Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования
«Московский физико-технический институт
(государственный университет)»

Факультет управления и прикладной математики

Кафедра теоретической и прикладной информатики

СЛИЯНИЕ ИСХОДНЫХ ФАЙЛОВ В ПРИСУТСТВИИ КОНФЛИКТУЮЩИХ СИНТАКСИЧЕСКИХ БЛОКОВ

Выпускная квалификационная работа
(бакалаврская работа)

Направление подготовки: 03.03.01 Прикладные математика и физика

Выполнил:
студент 376 группы Водопьян Даниил Григорьевич

Научный руководитель:
асс. Соболев Артемий Анатольевич

Москва 2017

Аннотация

Современные проекты разработки программного обеспечения (ПО) практически всегда выполняются командой разработчиков. При командной разработке ПО важна координация и синхронизация усилий членов команды. Важным инструментом организации разработки во многих методологиях является программные системы управления версиями. При наличии множественных версий проекта, разрабатываемых параллельно, возникают противоречивые изменения кодовой базы проекта, или конфликты. Для устранений конфликтов версий используются различные алгоритмы и техники слияния версий. Благодаря им слияние версий разрабатываемого программного продукта в значительном количестве случаев проводится в автоматическом режиме, но на практике встречаются ряд ситуаций, когда автоматическое слияние оказывается невозможным и необходимо устранить конфликты вручную, потратив значительное время. Таким образом, совершенствование программных систем управления версиями является важной частью оптимизации разработки ПО.

В результате выполнения работы были рассмотрены проблемы слияния конфликтов в файлах исходного кода и изучены конфликты с пересекающимися синтаксическими блоками. Был разработан алгоритм расширения синтаксических блоков, улучшающий эргономику слияния конфликтов с пересекающимися синтаксическими блоками. Алгоритм был реализован в виде экспериментальной утилиты MergeTool. Утилита MergeTool была протестирована на тестовом наборе из 16 базовых тестов 137 алгоритмических тестов, ее работа была улучшена в течении 7 последовательных версий.

Оглавление

Введение

Современные проекты разработки программного обеспечения (ПО) практически всегда выполняются командой разработчиков. При командной разработке ПО важна координация и синхронизация усилий членов команды. Важным инструментом организации разработки во многих методологиях (моделях разработки ПО и менеджмента) является программные системы управления версиями (Version Control System, VCS или Revision Control System). При наличии множественных версий проекта, разрабатываемых параллельно, возникают противоречивые изменения кодовой базы проекта, или конфликты. Для устранений конфликтов версий используются различные алгоритмы и техники слияния версий. Благодаря им слияние версий разрабатываемого программного продукта в значительном количестве случаев проводится в автоматическом режиме, но на практике встречаются ряд ситуаций, когда автоматическое слияние оказывается невозможным и необходимо устранить конфликты вручную, потратив значительное время. Таким образом, совершенствование программных систем управления версиями является важной частью оптимизации разработки ПО.

В рамках этой работы было рассмотрено ускорение процесса слияния путем повышения эргономичности ручных операций и уменьшения когнитивной нагрузки на исполняющего разработчика. Был разработан и создан прототип утилиты, которая в совокупности с существующими средствами слияния конфликтов предоставляет более естественное представление конфликтов и улучшает возможность их автоматического разрешения. Основное конкурентное преимущество разработанного решения является использования синтаксического парсера для анализа содержимого сливаемых файлов.

Актуальность темы работы

Темы данной работы посвящена решению одной из проблем систем управления версиями, которые используются более 90% команд-разработчиков в процессе подготовки новых программных продуктов. Поэтому даже незначительное совершенствование VCS имеет существенный эффект.

1. Современные модели жизненного цикла программного обеспечения, которые используются IT-компаниями, предполагают использование процесса управления версиями

В современных IT-компаниях разработку крупных проектов программного обеспечения осуществляют силами больших коллективов разработчиков. Например в разработке операционной системы Windows 7 корпорации Microsoft участвовало 2500 человек [1].

В подобных сложных ситуациях для достижения цели фирмы используют не только наиболее подходящую модели жизненного цикла программного обеспечения, но и специально разработанные модели управления коллективом разработчиков.

На практике используется несколько известных моделей жизненного цикла программного обеспечения: каскадная модель, модель итеративной и инкрементальной разработки, ее разновидность - спиральная модель. Каждая модель состоит из нескольких фаз:

1. Формирование требований;
2. Проектирование;
3. Реализация;
4. Тестирование;
5. Внедрение;
6. Эксплуатация
7. Сопровождение.

В каскадной модели начало каждой следующей фазы разработки совпадает с полным окончанием предыдущей. На практике каскадная модель практически не встречается в чистом виде, так как часто приходится возвращаться к предыдущей фазе для того, чтобы исправить вновь найденные ошибки.

Итерационная модель предполагает несколько циклов разработки (итераций), когда каждая последующая итерация обладает расширенным функционалом по сравнению с предыдущей. Таким образом, во время первых трех фаз жизненного цикла ПО, даже если в команде всего один программист, в итерационной модели обязательно проводится процедура управления версиями программного продукта.

2. Современные модели управления большими командами разработчиков предполагают использование программных средств для управления версиями разрабатываемого программного продукта

Модель управление командой разработчиков ПО в каждой компании может быть своя. Например, в Майкрософт использует собственную методологию разработки IT-проектов, которая зафиксирована в двух взаимосвязанных структурах: Microsoft Solutions Framework (MSF) и Microsoft Operations Framework (MOF)[1]. Модель Майкрософт дополненная методологией Agile получила широкое распространение среди компаний-разработчиков программного обеспечения. MSF и Agile предполагают взаимодействие небольших команд разработчиков ПО (от 3 до 10 человек), которые регулярно согласовывают свои участки общей разработки, то есть осуществляют процедуру согласования сделанного за определенный промежуток времени или управление версиями программного продукта. Для этого широко применяются системы контроля версий кода для хранения исходных кодов разрабатываемой программы. В небольшой команде, которая работает в одной комнате, можно согласовать все сделанные в программе изменения и без специальных программных средств. Но часто IT-компании используют “офшорное” программирование, когда разработчики одной команды находятся в разных странах и часовых поясах. В этой ситуации без специальных систем управления версиями не обойтись.

Система управления версиями (VCS) — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое. [3]

3. Исследование современных практик, которые используются разработчиками программного обеспечения, показывает высокую долю команд, применяющих средства VCS, и тратит на операции слияния версий значительное количество рабочего времени[2].

В обзоре практик программирования "Continuous Delivery: A Maturity Assessment Model" (March 2013) приводится данные о том, как команды разработчиков отвечали на вопрос об использовании специального программного обеспечения в своей работе.

[1] https://ru.wikipedia.org/wiki/Microsoft_Solutions_Framework

[2] https://www.guiffy.com/SureMergeWP.html

Распределение ответов на этот вопрос приведено на рис. 1. Для целей данной работы представляет интерес ответы разработчиков по пунктам 1 и 3. Вопрос по пункту 1 может быть сформулирован следующим образом: “Как часто Вы используете программное обеспечение для управления версиями, разрабатываемого продукта?” Из ответов видно, что 91% опрошенных разработчиков используют системы контроля версий в своих проектах. Из них 40% используют VSC ежедневно, а 21% - еженедельно (рис. 2). Также, 93% опрошенных используют автоматические тесты для поддержки работоспособности кодовой базы).

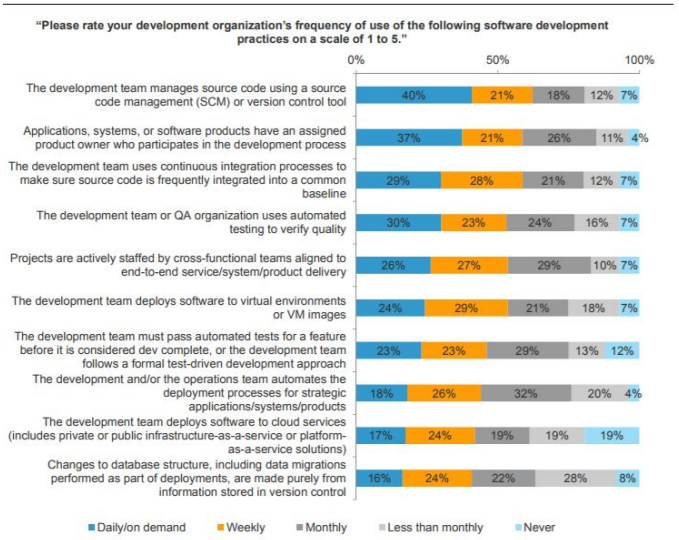

Рис. 1 Диаграмма распределения ответов на вопрос о наиболее часто используемых организационных процедурах при разработке программного обеспечения

Рис. 2 Распределение команд-разработчиков по частоте использования VCS

4. В современные модели организационной зрелости управления программными проектами входит использование VCS.

L. Ward предложил включить в систему оценки уровня организационной зрелости управлением программного проекта использование различных форм управления версиями[3] и разработал специальную для этого шкалу. Также С. Макконнелл[4] в своей книге отмечает, что для успешного осуществления IT-проекта руководитель разработки должен учитывать по крайней мере 33 требования, среди которых: ...3.6. Установлена процедура управления изменениями в проекте.

Таким образом, актуальность темы работы заключается в том, что современные модели жизненного цикла и разработки программного обеспечения включают трудоемкий процесс управления версиями ведущегося программного проекта, используя для этого специальное программное обеспечение VCS, даже незначительный прогресс которого способен повысить эффективность работы подавляющего количества IT-компаний.

Проблема систем управления версиями

На практике используются два основных рабочих цикла члена команды разработчиков программного обеспечения.

“Централизованный” рабочий цикл, в основе которого клиент-серверная архитектура, обычно выглядит следующим образом:

1. В начале работы программист вносит изменения в основную версию программного продукта, который находится в виде рабочей копии на его компьютере;
2. Модификация проекта;

[3] http://web.archive.org/web/20100322230458/http://www.lucasward.net/2010/02/maturity-model-for-source-control-scmm.html

[4] С. Макконнелл «Остаться в живых. Руководство для менеджеров программных проектов», «Питер», 2006.

3. Завершив очередной этап работы разработчик передает измененные файлы на сервер, в основную ветвь либо в отдельную ветвь разработки отдельного задания. При этом обычно программное обеспечение сервера, ответственное за управление версиями (VCS) перед фиксацией изменений требует от разработчика обновить свою локальную копию с учетом изменений, которые внесли другие члены команды. Если это не произойдет, то возможно возникновение конфликтов с основной версией. Кроме этого VCS сохраняет на сервере все версии проекта и его файлов в специальном репозитории. Это позволяет вернуться к предыдущей стадии разработки в случае необходимости.

“Распределенный” рабочий цикл отличается от “централизованного” только тем, что полная копия репозитория проекта хранится на каждом компьютере проекта при этом сервер не используется. Это означает, что рабочий цикл выглядит так:

1. Программист вносит изменения в основную версию программного продукта, который находится в виде рабочей копии на его компьютере;
2. Модификация проекта;
3. Завершив очередной этап работы разработчик передает измененные файлы в репозиторий, в основную ветвь либо в отдельную ветвь разработки отдельного задания, VCS перед фиксацией изменений требует от разработчика обновить свою локальную копию не только программного продукта, над которым работает команда, но полную копию репозитория.

Абсолютное большинство современных систем управления версиями ориентировано, в первую очередь, на проекты разработки программного обеспечения, в которых основным видом содержимого файла является текст. Соответственно, механизмы автоматического слияния изменений ориентируются на обработку текстовых файлов, то есть файлов, содержащих текст, состоящий из строк буквенно-цифровых символов, пробелов и табуляций, разделенных символами перевода строки.

При определении допустимости слияния изменений в пределах одного и того же текстового файла работает типовой механизм построчного сравнения текстов (примером его реализации является системная утилита GNU diff), который сравнивает объединяемые версии с базовой и строит список изменений, то есть добавленных, удаленных и измененных наборов строк. Минимальной единицей данных для этого алгоритма является строка, даже самое малое отличие делает строки различными.

Те найденные наборы изменённых строк, которые не пересекаются между собой, считаются совместимыми и их слияние делается автоматически.[5][3]

Если в сливаемых файлах находятся изменения, затрагивающие одну и ту же строку файла программного кода, это приводит к синтаксическому конфликту.

Также при слиянии может возникнуть семантический конфликт. Например, после слияния двух версий программы может возникнуть новая версия, текст которой будет верен синтаксически, но не будет удовлетворять требованиям, поставленным к работе программы. Такие файлы могут быть объединены только вручную.

Верхняя оценка количества всех возникающих конфликтов в этом примере равна произведению количества изменений в каждой из сливаемых версий. В этом оценочном расчете мы считаем, что каждое изменение конфликтует с каждым и слияния происходят попарно. Если в проекте участвуют не два программиста, а больше, то верхняя оценка количества возможных конфликтов многократно повышается и вероятность возникновения неразрешимого в автоматическом режиме значительно увеличивается.

Из практики известно что количество времени, затраченное на слияние двух версий программ, или трудоемкость, возрастает а) прямо пропорционально количеству конфликтов и б) экспоненциально в зависимости от размера изменений. Размер изменений оценивается по количеству строк кода программы.

В современных IT-компаниях с ростом рабочего времени, затраченного на слияние версий, пытаются бороться с помощью уменьшения количества изменений, требуя от программистов-разработчиков как можно чаще проводить слияния, чтобы уменьшить количество изменений и их размер. Так, например, в обзоре практик программирования "Continuous Delivery: A Maturity Assessment Model" (March 2013) приведен ответ разработчиков программного обеспечения на вопрос; “Как часто команда программистов устоявшийся процесс слияния версий разрабатываемого продукта?” (п.3 на рис.1 и рис. 3). Из диаграммы видно, что 28% опрошенных команд проводят слияние версий раз день и чаще.

Но из тех же диаграмм (рис. 2 и 3) видно, что более 30% команд редко используют программное обеспечение по управлению версиями, либо у них нет установившейся практики слияния версий. Это означает, что в большом количестве случаев

[5] Система управления версиями. Слияние версий

проводится слияние значительных изменений, при которых неизбежно возникают и синтаксические, и семантические конфликты, и, следовательно, каждое слияние требует большой объема ручной работы[6].

Рис. 3 Использование командами-разработчиками программного обеспечения устоявшейся практики слияния версий разрабатываемого продукта

Таким образом, проблема, одно из возможных решений которой представлено в данной работе, заключается в том, что специализированные программные решения по управлению версиями и соответствующие внутрикорпоративные модели работы, направленные на то, чтобы уменьшить трудозатраты при слиянии новых версий, разрабатываемого большим коллективом разработчиков, программного продукта, позволяют решить значительную долю возникающих конфликтов, но возникающие при слиянии версий семантические и синтаксические конфликты с помощью существующего программного обеспечения разрешить не удается, поэтому они разрешаются только “вручную” при значительных затратах рабочего времени.

[6] По оценке B. Ritcher на слияние версий без использования надежных программных инструментов уходит до недели рабочего времени команды программистов https://www.guiffy.com/SureMergeWP.html

Цель работы

Для того чтобы сформулировать цель работы, рассмотрим характерные особенности процесса слияния двух файлов и возможные конфликты, которые при этом могут проявиться.

Пусть нам надо сравнить перед слиянием два текстовых файла содержащих код программы. Как уже было отмечено выше стандартный алгоритм (например, GNU DIFF) сравнивает файлы построчно, игнорируя пробелы. Изменения в пределах одного текстового файла, сделанные в разных версиях, могут быть объединены, если они находятся в разных местах этого файла и не пересекаются. В этом случае в объединенную версию вносятся все сделанные изменения. Действительно, будем считать, что изменения были сделаны двумя квалифицированными программистами в непересекающихся частях двух копий одного и того же файла, то есть каждая из программ по отдельности проходит полный набор тестов. Новый файл (или модифицированная программа), созданный по стандартному алгоритму слияния, будет содержать оба изменения. При этом полученная таким способом новая программа может работать неправильно, например, из-за того, что программисты, работая независимо друг от друга, не согласовали передачу данных от одной модифицированной части к другой. Устранение такого конфликта происходит в результате “ручного” анализа нового текста программы.

Но возможна ситуация, когда при слиянии двух версий программы сделанные в ней изменения пересекаются между собой, то есть утилита GNU DIFF находит одну или несколько строк, которые изменены обоими программистами. Она расценивает это как конфликт, который невозможно устранить автоматически, операция слияния прекращается, утилита обращается к разработчику с запросом на устранение конфликта. В этой ситуации у разработчика есть следующий выбор:

1. Отредактировать строку с учетом обоих изменений;
2. Выбрать в ручном режиме один из двух вариантов модифицированной строки;
3. Предложить утилите механически объединить два варианта. Например, у конфликтной строки начало изменено первым программистом, середина не тронута, а конец - вторым программистом.

Первый полностью ручной вариант весьма трудозатратен, поэтому два последних полуавтоматических варианта разрешения конфликта очень привлекательна для

разработчика, особенно, при большом количестве конфликтов. К сожалению, операция механического объединения двух вариантов часто приводит к некорректному синтаксису программы. Нарушение синтаксиса осложняют разрешение конфликтов и отладку программы.

Во-первых, программу с нарушенным синтаксисом невозможно запустить, и соответственно, невозможно проверить корректность поведения программы с помощью автоматических тестов. На практике при возникновении синтаксического нарушения в модифицированном файле, разработчик вынужден исправлять синтаксические ошибки, чтобы сначала запустить программу, а затем пройти тестирование и выявить семантические ошибки, которые исправляются на основе результатов тестирования.

Во-вторых, процесс исправления синтаксических ошибок плохо поддается оптимизации. Наиболее распространенным способом устранения синтаксических ошибок является запуск компилятора или любого другого анализатора синтаксиса. После запуска анализатор выдает список ошибок синтаксиса, которые необходимо исправить. Так как большинство анализирующих инструментов обрабатывают текст до первой ошибки, возможности разработчика исправлять несколько ошибок за раз весьма ограниченны. Это приводит к многократным перезапускам анализирующего инструмента, что может выражаться в значительные временные затраты, в зависимости от скорости анализа.

Таким образом, минимизация синтаксических конфликтов и упрощение их разрешения могут привести к значительной экономии времени программистов[7]. Поэтому целью данной работы является разработка продвинутого алгоритма слияния синтаксических конфликтов с использованием анализатора синтаксиса.

[7] См. Здесь https://www.guiffy.com/SureMergeWP.html один из примеров сокращения рабочего времени, затраченного на слияние, после внедрения усовершенствованного алгоритма

Постановка задачи

- Найти перспективную категорию синтаксических конфликтов. Требования:
    - Такие конфликты достаточно часто встречаются в практической разработке ПО
    - Разрешение подобных конфликтов содержит рутинную часть, поддающуюся автоматизации
- Составить набор алгоритмических тестов
    - Рассмотреть C-подобные языки программирования: C/C++, Objective-C, Swift
    - Сосредоточится на частых блоковых конструкциях: цикл for, цикл while, цикл do-while, условие if
    - Покрыть различные стили форматирования кода
- Разработать алгоритм удовлетворяющий 95% тестовых случаев
    - Алгоритм должен интегрироваться с промышленными программами разрешения конфликтов слияния (например SourceTree)
- Реализовать прототип алгоритма
    - Прототип должен функционировать на операционной системе Mac OS X
    - Код прототипа должен легко переносится на операционные системы Linux и Windows
- Разработать инфраструктуру для применения алгоритма в проектах
    - Поддержать проекты с множественными файлами и сложной структурой
    - Добавить интерфейс для автоматического разрешения конфликтов в проекте
    - Добавить возможность компиляции проекта в процессе разрешения конфликтов

Обзор

Теоретические исследования

[1] Bill Ritcher "A Trustworthy 3-Way Merge" https://www.guiffy.com/SureMergeWP.html

В статье кратко описаны известные алгоритмы слияния оригинального текстового файла и двух его модификаций. Автор указывает на их недостатки и нежелательные последствия, которые возникают при их использовании: а) ограничение возможностей параллельной работы команды программистов, б) большой объем операций слияния, которые слишком часто должны делать разработчики блюз, и/или С) дополнительные трудозатраты на ручную правку результатов слияния. Автор считает, что существующие алгоритмы слияния значительно снижают продуктивность процесса разработки.

В статье описан усовершенствованный алгоритм слияния трех файлов, обсуждаются многие проблемы, которыми сопровождается внедрение и использование алгоритма слияния, а также набор тестов, созданный для того, чтобы выявить проблемы слияния. В конце статьи показано, каким образом были решены проблемы слияния в новом программном продукте Guiffy SureMerge и описано его внедрение в реальный рабочий процесс. В статье отмечено, что в результате использования усовершенствованного алгоритма время слияния сократилось до двух раз.

[2] Robin La Fontaine "Merging XML files: a new approach providing intelligent merge of XML data sets" https://www.deltaxml.com/support/documents/articles-and-papers/merging-xml-files.pdf

В статье обосновывается необходимость мощного инструмента для слияния XML-файлов Слияния XML- это чрезвычайно сложный процесс, но часто необходимо консолидировать потоки данных из гетерогенных систем, или синхронизировать представления (структуру) XML-фрагментов, которые являются составными частями большого документа.

В статье описан разработанный авторами методы, поддающиеся автоматизации, которые задают и контролирует процесс слияния. Продемонстрирована возможность описанными методы обеспечить последовательное, гибкое и устойчивое решением

проблемы слияния XML-файлов, а также интеграцию их в информационные процессы при практически неограниченных возможностях пользовательской настройки.

Авторы утверждают, что благодаря описанным методам, интеллектуальное слияние наборов данных XML становится реальностью.

Фактически, в статье предложен подход к слиянию, основанный на использовании промежуточного XML-файла, который содержит данные двух сливаемых файлов в виде формальной структуры, по которой можно однозначно идентифицировать общие данные для обоих файлов и данные, которые является уникальным для каждого из файлов. Достоинство использования такого промежуточного файла заключается в том, что многие конфликты, которые обычно возникают, когда объединяются XML-данные могут быть выявлены и устранены. Разрешение этих конфликтов является ключом к достижению успешного слияния.

Также в статье рассматриваются проблемы работы с реальными данными с точки зрения контроля корректности связей (структуры) между данными внутри файла. Поиск корректных связей является необходимым подготовительным этапом, обеспечивающим выполнение успешного слияния.

В статье показано, что новые методы могут использоваться для автоматической генерации XML-файлов из базы данных, в которой хранятся, например, XML-схемы или файлы SVG.

[3] J. W. Hunt, M. D. McIlroy "An Algorithm for Differential File Comparison", Department of Electrical Engineering, Stanford University, Stanford, California Bell Laboratories, Murray Hill, New Jersey 07974 http://www.cs.dartmouth.edu/~doug/diff.pdf

В статье описано использование и алгоритм работы утилиты DIFF. Цель использования утилиты - построить список строк, которыми отличаются два текстовых файла. Центральный алгоритм утилиты, фактически, представляет из себя поиск самой длинной неизменной (общей) последовательности в двух файлах. Таким образом она ищет строки, которые не отличаются у двух файлов. Алгоритм достаточно эффективный в смысле времени работы и использования дискового пространства.

[4] Sanjeev Khanna , Keshav Kunal , and Benjamin C. Pierce "A Formal Investigation of Diff3", University of Pennsylvania
http://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf

В статье описаны свойства алгоритм diff3 . Основным результатом статьи является, что использование алгоритма DIFF3 при объединении двух текстовых файлов их измененные фрагменты никогда не конфликтуют, если они “хорошо разделены”.

[5] Д.В. Кознов, Е.В. Ларчик, М.М. Плискин, Н.И. Артамонов "О Задаче Слияния Карт Памяти (Mind Maps) При Коллективной Разработке", ПРОГРАММИРОВАНИЕ, 2011, No 6, с. 56–66

В статье представлено решение задачи слияния карт памяти (Mind Maps) в процессе коллективной Интернет-разработки. Задача решена на основе известного алгоритма слияния XML-файлов 3DM, модифицированного в соответствии с особенностями задачи: (1) необходимость двух режимов работы – по умолчанию и с предоставлением пользователю возможности самостоятельно разрешать конфликты; (2) необходимость максимального сохранения изменений, сделанных в поддеревьях при разрешении конфликтов Update/Delete; (3) несимметричность сливаемых деревьев (серверная копия считается более приоритетной); (4) необходимость применения edit-скрипта к исходной версии для сохранения истории изменений; (5) наличие уникальных идентификаторов у узлов сливаемых деревьев (то есть более упрощенная процедура идентификации) и потребностью ” разносить“ узлы с одним и тем же значением идентификатора, но сильно разным содержанием.

Программные продукты

1. Compare++ http://cmpp.coodesoft.com/

Проприетарная утилита для поиска различий между двумя папками или файлами. Тестирование оказалось невозможным по причине закрытости кода. Поддерживает только систему Windows. Не интегрируется с существующими утилитами так как обладает собственным графическим интерфейсом.

2. Pretty Diff http://prettydiff.com/

Утилита для сравнения программного кода с открытым исходным кодом. Поддерживает синтаксис JavaScript и HTML. Работает в режиме онлайн. Утилита способна минимизировать и форматировать код без потери функциональности, а также сравнивать два файла.

3. SemanticMerge: https://www.semanticmerge.com/

Проприетарная утилита для поиска различия в двух программах и их слияния на основе анализа языка. Поддерживает синтаксис языков Java, C#, C, C++, JS. SemanticMerge анализирует код программы и ее структуру. Обрабатывает структурные конфликты связанные с сигнатурами методов, но использует традиционные алгоритмы слияния для разрешения конфликтов внутри тел функций.

4. OxygenXml oxygenxml.co

Мощный инструмент для редактирования XML. Умеет сравнивать директории, архивы, файлы, изображения. Может проводить операцию слияния, но только в “ручном” режиме.

Техническая документация

1. Ветвление в Git - Основы ветвления и слияния
2. Continuous Delivery: A Maturity Assessment Model
3. Система управления версиями (wikipedia)
4. http://clang.llvm.org/docs/IntroductionToTheClangAST.html
5. http://clang.llvm.org/docs/LibASTMatchers.html
6. Документация по работе с git-merge. https://git-scm.com/docs/git-merge
7. Документация по интеграции с git-mergetool. https://git-scm.com/docs/git-mergetool
8. Официальный сайт проекта Clang и LLVM. Доступно: http://clang.llvm.org/
9. Неофициальная инструкция по интеграции с Python API для Clang. Доступно: http://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang/

Описание алгоритма

Схема алгоритма

Для решения поставленной задачи улучшения эргономичности слияния конфликтов с пересечением синтаксических блоков был разработан алгоритм расширения синтаксических блоков.

Элементарной операцией алгоритма является перенесение границ конфликта. В случае расширения зоны конфликта, например вниз, нижняя граница переносится на строку ниже так чтобы строка за старой границей конфликта оказалась в зоне конфликта. При этом новая строка конфликта дублируется как принадлежащая обеим версиям файла. Схожая последовательность действий позволяет сместить верхнюю границу и увеличить зону конфликта вверх. Заметим что такие операции расширения зоны конфликта всегда правомерны, так как из преобразованного конфликта можно восстановить исходные конфликтующие версии файлов.

Описанный алгоритм оперирует со строками, но ничего принципиально не мешает уменьшить гранулярность алгоритма до отдельных символов. Оперирование строками было предпочтено из соображений упрощения реализации.

Алгоритм расширения синтаксических блоков весьма прямолинеен. При обнаружении синтаксической конструкции, граница которой пересекает зону конфликта, алгоритм расширяет зону конфликта чтобы включить и вторую границу конструкции. Тогда конфликт перестает пересекать такую синтаксическую конструкцию, что и решает поставленную задачу.

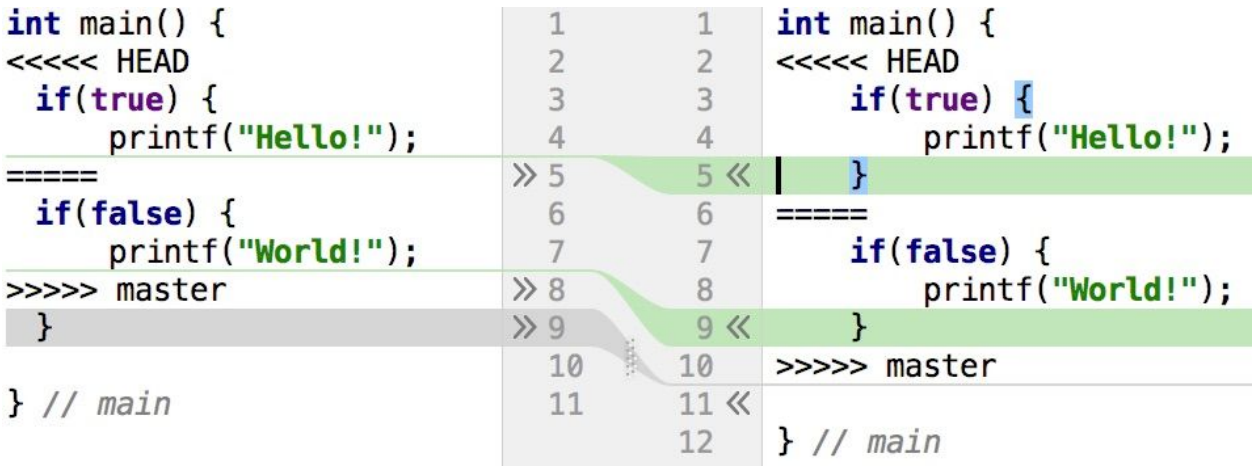

Рис. 4 Пример работы алгоритма

На рис.4 выше можно увидеть пример работы алгоритма. Левая часть диаграммы представляет собой исходное представление конфликта, правая часть представляет результат работы алгоритма. Как видно из сравнения по строкам, исходный конфликт не включал в себя замыкающую фигурную скобку на 9-ой строке. Согласно алгоритму, конфликт был расширен вниз до включен замыкающей скобки. Так как 9-ая строка была добавлена в зону конфликта, она была продублирована на стоках 5 и 9 для сохранения корректности конфликтующих изменений.

Версия 1 "Изначальная"

Алгоритм преобразует конфликты в каждом исходном файле в соответствие со схемой, описанной выше.

На вход алгоритма подается дерево синтаксического разбора (AST, Abstract Syntax Tree) исходного файла и список конфликтов слияния внутри этого файла. Из всех вершин AST выбираются вершины, тип которых соответствует типу конструкций условий if. Далее для каждой такой вершины выполняются две процедуры по обработке синтаксических блоков с конфликтами в конце и начале блока соответственно. Каждой процедуре передается текущий блок и список всех конфликтов в файле. Результатом работ процедур является изменения соответствующих конфликтов в соответствии со схемой, описанной выше.

На рис. 5:

- AST blocks - дерево синтаксического разбора исходного файла
- file conflicts - список конфликтов слияния внутри исходного файла
- block.kind - тип вершины AST данного блока
- IF_STMT - тип вершины AST, соответствующий конструкции условия if

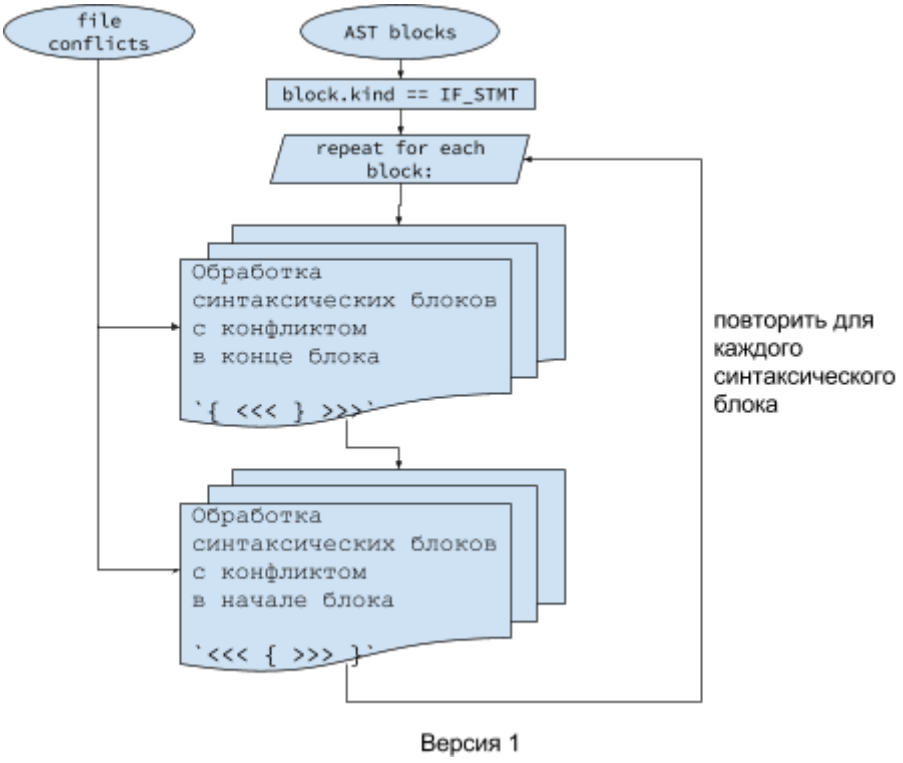

Рис. 5 Схема работы основного алгоритма версии 1

Процедура обработки синтаксических блоков с конфликтами в конце блока устроена следующим образом. Из всех конфликтов в файле отбираются конфликты, которые пересекаются с концом данного блока. Пересечение конфликта и блока проверяется условием того что начало блока находится левее начала конфликта, которое находится левее конца блока, который находится левее конца конфликта. Из всех конфликтов, пересекающих данный блок, берется первый. На практике правомерно брать только один конфликт, так как конфликты не могут быть расположены слишком быстро друг к другу и не быть распознанными как один больший конфликт.

Далее вычисляется количество строк, на которой надо изменить зону конфликта. Это количество строк равно расстоянию между началом блока и началом конфликта. Математически это выражается как разность номера строки начала конфликта и номера строки начала блока с добавлением единицы.

После этого зона конфликта расширяется на нужное количество строк вверх. Это происходит в две операции: сначала кусок текста перед зоной конфликта обрезается с хвоста на нужное количество строк, после чего обрезанный кусок текста добавляется к голове конфликта.

На рис. 6:

- block - данный синтаксический блок
- conflict - рассматриваемый конфликт
- file_bit - кусок текста перед рассматриваемым конфликтом
- block.start, block.end - номера строк начала и конца блока соответственно
- conflict.start, conflict.end - номера строк начала и конца конфликта соответственно
- num - количество строк, на которое требуется расширить conflict
- chunk - кусок текста, отсекаемого от file_bit

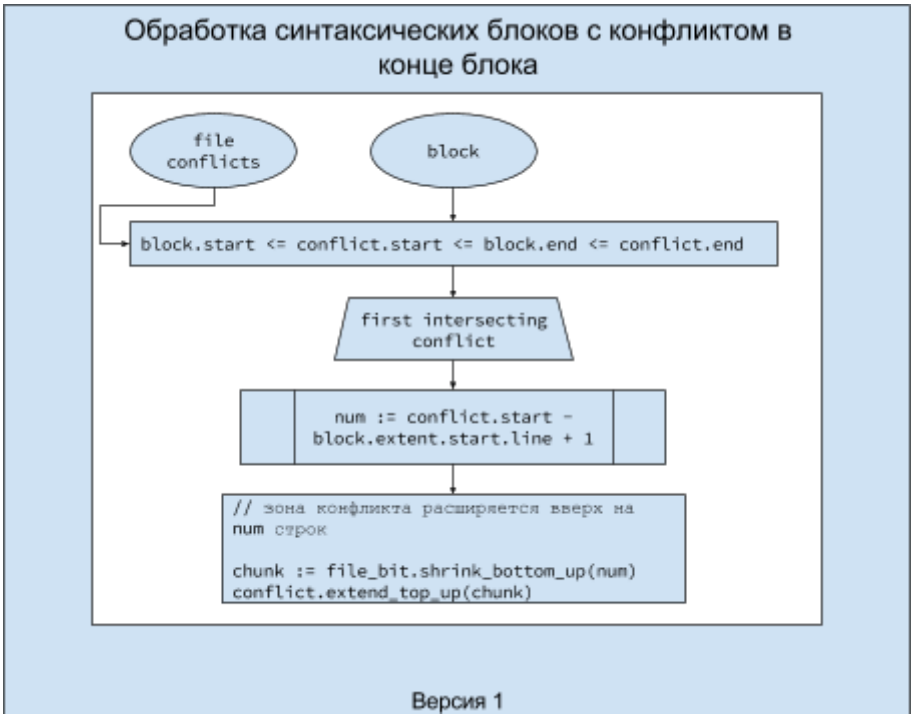

Рис. 6 Схема работы процедуры обработки синтаксических блоков с конфликтами в конце блока версии 1

Процедура обработки синтаксических блоков с конфликтами в начале блока устроена сходным образом с процедурой обработки синтаксических блоков с конфликтами в конце блока. Из всех конфликтов в файле отбираются конфликты, которые пересекаются с началом данного блока. Пересечение конфликта и блока проверяется условием того что начало конфликта находится левее начала блока, которое находится левее конца конфликта, который находится левее конца блока. Из всех конфликтов, пересекающих данный блок, берется первый. На практике правомерно брать только один конфликт, так как конфликты не могут быть расположены слишком быстро друг к другу и не быть распознанными как один больший конфликт.

Далее вычисляется количество строк, на которой надо изменить зону конфликта. Это количество строк равно расстоянию между концом блока и концом конфликта. Математически это выражается как разность номера строки конца блока и номера строки конца конфликта.

После этого зона конфликта расширяется вниз на нужное количество строк. Это происходит в две операции: сначала кусок текста после зоны конфликта обрезается с головы на нужное количество строк, после чего обрезанный кусок текста добавляется к хвосту конфликта.

На рис. 7:

- block - данный синтаксический блок
- conflict - рассматриваемый конфликт
- file_bit - кусок текста перед рассматриваемым конфликтом
- block.start, block.end - номера строк начала и конца блока соответственно
- conflict.start, conflict.end - номера строк начала и конца конфликта соответственно
- num - количество строк, на которое требуется расширить conflict
- chunk - кусок текста, отсекаемого от file_bit

Рис. 7 Схема работы процедуры обработки синтаксических блоков с конфликтами в начале блока версии 1

Версия 2 "С исправленной нумерацией строк"

После тестирования первой версии алгоритма выяснилось что почти все базовые тесты дают неверный результат. Это было связано с тем что парсел CLang LLVM начинает нумерацию строк файла с единицы, а не с нуля, как предполагалось изначально.

В процедуре обработки синтаксических блоков с конфликтом в конце блока необходимые изменения состояли из вычитания единицы из старого значения num и замены нестрогого отношения между началом блока и началом конфликта на строгое отношение порядка.

Рис. 8 Схема работы процедуры обработки синтаксических блоков с конфликтами в конце блока версии 2

В процедуре обработки синтаксических блоков с конфликтом в начале блока необходимые изменения состояли из прибавления единицы к старому значению num и замены нестрогого отношения между началом блока и концом конфликта на строгое отношение порядка.

Рис. 9 Схема работы процедуры обработки синтаксических блоков с конфликтами в начале блока версии 2

После описанных изменений все базовые тесты кроме test_parse стали давать верные результаты. Всего 15 базовых тестов дало положительный результат, один базовый тест дал отрицательный результат, 70 алгоритмических тестов дало положительный результат и 67 алгоритмических тестов дало отрицательный результат.

Версия 3 "С исправленными нейтральными случаями"

После тестирования второй версии стало понятно, что она изменяет некоторые нейтральные тестовые случаи, которые не содержат пересечения синтаксических блоков и конфликтов, и соответственно, должны оставаться неизменными.

Для исправления этой ошибки в процедуре обработки синтаксических блоков с конфликтом в конце блока старое условие отбора конфликтов было заменено на условие того что начало блока строго левее начала конфликта, которое левей конца блока, которое строго левее конца конфликта. Процедура обработки синтаксических блоков с конфликтом в начале блока не потребовала исправлений так описанная ошибка происходила только в тестах с с конфликтами в начале синтаксических блоков.

Рис. 10 Схема работы процедуры обработки синтаксических блоков с конфликтами в конце блока версии 3

После описанных изменений все базовые тесты стали давать верные результаты. Также 5 алгоритмических тестов стали проходить тестирование. Всего 16 базовых тестов дало положительный результат, 75 алгоритмических тестов дало положительный результат, и 62 алгоритмических тестов дало отрицательный результат.

Версия 4 "Корректно обрабатывающая ветки ветвления else"

После тестирования третьей версии обнаружилось что блоки else в условиях if-else обрабатывались некорректно. Эта ошибка была вызвана тем в терминах библиотеки CLang LLVM синтаксическая конструкция if состоит только из основной ветки, а опциональная ветка else представлена как ребенок основной конструкции.

Для устранения этой ошибки в список отфильтрованных вершин AST были добавлены все дети этих вершин, что в случае конструкции условия if являлось прямой и альтернативной ветками ветвления. При отсутствии альтернативной ветки else эта операция ничего не добавляла.

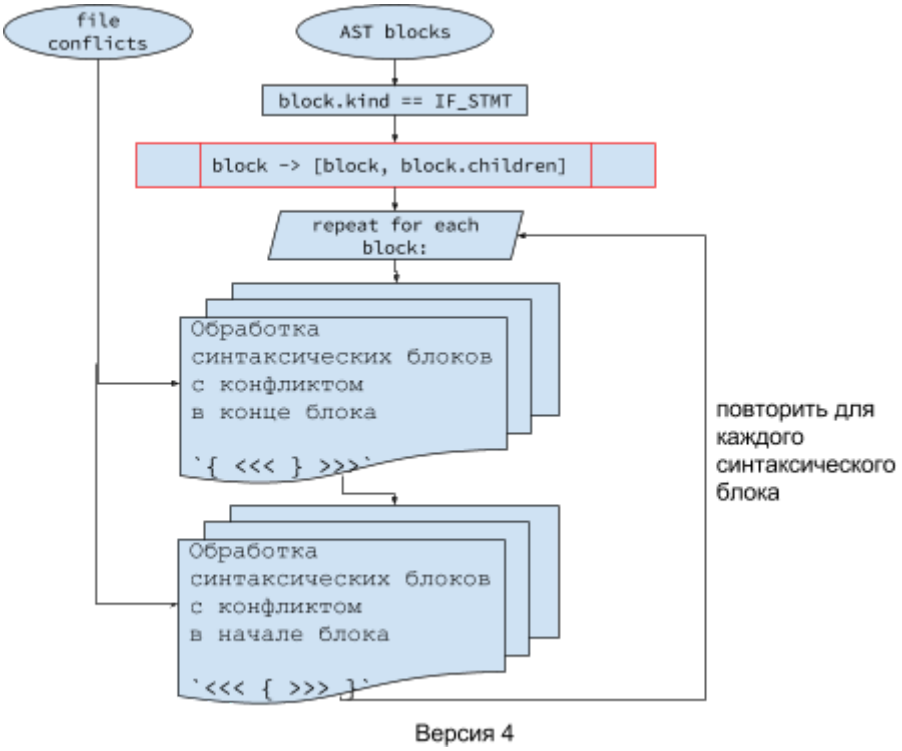

Рис. 11 Схема работы основного алгоритма версии 4

После описанных изменений один новый алгоритмических тесто стали проходить тестирование. Также все базовые тесты продолжили давать верные результаты. Всего 16 базовых тестов дало положительный результат, 76 алгоритмических тестов дало положительный результат, и 61 алгоритмических тестов дало отрицательный результат.

Версия 5 "С поддержкой циклов FOR"

Четвертая версия алгоритма успешно проходила тесты на обработку конструкции if-else, но давала неверные результаты на тестах, связанных с другими синтаксическими конструкциями.

Для добавления поддержки циклов for условие фильтрации блоков AST было расширено чтобы включить конструкции цикла for.

На рис. 12:

- FOR_STMT - тип вершины AST, соответствующий конструкции цикла for

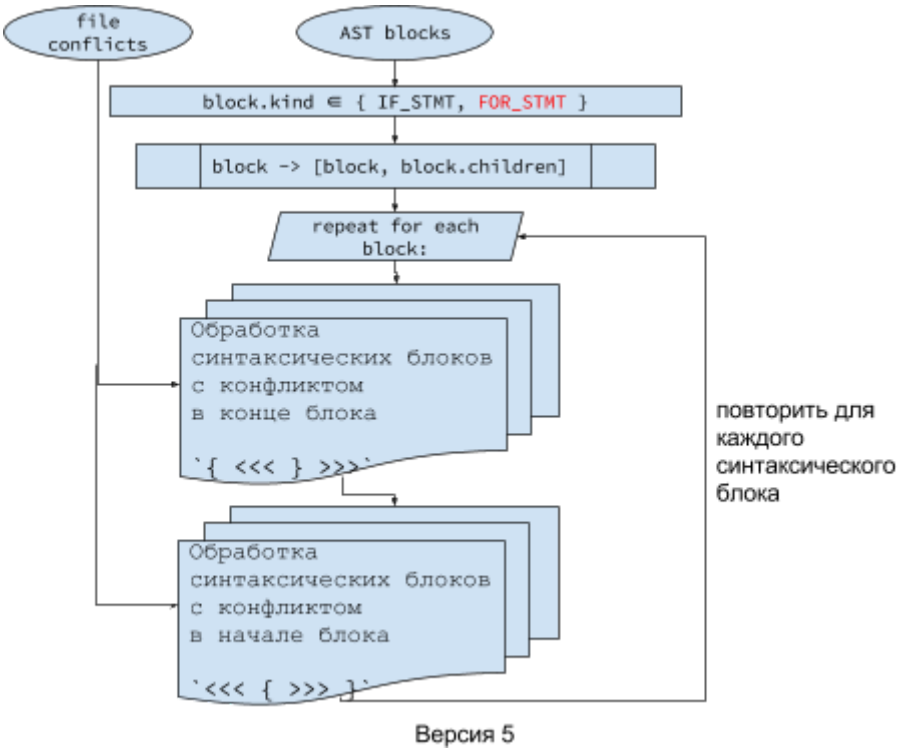

Рис. 12 Схема работы основного алгоритма версии 5

После описанных изменений 21 новых алгоритмических тестов стали проходить тестирование. Также все базовые тесты продолжили давать верные результаты. Всего 16 базовых тестов дало положительный результат, 97 алгоритмических тестов дало положительный результат, и 40 алгоритмических тестов дало отрицательный результат.

Версия 6 "С поддержкой циклов WHILE"

Пятая версия алгоритма успешно проходила тесты на обработку конструкции if-else и циклов for, но давала неверные результаты на тестах, связанных с другими синтаксическими конструкциями.

Для добавления поддержки циклов while условие фильтрации блоков AST было расширено чтобы включить конструкции цикла while.

На рис. 13:

- WHILE_STMT - тип вершины AST, соответствующий конструкции цикла while

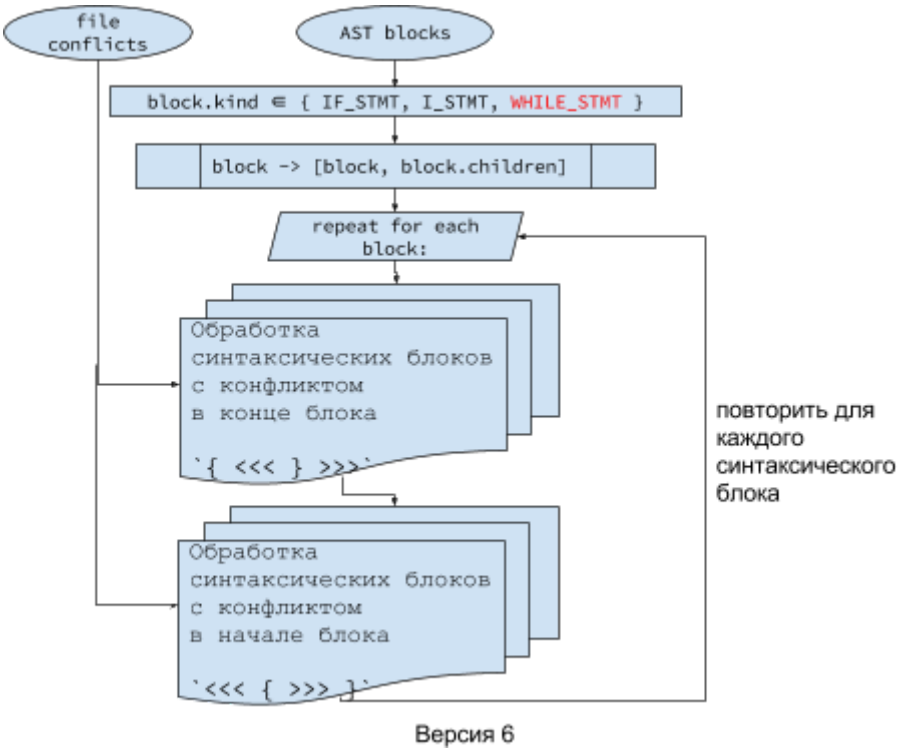

Рис. 13 Схема работы основного алгоритма версии 6

После описанных изменений 20 новых алгоритмических тестов стали проходить тестирование. Также все базовые тесты продолжили давать верные результаты. Всего 16 базовых тестов дало положительный результат, 117 алгоритмических тестов дало положительный результат, и 20 алгоритмических тестов дало отрицательный результат.

Версия 7 "С поддержкой циклов DO-WHILE"

Шестая версия алгоритма успешно проходила тесты на обработку конструкции if-else, циклов for и циклов while, но давала неверные результаты на тестах на обработку циклов do-while.

Для добавления поддержки циклов do-while условие фильтрации блоков AST было расширено чтобы включить конструкции цикла do-while.

На рис. 14:

- DO_STMT - тип вершины AST, соответствующий конструкции цикла do-while

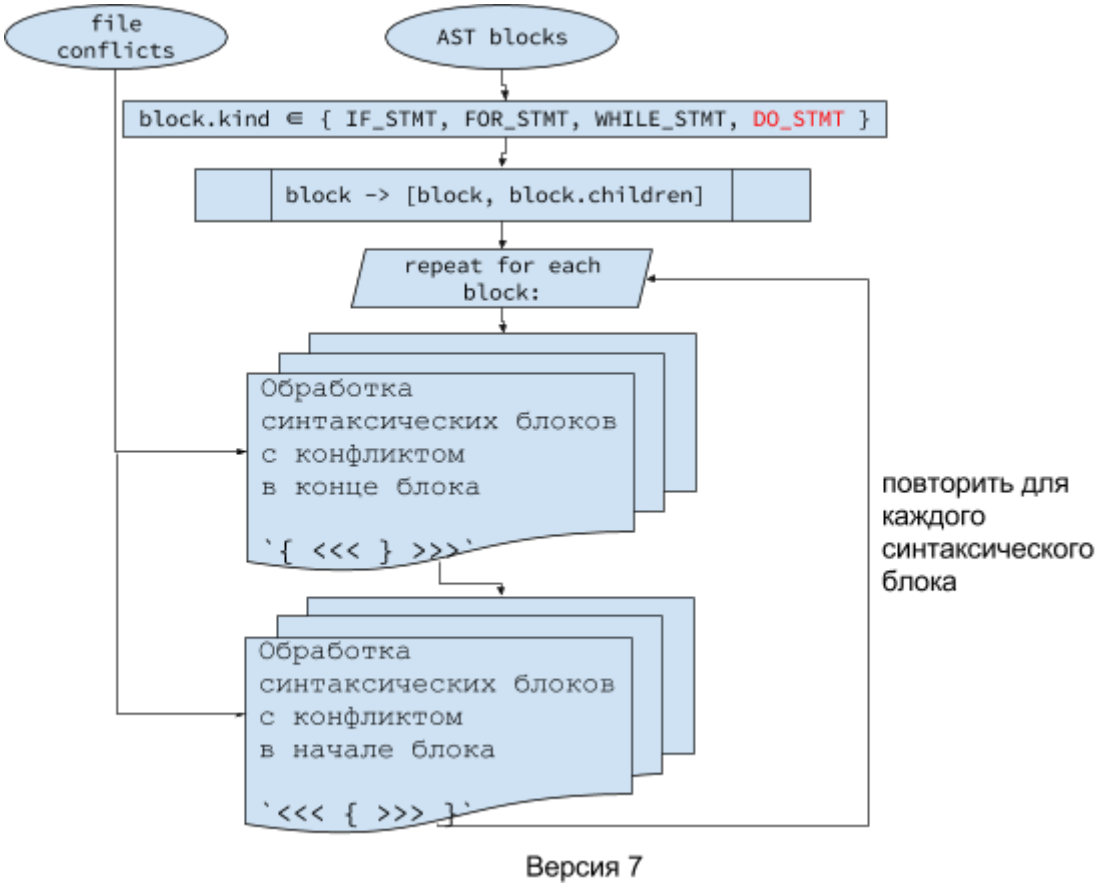

Рис. 14 Схема работы основного алгоритма версии 7

После описанных изменений 20 новых алгоритмических тестов стали проходить тестирование, что составило полный тестовый набор. Также все базовые тесты продолжили давать верные результаты. Всего 16 базовых тестов дало положительный результат и 137 алгоритмических тестов дало положительный результат.

Тестирующий стенд

Процесс тестирования

Апробация и тестирование алгоритма были выполнены при помощи прототипа реализации с кодовым названием MergeTool.

Прототип MergeTool реализован на языке Python 3.5 с использованием библиотеки парсеров CLang LLVM.

Библиотека CLang LLVM была выбрана как лидирующая реализация парсеров для языков семейства C: C/C++, Objective-C, Swift. Так как CLang LLVM предоставляет официальный интерфейс для языков Python и C, для разработки был использован язык Python, скорость прототипирования на котором считается выше, чем скорость прототипирования на C.

Прототип поддерживает синтаксис языка C, потому как C является имеет наиболее простой синтаксис из всех языков семейства C. Поскольку для части функционала прототипа необходима возможность компилирования файлов, была реализована интеграция с системой сборки GNU CMake.

Результаты тестирования

Прототип MergeTool был протестирована на тестовом наборе из 16 базовых тестов 137 алгоритмических тестов, его алгоритма была отлажена в течении 7 последовательных версий алгоритма.

В таблице 1 строки пронумерованы от v1 до v7 и относятся к результатам тестирования версий алгоритма от первой до седьмой соответственно. Столбцы таблицы относятся к базовому, алгоритмическому и общему тестовым наборам соответственно. Значения ячеек отображаю количество тестов с неверным результатом.

version	Basic tests	Algorithmic tests	All tests
# of tests	16	137	153
v1	14	137	151
v2	1	67	68
v3	0	62	62
v4	0	61	61
v5	0	40	40
v6	0	20	20
v7	0	0	0
IDEAL	0	0	0

Таблица 1. Необработанные результаты тестирования версий алгоритма MergeTool

На рис. 15 наглядно видно постепенное улучшение работы алгоритма. Первая версия алгоритма удовлетворяла только 2 базовым тестом, тогда как финальная 7 версия удовлетворяет полному набору из 16 базовых тестов и 137 алгоритмических тестов. Финальная версия алгоритма покрывает все тесты, разработанные для валидации его работы, таким образом выполняя требование удовлетворению более 95% тестов.

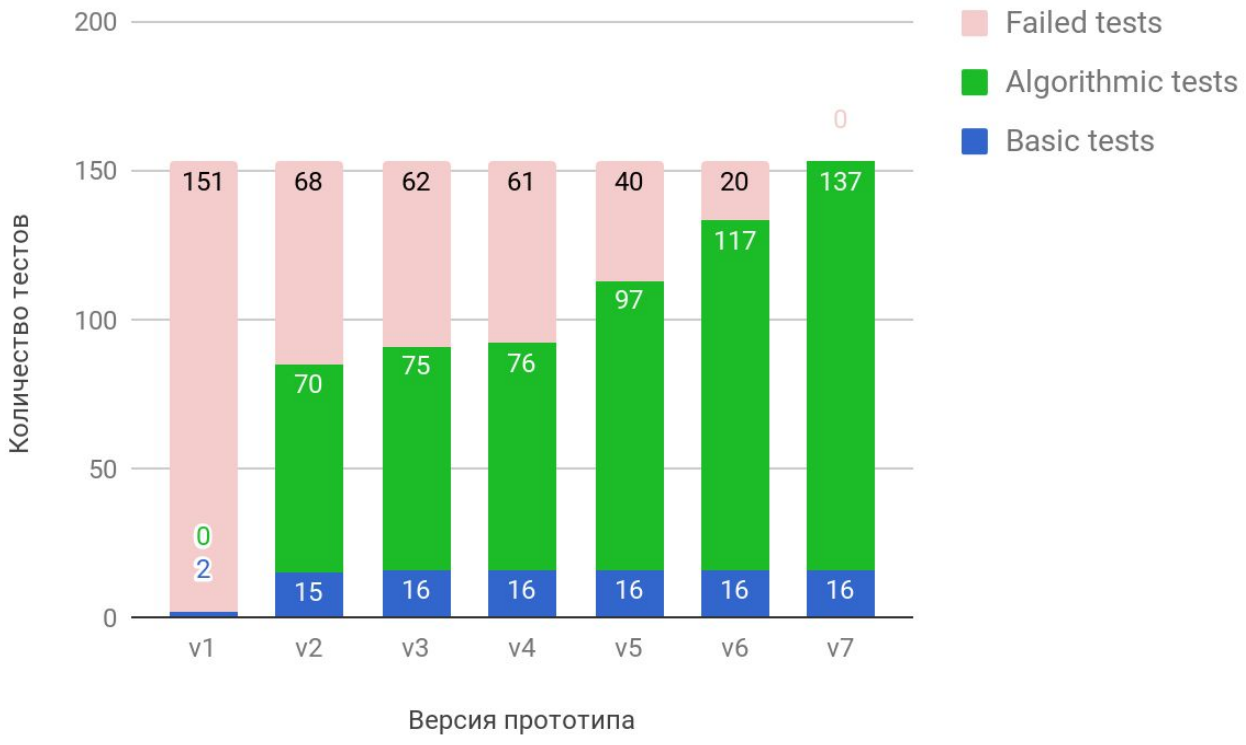

Рис. 15 Результаты тестирования версий алгоритма MergeTool

Тестовые наборы

В разработке и тестировании MergeTool были использованы два специальных набора тестов. Первый набор из 16 базовых тестов (Basic tests) проверяет базовую функциональность и инфраструктуру MergeTool, работу внутренних функций и разрешение простейших ситуаций. Второй набор из 137 алгоритмических тестов (Algorithmic tests) всесторонне тестируется сам алгоритм обработки конфликтов.

Алгоритмические тесты

Алгоритмические тесты проверяют работоспособность основного алгоритма MergeTool. На данный момент алгоритм поддерживает четыре различных конструкций языка C: цикл for, цикл while, цикл do-while и конструкцию if-else. Соответственно все алгоритмические тесты разбиты на пять файлов, по одному файлу на каждую из четырех поддерживаемых конструкций языка C, а также один дополнительный файл на смешанные тесты, включающие несколько конструкций.

Тестовые файлы для циклов for, while и do-while имеют схожую структуру. Каждый файл содержит три класса тестов: тесты которые не должны меняться при работе алгоритма, тесты, содержащие в себе единственную конструкцию, и тесты, содержащие несколько последовательных или вложенных конструкций.

Конструкция if-else немного отличается от циклов наличием опционального блока else, поэтому ее файл содержит дополнительный четвертый класс с тестами, проверяющими поведение алгоритма на границе блоков if и else.

Тестовый файл со смешанными конструкциями проверяет взаимодействие цикловых конструкций и конструкции if-else, поэтому он содержит 3 класса: if-else и цикл for, if-else и цикл while, if-else и цикл do-while.

Классы тестовых случаев, не меняющихся после обработки, содержат три группы тестов: конфликт находится сразу после основной конструкции, конфликт находится внутри основной конструкции, и конфликт находится непосредственно перед основной конструкцией.

Классы тестовых случаев с одной конструкцией содержат две группы тестов. В первой группе конфликт пересекается с началом конструкции, и во второй группе конфликт пересекается с окончанием конструкции.

Классы тестов с несколькими конструкциями содержат вариации трех основных тестовых ситуаций: две конструкции подряд и конфликт затрагивает конец одной и начало другой; две вложенные конструкции и конфликт затрагивает окончания обеих конструкций; две вложенные конструкции и конфликт затрагивает начала обеих конструкций.

Класс тестов для конструкции if-else состоит из вариаций трех тестовых случаев: зона конфликта захватила конец блока if и начало блока else; конфлик захватывает весь блок if и начало блока else; конфлик захватывает весь блок else и конец блока else.

Классы тестов в файле со смешанными конструкциями подобны друг другу и состоят из вариаций 3 основных конфигураций цикла и условия if-else, а также 3 их зеркального отражения в которых условие и цикл поменяны местами. Основные конфигурации: конфликт на границе последовательных условия и цикла; условие вложено в цикл с конфликтом на окончаниях конструкций; условие вложено в цикл с конфликтом на началах обеих конструкций.

Так как блоки кода в конструкциях for, while, do-while и if-else опциональны, то тесты дублируются для вариантов конструкций с блоками кода (содержат "block" в имени теста) и с единственными инструкциями вместо блока (содержат "raw" в имени теста). Для тестов с конструкцией if-else добавляются случаи когда первая часть конструкция содержит блок, а вторая единственную инструкцию, и наоборот.

Все тесты имеют по несколько вариантов написания в зависимости от стиля форматирования кода. Используется стандарты форматирования K&R, Allman, Pico и различные нестандартизованные смешения разных стандартов. Некоторые тесты дополнительно записаны на одной строке чтобы проверить устойчивость алгоритма.

Схема основных алгоритмических тестов

1. Файл цикла for
    1.1. Класс тестов, не меняющихся при работе алгоритмом
        1.1.1. Конфликт после цикла for
        1.1.2. Конфликт внутри цикла for
        1.1.3. Конфликт перед циклом for

1.2. Класс тестов, содержащих один цикл for

1.2.1. Конфликт пересекается с началом цикла for

1.2.2. Конфликт пересекается с концом цикла for

1.3. Класс тестов, содержащих несколько циклов for

1.3.1. Конфликт на границе двух последовательных циклов for

1.3.2. Конфликт пересекается с окончаниями двух вложенных циклов for

1.3.3. Конфликт пересекается с началами двух вложенных циклов for

2. Файл цикла while

2.1. Класс тестов, не меняющихся при работе алгоритмом

2.1.1. Конфликт после цикла while

2.1.2. Конфликт внутри цикла while

2.1.3. Конфликт перед циклом while

2.2. Класс тестов, содержащих один цикл while

2.2.1. Конфликт пересекается с началом цикла while

2.2.2. Конфликт пересекается с концом цикла while

2.3. Класс тестов, содержащих несколько циклов while

2.3.1. Конфликт на границе двух последовательных циклов while

2.3.2. Конфликт пересекается с окончаниями двух вложенных циклов while

2.3.3. Конфликт пересекается с началами двух вложенных циклов while

3. Файл цикла do-while

3.1. Класс тестов, не меняющихся при работе алгоритмом

3.1.1. Конфликт после цикла do-while

3.1.2. Конфликт внутри цикла do-while

3.1.3. Конфликт перед циклом do-while

3.2. Класс тестов, содержащих один цикл do-while

3.2.1. Конфликт пересекается с началом цикла do-while

3.2.2. Конфликт пересекается с концом цикла do-while

3.3. Класс тестов, содержащих несколько циклов for do-while

3.3.1. Конфликт на границе двух последовательных циклов do-while

3.3.2. Конфликт пересекается с окончаниями двух вложенных циклов do-while

3.3.3. Конфликт пересекается с началами двух вложенных циклов do-while

4. Файл условия if-else

4.1. Класс тестов, не меняющихся при работе алгоритмом

4.1.1. Конфликт после условия if-else

4.1.2. Конфликт внутри условия if-else

4.1.3. Конфликт перед условия if-else

4.2. Класс тестов, содержащих одно условие if-else

4.2.1. Конфликт пересекается с началом условия if-else

4.2.2. Конфликт пересекается с концом условия if-else

4.3. Класс тестов, содержащих несколько условий if-else

4.3.1. Конфликт на границе двух последовательных условий if-else

4.3.2. Конфликт пересекается с окончаниями двух вложенных условий if-else

4.3.3. Конфликт пересекается с началами двух вложенных условий if-else

4.4. Класс тестов, содержащих комбинации блоков if и else

5. Файл смешанных тестов

5.1. Условие if и цикл for

5.1.1. Конфликт на границах последовательных условия if и цикла for

5.1.2. Конфликт на окончаниях условия if и вложенного в него цикла for

5.1.3. Конфликт на началах условия if и вложенного в него цикла for

5.1.4. Конфликт на границах последовательных цикла for и условия if

5.1.5. Конфликт на окончаниях цикла for и вложенного в него условия if

5.1.6. Конфликт на началах цикла for и вложенного в него условия if

5.2. Условие if и цикл while

5.2.1. Конфликт на границах последовательных условия if и цикла while

5.2.2. Конфликт на окончаниях условия if и вложенного в него цикла while

5.2.3. Конфликт на началах условия if и вложенного в него цикла while

5.2.4. Конфликт на границах последовательных цикла while и условия if

5.2.5. Конфликт на окончаниях цикла while и вложенного в него условия if

5.2.6. Конфликт на началах цикла while и вложенного в него условия if

5.3. Условие if и цикл do-while

5.3.1. Конфликт на границах последовательных условия if и цикла do-while

5.3.2. Конфликт на окончаниях условия if и вложенного в него цикла do-while

5.3.3. Конфликт на началах условия if и вложенного в него цикла do-while

5.3.4. Конфликт на границах последовательных цикла do-while и условия if

5.3.5. Конфликт на окончаниях цикла do-while и вложенного в него условия if

5.3.6. Конфликт на началах цикла do-while и вложенного в него условия if

Базовые тесты

test/test_file_bit.py

Исходные данные тестов
file_bit = FileBit(239, ("   TestFileBitif(true)\n" "   {\n" "   cin >> n;\n" "   n += 1;\n" "   printf(\"left\");\n" "   }"))

Таблица 2. Инициализация тестов класса TestFileBit

Имя теста и описание	Код теста	Ожидаемый результат

TestFileBit
.test_shrink_top_down
 	
Тестирует уменьшения части текста сверху вниз
	file_bit
	.shrink_top_down(1)
		{
		cin >> n;
		n += 1;
		printf(\"left\");
		}

TestFileBit
.test_shrink_bottom_up
 	
Тестирует уменьшения части текста снизу вверх
	file_bit
	.shrink_bottom_up(1)
		if(true)
		{
		cin >> n;
		n += 1;
		printf(\"left\");\n

TestFileBit .test_shrink_0 Тестирует 0 как крайний случай функций уменьшения части текста	file_bit .shrink_bottom_up(0) file_bit .shrink_top_down(0)	AssertionError

Таблица 3. Тесты класса TestFileBit

test/test_file_merge.py

Исходные данные тестов

```
fb1 = FileBit(1, ("int main() {\n"
                  "   if(true)\n"
                  "   {\n"
                  "       cin >> n;\n"
                  "       n += 1;\n"
                  "       printf(\"left\");\n"
                  "   }\n"))

ct1 = Conflict2Way(8, 8, 8,
                   "   int x = 0;\n"
                   "   x = 0;\n",
                   "   int y = 3;\n",
                   "<<<<<<<\n", "=======\n", ">>>>>>>\n")

fb2 = FileBit(14, ("   while(true) {\n"
                   "      cin >> n;\n"
                   "      n += 1;\n"
                   "      printf(\"left\");\n"
                   "   }\n"))

ct2 = Conflict3Way(19, 15, 14, "
                    x = 0;\n", """   printf("Hello!")\n""", "   y = 2;\n",
                   "<<<<<<<\n", "|||||||\n", "=======\n", ">>>>>>>\n")

fb3 = FileBit(26, "   return 0;\n"
                  "}\n")

file_merge = FileMerge(Path("./testproject/prog.cpp"),
                       [fb1, fb2, fb3], [ct1, ct2])
```

Таблица 4. Инициализация тестов класса TestFileMerge

Имя теста и описание	Код теста	Ожидаемый результат
TestFileMerge .test_is_resolved Тест выбирает	ct1.select(Choice.both) ct2.select(Choice.left)	True == file_merge.is_resolved()

стратегии слияния для двух конфликтов и проверяет что совокупный файл помечается как имеющий стратегию слияния		
TestFileMerge .test_select_all Тест выбирает глобальную стратегию слияния файла и проверяет что отдельные конфликты помечены как имеющие стратегию слияния	file_merge.select_all(Choice.right)	True == ct1.is_resolved(), True == ct2.is_resolved()

TestFileMerge .test_result_default
 	
Тест применяет две различные стратегии (both, undecided) к файлу file_merge и проверяет что результирующий текст соответствует выбранным стратегиям
	ct1.select(Choice.both)
	
	ct2.select(Choice.undecided)
	
	text = file_merge.result()
		int main() {
		if(true)
		{
		cin >> n;
		n += 1;
		
		printf(\"left\");
		}
		int x = 0;
		x = 0;
		int y = 3;
		while(true) {
		cin >> n;
		n += 1;
		
		printf(\"left\");
		}
		<<<<<<<
		x = 0;
		|||||||
		printf(\"Hello!\")
		=======

		y = 2;
		>>>>>>>
		return 0;
		}

TestFileMerge
.test_result_left
 	
Тест применяет две различные стратегии (both, undecided) к файлу file_merge и проверяет что результирующий текст соответствует выбранным стратегиям при том что стратегия по умолчанию выставлена на left вместо undecided
	ct1.select(Choice.both)
	ct2.select(Choice.undecided)
	
	text =
	file_merge.result(Choice.left)
		int main() {
		if(true)
		{
		cin >> n;
		n += 1;
		printf(\"left\");
		}
		int x = 0;
		x = 0;
		while(true) {
		cin >> n;
		n += 1;
		printf(\"left\");
		}
		x = 0;
		return 0;
		}

TestFileMerge
.test_result_right
 	
Тест применяет две различные стратегии (left, right) к файлу file_merge и проверяет что результирующий текст соответствует выбранным стратегиям при том что стратегия по умолчанию выставлена на right вместо undecided
	ct1.select(Choice.left)
	ct2.select(Choice.right)
	
	text =
	file_merge.result(Choice.right)
		int main() {
		if(true)
		{
		cin >> n;
		n += 1;
		printf(\"left\");
		}
		int y = 3;
		while(true) {
		cin >> n;
		n += 1;
		printf(\"left\");

		}
		
		y = 2;
		
		return 0;
		
		}

TestFileMerge .test_abstract_syntax_tree _left Тест проверяет корректность работы синтаксического парсера. Файл с заданным текстовым содержанием file_merge разбирается в его AST, вершины которого проверяются на соответствие нужным типам.	root = file_merge.abstract_syntax_tree(C hoice.left).cursor main_body = root.child(0).child(0) if1 = main_body.child(0) var1 = main_body.child(1) ass1 = main_body.child(2) while1 = main_body.child(3) ass2 = main_body.child(4) return1 = main_body.child(5)	if1.kind == CursorKind.IF_STMT, var1.kind == CursorKind.DECL_STMT, ass1.kind == CursorKind.BINARY_OPERATO R, while1.kind == CursorKind.WHILE_STMT, ass2.kind == CursorKind.BINARY_OPERATO R, return1.kind == CursorKind.RETURN_STMT
TestFileMerge .test_abstract_syntax_tree _right Тест проверяет корректность работы синтаксического парсера. Файл с заданным текстовым содержанием file_merge разбирается в его AST, вершины которого проверяются на соответствие нужным типам.	root = file_merge.abstract_syntax_tree(C hoice.right).cursor main_body = root.child(0).child(0) if1 = main_body.child(0) var1 = main_body.child(1) while1 = main_body.child(2) ass2 = main_body.child(3) return1 = main_body.child(4)	if1.kind == CursorKind.IF_STMT, var1.kind == CursorKind.DECL_STMT, while1.kind == CursorKind.WHILE_STMT, ass2.kind == CursorKind.BINARY_OPERATO R, return1.kind == CursorKind.RETURN_STMT
TestFileMerge .test_abstract_syntax_tree _default Тест проверяет по умолчанию используется стратегия left для разбора файла в его AST	default = file_merge.abstract_syntax_tree() left = file_merge.abstract_syntax_tree(C hoice.left) kinds = [CursorKind.FUNCTION_DECL, CursorKind.IF_STMT, CursorKind.WHILE_STMT, CursorKind.BINARY_OPERATOR]	[ch.kind **for** ch **in** FileMerge.extract_childre n(default.cursor, kinds)] == [ch.kind **for** ch **in** FileMerge.extract_childre n(left.cursor, kinds)])

TestFileMerge .test_abstract_syntax_tree _cache Тест проверяет что кеширование результатов разбора синтаксиса работает корректно	left1 = file_merge.abstract_syntax_tree(C hoice.left) left2 = file_merge.abstract_syntax_tree(C hoice.left) right = file_merge.abstract_syntax_tree(C hoice.right) left3 = file_merge.abstract_syntax_tree(C hoice.left)	left1 == left2, left2 != right, right != left3
TestFileMerge .test_refactor_syntax_bloc ks Тест проверяет что при обработке файла file_merge, текст которого не содержит конфликтов, ничего не изменяется	before = copy(file_merge) file_merge.refactor_syntax_blocks ()	file_merge.path == before.path, file_merge.file_bits == before.file_bits, file_merge.conflicts == before.conflicts
TestFileMerge .test_extract_children Тест проверяет работу функции извлечения и фильтрации вершин AST	root = file_merge.abstract_syntax_tree(C hoice.left).cursor children = FileMerge.extract_children(root, [CursorKind.FUNCTION_DECL, CursorKind.IF_STMT, CursorKind.WHILE_STMT]) extracted_features = [(ch.kind, ch.extent.start.line) for ch in children]	extracted_features == [(CursorKind.FUNCTION_DEC L, 1), (CursorKind.IF_STMT, 2), (CursorKind.WHILE_STMT, 10)]
TestFileMerge .test_parse Тест парсит текст идентичный тексту файла file_merge и проверяет что внутренняя структура полученного объекта идентична внутренней	text = ("int main() {\n" " if(true)\n" " {\n" " cin >> n;\n" " n += 1;\n" " printf(\"left\");\n" " }\n"	parsed_file.path == file_merge.path, parsed_file.file_bits == file_merge.file_bits parsed_file.conflicts == file_merge.conflicts

структуре file_merge	"<<<<<<<\n" " int x = 0;\n" " x = 0;\n" "=======\n" " int y = 3;\n" ">>>>>>>\n" " while(true) {\n" " cin >> n;\n" " n += 1;\n" " printf(\"left\");\n" " }\n" "<<<<<<<\n" " x = 0;\n" "|||||||\n" " printf(\"Hello!\")\n" "=======\n" " y = 2;\n" ">>>>>>>\n" " return 0;\n" "}\n") parsed_file = FileMerge.parse(Path("./testproject/prog.cpp"), StringIO(text))	
TestFileMerge .test_can_parse Тест проверяет что алгоритм принимает файлы поддерживаемых расширений и отвергает любые другие файлы		True == FileMerge.can_parse(Path("./testproject/prog.cpp")), True == FileMerge.can_parse(Path("/Users/admin/testproject/prog.hpp")), True == FileMerge.can_parse(Path("testproject/prog.c")),

		True == FileMerge.can_parse(Path("testproject/prog.h")), False == FileMerge.can_parse(Path("testproject/h")), False == FileMerge.can_parse(Path("testproject/prog.log"))

Таблица 5. Тесты класса TestFileMerge

Заключение

В данной работе были рассмотрены проблемы слияния конфликтов в файлах исходного кода и изучены конфликты с пересекающимися синтаксическими блоками. Был разработан алгоритм расширения синтаксических блоков, улучшающий эргономику слияния конфликтов с пересекающимися синтаксическими блоками. Алгоритм был реализован в виде экспериментальной утилиты MergeTool. Утилита MergeTool была протестирована на тестовом наборе из 16 базовых тестов 137 алгоритмических тестов, ее работа была улучшена в течении 7 последовательных версий.

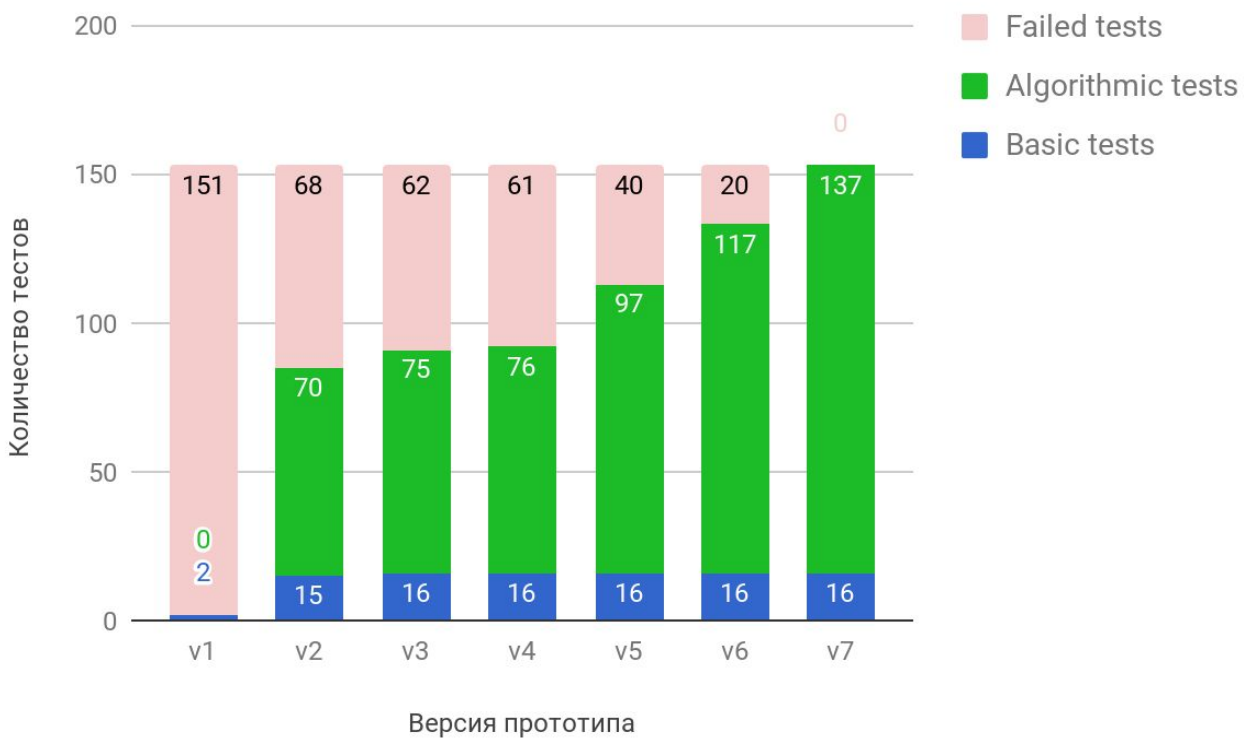

Рис. 16 Результаты тестирования различных версий алгоритма MergeTool

Результаты выполнения работы

1. Найдены категории синтаксических конфликтов, удовлетворяющие поставленным требованиям:
    a. часто встречаются в практической разработке ПО
    b. разрешение подобных конфликтов содержит рутинную часть, поддающуюся автоматизации
2. Составлен набор алгоритмических тестов
    a. Использован синтаксис языка C
    b. Покрыты конструкции цикла for, цикла while, цикла do-while, условия if
    c. Покрыты различные стили форматирования кода
3. Разработан алгоритм удовлетворяющий более 95% тестовых случаев
    a. Алгоритм поддерживает с формат широко применяемой утилиты GNU diff, формат которой распознается промышленными программами разрешения конфликтов слияния, такими как SourceTree

4. Реализован прототип алгоритма
    a. Прототип оттестирован на операционной системе Mac OS X
    b. Основной код прототипа написан на языке Python, который является мультиплатформенным и легко переносится на операционные системы Linux и Windows
5. Разработана инфраструктура для применения алгоритма в проектах
    a. Поддержаны проекты, использующие систему сборки GNU Make
    b. Реализован интерфейс для автоматического разрешения конфликтов в проекте
    c. Добавлена возможность компиляции проекта в процессе разрешения конфликтов

Литература

[1] Bill Ritcher "A Trustworthy 3-Way Merge" https://www.guiffy.com/SureMergeWP.html

[2] Robin La Fontaine "Merging XML files: a new approach providing intelligent merge of XML data sets" https://www.deltaxml.com/support/documents/articles-and-papers/merging-xml-files.pdf

[3] J. W. Hunt, M. D. McIlroy "An Algorithm for Differential File Comparison" http://www.cs.dartmouth.edu/~doug/diff.pdf

[4] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce "A Formal Investigation of Diff3" http://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf

[5] Д.В. Кознов, Е.В. Ларчик, М.М. Плискин, Н.И. Артамонов "О Задаче Слияния Карт Памяти (Mind Maps) При Коллективной Разработке" http://www.math.spbu.ru/user/dkoznov/papers/KoznovLarchik2011.pdf

[6] Forrester Consulting "Continuous Delivery: A Maturity Assessment Model" http://info.thoughtworks.com/rs/thoughtworks2/images/Continuous%20Delivery%20_%20A%20Maturity%20Assessment%20ModelFINAL.pdf

[7] "Ветвление в Git - Основы ветвления и слияния" https://git-scm.com/book/ru/v1/Ветвление-в-Git-Основы-ветвления-и-слияния

[8] Manuel Klimek "Introduction to the Clang AST" http://clang.llvm.org/docs/IntroductionToTheClangAST.html

[9] Eli Bendersky "Parsing C++ in Python with Clang" http://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang/