

Санкт-Петербургский
Государственный Университет

Математико-механический факультет
Кафедра Системного Программирования

Интеграция структурного слияния
файлов в систему контроля версий
Subversion

Дипломная работа студента 544 группы
Вадишева Семена Борисовича

Научный руководитель: А. Ю. КИТАЕВ

/подпись/

Рецензент: АСП. Л. С. ГУРЕВИЧ

/подпись/

„Допустить к защите“

Заведующий кафедрой: Д.Ф.-М.Н., ПРОФ. А. Н. ТЕРЕХОВ

/подпись/

Санкт-Петербург
2009 г.

Аннотация

В настоящей работе описана задача трехстороннего слияния текстовых файлов, дан обзор существующих подходов для решения представленной задачи, после чего предложен алгоритм построения и слияния внутренней структуры текстового файла, адаптированный для интеграции в систему контроля версий **Subversion**.

Содержание

1	Введение	4
1.1	Системы контроля версий	4
1.2	Постановка задачи	5
2	Обзор алгоритмов	7
2.1	Алгоритмы бинарного слияния файлов	7
2.2	Алгоритмы структурного сравнения	8
2.3	Алгоритмы структурного слияния	11
3	Подход к решению	15
3.1	Структура представления содержимого файла	16
3.2	Алгоритм структурного сравнения	18
3.3	Алгоритм структурного слияния	20
4	Результаты	25
4.1	Реализация	25
4.2	Применение	25
4.3	Преимущества и недостатки	28
5	Заключение	30

1 Введение

История систем контроля версий насчитывает уже несколько десятилетий, за это время было разработано более сотни подобных систем. Прежде всего это обусловлено тем классом задач, для решения которых они создавались, а именно хранение истории изменений файла или набора файлов, эффективные способы получения предыдущих версий, а также возможность получения метайнформации о изменениях между версиями. Подобные задачи встречаются повсеместно — от хранения изменений единственного файла с помощью системы версионирования, встроенной непосредственно в редактор, до поддержания разработки огромных программных проектов, над которыми работают десятки программистов, расположенных по всему земному шару. Последнее, безусловно, стало самым распространенным применением систем контроля версий.

1.1 Системы контроля версий

Любая достаточно развитая система контроля версий имеет хранилище, позволяющее эффективно хранить и обрабатывать различные версии дерева файлов и директорий. Такое хранилище принято называть *репозиторием*. Каждая версия в нем некоторым образом именована, в качестве имени, например, может выступать целое неотрицательное число или *SHA-1 хэш-код*, подсчитанный на основе содержимого файлов в данной версии. Именованные версии принято называть *ревизиями* [1].

В зависимости от числа *репозитория*в, поддерживаемых данной системой контроля версий, она может быть *централизованной* или *распределенной* [2]. В настоящей работе мы сосредоточим внимание на первом типе.

Со стороны клиента системы контроля версий предоставляют интерфейс *рабочей копии*, то есть локального представления некоторой версии проекта. Неотъемлемой частью *рабочей копии* является *базовая версия*, или *baseline*, — версия, относительно которой определяются изменения файлов.

Такая архитектура предоставляет возможность использования двух моделей версионирования [3]:

1. Оптимистичная модель версионирования.

Модель конкурентного изменения файлов, то есть создания пользователями двух альтернативных версий некоторого файла, которые в дальнейшем необходимо будет совместить в одну.

2. Пессимистичная модель версионирования.

Этот подход также называют *моделью блокировок*, он не позволяет совершать одновременные изменения путем установления *блокировки* на определенном файле, что в свою очередь запрещает вносить изменения в этот файл кем-либо кроме *автора блокировки*.

Далее в этой работе мы не будем рассматривать *пессимистичную* модель версионирования, но сосредоточимся на *оптимистичной*.

1.2 Постановка задачи

Реализация любой системы контроля версий, которая поддерживает *оптимистичную модель версионирования*, включает в себя решение двух типичных задач:

1. Задача сравнения двух файлов.

В формальном виде описать разницу в содержании двух данных файлов так, чтобы это описание можно было *сериализовать*, то есть хранить или передавать по сети.

2. Задача трехстороннего слияния файлов.

Для данных базового файла и двух его альтернативных версий создать такой файл, что в описании его разницы с базовым файлом содержались описания разницы альтернативных версий с этим же файлом. На рисунке 1 представлена схема трехстороннего слияния файлов.

Типичным решением для большинства существующих систем контроля версий являются различные вариации бинарного сравнения и трехстороннего слияния файлов. Однако, наряду с подобными подходами существуют также и алгоритмы структурного сравнения и слияния [4], [5]. Эти алгоритмы используют тот факт, что для большинства версионируемых текстовых файлов можно создать некоторую структуру, представляющую содержимое файлов. Подобными файлами могут быть исходные коды программ, для них структурой представления содержимого является *абстрактное синтаксическое дерево*.

Перед автором данной работы была поставлена задача адаптировать существующие структурные алгоритмы для дальнейшей их интеграции в систему контроля версий **Subversion** с выполнением следующих требований:

- Алгоритм должен учитывать способ хранения и обработки представления дерева файлов и директорий в *репозитории*.

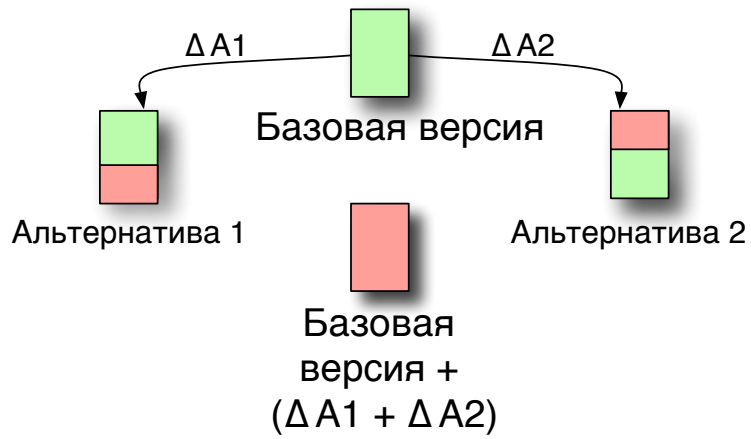


Рис. 1: Задача трехстороннего слияния файлов.

- История изменений между версиями должна включать модификации внутренней структуры файла.
- Алгоритм должен предоставлять возможность обнаружения переименований переменных и функций.
- Реализация должна быть пригодной для внедрения в существующие средства разработки.

2 Обзор алгоритмов

2.1 Алгоритмы бинарного слияния файлов

Исследования в области бинарного слияния файлов проводились еще до создания систем контроля версий и были направлены в частности на разработку *2-way merge* и *3-way merge* приложений. Одними из самых популярных подобных приложений стали утилиты *diff* и *patch*, изначально получившие распространение на Unix-подобных операционных системах.

В основе програм *diff* и *patch* заложен алгоритм нахождения *наибольшей общей подпоследовательности* (англ. *longest common subsequence*, проблема *LCS*) [6].

Пусть даны две последовательности элементов:

a b c d f g h j q z

a b c d e f g i j k r x y z

Необходимо найти наиболее длинную последовательность элементов, которая представлена в обеих последовательностях в одинаковом порядке. То есть задача заключается в нахождении новой последовательности, которая может быть получена из первой последовательности удалением некоторых элементов или из второй последовательности удалением других элементов. В данном случае такой последовательностью будет являться

a b c d f g j z

После получения наибольшей общей подпоследовательности остаётся небольшой шаг до определения формальной разницы между последовательностями:

<i>e</i>	<i>h</i>	<i>i</i>	<i>k</i>	<i>q</i>	<i>r</i>	<i>x</i>	<i>y</i>
+	-	+	+	-	+	+	+

Заметим что, разницей между двумя последовательностями символов является упорядоченный набор модификаций, которые необходимо применить к первой последовательности, чтобы в итоге получить вторую. В терминах модификаций задача нахождения *наибольшей общей последовательности* формулируется как задача нахождения минимального набора модификаций для перехода от первой последовательности ко второй.

Нахождение минимального набора модификаций для произвольных двух строк текстового файла дает возможность сопоставлять эти строки по принципу *похожести*. Для файла в целом это позволяет определить какие строки изменились и каким образом. Наиболее широко подобный подход представлен в работе Юджина Майерса [7], различные вариации указанных им алгоритмов используются большинством *2-way merge* и *3-way merge* приложений.

Логичным развитием описанного подхода стало сравнение произвольных *бинарных* файлов. Результатом этому стали работы над алгоритмами *дельта-кодирования* — способа хранения и передачи изменяемых во времени данных с использованием *бинарных дельт*.

В конечном результате последние работы в этом направлении привели к созданию ряда стандартов, одним из которых стал стандарт формата представления *бинарной дельты* описанный в RFC 3284 [8]. Именно этот формат используется в системе контроля версий **Subversion** для хранения в *репозитории* и передачи между приложением-клиентом и *репозиторием* разницы между двумя версиями одного и того же файла.

Заметим, что независимо от формата *дельта* представляет собой набор модификаций, или скрипт, который описывает те преобразования, которые необходимо применить к исходному файлу, чтобы получить соответствующую его версию. Это используется в решении задач трехстороннего слияния. Создав для каждой альтернативной версии скрипт изменений, достаточно применить их конкурентно на базовом файле, чтобы получить результат слияния. В тех случаях, когда оба скрипта изменяют один и тот же участок исходного файла возникает *конфликт слияния*, то есть ситуация, когда автоматическое слияние невозможно, что приводит к необходимости участия человека для разрешения *конфликта слияния*.

2.2 Алгоритмы структурного сравнения

В качестве структуры, представляющей внутреннее содержание файла, в подавляющем большинстве подобных алгоритмов выбирается *дерево*, для каждой вершины которого выполняется одно из условий:

1. Дети вершины имеют идентификатор, уникально определяющий ребенка этой вершины.
2. Дети вершины упорядочены.

Наибольшую сложность представляют деревья с вершинами второго вида, именно поэтому как и в большинстве исследований, в этой работе

мы сосредоточимся на рассмотрении *упорядоченных деревьев*, то есть тех деревьев, для каждой вершины которых, все дети упорядочены. В существующих моделях каждой вершине *дерева* соответствует некоторое содержание, которое может изменяться от версии к версии.

Будем использовать следующие обозначения: T — дерево, V_T — множество его вершин, E_T — множество его ребер. Исходное дерево мы будем обозначать как T_b , а альтернативные версии дерева как T_w или T_1, T_2 и т.д. За V_b обозначим множество вершин базового дерева, E_b множество ребер базового дерева, множество вершин и множество ребер альтернативной версии дерева будем обозначать как V_w и E_w соответственно. Пусть $v \in V$, тогда величина $content(v)$ соответствует содержанию вершины v .

Введем в рассмотрение отображение $name : V \rightarrow \mathcal{A}^*$ — *инъективную* функцию именования вершин, которая каждой вершине сопоставляет цепочку элементов некоторого упорядоченного алфавита символов $\mathcal{A}$. Будем считать, что перед началом каждого алгоритма мы переименовали все вершины так, что имя вершины в точности определяет ее расположение в дереве. Пример подобного именования показан на рисунке 2. Далее в работе будем отождествлять вершину дерева v с ее идентификатором, то есть значением $name(v)$.

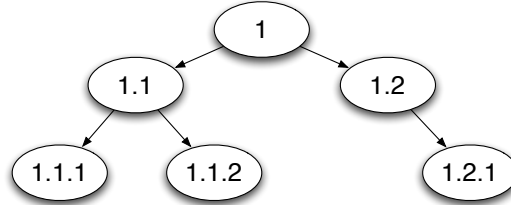


Рис. 2: Именованние вершин обходом в ширину.

Пусть $v \in V$, множество ее детей обозначим как C_v . Порядок на множестве C_v будем обозначать с помощью стандартной инфиксной записи: $\forall v_1, v_2 \in C_v : v_1 < v_2$ или $v_1 > v_2$.

Формальной разницей между двумя данными деревьями T_b и T_w назовем набор модификаций E такой, что $E = (edit_1, edit_2, \dots, edit_n) : \forall i \in [1, \dots, n]$ $edit_i$ соответствует одному из четырех типов модификаций:

1. $update(v)$.
Изменение содержимого вершины v .
2. $add(v)$.
Добавление вершины v .

3. $delete(v)$.

Удаление вершины v .

4. $move(v_1, v_2)$.

Перемещение вершины v_1 в вершину v_2 .

Упорядоченный набор модификаций принято называть *скриптом*. Для скрипта $E = (edit_1, edit_2, \dots, edit_n)$, который преобразует дерево T_b в дерево T_w , будем писать $T_b \xrightarrow{E} T_w$ или $T_b \xrightarrow{edit_1} T_1 \xrightarrow{edit_2} \dots \xrightarrow{edit_n} T_w$.

Пусть для данных T_b и T_w есть некоторое множество скриптов, описывающих разницу между ними. Обозначим их множество за $\mathcal{E}$. Введем функцию $cost : \mathcal{E} \rightarrow \mathbf{R}$, которая каждому такому скрипту сопоставляет некоторое вещественное число. Зачастую в качестве такой функции рассматривается $cost(E) = |E|$.

Алгоритм сравнения двух деревьев T_b и T_w в общем виде состоит из двух частей [9]:

1. Построение множества пар соответствий $Matching(T_b, T_w) \in V_b \times V_w$, то есть множества пар *похожих* вершин, для краткости будем обозначать это множество как M . В идеальном случае $v_b \in V_b$ и $v_w \in V_w$ считаются *похожими*, если

$$\exists E = (edit_1, edit_2, \dots, edit_n) : T_{v_b} \xrightarrow{E} T_{v_w},$$

$$cost(E) \rightarrow min, \quad cost(E) \leq N,$$

где T_{v_b}, T_{v_w} — поддеревья T_b и T_w с корнями в v_b и v_w соответственно, а N — некоторое пороговое вещественное число.

Большинство алгоритмов не производят поиск пар соответствий в виде, представленном выше, связано это прежде всего с большой трудоемкостью [10]. Вместо этого вводятся различного рода эвристики, позволяющие сопоставить вершины T_b и T_w некоторым образом. Пример подобной эвристики будет представлен далее в работе.

2. Нахождение скрипта модификаций минимальной стоимости на основе построенного множества пар соответствий. Вариации алгоритма 1 наиболее часто используются в целях решения этой задачи [4].

Пусть r_b и r_w корни T_b и T_w соответственно. По умолчанию $(r_b, r_w) \in M$. В начале алгоритма скрипт модификаций E пуст, на каждом

шаге в него добавляются модификации одного из четырех представленных выше типов. Таким образом алгоритм строит последовательность $T_b \xrightarrow{edit_1} \dots \xrightarrow{edit_n} T_w$, причем для каждого промежуточного дерева T_i рассматривается $M' = Matching(T_b, T_i)$. Формально доказанным фактом является то, что полученный скрипт модификаций $E = (edit_1, \dots, edit_n)$ имеет минимальную стоимость при $cost(E) = |E|$ [11].

2.3 Алгоритмы структурного слияния

Теперь, зная алгоритм сравнения двух упорядоченных деревьев, перейдем к решению задачи структурного трехстороннего слияния.

Пусть даны исходное дерево T_b и две его альтернативных версии T_1 и T_2 . Построим для каждой из альтернатив скрипты модификаций $E_1 = (edit_1^1, \dots, edit_{n_1}^1)$ и $E_2 = (edit_1^2, \dots, edit_{n_2}^2)$. Пусть $edit_i^k$ изменяет некоторым образом вершину v , добавляет или удаляет ее (в данном случае $move(v_1, v_2)$ рассматривается как удаление v_1 и добавление v_2), либо изменяет содержимое этой вершины. Запишем этот факт как $v \leftarrow edit_i^k$. Алгоритм 3 представляет собой схему пошагового построения цепочки $T_b \xrightarrow{e_1} \dots \xrightarrow{e_m} T'_b$, где в качестве e_i могут выступать модификации приведенных четырех типов и дополнительная модификация, которая устанавливает маркер конфликтного состояния на данной вершине. Таким образом результат слияния дерево T'_b содержит изменения и E_1 , и E_2 , с некоторым набором вершин, находящимся в конфликтном состоянии [5], [9].

Наиболее интересна функция $mergeMods(edit_1, edit_2)$, которая по двум модификациям изменяющим некоторую вершину, определяет как должна выглядеть эта вершина в результирующем дереве T'_b [5]. Наглядно представить работу этой функции можно в виде таблицы 1.

Algorithm 1 Compare(T_b, T_w)

$E := \emptyset$
 $M' := M$
{Обходим вершины T_w в ширину}
for $v_i := \text{BreadthFirstSearch}(T_w)$ **do**
 $p_i : v_i \in C_{p_i}$
 $p_m : (p_m, p_i) \in M'$
 if $\forall v \in V_b \ (v, v_i) \notin M'$ **then**
 {Добавляем в T_b вершину с идентификатором, соответствующим
 v_i , назовем ее v_b }
 $edit := add(v_b)$
 $E := E \cup \{edit\}$
 $M' := M' \cup (v_b, v_i)$
 else
 if $v_i \neq r_w$ **then**
 $v_b : (v_b, v_i) \in M'$
 $p_b : v_b \in C_{p_b}$
 if $content(v_b) \neq content(v_i)$ **then**
 {Обновляем содержание вершины v_b }
 $edit := update(v_b)$
 $E := E \cup \{edit\}$
 if $(p_b, p_i) \notin M'$ **then**
 {Родители v_b и v_i не образуют пару соответствия}
 {Перемещаем v_b в вершину с идентификатором, соответствующим
 v_i , назовем ее v'_b }
 $edit := move(v_b, v'_b)$
 $E := E \cup \{edit\}$
 {Перемещаем детей v_b и v_i , затратив при этом минимальное количество
 модификаций}
 AlignChildren(v_b, v_i)
 {Обходим T_b для удаления вершин}
 for $v_j \in V_b$ **do**
 if $\forall v \in V_w \ (v_j, v) \notin M'$ **then**
 {Удаляем все вершины, для которых не нашлось пары соответствия}
 $edit := delete(v_j)$
 $E := E \cup \{edit\}$

Algorithm 2 AlignChildren(v_b, v_w)

{Выделим последовательности детей v_b и v_w имеющих соответствия между собой}
 $S_b : S_b \subset C_{v_b}, \quad \forall c_b \in S_b \exists c_w \in C_{v_w} \mid (c_b, c_w) \in M'$
 $S_w : S_w \subset C_{v_w}, \quad \forall c_w \in S_w \exists c_b \in C_{v_b} \mid (c_b, c_w) \in M'$
{В этих последовательностях одинаковыми назовем те вершины, которые образуют пары соответствия}
 $equal(a, b) \Leftrightarrow (a, b) \in M'$
{Находим наибольшую последовательность вершин, входящую и в S_b , и в S_w }
 $S := LCS(S_b, S_w, equal)$
for $\forall x \in S_b, \forall y \in S_w \mid (x, y) \in M', (x, y) \notin S$ **do**
 {Перемещаем детей, которые не вошли в общую последовательность}
 $edit := move(x, y')$
 $E := E \cup \{edit\}$

Algorithm 3 Merge(T_b, T_1, T_2)

$E_1 = Compare(T_b, T_1)$
 $E_2 = Compare(T_b, T_2)$
for $\forall edit_1 \in E_1$ **do**
 {Определяем какую вершину изменяет $edit_1$ и проверяем есть ли модификация из E_2 , действующая на ту же вершину}
 $v : v \leftarrow edit_1$
 if $\exists edit_2 \in E_2 : v \leftarrow edit_2$ **then**
 $e = mergeMods(edit_1, edit_2)$
 $T_{previous} \xrightarrow{e} T_{next}$
 else
 $T_{previous} \xrightarrow{edit_1} T_{next}$
 for $\forall edit_2 \in E_2 : edit_2$ еще не была применена **do**
 {Применяем оставшиеся изменения}
 $T_{previous} \xrightarrow{edit_2} T_{next}$

Таблица 1: Принцип работы функции $mergeMods(edit_1, edit_2)$.

$edit_1$	$edit_2$	Результирующее действие
$update(v)$	$update(v)$	если автоматическое слияние содержимого возможно, то $update(v)$, иначе отметить конфликт на v
$update(v)$	$add(v)$	отметить конфликт на v
$update(v)$	$delete(v)$	отметить конфликт на v
$add(v)$	$add(v)$	если содержимое для каждой версии одинаково, то $add(v)$, иначе отметить конфликт на v
$add(v)$	$delete(v)$	отметить конфликт на v
$delete(v)$	$delete(v)$	удалить вершину v

3 Подход к решению

Настоящую работу следует рассматривать в контексте разработки системы **Jetversion**, основной целью которой является изменения клиентской части системы **Subversion** для добавления функциональности, которая бы позволяла версионировать внутреннее представление файлов. Общая схема взаимодействия клиента системы **Jetversion** с *репозиторием* системы **Subversion** представлена на рисунке 3.

Решение задачи, поставленной в начале настоящей работы, включает в себя три аспекта:

- Разработка структуры представления содержимого файла.
- Реализация алгоритма сравнения структуры *рабочей версии* файла с *базовой версией* его структуры.
- Реализация алгоритма применения приходящих от *репозитория* изменений к структуре файла *рабочей копии*.

Далее в этой главе речь пойдет о каждом из вышеперечисленных аспектов.

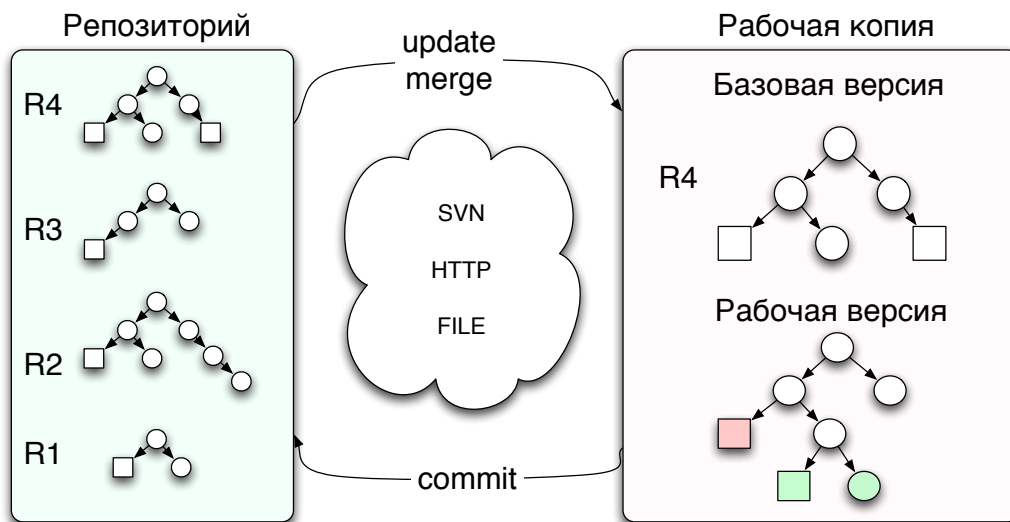


Рис. 3: Схема взаимодействия клиента с репозиторием.

3.1 Структура представления содержимого файла

Репозиторий системы контроля версий **Subversion** позволяет хранить дерево файлов и директорий. Существенным преимуществом *репозитория* этой системы по сравнению с рядом других систем является возможность обработки версионизируемых свойств, то есть набора пар *ключ - значение*, которые могут изменяться от версии к версии вместе с содержимым файла.

Как и в большинстве подобных исследований, в данной работе мы сосредоточимся на версионировании определенных типов файлов. Так как проект **Jetversion** разрабатывается на языке программирования **Java**, то естественным выбором стали файлы исходных кодов программ, написанных именно на этом языке программирования.

В результате обсуждений касательно той модели, которая будет использоваться для представления содержимого файла исходных кодов программ, была выбрана структура, наиболее удачно подходящая для решения поставленных задач. В рамках настоящей работы для этой структуры принято название *GAT-дерево* (англ. *graph-like annotated tree*). Схематическое изображение такой структуры представлено на рисунке 4.

Более формально, *GAT-дерево* T — это дерево со следующим набором свойств:

- Дети любой вершины имеют уникальные между собой идентификаторы. Идентификатором вершины является имя соответствующего файла или директории в *репозитории*.
- Каждой вершине поставлен в соответствие набор свойств, каждое из которых представляет собой пару *ключ - значение*.
- Множество вершин разбито на два типа: *blocks* — вершины, которые обладают только содержанием и не имеют детей, и *nodes* — вершины, которые не обладают содержанием, но имеют ноль или более детей:

$$V = B \cup N \mid B \cap N = \emptyset, \forall b \in B \ C_b = \emptyset, \forall n \in N \ content(n) = \emptyset,$$

где B — множество вершин, обладающих только содержанием, а N — множество вершин, не обладающих содержанием. В *репозитории* элементам множества B соответствуют файлы, а элементам множества N директории.

- Множество вершин разбито на классы:

$$V = C_1 \cup \dots \cup C_n \mid \forall i, j \in [1, \dots, n] \ C_i \cap C_j = \emptyset.$$

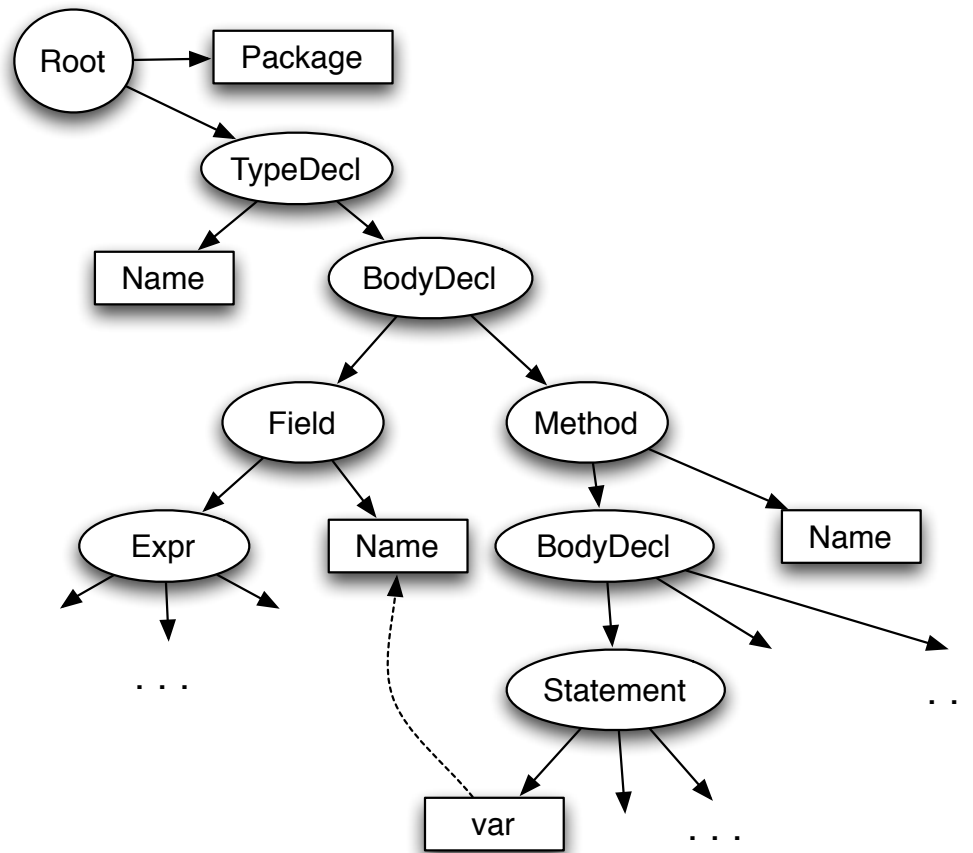


Рис. 4: Структура представления содержимого файла.

Для поддержания этой особенности введено свойство *jvn:class*, значением которого для данной директории или файла является имя класса, которому принадлежит соответствующая вершина.

- Дети вершины упорядочены тогда и только тогда, когда эта вершина принадлежит классу из заранее установленного набора. Для поддержания этой особенности введено свойство *jvn:position*, значениями которого, не умаляя общности, являются вещественные числа, что в свою очередь позволяет ввести порядок на множестве детей соответствующей вершины.
- Для вершин из множества *B* возможно установить *ссылку* на другой элемент из множества *B*. Для реализации этой особенности введено свойство *jvn:ref*, которое устанавливается на файле в репозитории, его значением является путь от корня дерева до файла,

на который указывает *ссылка*.

3.2 Алгоритм структурного сравнения

В системе контроля версий **Subversion** для определения разницы между двумя версиями дерева файлов и директорий используется интерфейс *editor* [12].

Интерфейс *editor* предоставляет набор операций для обхода дерева файлов и директорий, добавления новых файлов, изменения содержания и набора свойств существующих и т. д. Именно этот интерфейс используется для описания *скрипта* модификаций между двумя *GAT-деревьями*.

На основе указанного интерфейса естественным образом определены четыре типа модификаций, представленные в предыдущей главе. Стоит отметить модификации типа $move(v_1, v_2)$, так как существует две различные их реализации:

- Изменение порядка вершин без изменения их идентификаторов влечет лишь изменения свойств *svn:position* и не требует добавления и удаления соответствующих элементов *GAT дерева*.
- Изменение идентификаторов вершин является задачей на порядок сложнее. Крайне важным фактом является то, что **Subversion** не поддерживает так называемый *true rename* файлов и директорий [3]. Поэтому модификации такого рода реализованы как последовательность двух операций:

1. $delete(v_1)$.
Удаление вершины v_1 .
2. $addWithHistory(v_2, v_1)$.
Добавление вершины v_2 с сохранением за ней истории вершины v_1 . После подобного добавления *репозиторий* может передать клиенту информацию, что вершина v_2 в данной ревизии была на месте v_1 .

В отличие от рассматриваемых ранее *скриптов* модификаций, *скрипт*, описанный с помощью интерфейса *editor*, имеет строгую последовательность, которая определяет обход исходного *GAT дерева* T_b с последовательным применением модификаций, по окончании обхода на выходе должна получиться альтернативная версия *GAT-дерева* T_w . Эту же самую последовательность можно описать с помощью *дельта дерева* — дерева, вершины которого помечены идентификаторами вершин T_b и T_w , а

некоторые ребра помечены соответствующими модификациями. Рисунок 5 иллюстрирует сказанное выше.

Важно учесть, что добавление и удаление *ссылок* между блоками представляется в *дельта дереве* как модификация типа $update(v)$, изменяющая свойство $jvn:ref$. В противоположность этому изменение свойства $jvn:class$ приводит к удалению старой вершины и созданию на ее месте новой с тем же идентификатором.

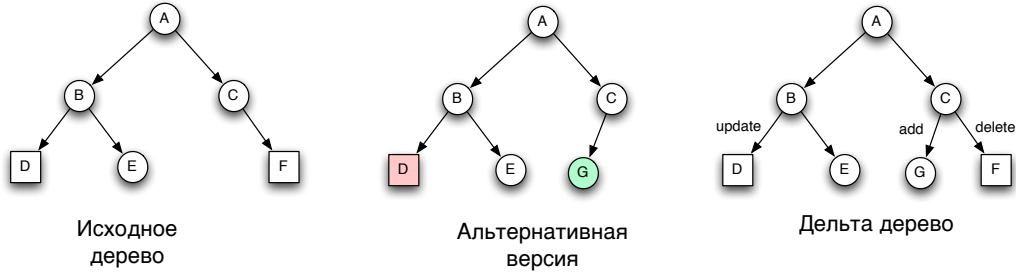


Рис. 5: Пример дельта дерева.

Будем обозначать *дельта дерево* как ΔT и писать $T_b \xrightarrow{\Delta T} T_w$. Таким образом алгоритм структурного сравнения сводится к построению ΔT .

Для построения *дельта дерева* ΔT между двумя данными *GAT* деревьями T_b и T_w обратимся к подходу сравнения упорядоченных деревьев, описанному в предыдущей главе. Важным результатом настоящей работы стало использование алгоритма построения множества пар соответствия $Matching(T_b, T_w)$, позволяющего с некоторыми изменениями алгоритма 1 строить *дельта деревья*.

Введем ограничения на пары вершин (v_b, v_w) , которые могут попасть в множество пар соответствия M :

- Обе вершины должны быть одновременно либо *файлами*, либо *директориями*, иначе говоря $v_b \in B, v_w \in B$ или $v_b \in N, v_w \in N$.
- Обе вершины должны принадлежать одному классу, то есть $\exists k : v_b \in C_k, v_w \in C_k$.
- Родители тех вершин, которые образуют пару соответствия, должны также содержаться в M , то есть $v_b \in C_{p_b}, v_w \in C_{p_w}$, тогда $(v_b, v_w) \in M \Rightarrow (p_b, p_w) \in M$.

Для построения множества пар соответствия M используется следующий подход:

1. Корни деревьев образуют пару соответствия: $(r_b, r_w) \in M$.
2. Добавляем в M пары вершин из множеств $N_b \subset V_b$ и $N_w \subset V_w$ с одинаковыми идентификаторами.
3. Введем в рассмотрение промежуточное множество пар M' и добавим в него пары вершин из множеств $B_b \subset V_b$ и $B_w \subset V_w$ с *похожим* содержанием. Напомним, что *блоки* с *похожим* содержанием находятся при решении задачи *LCS*, описанной в предыдущей главе.
4. Пусть $(p_w, p_b) \in M$, C_w и C_b множества детей вершин p_w и p_b соответственно. Сопоставим вершины $v_i \in C_b$ и $v_j \in C_w$, которые имеют разные идентификаторы $name(v_i) \neq name(v_j)$. Этот этап построения множества M описан в виде алгоритма 4. На выходе данного алгоритма получаем множество пар соответствий M .

Описанный подход позволяет изменить алгоритм 1 следующим образом:

- Обход вершин альтернативного *GAT дерева* происходит в глубину за счет того, что перемещения могут происходить только среди детей одной вершины.
- Достаточно одного обхода альтернативного *GAT дерева*, что позволяет построить *дельта дерево* с использованием интерфейса *editor*.
- Множество пар соответствия $Matching(T_b, T_w)$ содержит перемещенные вершины, что позволяет непосредственно во время обхода изменять *ссылки* между *блоками* (свойства *jvp.ref*).

Таким образом модифицированный алгоритм, разработанный в рамках настоящей работы, позволяет описать разницу между двумя *GAT деревьями* с помощью последовательности вызовов интерфейса *editor*, что в свою очередь позволяет отправить на *репозиторий* информацию, необходимую для операции *commit*.

3.3 Алгоритм структурного слияния

Следующей задачей является применение последовательности команд интерфейса *editor*, описывающих некоторое *дельта дерево* ΔT , к *GAT дереву*, соответствующему файлу *рабочей копии*. Как и прежде будем обозначать *базовую версию* как T_b , а *рабочую версию* как T_w .

Следует осознавать разницу в двух следующих операциях:

Algorithm 4 $\text{match}(p_b, p_w, M')$

{Рассмотрим детей вершин}

$C_w := C_{p_b}$

$C_b := C_{p_w}$

{Перебираем всевозможные пары детей с учетом введенных ограничений}

for $v_b \in C_b$ **do**

if $v_b \in B_b$ **then**

$v_w^* \in C_w : (v_b, v_w^*) \in M'$, разница между v_b и v_w^* минимальна

$M := M \cup \{(v_b, v_w^*)\}$

 {Удаляем из M' все пары содержащие v_b или v_w^* }

$\text{remove}(M', v_b, v_w^*)$

else

for $v_w \in C_w \cap N_w$ **do**

 {Определяем коэффициент соответствия $K_{b,w}$ }

$K_{b,w} := \frac{|V_{\text{match}}|}{|V_{\text{total}}|}$

 { V_{match} — множество вершин поддеревьев $v_{b,i}$ и $v_{w,j}$, таких что $(v_{b,i}, v_{w,j}) \in M \cup M'$, а V_{total} — множество всех вершин, входящих в каждое из поддеревьев }

$v_{w^*} : K_{b,w^*}$ максимален

 {Если коэффициент больше некоторого порогового значения K , то v_b и v_w^* образуют пару соответствия}

if $K_{b,w^*} > K$ **then**

$M = M \cup \{(v_b, v_{w^*})\}$

$\text{match}(v_b, v_{w^*}, M')$

else

 {Для v_b не найдено соответствия}

1. *update*.

Дельта дерево ΔT описывает разницу между T_b и некоторым *GAT* деревом T_1 , которое является версией структуры файла в указанной ревизии.

2. *merge*.

Дельта дерево ΔT описывает разницу между двумя *GAT* деревьями T_1 и T_2 , которые соответствуют ревизиям, указанным в параметрах операции.

В данном случае не имеет смысла строить $\Delta T_w : T_b \xrightarrow{\Delta T_w} T_w$, так как не удастся одновременно применять и ΔT , и ΔT_w . Связано это с тем, что обход *дельта деревьев* строго определен, и в общем случае обходы деревьев ΔT и ΔT_w различны.

В настоящей работе для слияния *GAT деревьев* применяется алгоритм 5. На вход ему подаются T_b , T_w и $\Delta T : T_1 \xrightarrow{\Delta T} T_2$, где T_1 и T_2 являются некоторыми *GAT деревьями*, хранимыми в *репозитории*. На выходе получаем дерево T'_w , являющееся результатом слияния. По ходу работы алгоритм изменяет дерево T_w , промежуточные деревья обозначены как T'_w .

Рассмотрим функции, которые используются в данном алгоритме:

matchWithHistory(v_δ, T'_w).

v_δ — вершина, переданная из *репозитория*, необходимо найти ей соответствие среди вершин T'_w с тем же ограничением, что вводились для алгоритма 4, то есть ищем соответствующую вершину среди детей вершины, которая соответствует родителю v_δ . Запрашивать у *репозитория* поддерево вершины v_δ крайне накладно, поэтому реализация использует иной подход:

- Сопоставляем вершину с идентификаторами $name(v_\delta)$, если такая существует.
- Запрашиваем историю перемещений, для каждой вершины с учетом ограничений. Ищем $name(v_\delta)$ среди историй вершин. Чем ближе вхождение идентификатора v_δ в истории вершины, тем более *похожа* эта вершины на v_δ . Сопоставляем наиболее *похожую* вершину.
- Если идентификатор вершины не нашелся ни в одной из историй, то соответствия не найдено.

Важно учесть, что операция получения истории данной вершины представляет один запрос к *репозиторию*, что довольно эффективно.

subtreeHasLocalMods(v_w)

Необходимо определить, изменилось ли поддерево вершины $v_w \in V_w$. Для этого находим вершину $v_b \in V_b : (v_b, v_w) \in Matching(T_b, T_w)$, если такой не нашлось, то возвращаем *true*, иначе строим $\Delta T = Compare(T_b, T_w)$ и возвращаем *true*, если ΔT не пусто, и *false* иначе.

localMod(v_w)

Необходимо определить, изменилась ли вершина вершины $v_w \in V_w$ относительно $v_b \in V_b$. Для этого находим вершину $v_b \in V_b : (v_b, v_w) \in Matching(T_b, T_w)$, если такой не нашлось, возвращаем *add*(v_w),

postMerge(T'_w)

После того, как мы обошли ΔT и применили к T_w изменения, мы получили T'_w . Теперь необходимо обновить значения свойств *jvn:ref* на тех вершинах, которые первый обход не изменил. Для этого мы обходим T'_w и для всех *блоков*, имеющих *ссылки*, обновляем их значения:

- Если вершина, на которую указывала ссылка, удалена, то помечаем вершину с этой *ссылкой* как конфликтную.
- Если вершина, на которую указывала ссылка, перемещена, то устанавливаем *ссылку* на ту вершину, куда произошло перемещение.

logConflict($v_\delta, edit_\delta$)

Добавление информации о конфликте, которая необходима во время кодогенерации для внесения маркеров конфликтов.

В алгоритме также используется функция *mergeMods*($edit_1, edit_2$), описание ее поведения представлено в таблице 1.

Algorithm 5 Merge($\Delta T, V_b, V_w$)

{Строим множество пар соответствия $Matching(T_b, T_w)$ }
 $M := Matching(T_b, T_w)$
{Обходим ΔT в порядке, переданном из *репозитория*}
for $v_\delta \in \Delta T$ **do**
 {Рассмотрим модификацию, действующую на v_δ }
 $edit_\delta : v_\delta \leftarrow edit_\delta$
 {Находим вершину $v_w \in T'_w$, соответствующую v_δ }
 $v_w := matchWithHistory(v_\delta, T'_w)$
 if $v_w = null$ **then**
 if $edit_\delta = add(v_\delta)$ **then**
 {Применяем добавление вершины v_δ к T'_w }
 else
 $logConflict(v_\delta, edit_\delta)$
 else
 if $edit_\delta = delete(v_\delta)$ и $subtreeHasLocalMods(v_w)$ **then**
 $logConflict(v_\delta, edit_\delta)$
 else
 {Определяем $edit$ — локальное изменение v_w }
 $edit := localMod(v_w)$
 $e := mergeMods(edit, edit_\delta)$
 {Применяем модификацию e к T'_w }
 $postMerge(T'_w)$

4 Результаты

4.1 Реализация

В состав данной работы включены исходные коды системы Jetversion, которые представляют собой модификацию библиотеки SVNKit — Java версии клиента системы Subversion [13]. Для более подробного изучения результатов, полученных автором, следует рассмотреть следующие компоненты программы:

- Реализация *GAT деревьев* представлена в исходном коде в виде набора *провайдеров вершин GAT дерева*, реализации интерфейсов *IOVSEntryProvider* и *IOVSEntryProviderLookup*.
- Реализация основной части алгоритма сравнения *базового GAT дерева* и его *рабочей версии* представлена в классе *OVSDiffContext*.
- Реализация алгоритма слияния *GAT деревьев* 5 представлена в виде изменения класса *SVNMergeDriver* и большей частью описана в классе *SVNExtendedMergeDriver*.

4.2 Применение

Типичный процесс работы пользователя клиента системы Jetversion представлен ниже.

```
$ jsvn checkout http://repository working-copy
```

Создали *рабочую копию*. На данном этапе действуют те бинарные алгоритмы, которые используются системой Subversion.

```
$ jsvn propset jvn:gat ru.spbu.jvn.gat.JavaGAT Example.java
```

Установили на файле Example.java специальное свойство, которое определяет, что для данного файла необходимо версионировать структуру его содержания. Значение ru.spbu.jvn.gat.JavaGAT определяет парсер, кодогенератор и набор *провайдеров*, которые будут использоваться.

```
$ jsvn commit Example.java
```

Отправляем на *репозиторий* описание структуры *GAT дерева*, соответствующего файлу Example.java. На данном этапе в работу вступает алгоритм сранения *GAT деревьев*.

```
$ jsvn update Example.java
```

Получение из *репозитория* изменений сделанных другими пользователями. На данном этапе работает алгоритм слияния *GAT деревьев*.

```
$ jsvn merge -r N:M branch/Example.java Example.java
```

Получение из *репозитория* изменений сделанных другими пользователями в альтернативной ветке развития branch. На данном этапе также работает алгоритм слияния *GAT деревьев*.

```
$ jsvn resolve Example.java
```

Если на предыдущих двух этапах произошли конфликты, то для дальнейшей работы их необходимо исправить и указать это клиенту.

Как видно, интерфейс видоизмененного клиента идентичен интерфейсу исходного клиента, но реализованные в нем алгоритмы приводят к ряду изменений в работе команд, укажем основные из них.

Операция *commit* не будет отправлять на *репозиторий* изменений в файле, если его содержание изменилось таким образом, что структура внутреннего представления файла осталась прежней. Такое может произойти, например, когда изменилось форматирование кода, или, когда изменился порядок декларации методов и переменных. Пример 1 иллюстрирует такое изменение.

Базовая версия	Рабочая версия
<pre>class Example { String name; String toString() { return name; } }</pre>	<pre>class Example { String toString() { return name; } String name; }</pre>

Пример 1: Модификации в коде,
не изменяющие структуру представления файла.

На примерах 2 и 3 представлен результат операции слияния файлов для случая переименования переменной. Зеленым цветом выделены добавленные участки кода, красным цветом отмечены удаленные, синим

выделен элемент, находящийся в конфликтном состоянии, а изменения в коде отмечены жирным шрифтом.

Версия 1	Версия 2	Результат слияния
class Example {	class Example {	class Example
String name;	String myName ;	String myName ;
String toString() { return name; }	String toString() { return myName ; }	String toString() { return myName ; }
String getName() { return name; }		String getName() { return myName ; }
}	}	}

Пример 2: Переименование переменной в добавленном методе.

Версия 1	Версия 2	Результат слияния
class Example {	class Example {	class Example
String name;	String name;	String name;
String toString() { return name; }	String toString() { return " Simon "; }	String toString() { return " Simon "; }
String getName() { return name; }		String getName() { return name ; }
}	}	}

Пример 3: Конфликт удаления переменной из добавленного метода.

4.3 Преимущества и недостатки

Представленная функциональность обладает рядом преимуществ:

- Как говорилось в начале работы, структурное слияние файлов не является абсолютно новым подходом, алгоритмы представленные в данной работе наследуют свойства своих аналогов:
 - Различное форматирование, применяемой разными пользователями, не вызывает конфликтов слияния.
 - Возможность определения того форматирования, которое более предпочтительно для конкретного пользователя.
- Возможность распознавания и применения переименований переменных и функций на этапе слияния файлов.
- Система разрабатывалась именно как расширение функциональности исходной системы, поэтому стало возможным совмещать как структурные, так и бинарные алгоритмы.
- Так как *репозиторий* способен получать информацию о внутреннем устройстве файла, он также предоставляет возможность получения метаданных для изменений внутри файла. Иначе говоря, просматривая лог, пользователь способен узнать, какая часть файла была изменена, а также кем она была изменена и когда.

Однако, наряду с этими фактами существует совокупность в разной степени отрицательных особенностей:

- Изменения в хранении файлов привели к тому, что некоторая часть функциональности системы **Subversion** была запрещена, были частично удалены следующие особенности [3]:
 - Merge tracking: большое число копирований файлов и директорий приводит к увеличению значения свойства *svn:mergeinfo*, что в итоге приводит к замедленной работе операции *merge*.
 - Depth support: так как файлы в *репозитории* представлены как директории, он не способен определить глубину обхода директорий, которые необходимо передать клиенту.
- Клиент системы *Jetversion* не способен генерировать описание разницы между двумя файлами в формате, поддерживаемом другими приложениями.

- Так как содержание файла определяется кодогенератором, то у пользователей различные части кода могут соответствовать различным строкам в файле, что делает невозможным использование привычных ссылок на линию в файле.

5 Заключение

Данная работа представляет собой глубокое изменение клиентской части системы **Subversion**, позволяющее этой системе работать с внутренним представлением файлов. В то же время эта работа дает огромное пространство для дальнейших исследований и разработок. Наиболее важным направлением развития, несомненно, является внедрение системы в интегрированные средства разработки. Использование структур, с которыми работают подобные средства, а также получение информации о изменениях в этих структурах позволит качественно улучшить существующее решение.

Список литературы

- [1] Version control system.
http://en.wikipedia.org/wiki/version_control_system.
- [2] Distributed source control management.
http://en.wikipedia.org/wiki/distributed_revision_control.
- [3] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Version control with subversion, 2009.
- [4] Beat Fluri, Michael Wursch, Martin Pinzger, and Harald C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE transactions on software engineering*, 33:725–741, 2007.
- [5] Albert Zündorf, Jörg Wadsack, and Ingo Rockel. Merging graph-like object structures. In *In Proceedings of the Tenth International Workshop on Software Configuration Management*, 2001.
- [6] Campbell B. Fraser, Robert W. Irving, and Martin Middendorf. Maximal common subsequences and minimal common supersequences, 1995.
- [7] Eugene Myers. An $O(ND)$ difference algorithm and its variations. In *Algorithmica*, pages 251–266, 1986.
- [8] D. Korn, J. MacDonald, J. Mogul, and K. Vo. RFC 3284 – the VCDIFF generic differencing and compression data, 2002.
- [9] Mervi Ranta and Tancred Lindholm. A 3-way merging algorithm for synchronizing ordered trees - the 3dm merging and differencing tool for XML, 2001.
- [10] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. In *In Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 146–157, 2007.
- [11] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-molina, and Jennifer Widom. Change detection in hierarchically structured information. In *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.

- [12] Tree editor interface.
http://svn.collab.net/repos/svn/trunk/subversion/libsvn_delta/editor.c.
- [13] SVNKit – Java software library for working with the Subversion VCS.
<http://en.wikipedia.org/wiki/svnkit>.