей, так как требует компетенций в области системного параллельного программирования для обеспечения корректной работы взаимодействующих процессов, эффективного распределения ресурсов системы для выполнения программы. Для преодоления данных трудностей прибегают к средствам автоматического конструирования параллельных программ. Одним из таких средств является поддержка идеи фрагментированного программирования — система LuNA.

LuNA (Language for Numerical Algorithms)[1] – система автоматического конструирования параллельных программ, разрабатываемая в ИВМиМГ СО РАН. Язык LuNA создан для реализации задач численного моделирования.

Ввиду фундаментальных причин[2] система LuNA не может построить оптимальную программу с помощью общих алгоритмов для всех задач численного моделирования, что будет приводит к плохой производительности на реальных приложениях. Для преодоления данных трудностей следует использовать частные решения для конкретного класса задач, специализированные алгоритмы которых позволяют улучшить нефункциональные характеристики исполнение программы.

На текущий момент система LuNA имеет как общие алгоритмы, так и несколько частных для конкретного класса задач, например система, решающая задачу “методом частиц в ячейках»[3]. В перспективе система LuNA и другие системы автоматического конструирования параллельных программ будут дополняться новыми частными решениями, целесообразно

интегрировать множество таких решений в общую систему. Для этого система должна предоставлять удобный программный интерфейс инъекции программных модулей частной системы, с помощью специального модуля, который будем называть модуль сборки LuNA-программ. Модуль сборки должен поддерживать включение множества частных решений, позволять манипулировать сборкой и исполнением LuNA-программы. А также должен иметь возможность развития для поддержки более широкого класса задач по эксплуатации LuNA-программы: профилирование, тестирование, сборка, исполнение и так далее.

Выполнение данной работы подразумевает под собой: 1) Проектирование интерфейса и реализация модуля удовлетворяющего ему, который предоставляет всю необходимую инфраструктуру для интеграции частных решений. 2) Реализация частных решений и интеграция с помощью спроектированного интерфейса. В рамках описанной работы реализована первая часть.

Цель работы – разработать модуль, обеспечивающий вариативную сборку LuNA-программы из специализированных решений/системных алгоритмов.

Для достижения этой цели были поставлены следующие задачи:

1. Выполнить комплексный анализ существующих частных и родственных решений, выявить их потребности для встраивания в общую систему.
2. Спроектировать программный интерфейс модуля сборки LuNA-программы.
3. Реализовать программный модуль сборки LuNA-программы.
4. Произвести тестирование модуля сборки LuNA-программ.

Практическая ценность работы состоит в том, что модуль сборки позволит получить единый интерфейс для решения широкого класса задач по эксплуатации LuNA-программы.

Научная новизна работы состоит в проектировании нового программного интерфейса для решения задач: сборки и исполнения, и последующей эксплуатации LuNA-программы, учитывающий особенности интегрируемых модулей и удовлетворяющий поставленным требованиям.

Структура и объем работы. Работа состоит из введения, обзора существующих решений, теоретической части, практической части и заключения. В обзоре будут приведены основные существующие на данных момент решения и их краткая характеристика. В теоретической части основные термины определения и понятия предметной области. В практической части будут представлены результаты разработки и тестирования.

2 Анализ предметной области

В данном разделе будут обзорно рассмотрены решения и идеи создания конвейеры сборки программ или других расширяемых и дополняемых систем компиляции.

2.1 Compilation pipeline

Конвейер компиляции для низкоуровневых программ на языке ассемблер. Согласно Игорю Жиркову[4] конвейер стоит на три основных этапа: препроцессинг, компиляция, связывание. Для примера рассмотрим два исходных файла first.asm и second.asm, каждый из них обрабатывается отдельно перед этапом компиляции(проиллюстрировано на рисунке 1)

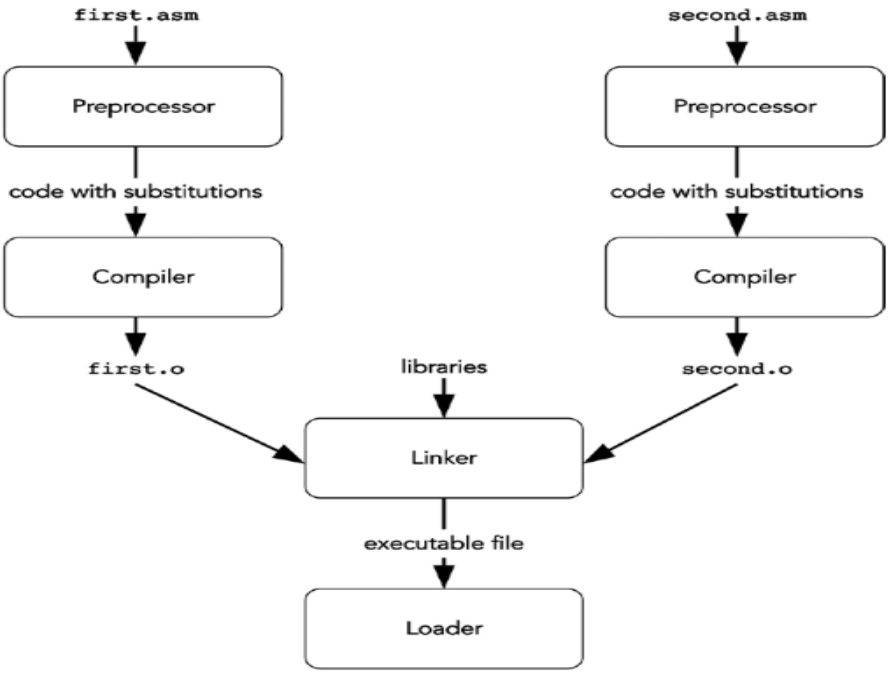

Рисунок 1 — конвейер компиляции [4]

Препроцессор преобразует исходный код программы для получения другой программы на том же языке. Преобразования представляют собой подстановки одной строки вместо других.

Компилятор преобразует каждый исходный файл в файл с закодированным машинными инструкциями. Однако такой файл ещё не готов к выполнению, поскольку ему не хватает связей с другими отдельно скомпилированным файлами. Речь идет о случаях, когда инструкции обращаются к данными или инструкциям, которые объявлены в других файлах. Например для first.asm - second.asm.

Компоновщик устанавливает соединения между файлами и создает исполняемый файл. После этого программа готова к запуску. Компоновщики работают с объектными файлами, типичными формата, которых являются ELF.

Compilation Pipeline для сборки использует разбиение на этапы, каждый из которых имеет входные/выходные данные, а также свой функциональный вклад в сборку программы, за счет этого поддерживается разделение ответственности между этапами сборки. Недостатком Compilation Pipeline является отсутствие поддержки вариативности сборки программы, что является ключевым для реализации модуля.

2.2 Конвейер компиляции OCAML

OCaml (Objective Caml) — объектно-ориентированных язык программирования общего назначения[5]. Был разработан с учётом безопасности исполнения и надежности программ. Поддерживает функциональную, императивную и объектно-ориентированную парадигмы программирования.

Конвейер OCaml[6] представляет из себя серию этапов, где каждый этап выполняет уникальную работу, независимо от других этапов конвейера. К примеру, один этап проверяет безопасность типов, используя знание

системы типов OCaml, а последний этап компилятора выводит ассемблерный код, который ничего не знает об объектах OCaml. Процесс компиляции проиллюстрирован на рисунке 2.

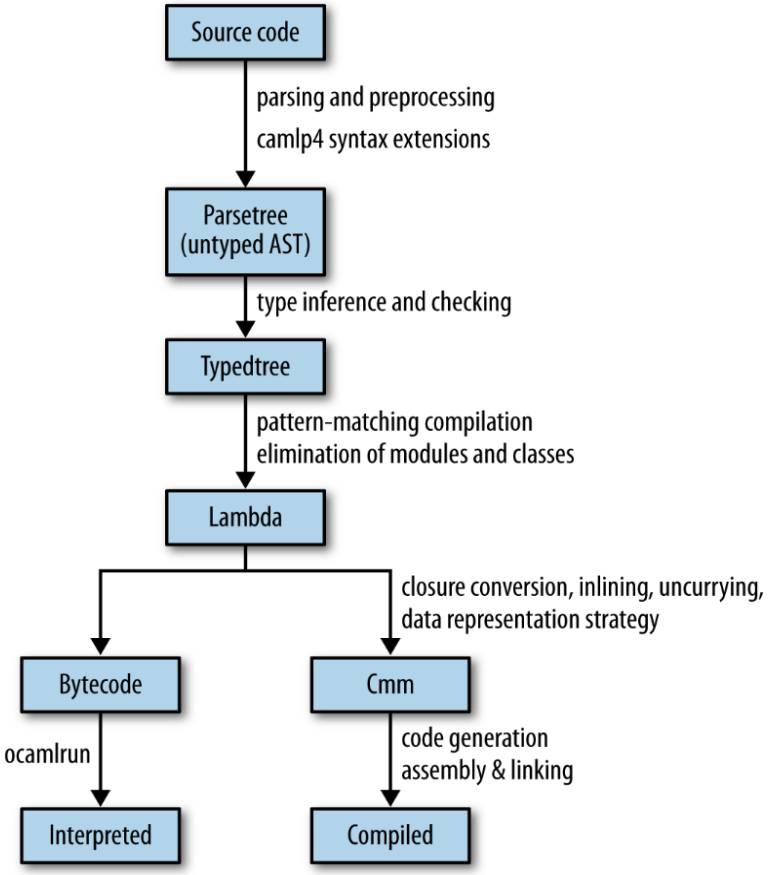

Рисунок 2 — Конвейер компиляции OCaml [6]

Интерес представляет ветвление на два абсолютно разных типа исполнения программы. Компилятор OCaml после создания лямбда-формы конвейер компилятора OCaml разветвляется на два отдельных компилятора: один для байт-кода, а другой — для машинного кода.

Преимущество *байт-кода* в том, что он прост, переносим и легко компилируется, а недостатком является то, что он медленнее выполняется и требует среды выполнения OCaml. Байт-код, сгенерированный компилятором, запускается средой выполнения OCaml, которая включает в себя интерпретатор, средства сборки мусора и некоторые функции C, реализующие низкоуровневые системные вызовы.

Компиляция в *бинарник* обычно используется для производственных сценариев использования.

Процесс компиляции медленнее, но полученный двоичный файл оптимизирован и обеспечивает более высокую производительность.

Исходя из обзора конвейера компиляции OCaml можно выделить, то что конвейер декомпозирован на этапы и у каждого этапа присутствует своя ответственность, а также имеется вариативность сборки.

Однако конвейер OCaml не гибок для интегрирования новых этапов. Отсутствует операции отката, во время возникновения ошибки на одном из этапов, тем самым конвейер рассматривается как единая сильносвязанная система, где каждый шаг сборки программ даже потенциально не имеет никаких альтернатив.

2.3 CI/CD-конвейер

CI/CD[7](continuous integration/continuous delivery(deployment)) — одна из практик DevOps[8], подразумевающая непрерывную интеграцию и доставку. Этот набор принципов предназначен для повышения удобства, частоты и надежности развертывания изменений программного обеспечения или продукта. CI/CD позволяет разработчикам уделять внимание реализации бизнес-требований, качеству кода и безопасности продукта.

Конвейер CI/CD — это автоматизированный, повторяющийся метод разработки, доставки и развертывания, применяемый на протяжении всего жизненного цикла приложения — от решения о его создании до выведения из эксплуатации. Для реализации конвейера CI/CD используется Jenkins[7], либо аналогичный продукт.

На рисунке 3 продемонстрирован пример CI/CD-конвейера.

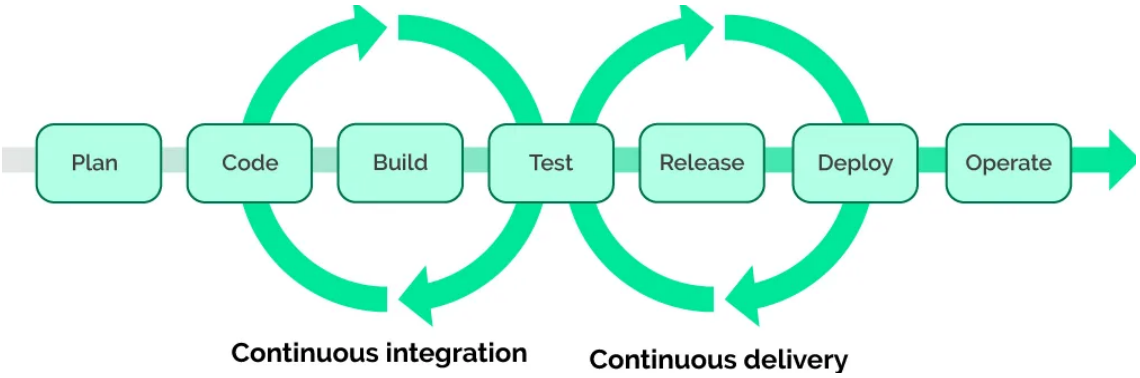

Рисунок 3 — CI/CD-конвейер [7]

Этапы CI/CD

Написание кода.

Каждый из разработчиков пишет код своего модуля, проводит ручное тестирование, а затем соединяет результат работы с текущей версией проекта в основной ветке.

Для контроля версий используется система Git, либо аналогичные решения. Когда участники команды опубликуют код своих модулей в основной ветке, начнется следующий этап.

Сборка.

Система контроля версий запускает автоматическую сборку и тестирование проекта. Триггеры для начала сборки настраиваются командой индивидуально — фиксация изменений в основной ветке проекта, сборка по расписанию, по запросу и т.д.

Ручное тестирование.

Когда CI система успешно проверила работоспособность тестовой версии, то код отправляется тестировщикам для ручного обследования. При этом тестовая сборка получает номер кандидата для дальнейшего релиза продукта (например, v.1.0.0-1).

Релиз.

По итогам ручного тестирования сборка получает исправления, а итоговый номер версии кандидата повышается (например, после первого исправления версия v.1.0.0-1 становится v.1.0.0-2). После этого выпускается версия кода для клиента (например, v.1.0.0) и начинается следующий этап цикла.

Развертывание.

На этом этапе рабочая версия продукта для клиентов автоматически публикуется на production серверах разработчика. После этого клиент может взаимодействовать с программой и ознакомиться с ее функционалом как непосредственно через готовый интерфейс, так и через облачные сервисы.

Поддержка и мониторинг.

Конечные пользователи начинают работать с продуктом. Команда разработки поддерживает его и анализирует пользовательский опыт.

Планирование.

На основе пользовательского опыта формируются запросы на новый функционал для продукта, готовится план доработок. После этого цикл замыкается и переходит в начальную стадию — написание кода. Далее начинается новая итерация CI/CD разработки.

Практика CI/CD имеет имеет очень полезные идеи для сборки и эксплуатации LuNA-программы. Сборку программы можно разбить на этапы таким образом, чтобы помимо локального запуска, выполнялись заранее описанные автоматические тесты и далее по статусу их прохождения выполнялся запуск на удалённом суперкомпьютере, с последующим анализом профиля программы. Но конвейеры CI/CD, обычно, не вариативны и не имеют альтернативных этапов.

2.4 Утилита MAKE

make[9]— утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Утилита использует специальные make-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла make определяет и запускает необходимые программы.

Утилита make работает по правилам, записанным в специальном конфигурационном файле. Правила определяют цели, зависимости между целями и набор команд для выполнения каждой цели.

Цели могут соответствовать определенным файлам. Кроме того, цели могут не соответствовать ни одному файлу и использоваться для группировки других целей или определенной последовательности команд. Такие цели называются phony targets. На рисунке 4 приведен пример целей в makefile’е.

```
target : source1 source2 ... sourceN
        command
        command
        ...
```

- Example:

```
myprogram : file1.c file2.c file3.c
        gcc -o myprogram file1.c file2.c file3.c
```

Рисунок 4 — пример цели в makefile’е

Каждая цель может зависеть от выполнения других целей. Выполнение цели требует предварительного выполнения других целей, от которых она зависит.

В случае зависимости между целями, соответствующими файлам, цель выполняется только в том случае, если файлы, от которых она зависит, новее, чем файл, соответствующий цели. Это позволяет перегенерировать только файлы, зависящие от измененных файлов, и не выполнять потенциально долгий процесс пересборки всех файлов проекта.

Таким образом, makefile определяет граф зависимостей, по которому утилита make выполняет ту или иную цель, по возможности минимизируя количество операций сборки.

Таким образом для реализации вариативности сборки, тестирования, анализа программы и прочих программ, возможно описать несколько целей. Но интерфейс утилиты не поддерживает более сложную логику выбора между вариантами этапов.

2.5 Выводы по результатам обзора

Исходя из результатов обзора, можно сделать вывод, что с нет подходящего модели, системы или готового интерфейса для удобного включения системы специализированных решений в систему LuNA, но в перечисленных решениях есть полезные идеи, которые нужно учесть при реализации собственного модуля.

3 Основные понятия

Для дальнейшего описания работы необходимо определиться с терминологией и ввести основные понятия.

3.1 Фрагментированное программирование и система luna

3.1.1 Фрагментированная программа

Фрагментированная программа - рекурсивное множество триплетов вида <*in, out, mod*>**.** Множество триплетов можно представить в виде потенциально бесконечного графа задач, где между разными фрагментами вычислений(определён далее) присутствует информационная зависимость. Как только последний фрагмент вычислений заканчивает свое выполнение, фрагментированная программа на языке LuNA считается завершенной.

Фрагментированная структура алгоритма сохраняется и во время его выполнения, то есть фрагменты данных (определён далее) и вычислений явно присутствуют на узлах мультикомпьютера как независимые и взаимодействующие процессы. Исполнение алгоритма автоматически осуществляет система. Она размещает фрагменты данных в распределенной памяти мультикомпьютера, отслеживает информационные зависимости между фрагментами вычислений и исполняет последние. Исполнение фрагмента вычислений состоит в том, что ему назначается вычислений узел, в локальную память этого узла передаются все входные фрагменты данных для этого фрагмента вычислений, запускается последовательная процедура над этими фрагментами данных.

Такая схема выполнения прикладного алгоритма позволяет системе распределять и перераспределять фрагменты данных и вычислений по узлам мультикомпьютера и выбирать порядок их выполнения. Это позволяет осуществлять автоматически балансировку нагрузки на вычислительные

узлы, управление памятью, планирование вычислений, настройку программы на конфигурацию вычислителя.

Во время исполнения программы фрагменты вычислений рекурсивно разворачиваются во множества атомарных фрагментов. Фрагменты отображаются на реальные ресурсы — фрагменты вычислений на потоки исполнения, фрагменты данных распределяются по памяти процессов.

Фрагменты данных могут передаваться между вычислительными узлами компьютера, если необходимого фрагменту вычислений входного фрагмента данных нет в памяти процесса, в рамках которого выполняется данный фрагмент вычислений. Также, нужно отметить, что при реализации программы в системе LuNA фрагменты вычислений могут разделяться на стадии — помимо стадии исполнения, при наличии у него входных фрагментов данных добавляются стадии запросов входных фрагментов данных и их ожидания.

3.1.2 Фрагмент вычислений

Фрагмент вычислений — независимая единица программы содержит описание входных/выходных фрагментов данных и кода фрагмента. Каждый фрагмент кода является процедурой без побочных эффектов, аналог “чистой функции”.

Фрагменты вычислений делятся на два основных типа:

- Атомарные фрагменты, представляющие собой функции на языке C++. Работой атомарного фрагмента вычислений будем считать выполнение соответствующей ему пользовательской функции, работой структурированного - создание дочерних фрагментов вычислений.
- Структурированные фрагменты, порождаемые операторами языка LuNA, например: циклы “for” и “while” - множество фрагментов

вычислений, “sub” подпрограмма - потенциально бесконечное множество фрагментов вычислений.

3.1.3 Фрагмент данных

Фрагмент данных — блоки данных (агрегаты переменных), которые являются входными/выходными данными фрагментов вычислений.

В ходе исполнения фрагментированной программы фрагменты данных производятся одним фрагментом вычислений и передаются на вход другому, таким образом образуются информационные зависимости между фрагментами вычислений. Такое представление позволяет представить фрагментированный алгоритм в виде двудольного ориентированного графа.

Фрагменты данных могут передаваться между параллельными процессами, если необходимого фрагменту вычислений входного фрагмента данных нет в памяти процесса, в рамках которого выполняется данный фрагмент вычислений. На рисунке 5 приведён пример фрагментированного алгоритма.

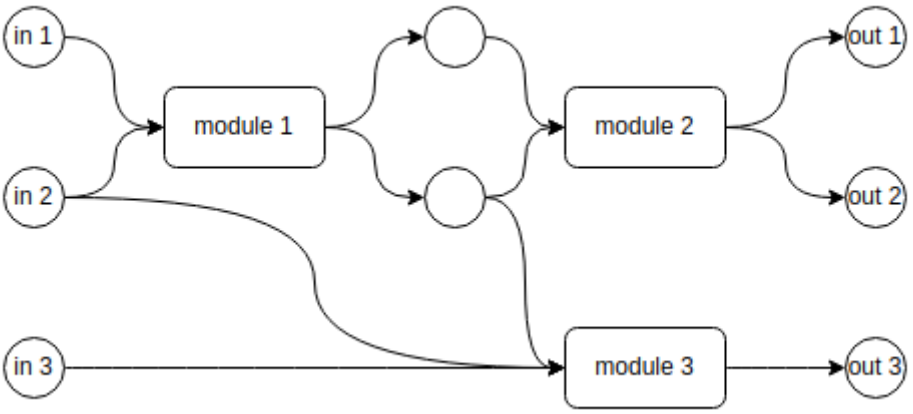

Рисунок 5 — пример фрагментированного алгоритма

3.1.4 Система рекомендаций

Система рекомендация — это расширение языка, которое было введено, для того чтобы облегчить алгоритмическую сложность некоторых задач, в частности-переборных. Рекомендации являются частью фрагментированной программы и используются как подсказки для решения оптимизационных задач, например с помощью рекомендации *locator_cyclic*, определяется расположения фрагмента данных, тем самым можно уменьшить накладные расходы по доставке фрагмента данных. Рекомендации не влияют на функциональную часть работы алгоритма.

Система LuNA состоит из двух основных компонент — это компилятор и исполнительная система. Компилятор выполняет статический анализ программы и генерирует ряд рекомендаций, далее исполнительная фрагментированная программа передается исполнительной системе.

3.1.5 Исполняемое представление luna-программы

Исполняемое представление LuNA-программы в последней своей реализации основывается на мультиагентной системе [2], где каждый вычислительный процесс (фрагмент вычислений) — это агент.

Агента управляются программой, куда заложены статические решения и коллективное принятие динамических решений.

3.2 Этапы работы с luna-программой

1. Пользователь описывает фрагментированный алгоритм
2. Пользователь добавляет рекомендации
3. Компилятор формирует исполняемое представление, в которое вкладывает статические решение о способе исполнения

4. Исполнительная (RTS) система окончательно определяет и реализует поведение программы.
5. Пользователь осуществляет анализ исполнения.

3.3 Процесс сборки luna-программы

3.3.1 Препроцессинг

Препроцессинг — один из этапов компиляции программы, задача которого подготовить исходный код программы для следующего этапа компиляции. О данных на выходе препроцессора говорят, что они находятся в *препроцессированой* форме, пригодной для обработки последующими программами (лексер, парсер, компилятор и так далее). Результат и вид обработки зависят от вида препроцессора; так, некоторые препроцессоры могут только выполнить простую текстовую подстановку, другие способны по возможностям сравниться с языками программирования.

Перед тем как исходный код LuNA-программы поступит на вход парсеру, программа проходит стадию *препроцессинга*, в результате которой осуществляются макро-подстановки, убираются комментарии и выделяются внешние блоки. Препроцессор формирует препроцессированую программу, сохраняет позиционную информацию о тексте исходной программы и выносит внешние блоки с позиционной информацией.

3.3.2 Парсер

Абстрактное синтаксическое дерево – представление программы, доступное в момент компиляции для чтения и модификации. На верхнем уровне, дерево состоит из списка процедур и подпрограмм, объявленных в программе.

Синтаксический анализ — стадия компиляции, при котором анализатор преобразует человеко-ориентированный текст программы в

машинно-ориентированный. Исходный текст программы разбивается на токены и из не формируется абстрактное синтаксическое дерево.

Исходный текст программы LuNA подвергается изначально лексическому анализу, где с помощью регулярных выражений осуществляется поиск шаблонов символов. Лексический анализатор создаётся с помощью программы *flex* [10] , на вход которой подаётся шаблоны из регулярных выражений. В результате работы лексического анализатора исходный текст разбивается на распознанные группы - *лексемы* - с целью получения на выходе идентифицированных последовательностей, называемых *токенами*

Задача грамматического анализа — найти взаимосвязь между входными токенами. Распространенной структурой, выражающей эту связь, является дерево, такое как дерево синтаксиса с правилами приоритета. Чтобы написать синтаксический анализатор, необходим определенный метод для описания правил, используемых синтаксическим анализатором для преобразования серии токенов в числа синтаксического анализа. Для этого используется контекстно-свободная грамматика стандартный формат написания этой грамматики — *грамматика BFN*. Для описания грамматики используется язык *bison* [11]**.**

3.3.3 Компиляция

В результате компиляции LuNA-программы анализируется абстрактное синтаксическое дерево и принимается решение относительно нефункционального поведения программы, осуществляется это с помощью системы рекомендаций. Компилятор, анализируя информационные зависимости, принимает решения по поводу порядка и места выполнения фрагментов вычислений, место хранения и миграции фрагментов данных, а также их удаления.

Так в процессе компиляции происходит частичный переход от декларативного представления программы к императивному с помощью генерации программ агентов. Программа агента генерируется на C++ в виде набора функций. Состояние агента хранится в сериализованном объекте, а вспомогательный функционал вынесен в исполнительное окружение.

3.4 Метод частиц в ячейках

Модуль сборки LuNA-программы призван добавить поддержку специализированных решений для задач математического моделирования. Одним из таких классов задач являются задачи, решаемые методом “*частиц-в-ячейках”*. Далее будет представлен краткий обзор данного метода.

При изучении астрофизических процессов, в частности динамики галактик и протопланетных дисков большую роль играет численное моделирование вращения газопылевого облака. В качестве способа моделирования подобных процессов ниже обзорно будет рассмотрен метод “частиц-в-ячейках”. Метод “частиц-в-ячейках” характеризуется тем, что эволюция системы частиц на каждом временном шаге разбивается на два этапа. На одном из них при фиксированном положении частиц предварительно вычисляется их взаимодействие и результат их коллективного воздействия на среду. Расчёт ведётся на неподвижной (**“***эйлеровой***”**) сетке. На **“***лагранжевом***”** этапе выполняется интегрирование на очередной временной шаг динамической системы, правая часть которого вычислена на **“***эйлеровом***”** этапе.

При моделировании астрофизических систем, состоящих из большого количества объектов, эффекты столкновений оказываются пренебрежительно малыми. Частицы используемые в моделировании, представляют собой, скорее перенос средней массовой плотности, чем орбиты отдельных объектов.

В пространстве распределены модельные частицы, которые движутся под действием сил гравитации. Для каждой модельной частицы заданы пространственные координаты и скорость. Массы всех частиц одинаковы.

Для метода “частиц-в-ячейках” характерны большие объемы вычислительной работы. Вместе с тем метод частиц в ячейках реализуется на ЭВМ с параллельной архитектурой.

Существует несколько способов параллельной реализации данной задачи:

- Дублирование сетки по всем процессорам вычислительной системы и распределение частиц между ними. Недостатком является плохая масштабируемость.
- Пространственная декомпозиция — разрезание частной области на несколько подобластей по числу процессоров и распределение соответствующим образом сеток и частиц. В процессе счёта большая часть частиц может собраться одной подобласти, в результате соответствующий процессор окажется перегруженным, тогда как остальные будут простаивать.
- Область моделирования разрезается на непересекающиеся подобласти, каждая из которых обрабатывается группой из одного или нескольких процессоров. Соответствующие подобласти часть сетки дублируется а частицы равномерно распределяются между всеми процессорами группы. Алгоритм также хорошо масштабируется.

В данном случае выбран последний способ.

3.5 LUNA-ICLU

В качестве реализации специализированного решения для задач метода “частиц-в-ячейках” был выбран компилятор *LuNA-ICLU [12]*.

LuNA-ICLU призван преодолеть проблемы, влияющие на производительность системы LuNA связанные использованием универсальных системных алгоритмов фрагментированного выполнения программы. Идея системы *LuNA-ICLU* заключается в применении системных алгоритмов, которые могут автоматически генерировать статическую программу MPI[13] из строго определенного класса фрагментированных программ. Таким образом, разработчику прикладной программы не нужно решать такие системные задачи параллельного программирования, как разработка алгоритмов динамической балансировки нагрузки.

Чтобы сгенерировать статическую программу MPI из данной фрагментированной программы, необходимо проанализировать информационные зависимости между фрагментами вычислений, описанными во входной фрагментированной программе. Выражения языка LuNA используют фрагменты вычислений и фрагменты данных, в том числе индексированные, которые являются частями фрагментированных массивов. Выражения индекса могут быть сложными и трудными для анализа. Чтобы преодолеть эту проблему, определен ограниченный класс программ с фрагментированным вводом. Фрагментированная программа может содержать одномерные или двумерные фрагментированные массивы данных и итерационные процессы, описываемые с помощью оператора *while*. Значения фрагмента данных на текущей итерации вычисляются из набора значений фрагмента данных из одной или нескольких предыдущих итераций. Размеры массивов фрагментов данных строго разделены на временные, по которым проходят итерации, и пространственные.

В рамках итерации каждый элемент массива фрагментов данных может быть вычислен фрагмента вычислений из элементов массивов фрагмента данных с одинаковыми индексами пространственной размерности.

Такой класс алгоритмов достаточно прост для анализа компилятором и содержит решения многих прикладных задач.

Чтобы преодолеть проблемы статического анализа фрагментированных программ, язык LuNA был расширен новыми синтаксическими конструкциями:

- ☐ Оператор DFArray структуру и размеры массива DF(Фрагмента данных).
- ☐ Пространственные и временные размерности теперь чётко разделены. Пространственное измерение обозначается символами “[” и “]” и определяет набор DF, который соответствует одной и той же итерации. Временное измерение обозначается “(“ и “)”.
- ☐ Зависимости данных между элементами массива DF на разных итерациях цикла while явно указываются в заголовке цикла с использованием таких выражений, как <A(i-1), A(i) → A(i+1)>
- ☐ Оператор dynamic отмечает набор фрагментов вычислений в теле итерации, который может вызвать дисбаланс нагрузки.

Шаги для сборки *LuNA-ICLU* программ содержит следующие этапы:

- ☐ *Preprocessor*
- ☐ Parser
- ☐ Compiler
- ☐ MpiCompiler
- ☐ RunnerIclu

4 Предлагаемое решение модуля сборки

В данном разделе будет изложены требования к интерфейсу, основные идеи абстракций и алгоритмы, использующиеся при проектировании модуля сборки и поддержки специализированных и частных решений для системы LuNA.

4.1 Требования, предъявляемые к решению

Предлагаемое решение, должно удовлетворять следующим требованиям:

1. Типичная интегрируемая специализированная система включает в себя несколько шагов по сборке и исполнению. Описываемые шаги сборки должны быть слабо связанными, и позволять интегрировать новые шаги специализированных систем без изменений уже включенных шагов модуля.
2. Предлагаемое решение должно удовлетворять требованию вариативности, модуль должен обеспечивать поддержку нескольких специализированных решений.
3. Решение должно агрегировать все шаги сборки конкретного частного решения. При описании решений, пользователь должен иметь возможность четко отделять и группировать шаги его сборки и исполнения.
4. Возможность манипулировать принятием решения о выборе и исполняемых шагов конкретной частной системы. Манипулирование принятием решения может приниматься как при помощи знаний пользователя, так и автоматически.
5. Модуль должен закладывать возможность исполнения одних и тех же шагов сборки на разных окружениях. Решение в перспективе должно охватывать более широкий класс задач по эксплуатации программ,

одной из таких задач является исполнение программы на удаленной машине, именно поэтому оно должно поддерживать исполнение исполнение на разных окружениях.

4.2 Требования, предъявляемые к интерфейсу

Программный интерфейс модуля поддержки специализированных решений должен соответствовать следующим требованиям:

- ☐ Инкапсуляция. Каждый включаемый программный модуль, реализующий этап сборки, должен быть абстрагирован от реализации общего конвейера.
- ☐ Изолированность. Внедрение нового программного модуля не должно повлиять на реализацию остальных. Реализация каждого отдельного этапа не влияет на реализацию других.
- ☐ Определенность — описание общего дерева сборки производится централизованно. Во избежании конфликтов, циклов и неопределённости между ветками сборки децентрализованное описание исключается.
- ☐ Полнота — программный интерфейс представляемый конвейером сборки должен включать в себя поддержку стандартных операций над ветками(слияние, откат, ответвление), которые позволяли бы гибко манипулировать процессом сборки LuNA-программы. Предъявляемое требование несёт в себе рекомендательный характер.

4.3 Необходимые термины модуля сборки

Для дальнейшего описания работы необходимо ввести несколько терминов:

- ☐ Этап — отдельный шаг сборки LuNA-программы. Каждый этап представляет из себя команду для исполнения, а также требования входных и выходных данных. Например отдельным этапом компиляции

является препроцессинг. Шаги исполнения: выполнение скрипта pp.py, требование на входные данные: файл с ненулевым размером input.fa, требование на выходные данные: файлы с ненулевыми размерами preprocessed.fa, preprocessed.fa.ti, blocks.ti.

- Ветка — последовательность этапов, объединённая мета-информацией о реализуемом частном решении. При реализации модуля поддержки специализированных решений каждое решение также может состоять из нескольких этапов иметь свой препроцессор, парсер, интеллектуальный компилятор и исполнительную систему. Таким образом каждое решение будет иметь свою цепочку сборки отличающуюся от основной.
- Дерево сборки — набор этапов организованная в виде дерева с учетом отношений между ветками и последовательности этапов в каждой ветке.

4.4 Модуль сборки

4.4.1 Общее описание

Модуль сборки LuNA-программы призван предоставить интерфейс для включения частных решений в общих конвейер сборки программы. Во время сборки LuNA-программы модуль автоматически выбирает следующий этап сборки программы основываясь на алгоритме обхода дерева сборки, статистики выполнения предыдущих процессов сборки.

4.4.2 Этап сборки

Сборка каждого специализированного решения состоит из последовательности исполнения отдельных программных модулей, целесообразно инкапсулировать каждый шаг последовательности в абстракцию модуля сборки, называемую этапом сборки.

Каждый этап модуля представляет из себя атомарный и независимый шаг сборки, который декларирует входные зависимости и результат работы этапа.

Этап сборки включает в себя:

1. Команда исполнения этапа, описывается в виде императивной команды для исполнения отдельно компилируемого или интерпретируемого модуля, включающегося в процесс сборки частного решения.
2. Зависимости этапа, описываются в виде уникального имени или идентификатора. Далее во время исполнения этапа уникальный идентификатор используется для поиска нужной зависимости в заранее заданном окружении(локальная/удаленная файловая система, структуры в памяти программы).

   В случае отсутствия нужной зависимости попытка исполнения этапа завершается с ошибкой.
3. Результаты выполнения этапа, так же задаётся уникальным идентификатором используется для поиска нужной сущности (файл, элемент структуры в памяти программы, переменная окружения и т.д.). Таким образом проверяется необходимая валидация результата исполнения этапа.

Разбиение шагов компиляции LuNA-программы на этапы позволяет описывать отдельные шаги компиляции независимо от других. Вся реализация инкапсулирована внутри конкретного этапа, объявление зависимостей позволяет сделать дерево сборки слабо связанным, так как этапы взаимозаменяемы, что удовлетворяет первому требованию предлагаемого решения.

4.4.3 Исполнитель

Этапы сборки обрабатываются специальным объектом — исполнителем. Исполнитель отвечает за выбор следующего этапа сборки, проверку его зависимостей, исполнение и проверку результата.

Исполнитель имеет состояние:

1. Текущий этап — этап на которым производятся все текущии операции.
2. Журнал исполнения — в ходе исполнения этапов, каждая стадия(проверка зависимости, исполнение, проверка результата) его исполнения журналируется с помощью специального объекта, который реализует контракт: запись об ошибке, запись информации о исполнении этапа. С помощью этого объекта пользователь будет получать актуальную информацию о сборке программы.
3. Состояние исполнителя в будущем может расширяться в зависимости от потребностей. К примеру, исполнитель может переиспользовать опыт предыдущих сборок программ для принятия решения о следующем этапе сборки.

Введение объекта исполнителя позволяет снять ответственность за выполнение, поиск зависимостей, подстановку окружения и сбора статистики с этапов. Объекты этапов сборки имеют только декларативный характер. С помощью данной сущности модуль позволяет удовлетворить требованиям вариативности, возможности манипулирования принятием решения о выборе шагов специализированной системы, а также удовлетворяет требованию о исполнении на разных окружениях.

4.4.4 Окружение

Окружение — объект использующийся для взаимодействия со средой исполнения этапа(локальная / удаленная машина). С помощью окружения

исполнитель выполняет команду этапа в нужной среде исполнения. Возможные окружения исполнения задаются статически. Также некоторые сущности окружения конфигурируемы. Конфигурации содержат все необходимые параметры для использования данного окружения.

Пример: окружение удалённого вычислительного кластера, подключение которому осуществляется по протоколу ssh[14]. Конфигурация такого подключения будет содержать логин, пароль пользователя на вычислительном кластере или его приватный ключ.

4.4.5 Дерево сборки

Модуль сборки LuNA-программы должен поддерживать вариативную сборку из различных частных решений, именно для этого множество этапов сборки объединяется в структуру дерева, где у каждого этапа сборки может быть счетное число дочерних этапов. Множество дочерних этапов объединяется в структуру *очереди с приоритетом*, которая реализует в себе операции: 1) положить в структуру пару *"приоритет:этап"* 2) извлечь пару по максимальному или минимальному приоритету.

В дереве присутствует *нулевой этап* - корень дерева, который является родителем всех остальных этапов сборки. Так же данный этап играет инициализирующую роль в дереве сборки, с помощью него объявляются и определяются структуры программы.

Наличие структуры данных очередь с приоритетом - позволяет гибко контролировать порядок сборки LuNA-программы и выражать знание пользователя о порядке сборки частных решений.

Обход дерева осуществляется следующим алгоритмом:

1. В исполнителе текущим этапом устанавливается - нулевой этап сборки.
2. Проверяет зависимости, исполняет, проверяет результат работы этапа.
3. Из очереди дочерних элементов извлекается этап с максимальным приоритетом и устанавливает в текущий этап.

4. Исполнитель проверяет зависимости, исполняет, проверяет результат. При возникновении ошибки, в текущий этап устанавливается родитель.
5. Переход к шагу 3.

Итеративный обход дерева сборки позволяет избавиться от некоторых ошибок связанных с переполнением стека, как бы это было с использованием рекурсивного алгоритма, в случаях большого дерева сборки.

4.4.6 Ветка

Ветка — это абстракция над множеством этапов сборки, объединённая мета-информацией о реализуемом частном решении. Является основным объектом для конструирования дерева сборки из пользовательского кода.

Ветка содержит состояние:

1. Уникальный идентификатор. Реализуется с помощью алгоритмов типа UUID[15].
2. Родитель —объект этапа, от которого нужно сделать ответвление
3. Класс алгоритма — уникальный ключ частного решения. Используется определения принадлежности этапа к конкретному частному решению, после создания дерева сборки.
4. Этапы — набор этапов содержащихся в ветке. Имеют конкретную последовательность, с которой внедряются в дерево сборки.
5. Окружения — набор сущностей окружений, на который будут исполняться этапы ветки.

С помощью веток пользователь описывает этапы сборки. Далее множество веток преобразуются в дерево сборки по следующим правилам:

1. Каждому этапу содержащемуся в ветке присваивается мета-информация (уникальный идентификатор, класс алгоритма, родитель)

2. Каждый этап в ветке добавляется в соответствующую очередь каждого уровня потомков, если очереди потомка у этапа не существует, то она создаётся.
3. Приоритеты инкрементируются в соответствии с порядком добавления новых этапов. Таким образом более приоритетная ветка будет, та которая добавлена позже остальных. Также приоритеты этапам можно задавать вручную.

На рисунке 6 проиллюстрирован пример включения ветки в дерево сборки.

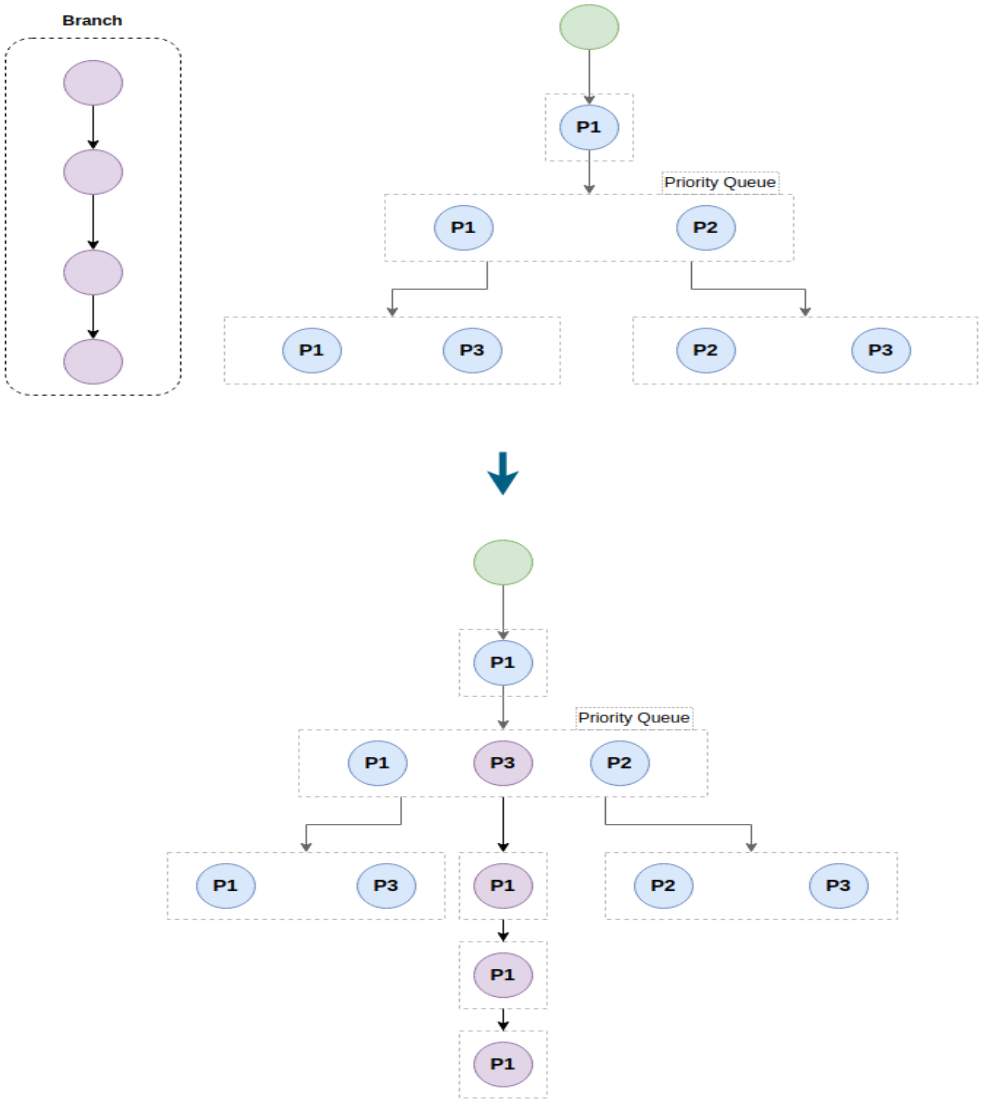

Рисунок 6 — пример включения ветки в дерево сборки

Сущность веток позволяет абстрагироваться от манипуляций с конкретными этапами сборки программы, что делает описание дерева сборки более удобным. Также ветки позволяют группировать этапы сборки по некоторому признаку, что важно при внедрении частных решений. Введение данной сущности позволяет интерфейсу удовлетворить требованию о группировке шагов конкретного частного решения.

4.4.7 Операции над этапами

Над этапами и ветками определён набор операций:

1. Откат — исполнитель устанавливает в текущий этап родителя ветки. Используется для отмены исполнения всей ветки. Ключевая операция модуля сборки. На рисунке 7 продемонстрирована операция отката ветки при возникновении ошибки.

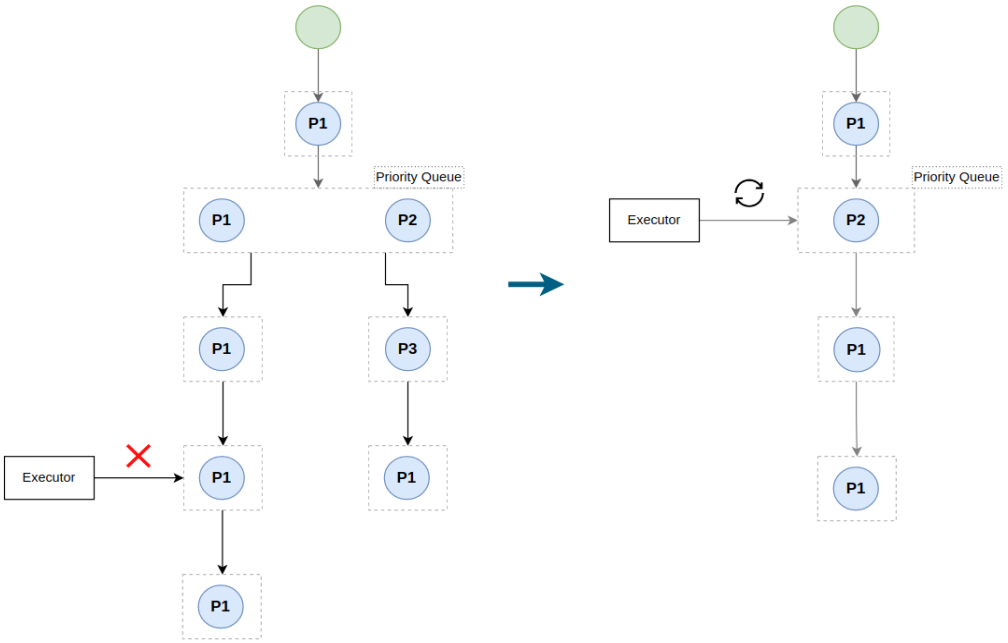

Рисунок 7 — пример операции отката

С помощью этой операции удовлетворяется требование о манипулирование процессом

2. Дублирование — исполнитель добавляет в очередь родительского этапа текущий этап с более низким приоритетом. Подобная возможность используется при исполнении одного и того же этапа в разных окружениях. На рисунке 8 проиллюстрирован пример операции дублирования.

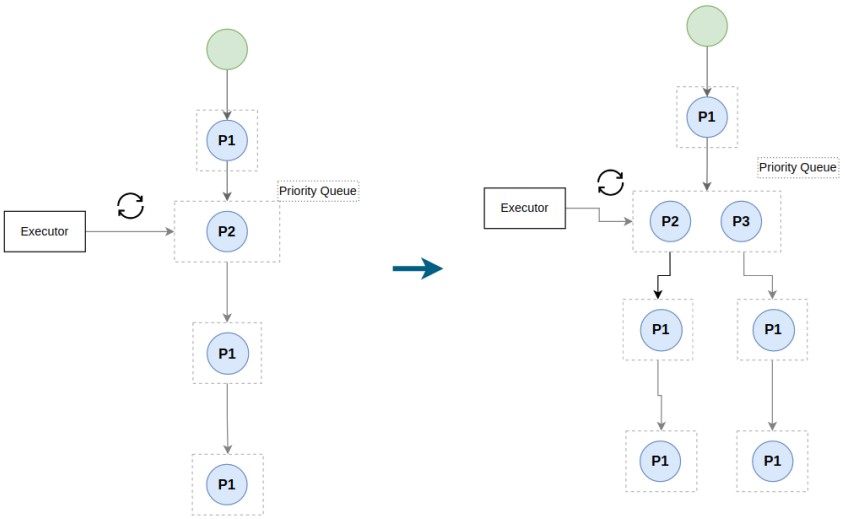

Рисунок 8 — пример операций дублирования

Множество операций над этапами может расширяться по мере необходимости. На данных момент определены основные операции для более гибкого манипулирования процессом сборки программы.

Операции позволяют гибко манипулировать процессом сборки программы, а также принятием решений о исполняемом специализированном решение.

4.4.8 Операции над ветками

Над ветками определён набор операций:

1. Слияние — ветка считается слитой в другую, если последний этап сливаемой ветки имеет один и тот же объект очереди с этапом другой ветки. Данная возможность позволяет встраивать в основную ветку альтернативные этапы, функциональность которых схожа между собой.

Пример: в основную ветку сборки программы может включаться несколько вариантов парсеров, препроцессоров, компиляторов и так далее. На рисунке 9 приведен пример операции слияния.

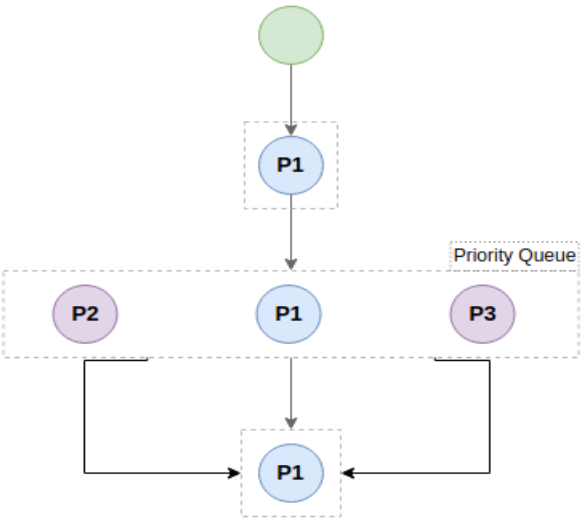

Рисунок 9 — пример операции слияния

4.4.9 Описание дерева сборки

Процесс сборки LuNA-программы происходит централизованно, для исключения коллизий и циклов сборки программы. Частные решения описываются в виде конечного множества веток, включающие нужные этапы сборки. Объединение этапов сборки в виде веток предоставляет удобный интерфейс для включения нового частного решения в общее дерево сборки. Каждой ветке присваивается мета-информация о реализующем частном решении, окружении и родителе ветки.

Далее устанавливаются отношения между ветками сборки. Для *ветки* в качестве родителя присваивается нужный этап, тем самым породив новое ответвление сборки программы. Множество веток с заданными отношениями преобразуются в *дерево сборки*, для родителя ветки добавляется новый потомок с наиболее высоким приоритетом. Для более гибкого контроля

процесса сборки приоритеты у этапов можно переопределять в ручную, такая возможность позволяет явно задавать порядок сборки.

Во время сборки программы исполнитель пытается выполнить каждый этап ветки, при возникновении ошибки в любом из этапов, он выполняет операцию “*откат*” и возвращается к этапу который является, родителем ветки. Тем самым поддерживается вариативность процесса сборки LuNA-программы.

4.4.10 Анализ

Соответствие требованиям из пункта 4.1 пояснено по ходу изложения всего раздела. Выбор модели сборки программы обусловлена ранее описанными требованиями:

1. Инкапсуляции и изолированности удовлетворяет присутствие сущности этапа в модели сборки программы.Каждый этап описывается независимо от исполнителя, дерева сборки и других этапов.
2. Определенность достигается за счет возможности определения всех веток программы в одной конкретной функции модуля сборки, например точка входа в модуль сборки.
3. Полнота достигается за счет внедрения сущности веток в модуль сборки и определенных операций над ними.

Предложенная модель сборки удовлетворяет ранее заданным требованиям и предоставляет удобных интерфейс для сборки программы. А также предлагаемое решение позволяет поддерживать вариативность сборки LuNA-программы, что удовлетворяет условиям поставленной задачи.

5 Программная реализация модуля сборки

Модуль сборки для системы LuNA был реализован на языке Python (язык выбран исходя из того, что большая часть кодовой базы системы LuNA использует этот язык) и представляет из себя набор объектов, взаимодействующих друг с другом. Модуль заменяет функциональность общего скрипта luna и может быть встроен в систему LuNA вместо него. В приложении А описано “Руководство для программиста”.

5.1 Реализация дерева сборки

Одной из ключевых абстракций модуля сборки LuNA-программы является этап сборки программы. Каждый этап сборки программ представляет из себя класс, который наследуется от базового класса *Stage* и переопределяет его абстрактные методы:

- get_command — метод возвращает объект класса Command, представляющий из себя агрегат информации о исполняемой этапом команде(скрипт, функция, подпрограмма).
- get_requirements — метод возвращает коллекцию объектов класса Requirement. Данный метод используется для проверки зависимостей определённого этапа. Класс Requirement является реализацией зависимостей декларируемой этапом.
- get_results — метод возвращает коллекцию объектов класса Result. Класс Result является реализацией результата исполнения отдельного этапа. Метод призван проверять корректность работы каждого этапа путём поиска созданных объектов.

Переопределение вышеизложенных методов объекта позволяет декларировать входные данные, результаты работы этапа и его поведение. Создание абстрактных классов реализовано при помощи библиотеки *ABC [16]*.

Каждый объект этапа в своём состоянии содержит: родителя (поле *parent*), для поддержания древовидной структуры, очередь с приоритетом(поле *next_stages*), для реализации вариативности сборки, идентификатор(поле *id*), метаинформация(поле *meta*), объект класса *Meta*, который содержит в себе информацию о реализуемом частном решении. На листинге 1 приведен пример реализации этапа.

```
      class GenerateCppBlocks(stage.Stage):
    def __init__(self):
        super().__init__()
        self.fcmp2 = Requirement(
            requirement=f'{cfg.luna_home}/scripts/fcmp2'
        )
             ...
        self.cpp_block_info = Requirement(
        requirement=f'{cfg.build_dir}/cpp_blocks_info.json'
        )
        self.cpp = Requirement(
            requirement=f'{cfg.build_dir}/test.cpp'
        )
        self.recommendations = Requirement(
            requirement=f'{cfg.build_dir}/program_recom.ja'
        )

    def get_requirements(self) -> [Requirement]:
        return [self.fcmp2, self.recommendations]

    def get_command(self) -> Command:
        cmd = f'{self.interpreter} {self.fcmp2} {self.recommendations}
 {self.cpp} {self.cpp_block_info}'
        return Command(
            bash_command=cmd
        )
```

Листинг 1 — пример реализации этапа сборки

Класс GenerateCppBlocks является реализацией отдельного этапа сборки, который декларирует зависимости в качестве атрибутов класса в методе __init__() и объявляет о том, что часть из них нужно проверить перед

исполнением с помощью метода get_requirements(). Метод get_command() декларирует о исполняемой команде используя ранее определенный атрибуты класса.

Реализацией абстракции *ветка* является - класс *Branch*, в нём описывается набор этапов (поле *stages*) - коллекция объектов *Stage*, набор окружений(поле *environments*). Также каждая ветка именуется с помощью тэга(поле *tag*), тэг представляет из себя строковое значение, которое идентифицирует частное решение, реализуемое веткой.

Для конструирования дерева используется объект класса *TreeBuilder*, в конструктор которому приходит коллекция объектов класса *Branch*. С помощью метода *build* происходит процесс сборки дерева.

В качестве корня дерева выступает объект класса *RootStage* этот объект реализует шаблон проектирования *Singleton[17]*, для избежания дублирования корня дерева. В процессе сбора дерева веткам, у которых не определен родитель, присваивается объект класса *RootStage*.

На листинге 2 приведен пример описания дерева сборки с помощью веток:

```
    def get_root():
main = Branch(
    tag='main',
    stages=[
       main_branch.Preprocessor(),
       main_branch.Parser(),
       main_branch.Substitution(),
       main_branch.GenerateBlocks(),
       main_branch.GenerateRecoms(),
       main_branch.GenerateCppBlocks(),
       main_branch.GenerateMakefile(),
       main_branch.BuildLibUcodes(),
     main_branch.RunRTS(),
    ],
    environment=enviroment.LocalEnvironment('local')
)
```

```
iclu = Branch(
    tag='luic',
    stages=[
        iclu_branch.Preprocessor(),
        iclu_branch.Parser(),
        iclu_branch.Compiler()
    ],
    environment= enviroment.LocalEnvironment('local')
)

builder = TreeBuilder(main, iclu)
return builder.build()
```

Листинг 2 — описание дерева сборки с помощью веток

Из листинга видно, что main, iclu — ветка сборки, которые имеют множество дочерних этапов и локальное окружение для исполнения. У обоих веток явно не указан родитель это значит, что они наследуются от объекта класса RootStage.

5.2 Конфигурация

Конфигурация приложения — одна из важных составляющих модуля сборки LuNA-программы. С помощью конфигурации этапам задаются зависимости, место сохранения результата работы. Также с помощью конфигурации приложению, задаются некоторые функциональные и нефункциональные возможности: настраивается уровень логирования приложения, устанавливается таймер для каждого этапа, на конкретных этапах устанавливаются возможности балансировки и так далее. Исходя из этого был сделан вывод, что модуль сборки нуждается в конфигурации и его нужно реализовать.

Конфигурация программного обеспечения — совокупность настроек программы, задаваемая пользователем, а также процесс изменения этих

настроек в соответствии с нуждами пользователя. Обычно это включает следующее:

- Значения, зависимые от среды развертывания.
- Идентификаторы подключения к ресурсам типа базы данных, кэш-памяти и другим сторонним службам.
- Регистрационные данные для подключения к внешним сервисам, например, к Amazon S3[18].

Иногда в UNIX-подобных ОС конфигурация задается на этапе сборки программы, и для её изменения программу необходимо пересобирать. Ярким примером может служить ядро *Linux*[19].

Почти для всех программ, собираемых с использованием сценариев autoconf[20], можно подключать или отключать те или иные внешние библиотеки.

При разработке программных продуктов существует несколько способов сконфигурировать приложение:

- Константы в коде. Данный способ нарушает методологию Twelve-Factor App methodology[21], которая требует строгого разделения конфигурации и кода. Конфигурация может существенно отличаться между развертыванием, код же не должен различаться
- Конфигурационные файлы. Этот способ наиболее подходит, чем хранение констант в коде, но он имеет ряд недостатков: конфигурационные файлы разбросаны по всему проекту, поэтому становится сложно управлять всеми настройками программы, также форматы конфигурационных файлов специфичны для конкретного языка или фреймворка.
- Переменные окружения. Наиболее удобных и безопасный способ хранения конфигурации. Переменные окружения легко изменить между запусками/развертыванием. В отличие от пользовательских

конфигурационных файлов они являются независимым от языка и операционной системы.

В модуле сборки LuNA-программы конфигурация осуществляется с помощью библиотеки *configargparse*[22], библиотека позволяет конфигурировать систему LuNA, как с помощью переменных окружений расположенных в файле *.env*, так и с помощью конфигурационных файлов разных форматов. Также данная библиотека позволяет конфигурировать запуск модуля сборки LuNA-программ с помощью параметров и ключей командной строки, что является важным для модуля имеющего CLI точку входа.

Конфигурация описывается с помощью специального конфигурационного класса *LunaConfig*, где внутри конструктора описываются конфигурируемые параметры и их источники. На рисунке 3 приведен пример описания конфигурации программы.

```
      class LunaConfig:
   def __new__(cls):
       if not hasattr(cls, 'instance'):
           cls.instance = super(LunaConfig, cls).__new__(cls)
       return cls.instance

   def __init__(self):
       self.parser = configargparse.ArgParser()
      . . .
       self.parser.add_argument("-lh", "--luna_home", required=True,
help="Luna home path", env_var="LUNA_HOME")
      self.parser.add_argument("--build-dir", env_var="BUILD_DIR",
default=os.getcwd())
      . . .
       self.cfg = self.parser.parse_args()
```

Листинг 3 — пример класса конфигурации

Из листинга видно, что класс *LunaConfig* декларирует множество конфигурируемых параметров, один из них *–luna_home*(домашняя директория основной системы LuNA), он заявлен как обязательный как через

переменную окружения *LUNA_HOME* так и через параметр командной строки.

5.3 Журналирование(логирование)

Журналирование — одна из важных составляющих модуля сборки LuNA-программы, просмотрев записи журнала, пользователь может отследить каким образом осуществлялся обход дерева сборки, сколько времени понадобилось модулю на выполнение конкретного этапа, а также отследить ошибки возникающие при исполнение этапа сборки. Исходя из этого был сделан вывод, что модуль сборки нуждается в журналировании и его нужно реализовать.

Журналирование — это процесс записи в хронологическом порядке событий, происходящих с каким-то объектом. В информационных технологиях в качестве такого объекта может представать файловая система, операционная система или отдельное *приложение*. Запись может совершаться в файл или в базу данных. По этим записям можно понять, что сейчас происходит с наблюдаемым объектом или же восстановить, что с ним происходило в какой-то интересующий нас момент времени.

Журнал — это поток агрегированных, упорядоченных по времени событий, собранных из потоков вывода собранных из потоков вывода всех запущенных процессов и вспомогательных сервисов. Журнал в своём сыром виде обычно представлен текстовым форматом с одним событием на строчку (хотя трассировки исключений могут занимать несколько строк). Журнал не имеет фиксированного начала и конца, поток сообщений непрерывен, пока работает приложение.

В модуле журналирование реализовано с помощью пакета стандартной библиотеке Python — *logging*[23]. Данный пакет предоставляет удобный интерфейс для конфигурирования журнала, настройки уровня журналирования, формат вывода записи журнала. Также пакет предоставляет

множество обработчиков для вывода записей в разные среды, например(StreamHandler — stdout(stderr), FileHandler — запись в файл, SysLogHandler — запись в syslog).

Настройка уровня журналирования происходит с помощью объекта конфигурации. Для агрегирования информации о исполнении этапа используются объекты класса *Status*. Класс *Status* содержит в себе сообщения, получаемые из потоков stdout, stderr во время исполнения, статус кода ответа и время исполнения этапа сборки. Ниже приведен на рисунке 10 приведен пример вывода записи журнала в stdout:

```
[INFO - (executor.py).Run(44) - Preprocessor is done.
Time: 0.04444011498708278
[ERROR - (executor.py).Run(47) - Parser Status command finished with code: 1
Error: ERROR (line 9): invalid symbol: @
{
    hello_world() <-- here

Message:
Time: 0.00535097499601021

[INFO - (executor.py).Run(44) - Preprocessor is done.
Time: 0.05688307699165307
[INFO - (executor.py).Run(44) - Parser is done.
Time: 0.005658017995301634
[INFO - (executor.py).Run(44) - Substitution is done.
Time: 0.0640331520116888
[INFO - (executor.py).Run(44) - GenerateBlocks is done.
Time: 0.04744183400180191
[INFO - (executor.py).Run(44) - GenerateRecoms is done.
Time: 0.06364716400275938
[INFO - (executor.py).Run(44) - GenerateCppBlocks is done.
Time: 0.061201564996736124
[INFO - (executor.py).Run(44) - GenerateMakefile is done.
Time: 0.21877568200579844
[INFO - (executor.py).Run(44) - BuildLibUcodes is done.
Time: 1.0842404209834058
Hello world!

[INFO - (executor.py).Run(44) - RunRTS is done.
Time: 0.48101699401740916
```

Рисунок 10 — пример лога процесса сборки

Из рисунка видно, что процесс сборки вначале попытался выполниться в сторону более приоритетной ветки, далее этап *Parser* завершился с ошибкой и модуль собрал исполнил по основной ветке.

5.4 Окружения

Объект класса *Environment* реализуют окружения модуля сборки. Класс *Environment* является базовым классом для частных окружений, и содержит следующие абстрактные методы:

- check — метод предназначенный для проверки зависимостей или результатов работы этапа. Принимает на вход объекты классов Requirement и Result Например: LocalEnvironment — класс реализующий локальное окружение, проверяет наличие файлов в файловой системе.
- execute_command — метод предназначенный для исполнения команды — объект класса Command. LocalEnvironment реализует данный метод с помощью исполнения shell команд в дочернем процессе операционной системы.

В рамках системы реализовано:

- LocalEnvironment — локальное окружение проверку и исполнение осущевствляет с помощью модуля стандартной библиотеки subprocess[24]. С помощью метода run() subprocess порождает дочерний процесс и блокируется до его завершения.
- RemoteEnvironment — окружение на удаленной машине. Для исполнения shell команд окружение устанавливает ssh[20] соединение с помощью библиотеки Fabric[25]. Fabric — это высокоуровневая библиотека Python, предназначенная для удаленного выполнения команд оболочки через SSH.

На листинге 4 приведена реализация LocalEnvironment:

```
 class LocalEnvironment(Environment):
def __init__(self, key_environment):
    local = environments[key_environment]
```

```
        self.user = local["user"]
        self.password = local["password"]

    def check(self, requirement) -> Status:
        if os.path.exists(str(requirement)):
            status = Status(return_code=0, message="Success")
        else:
            status = Status(return_code=1, message=f"Requirement
{requirement} is absent")
        return status

    def execute_command(self, cmd) -> Status:
        if type(cmd.bash_command) is str:
            shell_flag = True
        else:
            shell_flag = False

        start_time = time.perf_counter()
        process = subprocess.run(cmd.bash_command,
                                 stderr=subprocess.PIPE,
                                 stdout=subprocess.PIPE,
                                 shell=shell_flag,
                                 encoding='utf-8',
                                 input=cmd.input,
                                 env=cmd.environment,
                                 )
        end_time = time.perf_counter()
        if cmd.output_type is cmd.FILE:
            open(cmd.output_file,
                     'w').write(process.stdout)
        elif cmd.output_type is cmd.STDOUT:
            print(process.stdout)
        return super().collect_status(message=process.stdout,
                          err=process.stderr,
returncode=process.returncode,
execution_time=(end_time - start_time))
```

Листинг 4 — реализация LocalEnvironment

Из представленного листинга 4 видно, каким образом реализуются классы окружения. Методы execute_command(), check() ответственны за

исполнение команды и проверку зависимостей/результатов на данном окружении. В данном случае окружение выполняет операции на локальном компьютере посредством запуска дочерних процессов с помощью метода *run()* библиотеки *subprocess*.

5.5 Реализация функциональных тестов

В этом разделе описывается реализация функциональных тестов, результаты которых далее отражены в разделе *Тестирование.*

В рамках модуля сборки были написаны функциональные тесты при помощи фреймворка *PyTest*[26]. Был описаны класс *TestStage* с помощью него создаётся окружение для теста(тестовые ветки сборки). Также класс *StubEnvironment*, объекты который наследуется от базового класса Environment и переопределяет его методы, так чтобы с помощью теста возможно было снять показатели исполнения этапов сборки:

- success_executions — количество успешно исполнившихся этапов сборки тестового дерева
- check_failed_executions — количество исполнений этапов, закончившихся ошибкой по причине отсутствия необходимый зависимостей.
- list — список, который является средой для выполнения тестовых этапов. Окружение хранит в списке результаты выполнения этапов и зависимостей заданных заранее.

5.6 Анализ программной реализации

Реализация, которая описана в этом разделе, удовлетворяет предлагаемой модели сборки описанной в теоретической части. Реализованный модуль сборки позволяет удобно расширять и дополнять систему LuNA новыми частными решениями. Также модуль реализован таким образом, что его функциональность можно дополнять более широким

классом задач. Модуль полностью конфигурируем, а наличие журналирования делает исполнение дерева сборки прозрачным.

6 Тестирование

Для проверки функциональных возможностей модуля сборки были реализованы тесты двух видов:

1. Функциональные тесты, призванные испытать модуль сборки на возможные вырожденные и крайние случаи.
2. Тесты на настоящих программах, призванные испытать модуль в реальных условиях.

6.1 Функциональные тесты

Для фиксирования поведения модуля при крайних и вырожденных условиях для исполнения программы были описаны синтетические тестовые случаи:

1. Дерево содержит только корень (RootStage). Данный тестовый случай
2. Дерево содержит только один этап сборки отличный от корня (RootStage). Этот и предыдущий тестовые случаи призваны проверить поведение программы на вырожденных входных данных.
3. Дерево содержит последовательность из 5 этапов. Данный тестовый случай позволяет проверить поведение модуля на сборку программы без других веток, что соответствует поведению системы LuNA без модуля сборки.
4. Дерево содержит 5 веток по одному этапу в каждом, отношения между которыми задано как показано на рисунке 11.

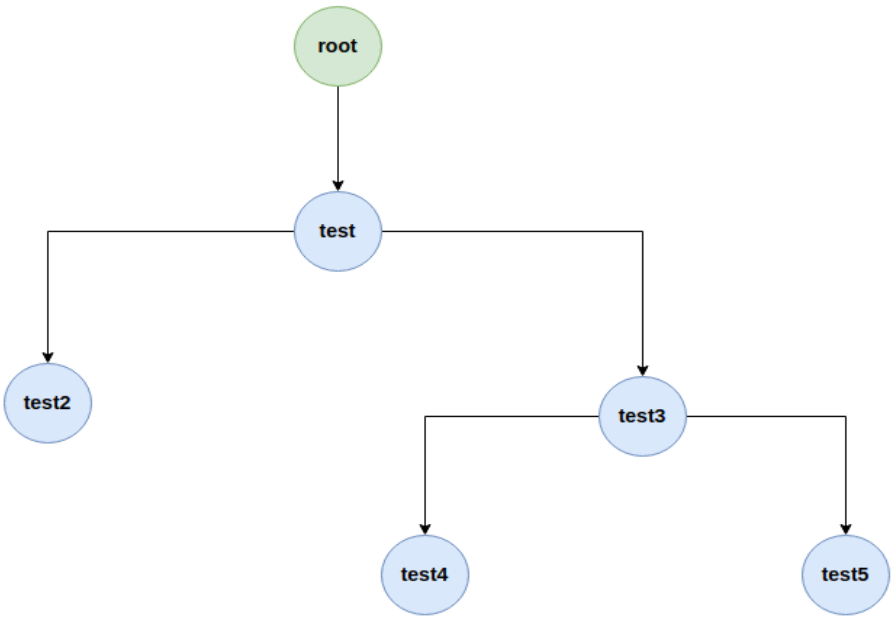

Рисунок 11 — отношения между этапами веток тестового случая

Результаты теста №4 приведены на рисунке 12:

```
============================= test session starts ==============================
collecting ... collected 1 item

module_test.py::test_fully_tree PASSED                                   [100%][INFO - (executor.py).Run(52) - Branch Tag: test TestStage is done.
Time: 0.0
[ERROR - (executor.py).Run(55) - Branch Tag: test2 TestStage is failed
 Status command finished with code: 1
Error: None
Message: Requirement ['notcontainsobj'] is absent
Time: 0

[INFO - (executor.py).Run(52) - Branch Tag: test3 TestStage is done.
Time: 0.0
[ERROR - (executor.py).Run(55) - Branch Tag: test4 TestStage is failed
 Status command finished with code: 1
Error: None
Message: Requirement notcontainsobj is absent
Time: 0

[INFO - (executor.py).Run(52) - Branch Tag: test5 TestStage is done.
Time: 0.0


============================== 1 passed in 0.04s ===============================
```

Рисунок 12 — результат вывода теста №4

Из результата следует, что 1-ый этап выполнился успешно, затем по очереди пытается выполнится 2-ой этап, и завершается с ошибкой по причине отсутствия необходимой зависимости зависимости, далее 3-этап

завершается с успехом, 4-ый с ошибкой и 5-ый с успехом. Тем самым по результатам обхода дерева получилось 3-этапа завершившихся с успехом и 2 - с ошибкой, что говорит о том, что тест выполняется с успехом.

В результате выполнения функциональных тестов, можно сделать вывод, что модуль сборки LuNA-программ работает корректно на описанных тестовых случаях.

6.2 Простейшие фрагментированные алгоритмы

Во время тестирования модуль сборки *LuNA*-программ в первую очередь был испытан простейшими программами на языке *LuNA*, в числе их HelloWorld для общей системы *LuNA* и специализированного компилятора *ICLU*. Листинги программ приведены ниже.

Цель данного теста проверить базовую функциональность модуля сборки и проследить исполнение этапов каждой ветки. На листингах 5 и 6 показаны программы на языке LuNA для основной ветки и ветки ICLU. На листинге 6 приведена программа для ветки iclu и на рисунке 13 изображен результат тестирования.

```
import c_helloworld() as hello_world;

sub main()
{
   hello_world() @ { locator_cyclic: 0; };
}
```

Листинг 5. Программа для основной ветки.

```
import c_helloworld() as hello_world;

sub main()
{
   hello_world();
}
```

Листинг 6 — программа для ветки ICLU

```
[INFO - (executor.py).Run(52) - Branch Tag: luic Preprocessor is done.
Time: 0.03629964900028426
[ERROR - (executor.py).Run(55) - Branch Tag: luic Parser is failed
 Status command finished with code: 1
Error: ERROR (line 9): invalid symbol: @
{
    hello_world() <-- here

Message:
Time: 0.005197416001465172

[INFO - (executor.py).Run(52) - Branch Tag: main Preprocessor is done.
Time: 0.0453796380024869
[INFO - (executor.py).Run(52) - Branch Tag: main Parser is done.
Time: 0.002695920004043728
[INFO - (executor.py).Run(52) - Branch Tag: main Substitution is done.
Time: 0.04287497600307688
[INFO - (executor.py).Run(52) - Branch Tag: main GenerateBlocks is done.
Time: 0.0311275099957129
[INFO - (executor.py).Run(52) - Branch Tag: main GenerateRecoms is done.
Time: 0.04190620100416709
[INFO - (executor.py).Run(52) - Branch Tag: main GenerateCppBlocks is done.
Time: 0.04778585500025656
[INFO - (executor.py).Run(52) - Branch Tag: main GenerateMakefile is done.
Time: 0.161317008009064
[INFO - (executor.py).Run(52) - Branch Tag: main BuildLibUcodes is done.
Time: 0.8031600039976183
Hello world!

[INFO - (executor.py).Run(52) - Branch Tag: main RunRTS is done.
Time: 0.4609511499875225
```

Рисунок 13 — результат сборка программы основной ветки

Из результатов этого теста можно сделать вывод, о том что программа *hello_world_iclu.fa* выполнилась без ошибок по ветке этапов *LuNA-ICLU* и вывела результат. Программа *hello_world.fa* в первую очередь начала исполняться по ветке этапов *LuNA-ICLU*, но завершилась с ошибкой на этапе Parser, далее продолжила исполняться по основной ветке этапов. Таким образом модуль сборки показал свою корректность работы на простейших программах.

6.3 Фрагментированный алгоритм для основной ветки

В данном тесте модуль сборки испытывается фрагментированным алгоритмом умножения матриц. Цель данного теста проверить вариативность сборки и принятий решения. В результате теста модуль

должен среагировать на ошибку в более приоритетной ветке *LuNA-ICLU*, и исполнится по основной. Листинг программы под номером 7 приведен ниже.

```
#!/usr/bin/luna
/*
For and while operators ("loops").
*/
import c_init(int, name) as init;
import c_show(string, int) as show;
sub main() {

  for i = 1 .. 2
     for j=1..3
        show("coord:", i*100+j);
}
```

Листинг 7 — фрагментированная программа для основной ветки

Результат тестирования проиллюстрирован на рисунке 14.

```
[INFO - (executor.py).Run(52) - Branch Tag: luic Preprocessor is done.
Time: 0.03479297104058787
[ERROR - (executor.py).Run(55) - Branch Tag: luic Parser is failed
 Status command finished with code: 1
Error: ERROR (line 1): invalid symbol: #
<-- here

Message:
Time: 0.0034371380461379886

[INFO - (executor.py).Run(52) - Branch Tag: main Preprocessor is done.
Time: 0.039845708990469575
[INFO - (executor.py).Run(52) - Branch Tag: main Parser is done.
Time: 0.004539916000794619
[INFO - (executor.py).Run(52) - Branch Tag: main Substitution is done.
Time: 0.05223294399911538
[INFO - (executor.py).Run(52) - Branch Tag: main GenerateBlocks is done.
Time: 0.03931760002160445
[INFO - (executor.py).Run(52) - Branch Tag: main GenerateRecoms is done.
Time: 0.05737686395877972
[INFO - (executor.py).Run(52) - Branch Tag: main GenerateCppBlocks is done.
Time: 0.04692070500459522
[INFO - (executor.py).Run(52) - Branch Tag: main GenerateMakefile is done.
Time: 0.20582091802498326
[INFO - (executor.py).Run(52) - Branch Tag: main BuildLibUcodes is done.
Time: 0.8861202570260502
coord: 102
coord: 103
coord: 101
coord: 201
coord: 202
coord: 203

[INFO - (executor.py).Run(52) - Branch Tag: main RunRTS is done.
Time: 0.3601294520194642
```

Рисунок 14 — результат выполнения процесса сборки и исполнения

Из результатов тесты видно, что сборка программы прервалась на ветке с тегом ‘luic’ из за ошибки на этапе Parser, далее произошёл откат до родителя RootStage и исполнитель перешёл к ветке с тэгом ‘main’. В результате данного теста модуль показал корректность работы на заданном тестовом случае.

6.3.1 Реализация уравнения пуассона в одномерной декомпозиции

Данный тест призван проверить работоспособность программы на реальной задаче, а именно уравнение Пуассона в одномерной декомпозиции. Данный тестовый случай должен, что тест при отсутствие ошибок модуль

начнет и закончит успешное исполнение на приоритетной ветке iclu. На рисунке 15 приведен результат теста.

```
[INFO - (executor.py).Run(52) - Branch Tag: luic Preprocessor is done.
Time: 0.040792019999571494
[INFO - (executor.py).Run(52) - Branch Tag: luic Parser is done.
Time: 0.010054079999463283
[INFO - (executor.py).Run(52) - Branch Tag: luic Compiler is done.
Time: 0.010981594999975641
[INFO - (executor.py).Run(52) - Branch Tag: luic MPICompiler is done.
Time: 1.314431060998686
[INFO - (executor.py).Run(52) - Branch Tag: luic Runner is done.
Time: 7.6001696519997495
DF: 4999
DF: 4999
DF: 4999
DF: 4999
```

Рисунок 15 — результат выполнения процесса сборки и исполнения программы реализующую уравнение Пуассона в одномерной декомпозиции

Из рисунка 15 видно, что процесс сборки и последующее исполнение началось и закончилось с успехом на ветке iclu.

По результатам данного теста можно сделать вывод, что модуль сборки LuNA-программы исполняется по ветке iclu, что удовлетворяет заданному тестовому случаю.

6.3.2 Выводы по результату тестирования

Исходя из промежуточных результатов по каждому пункту можно сделать вывод, что модуль сборки показал корректную работу как на синтетических тестовых случаях, так и на реальных программах.

ЗАКЛЮЧЕНИЕ

В результате работы был разработан модуль сборки LuNA, который позволяет интегрировать различные специализированные системы конструирования параллельных программ для определенного класса задач. С помощью модуля: разработчик LuNA-программы абстрагирован от множества частных систем и имеет единый интерфейс для сборки и исполнения программы, а разработчик системы LuNA интегрировать специализированные системы с помощью удобного интерфейса. В ходе работы выполнены все поставленные задачи тем самым была выполнена цель работы, которая вносит вклад в задачу специализированной поддержки автоматических систем конструирования параллельных программ.

Защищаемые положения:

1. Спроектирован интерфейс для обеспечения специализированной поддержки конструирования фрагментированных программ и реализован модуль сборки LuNA-программы.
2. Реализован модуль сборки LuNA-программ, реализующий спроектированному интерфейсу и удовлетворяющий предъявляемым требованиям.
3. Проведено экспериментальное исследование работоспособности модуля сборки LuNA-программ.

Возможные направления развития:

1. Поддержка других способов принятия решения о выборе специального решения собираемой задачи численного моделирования.
2. Интеграция новых частный решений в общую систему.

Выпускная квалификационная работа выполнена мной самостоятельно с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения

(концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Левченко Кирилл Константинович _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Ахмед-Заки Д. Ж., Лебедев Д. В., Малышкин В. Э., Перепелкин В. А. Автоматизация конструирования распределенных программ численного моделирования в системе LuNA на примере модельной задачи // журнал "Проблемы информатики", 2019, № 4, с.53-64. DOI: 10.24411/2073-0667-2019-00017
2. Malyshkin, V.E., Perepelkin, V.A. (2011). LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem. In: Malyshkin, V. (eds) Parallel Computing Technologies. PaCT 2011.
3. Киреев С. Е. Параллельная реализация метода частиц в ячейках для моделирования задач гравитационной космодинамики //Автометрия. – 2006. – Т. 42. – №. 3. – С. 32-39.
4. Zhirkov, I. (2017). Compilation Pipeline. In: Low-Level Programming. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-2403-8_5
5. OCaml: [Электронный ресурс]. URL: https://ocaml.org. (Дата обращения - 08.05.2022).
6. The OCaml Compiler Pipeline Kevin Sookocheff: [Электронный ресурс]. URL*: https://sookocheff.com/post/ocaml/the-ocaml-compiler-pipeline.* (Дата обращения - 08.05.2022).
7. Краткий обзор методологии CI/CD URL: https://sbercloud.ru/ru/warp/cicd-about. (Дата обращения - 08.05.2022).
8. DevOps: [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/DevOps. (Дата обращения - 08.05.2022).
9. Make: [Электронный ресурс]. URL: http://pushorigin.ru/bash/make. (Дата обращения - 08.05.2022).

10. Flex: [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Flex_(lexical_analyser_generator). (Дата обращения - 08.05.2022).
11. Bison: [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/GNU_Bison. (Дата обращения - 08.05.2022).
12. Belyaev, N., Kireev, S. (2019). LuNA-ICLU Compiler for Automated Generation of Iterative Fragmented Programs. In: Malyshkin, V. (eds) Parallel Computing Technologies. PaCT 2019. Lecture Notes in Computer Science(), vol 11657. Springer, Cham.
13. MPI: The Message Passing Interface - URL: https://parallel.ru/tech/tech_dev/mpi.html. (Дата обращения 08.05.2022).
14. The Secure Shell Transport Layer Protocol: [Электронный ресурс]. URL: https://www.rfc-editor.org/rfc/rfc4253. (Дата обращения 08.05.2022).
15. Universally unique identifier: [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Universally. (Дата обращения - 08.05.2022).
16. Abstract Base Classes: [Электронный ресурс]. URL: https://docs.python.org/3/library/abc.html. (Дата обращения - 08.05.2022).
17. Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2021. — 448 с.: ил. — (Серия «Библиотека программиста»). ISBN 978-5-4461-1595-2
18. Amazon S3: [Электронный ресурс]. URL: https://aws.amazon.com/ru/s3. (Дата обращения - 08.05.2022).
19. Ядро Linux: описание процесса разработки, 3-е изд. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2013. — 496 с. : ил. — Парал. тит. англ. ISBN 978-5-8459-1779-9 (рус.)
20. Autoconf: [Электронный ресурс]. URL: https://www.gnu.org/software/autoconf. (Дата обращения 08.05.2022).

21. The Twelve-Factor App: [Электронный ресурс]. URL: https://12factor.net/ru. (Дата обращения 08.05.2022).
22. ConfigArgParse: [Электронный ресурс]. URL: https://pypi.org/project/ConfigArgParse. (Дата обращения 08.05.2022).
23. Logging: [Электронный ресурс]. URL: https://docs.python.org/3/library/logging.html. (Дата обращения 08.05.2022).
24. Subprocess: [Электронный ресурс]. URL: https://docs.python.org/3/library/subprocess.html. (Дата обращения 08.05.2022).
25. Fabric: [Электронный ресурс]. URL: https://www.fabfile.org. (Дата обращения 08.05.2022).
26. PyTest: [Электронный ресурс]. URL: https://docs.pytest.org. (Дата обращения 08.05.2022).

ПРИЛОЖЕНИЕ А

Модуль сборки LuNA-программ

Руководство программиста

Листов 10

2022 г.

Аннотация

В данном программном документе приведено руководство программиста по применению и эксплуатации модуля сборки LuNA-программ. Исходным языком программы является Python. Разработан с помощью средства разработки кода PyCharm компания JETBRAINS.

В данном программном документе, в разделе “Назначение программы” указаны сведения о назначении программы и информация, достаточная для понимания функций программы и ее эксплуатации.

В разделе “Условия выполнения” программы указаны требования, необходимые для выполнения программы.

В разделе “Выполнение программы” указана последовательность действий оператора, обеспечивающих загрузку, запуск, выполнение и завершение программы

Оформление программного документа “Руководство оператора” произведено по требованиям ГОСТ 19.504-79 «ЕСПД. Руководство программиста» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам.

10 Назначение и условия программы

10.1 Назначение программы

Разработанная программа предназначена для внедрения специализированных программных модулей призванных контролировать параллельные программы для конкретного класса задач численного моделирования. Программа предоставляет интерфейс для описания процесса сборки программы.

10.2 Функции программы

Программа позволяет пошагово описать процесс сборки LuNA-программы из пользовательского кода. С помощью предоставляемого интерфейса программа позволяет внедрять в процесс сборки альтернативные варианты исполнения.

Также программа позволяет журналирует процесс сборки программы и позволяет просмотреть записи журнала.

10.3 Условия для выполнения программы

Для исполнения программы программисту необходимо иметь интерпретатор Python версии не ниже 3.8, а также пакеты ConfigArgParse, subprocess, PyTest.

11 Характеристика программы

11.1 Описание основных характеристик программы

Модуль сборки LuNA-программ объединяет все шаги компиляции LuNA-программы, включая альтернативные шаги компиляции для специализированных решений.

11.2 Описание основных особенностей программы

Текущая версия модуля на момент написания руководства модуль сборки LuNA-программы для запуска шагов сборки использует дочерние процессы операционной системы.

12 Обращение к программе

12.1 Конфигурирование и запуск модуля сборки

Для конфигурации модуля используются аргумента командной строки или переменные окружения процесса. Чтобы увидеть все возможное параметры для конфигурации и их наименования выполните команду:

python3 luna_build.py –help

В стандартном потоке вывода появится такое сообщение(если параметры конфигурации не переопределены или не добавлены новые). На рисунке 1 приведен пример вывод модуля с ключом -help.

```
usage: luna_build.py [-h] -lh LUNA_HOME --python PYTHON --cxx_flags CXX_FLAGS --cxx CXX --debug DEBUG [--cleanup CLEANUP]
                     [--log_level LOG_LEVEL] [--log_filename LOG_FILENAME] [--ld_library_path LD_LIBRARY_PATH] [-g G]
                     [--build-dir BUILD_DIR] --iclu_home ICLU_HOME
                     program [argv [argv ...]]

positional arguments:
  program
  argv

optional arguments:
  -h, --help            show this help message and exit
  -lh LUNA_HOME, --luna_home LUNA_HOME
                        Luna home path [env var: LUNA_HOME]
  --python PYTHON       Python interpreter path [env var: PYTHON]
  --cxx_flags CXX_FLAGS
                        cxx flags [env var: CXX_FLAGS]
  --cxx CXX             [env var: CXX]
  --debug DEBUG         debug flag [env var: DEBUG]
  --cleanup CLEANUP     cleanup flag, clear temporary directory [env var: CLEANUP]
  --log_level LOG_LEVEL
                        log level ERROR, INFO [env var: LOG_LEVEL]
  --log_filename LOG_FILENAME
                        log filename path [env var: LOG_FILENAME]
  --ld_library_path LD_LIBRARY_PATH
                        [env var: LD_LIBRARY_PATH]
  -g G                  [env var: DEBUG]
  --build-dir BUILD_DIR
                        build directory path [env var: BUILD_DIR]
  --iclu_home ICLU_HOME
                        home directory iclu-project [env var: ICLU_HOME]

 If an arg is specified in more than one place, then commandline values override environment variables which override
defaults.
```

Рисунок 1 — пример вывода с ключом –help

Конфигурация с использованием переменных окружений в файле .env приведена на листинге 1.

```
LUNA_HOME=/home/tobiska/LuNa-demo-project/luna

PYTHON=python3

CXX_FLAGS=" -I /home/tobiska/LuNa-demo-project/luna/include
-std=c++11 -fPIC -Wall -Wpedantic -Wno-vla -Wno-sign-compare
-Wno-unused-but-set-variable -Wno-unused-variable -O3"
CXX=g++
LDFLAGS="-ldl -L /home/tobiska/LuNa-demo-project/luna/lib"

DEBUG=true
CLEANUP=true
BALANCE=true
TIME=true
LUNA_NO_CLEANUP=false
BUILD_DIR=/home/tobiska/PycharmProjects/luna_interpreter/tmp

ICLU_HOME=/home/tobiska/iclu-project

LOG_LEVEL=DEBUG
```

Листинг 1 — пример .env файла

После конфигурации модуля, для запуска процесса сборки выполните команду:

python3 luna_build [<config_arg>...] program.fa [<argv>...]

Пример: python3 luna_build.py examples/mxm/test.fa

Для описания ветки сборки рекомендуется создать папку с названием *<tag>_branch* в ней файл stages.py и далее описывать все этапы сборки в нём.

Для описания этапа сборки нужно наследоваться от базового абстрактного класса Stage пакета stage. Далее переопределить абстрактные методы базового класса get_requirements, get_command, get_results.

Пример описания этапа приведен на листинге 2:

```
class Preprocessor(stage.Stage):
   def __init__(self):
       super().__init__()

       self.interpreter = Requirement(
           requirement=f'{cfg.python}'
       )
```

```
        self.preprocessor = Requirement(
            requirement=f'{cfg.iclu_home}/bin/preprocessor.py'
        )

        self.preprocessed_program = Result(
            result=f'{cfg.build_dir}/preprocessed.fa'
        )

        self.program = Requirement(
            requirement=f'{cfg.program}'
        )

    def get_requirements(self) -> [Requirement]:
        return [self.preprocessor, self.program]

    def get_results(self) -> [Result]:
        return [self.preprocessed_program]

    def get_command(self) -> Command:
        cmd = f'{self.interpreter} {self.preprocessor} <
{self.program} > {self.preprocessed_program}'
        return Command(
            bash_command=cmd
        )
```

Листинг 2 — пример описания этапа

12.2 Описание дерева сборки

Описание дерева сборки происходит централизованно в файле luna_build.py. Для определения ветки нужно импортировать класс Branch из пакета branch.py, а также множество этапов.

Для создания объекта ветки в конструктор данному классу нужно передать:

- коллекцию(<list>) этапов в том порядке, в котором они должны быть выполнены.
- тэг(<str>) ветки.
- окружение, которое можно импортировать из пакета environments
- родителя ветки(<Stage>), данный параметр опционален.

С помощью TreeBuilder из пакета branch и его метода build получаем корень дерева. Для запуска процесса сборки понадобиться создать объект Executor из пакета executor и передать ему в аргумент метода Run корень дерева. На листинге 3 пример описания дерева сборки:

```
       main = Branch(
   tag='main',
   stages=[
      main_branch.Preprocessor(),
      main_branch.Parser(),
      main_branch.Substitution(),
      main_branch.GenerateBlocks(),
      main_branch.GenerateRecoms(),
      main_branch.GenerateCppBlocks(),
      main_branch.GenerateMakefile(),
      main_branch.BuildLibUcodes(),
      main_branch.RunRTS(),
   ],
   environment=enviroment.LocalEnvironment('local')
)

iclu = Branch(
   tag='luic',
   stages=[
       iclu_branch.Preprocessor(),
       iclu_branch.Parser(),
       iclu_branch.Compiler(),
       #Runner
   ],
   environment= enviroment.LocalEnvironment('local')
)

builder = TreeBuilder(main, iclu)
return builder.build()
```

Листинг 3 — пример описания ветки

13 Требования к входным и выходным данным

13.1 Организация используемой входной информации

Входная информация - фрагментированная программа на языке LuNA в формате .fa, код атомарных фрагментов вычислений на языке C/C++ файл ucodes.cpp.

13.2 Организация используемой выходной информации

Выходная информация варьируется в зависимости от этапов выполняемых в модуле сборки. Также модуль журналирует процесс работы в stdout.