УДК 004.75
DOI 10.25205/1818-7900-2020-18-3-44-56

Разработка многопользовательской модели данных с использованием подхода Operational Transformation

К. Е. Салтук

Новосибирский государственный университет
Новосибирск, Россия

Аннотация

Предоставление пользователям возможности одновременно редактировать общую модель данных является непростой задачей для современных информационных систем. Большинство существующих алгоритмов требуют наличия постоянного сетевого постоянного сетевого подключения пользователей к системе, в противном случае, их намерения могут быть искажены и потеряны. Однако наличие постоянного сетевого подключения между пользователями накладывает сильные ограничения на возможные сценарии использования информационной системы.

Для решения этой проблемы в работе предлагается использование гибридного подхода: за основу был взят механизм Operational Transformation, который вводит концепции преобразования пользовательских намерений; вместе с тем, для обеспечения возможности работы с моделью без сетевого подключения используется концепция системы контроля версий, которая оперирует ветками и коммитами. Таким образом, в коммитах системы контроля версий предлагается хранить пользовательские намерения в виде операций. Для совместной работы этих двух систем была проанализирована работа оригинального алгоритма преобразования операций и предложена новая модель поддержания консистентности состояния которая поддерживает обработку возможных конфликтов пользовательских намерений. Благодаря использованию модифицированного алгоритма преобразования операций, появляется возможность производить слияние веток в системе контроля версий, таким образом реализуя необходимую функциональность для многопользовательской работы с общей моделью.

Использование описанного в работе комбинированного подхода позволит строить информационные системы, которые позволят пользователям модифицировать общую модель данных без постоянного сетевого подключения.

Ключевые слова

многопользовательская работа, operational transformation, система контроля версий

Для цитирования

Салтук К. Е. Разработка многопользовательской модели данных с использованием подхода Operational Transformation // Вестник НГУ. Серия: Информационные технологии. 2020. Т. 18, № 3. С. 44–56. DOI 10.25205/1818-7900-2020-18-3-44-56

Building Collaborative Data System on Top of the Operational Transformations Algorithm

K. E. Saltuk

Novosibirsk State University
Novosibirsk, Russian Federation

Abstract

Providing users with the way to collaboratively edit shared model is the challenge of complex informational systems. The majority of existing methods require online connection between users, otherwise, users' intentions can be lost af-

ter a certain delay. The aim of this research is to build a collaborative system that allows users to edit shared model in offline manner without losing their intentions. Firstly, commit-based system was selected as underlying data storage. Secondly, we examined existing collaborative algorithms and selected operational transformation as the most perspective for this scenario. Thirdly, we described the way to merge commits, that stores operations, using modified operational transformation algorithm. Key features of modified operational transformation algorithm include ability to handle resolving of merge conflicts, when intentions of two users are conflicting. To implement this, we consider the set of commits branches as ordered set. As a result, we developed a version control system, that stores user's intentions and can automatically resolve conflicts on merge. This approach can be applied to any informational system, which is able to save user's input as the set of operations.

Keywords

operational transformation, collaboration, data system, version control system

For citation

Saltuk K. E. Building Collaborative Data System on Top of the Operational Transformations Algorithm. *Vestnik NSU. Series: Information Technologies*, 2020, vol. 18, no. 3, p. 44–56. (in Russ.) DOI 10.25205/1818-7900-2020-18-3-44-56

Введение

Вместе с развитием информационных систем возрастает и их сложность. Время, затрачиваемое пользователями на работу, может значительно увеличиваться при работе над всё более сложными и крупными проектами. Один из способов ускорить работу заключается в организации многопользовательской работы над моделью. Большинство существующих алгоритмов требует наличия постоянного сетевого подключения, в противном случае, их намерения могут быть искажены и потеряны. Однако наличие постоянного сетевого подключения накладывает сильные ограничения на возможные сценарии использования информационной системы. Так, становится невозможным организовать автономную работу пользователей на сколь либо продолжительный промежуток времени.

Есть хорошо зарекомендовавший себя способ организации многопользовательской работы над текстовыми данными, который используется командами разработчиков для хранения файлов с исходным кодом – использование систем контроля версий. Основные концепции, которые вводятся системой контроля версий заключаются в том, что пользователи фиксируют свои изменения в коммитах. Коммиты имеют родителей, таким образом, граф коммитов может содержать развивающиеся параллельно ветви коммитов с исходным кодом [1; 2]. В случае когда автономная работа пользователя завершена, он может слить свою ветку с основной путём создания merge-коммита, у которого будет два родителя. Понятно, что при слиянии могут возникнуть конфликтные ситуации, когда в двух ветках был модифицирован один и тот же участок текстового файла. В таких случаях при формировании merge-коммита система контроля версий запросит пользовательский ввод, и разработчик должен будет решить его явно, вручную сформировав корректное состояние файла. Данный подход используется в таких системах контроля версий, как Git или Mercurial [1].

Понятно, что при использовании такого подхода в исходном виде, возможно как выявление конфликтов системой контроля версий, при отсутствии конфликтующих изменений с точки зрения модели, так и не обнаружение конфликта в тех случаях, когда модель становится неконсистентной. Примером первой ситуации может служить одновременное добавление двух методов в класс на любом объектно-ориентированном языке программирования, когда порядок методов не несёт никакой семантической нагрузки, но системой контроля версий данная ситуация будет воспринята как конфликт, так как текстовое описание этих методов будет находиться на совпадающих строчках документа. Примером обратной ситуации может служить переименование существующего метода в классе с одновременным добавлением использования этого метода в другом методе. С точки зрения системы контроля версий

это не является конфликтом, и после слияния во втором методе останется использование уже несуществующего имени.

Причина этих проблем заключается в том, что существующие системы контроля версий хранят лишь слепки файлов, а разрешение конфликтов происходит без использования семантики этих данных. Существующие исследования [3; 4] показывают, как можно улучшить ситуацию для языков с определённой грамматикой и избежать конфликтов.

Вместе с тем, существуют подходы для организации многопользовательской работы над моделями данных, не ограниченных текстовыми файлами. Среди таких подходов можно выделить Operational Transformation [5]. В данной работе показано, как можно построить систему контроля версий, работа которой будет базироваться на принципе Operational Transformation.

Operational Transformation

Подход Operational Transformation основан на такой концепции, согласно которой вся модель представлена как последовательность произведённых пользователем операций. Для каждой пары операций, которые возможно провести над моделью, необходимо определить трансформацию. В этой трансформации необходимо учесть, какой эффект одна операция может оказать на другую, и в соответствии с этим, преобразовать параллельно совершённую операцию [6]. Будем обозначать такую трансформацию как $T(O_1, O_2)$. Результатом трансформации будет считаться операция $O_1' = T(O_1, O_2)$.

Для демонстрации этого подхода рассмотрим следующий пример: пусть у нас есть модель, состояние которой содержит текст; также пусть над ней будут определены две операции: операция Insert(n, «c»), которая вставляет символ «c» на индекс *n*, и операция Remove(n), которая удаляет символ с индексом n. Рассмотрим ситуацию, когда два пользователя редактируют текст «SUN», при этом первый решает дописать в начало текста символ «N», применив операцию Insert(0, «N»), а второй пользователь хочет удалить последний символ, создав операцию Remove(2), после чего они обмениваются совершёнными операциями и применяют их к своим моделям. Если бы они применили эти операции «как есть», то их результаты были бы разными, как показано на левой части рис. 1.

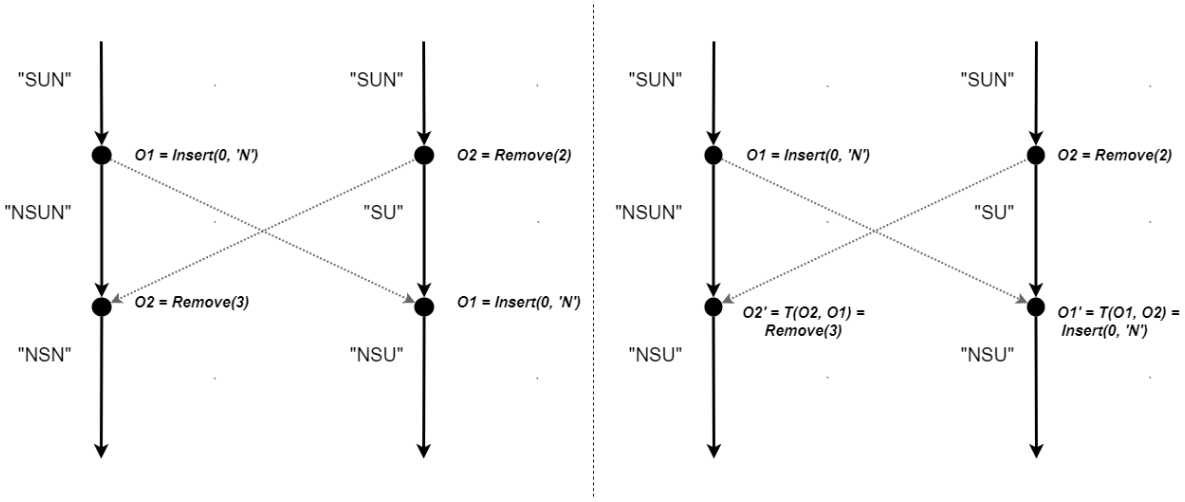

Рис. 1. Демонстрация работы трансформации операций для поддержания консистентности
Fig. 1. Using of operational transformation principle for consistency maintenance

Конечно, в таком виде модель не являлась бы консистентной, так как вместо ожидаемого результат «NSU» у первого клиента будет текст «NSN». Именно по этой причине в Operational Transformation используются трансформации операций. В данном случае в трансформации операции O2 поверх операции O1 нужно учесть, что в начало текста был вставлен один символ. Таким образом оригинальная операция должна быть трансформирована так, чтобы она удаляла символ не с индекса 2, а с индекса 3, как показано на правой части рис. 1.

Стоит заметить, что сам подход не накладывает никаких ограничений на модели, и для каждой конкретной задачи можно описать свою модель и множество операций над ней, определив трансформацию для каждой пары операций относительно друг друга.

Оригинальный алгоритм Operational Transformation предполагает, что трансформацию можно определить для любой пары операций [5; 6], однако на практике такое возможно далеко не всегда, ведь намерения пользователей могут противоречить друг другу. Предположим, что мы разрабатываем модель для систем автоматизированного проектирования. Модель систем автоматизированного проектирования будет содержать различные объекты с набором свойств. Пусть два пользователя независимо друг от друга меняют одно и то же свойство, устанавливая различные значения. При описании моделей консистентности Operational Transformation возможные конфликты пользователей разрешаются неявно, например, с использованием timestamp-вектора [5; 7]. Таким образом, в нашем примере намерение одного из пользователей было бы потеряно. Такой подход применим в системах, где пользователи модифицируют модель в режиме реального времени, однако при использовании системы контроля версий такая неявная потеря изменений недопустима. Поэтому в данной работе будет рассматриваться ситуация, когда трансформация операций не может быть совершена автоматически описанными ниже низ лежащим алгоритмом Operational Transformation и в соответствующем merge-коммите может храниться дополнительная информация о принятом пользователем решении для разрешения конфликта.

Устройство системы контроля версий данных

Основная функция, которую должна поддерживать система контроля версий – это создание коммитов с пользовательскими изменениями и переключение между созданными коммитами по всему графу коммитов. Понятно, что в коммитах такой системы должны храниться совершённые пользователем операции, которые просто будут применяться при переключении. Теперь рассмотрим коммит со слиянием в самом простом случае, когда он объединяет два коммита с одной операцией в каждом, как показано на рис. 2.

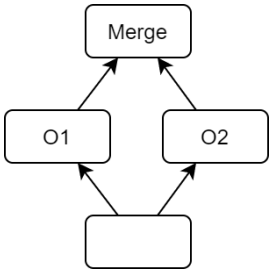

Рис. 2. Пример простого merge-коммита
Fig. 2. Example of simple merge-commit

Заметим, что в данном случае в результате переключения на коммит Merge над состоянием модели пользователя в зависимости от пути, по которому пользователь последовательно переключался, должны примениться операции O_1 и $O_2' = T(O_2, O_1)$, либо O_2 и $O_1' = T(O_1, O_2)$. В случае, когда функция трансформации операций T определена и детерминирована, в ком-

мите Merge можно не хранить никакой дополнительной информации, так как по дереву коммитов можно восстановить всю необходимую информацию. Но как уже было показано в предыдущей главе, для некоторых пар не получится определить полностью автоматическую трансформацию. В таких случаях необходимо сохранить в коммите Merge дополнительную информацию, которая позволит восстановить трансформацию с учётом принятого пользователем решения.

Алгоритм переключения коммитов

Для описания и демонстрации работы алгоритма переключения между коммитами будем использовать граф коммитов, изображённый на рис. 3.

Каждый коммит для простоты содержит ровно одну операцию, при использовании имени коммита в выражениях трансформации операций будет подразумеваться использование единственной операции из этого коммита.

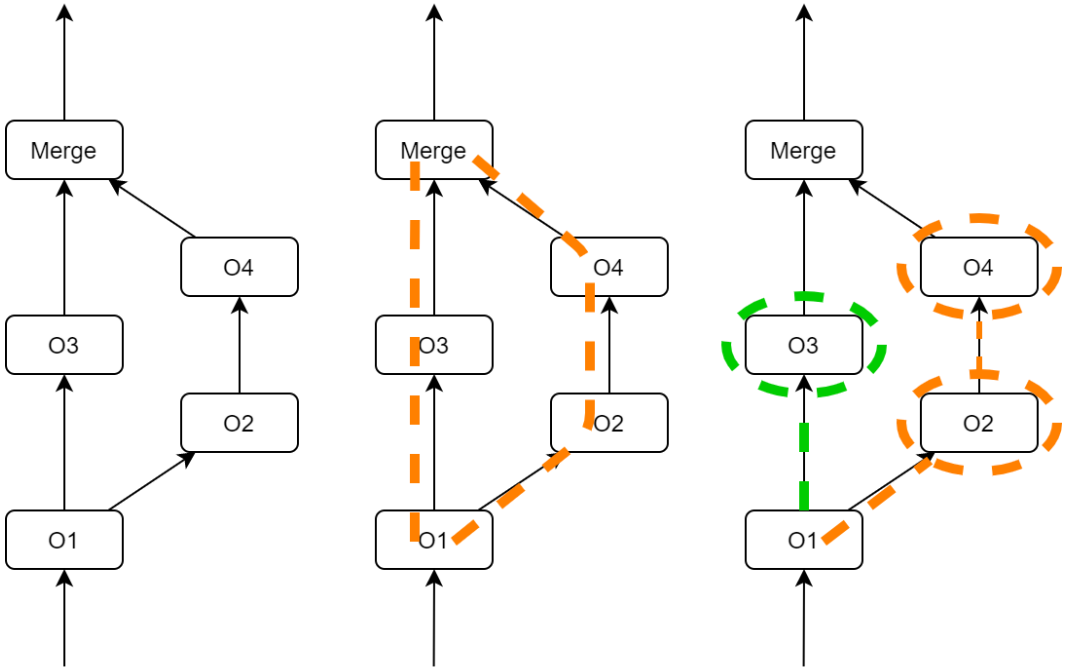

Рис. 3. Поиск коммитов для трансформации при прохождении через merge-commit
Fig. 3. Searching for commits to transform while checkout on merge-commit

Для начала заметим, что переключение между коммитами всегда может быть выражено в виде последовательных переключений по цепочке, связывающей эти коммиты. Заметим, что цепочек может быть несколько. В рамках данной главы будем рассматривать одну из них, консистентность такого подхода будет рассмотрена далее. Так, переключение с коммита O_1 на коммит Merge может быть выражено последовательным переключением с коммита O_1 на O_3, и переключением с O_3 на Merge. Таким образом, достаточно описать работу алгоритма по переключению на коммит с его непосредственного родителя. Соответственно, необходимо рассмотреть два случая: переключение на коммит, являющийся слиянием, и на коммит, не являющийся таковым.

Рассмотрим сценарий, когда происходит переключение на коммит, не являющийся слиянием (например, переключение с коммита O_1 на коммит O_3). Когда пользователь находится на коммите O_1, в состоянии его модели должны быть применены все операции из коммита O_1 и всех коммитов, предшествующих ему, как это реализовано в системах контроля версий [1]. При переключении на коммит O_3 должны быть применены операции из этого коммита и операции из всех коммитов, предшествующих ему. Таким образом для переключения коммита необходимо применить операцию O_3 поверх текущего состояния. Заметим, что её не нужно трансформировать в этом случае, так как она была создана непосредственно после O_1 и уже учитывает изменения из этого коммита.

Теперь рассмотрим второй сценарий, когда происходит переключение на коммит, являющийся слиянием. У таких коммитов есть два предка, в указанном примере это коммиты O_3 и O_4, которые являются предками коммита Merge. Пусть происходит переключение с коммита O_3 на коммит Merge. После переключения на этот коммит в состоянии модели пользователя должны быть применены операции из всех родителей и достижимых их них коммитов, то есть в рассматриваемом случае должны быть дополнительно применены коммиты O_2 и O_4. Но коммиты O_2 и O_4 должны быть применены на состоянии, которое получилось после O_3, хотя изначально они были созданы на состоянии, образованном коммитом O_1. Соответственно, тут можно применить подход Operational Transformation и трансформировать изначальные пользовательские намерения из прилегающей ветки.

Чтобы понять, поверх каких операций необходимо трансформировать операции из прилегающей ветки, рассмотрим рис. 3. На втором графе пунктирными линиями показан путь до общего предка двух прилегающих веток. Заметим, что в более сложных деревьях коммитов может существовать несколько альтернативных путей. В рамках этой главы будем рассматривать любой наикратчайший путь до общего предка. Консистентность такого подхода будет рассмотрена в следующей главе.

Таким образом, от двух родителей merge-коммита есть два соответствующих пути до их общего предка, на третьем графе рис. 3 коммиты из первого пути выделены зелёным цветом, и коммиты из второго пути выделены оранжевым цветом. Чтобы восстановить причинно-следственную связь, необходимо трансформировать операции из второго пути таким образом, чтобы они производились поверх состояния, с которого выполняется процесс обновления на коммит Merge. Таким образом, необходимо трансформировать все операции из второго пути поверх всех операций из первого пути, после чего применить их. В рассматриваемом случае, согласно этому правилу должны быть применены операции $O_2' = T(O_2, O_3)$ и $O_4' = T(O_4, O_3)$.

Таким образом алгоритм сможет восстанавливать состояние до любого коммита из всей истории изменений.

Консистентность модели

Как было отмечено в предыдущей главе, в дереве коммитов могу существовать разные пути, с помощью которых можно достичь одного и того же коммита. При этом может нарушиться консистентность модели. Для начала определим, что мы будем подразумевать под консистентностью модели данных.

В любой момент времени состояние модели данных характеризуется следующими параметрами:

- текущий активный коммит;
- локально применённые операции над рабочей копией, которые ещё не были зафиксированы в новом коммите;
- рабочая копия, в которой содержится восстановленное состояние модели. Данная копия изменяется при применении операций.

Для обеспечения консистентности модели данных необходимо, чтобы при эквивалентных активных коммитах и одинаковом наборе локально применённых операций, рабочая копия имела одно и то же состояние.

Понятно, что в случае, когда операции, которые применяет пользователь над своей рабочей копией, детерминированы, необходимым и достаточным условием для поддержания консистентности будет обеспечение одинакового состояния рабочей копии при переключении на один и тот же коммит. Поэтому далее сосредоточимся на обеспечении эквивалентного состояния рабочей копии при переключении на один и тот же коммит.

Коммутативность трансформаций

Рассмотрим минимальную ситуацию, в которой можно попасть на один коммит по двум разным путям: для этого достаточно одного коммита со слиянием, который будет соединять два различных коммита с общим предком, как показано на рис. 4.

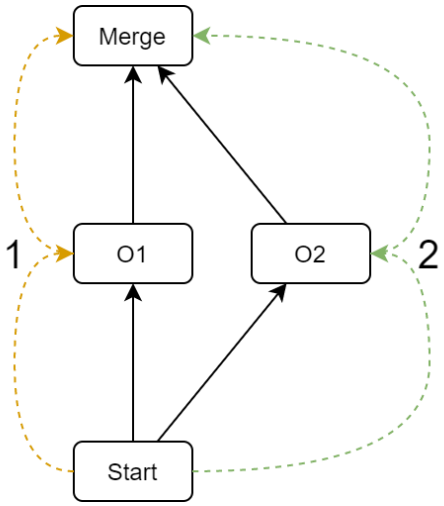

Рис. 4. Изображение двух путей для переключения на merge-коммит
Fig. 4. Two paths for checkout on merge-commit

Как видно, в таком графе коммитов есть два пути из коммита Start в коммит Merge. Путь 1 проходит через коммит O_1, в то время как путь 2 проходит через коммит O_2. В соответствии с алгоритмом переключения коммитов, описанном в предыдущей главе, при прохождении по пути 1 над рабочей копией будут применены операции O_1 и $O_2' = T(O_2, O_1)$, однако при прохождении по пути 2 будут применены O_2 и $O_1' = T(O_1, O_2)$. Таким образом для обеспечения консистентности является необходимым следующее условие:

> для любого состояния S и для любых операций O_1 и O_2, состояние S_1, полученное последовательным применением операций O_1 и $O_2' = T(O_2, O_1)$ на состоянии S, должно быть эквивалентным состоянию S_2, полученному последовательным применением операций O_2 и $O_1' = T(O_1, O_2)$ на состоянии S.

Заметим, что данное условие накладывает ограничение именно на реализацию трансформации операций T. Будем называть данное свойство трансформации коммутативностью.

В рассмотренном выше случае подразумевается, что операции O_2 и O_1 могут быть слиты без конфликтов. Но предположим, что это конфликтующие операции, тогда в коммите с Merge будет сохранена дополнительная информация о разрешении конфликта. Будем обозначать такую информацию как $R(O_1, O_2)$, причём порядок аргументов не будет иметь значе-

ния, так как считается, что выбор пользователя от него не зависит, поэтому для единообразия будем записывать аргументы в отсортированном порядке. Трансформацию, которая принимает дополнительный параметр R, будем обозначать как $T(O_1, O_2, R(O_1, O_2))$. Понятно, что для обеспечения консистентности свойство коммутативности должно выполняться и в случае разрешения конфликта:

> для любого состояния S и для любых операций O_1 и O_2 и разрешения конфликта $R(O_1, O_2)$, состояние S_1, полученное последовательным применением операций O_1 и $O_2' = T(O_2, O_1, R(O_1, O_2))$ на состоянии S, должно быть эквивалентным состоянию S_2, полученному последовательным применением операций O_2 и $O_1' = T(O_1, O_2, R(O_1, O_2))$ на состоянии S.

Транзитивность конфликтов

Рассмотрим граф коммитов, изображенный на рис. 5.

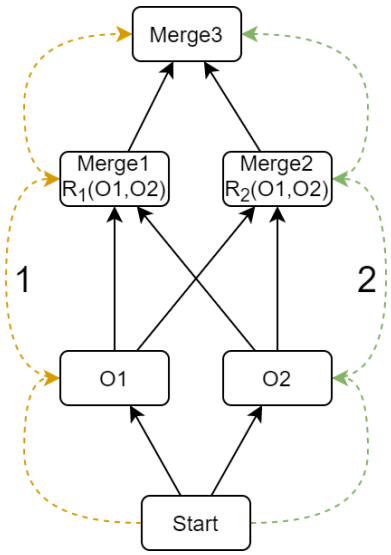

Рис. 5. Граф коммитов с двумя конфликтующими merge-коммитами
Fig. 5. Commits graph with two conflicting merge-commits

На данном графе также изображены два параллельных коммита O_1 и O_2, но в этот раз есть два коммита со слиянием: Merge1 и Merge2, причём операции O_1 и O_2 – конфликтующие. Таким образом в каждом из этих коммитов со слиянием есть разрешение конфликта операций O_1 и O_2 – $R_1(O_1, O_2)$ и $R_2(O_1, O_2)$, причём они могут быть различными. Рассмотрим цепочку коммитов 1. При переключении коммитов по этому пути над рабочей копией пользователя будут применены следующие операции: O_1, $O_2' = T(O_2, O_1, R_1(O_1, O_2))$. При переключении по пути 2 будут применены O_2, $O_1' = T(O_1, O_2, R_2(O_1, O_2))$. Но как уже была замечено, в коммитах Merge1 и Merge2 конфликты могут быть решены по-разному, из чего следует, что состояние на коммите Merge3 может различаться при переключении по первому и второму пути, что говорит о том, что в таком случае модель может стать неконсистентной.

Понятно, что для решения этой проблемы необходимо выбрать, какое из разрешений конфликтов должно попасть в результирующий коммит Merge3. Рассмотрим потенциальную возможность реализации этого подхода в общем случае. Пусть $R_1(O_1, O_2) \neq R_2(O_1, O_2)$, и в коммите Merge3 при очередном разрешении конфликта пользователь решил выбрать R_1.

Рассмотрим ситуацию, когда пользователь выбрал одно из конкурирующих разрешений конфликтов в коммите со слиянием, но переключается на этот коммит с коммита, в котором было применено другое разрешение конфликтов. Такая ситуация изображена на рис. 6.

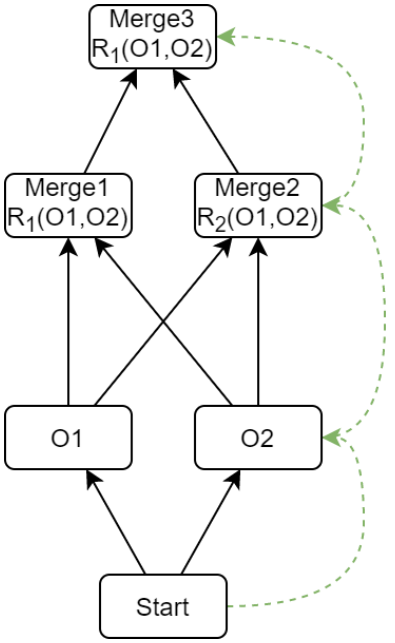

Рис. 6. Граф коммитов с разрешением транзитивного конфликта
Fig. 6. Commits graph with transitive conflict resolving

Как видно, когда пользователь находится на коммите Merge2, в его рабочей копии содержится результат последовательного применения следующих операций: O_2, $O_{1(R2)}$’ = = $T(O_1, O_2, R_2(O_1, O_2))$, но когда пользователь переключится на коммит Merge3, необходимо, чтобы в его рабочей копии был результат применения следующих операций: O_2, $O_{1(R1)}$’ = = $T(O_1, O_2, R_1(O_1, O_2))$. Понятно, что $O_{1(R2)}$’ в общем случае может не совпадать с $O_{1(R1)}$’. Таким образом, чтобы достичь необходимого состояния, необходимо изъять последнюю операцию $O_{1(R2)}$’ из пользовательского хранилища, вернув его в предыдущее состояние, после чего применить операцию, получившуюся в результате трансформации с учётом $R_1(O_1, O_2)$. Развивая этот сценарий дальше, рассмотрим слегка усложнённый граф коммитов, изображённый на рис. 7.

В данном графе коммитов после коммита Merge2 перед его слиянием с Merge 1 добавлен коммит O_3. Предположим, что O_3 не создаёт конфликтов с O_1. В таком случае, после переключения на коммит Merge3 по указанному на графе пути, в рабочей копии должно быть состояние, образованное следующими операциями: O_2, O_1’ = $T(O_1, O_2, R_2(O_1, O_2))$, и O_3. Но O_3 была сделана на состоянии, полученном применением O_1’ = $T(O_1, O_2, R_2(O_1, O_2))$. Однако на коммите Merge3 всё состояние должно быть рассчитано с условием, что разрешение конфликта между O_1 и O_2 было представлено через $R_1(O_1, O_2)$. Но операция O_3 уже основана на состоянии, в котором конфликт между O_1 и O_2 решён через $R_2(O_1, O_2)$.

Единственным решением для восстановления причинно-следственной связи может выступить обратная трансформация, которая бы позволила получить операцию O_3 над состоянием без O_1’ = $T(O_1, O_2, R_2(O_1, O_2))$, то есть необходимо введение функции обратной трансформации T^{-1}. В таком случае переключение с O_3 на Merge3 можно было бы реализовать, используя следующую трансформированную операцию: O_3’ = $T(T^{-1}(O_3, T(O_1, O_2, R_2(O_1, O_2))), T(O_1, O_2, R_1(O_1, O_2)))$.

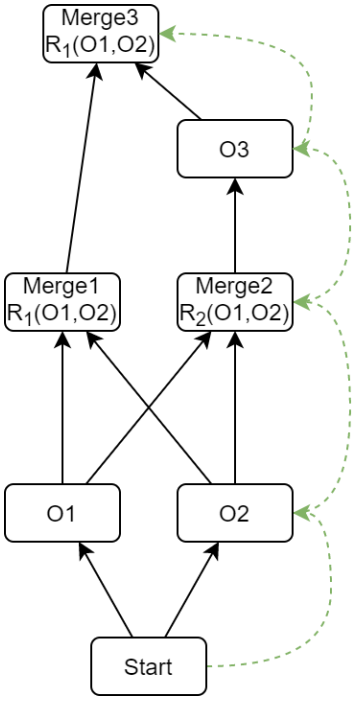

Рис. 7. Разрешением транзитивного конфликта с дополнительным промежуточным коммитом
Fig. 7. Transitive conflict resolving with additional inner commit

Из всего вышесказанного следует, что для обеспечения консистентности такой модели помимо коммутативности трансформаций необходимо введение ещё и обратной трансформации. Это в свою очередь накладывает дополнительные ограничения на реализацию различных моделей данных таким подходом.

Частичный порядок веток

Проанализируем, из-за чего возникает необходимость введения обратных трансформаций. Основная проблема заключается в том, что в параллельных ветках происходит разрешение одного и того же конфликта, вследствие чего дальнейшие операции уже производятся на несовместных состояниях, и в момент слияния таких веток одна из них в любом случае окажется в неконсистентном состоянии. Чтобы избежать таких ситуаций, нужно исключить принципиальную возможность разрешения конфликта более, чем один раз.

Чтобы гарантировать указанное выше свойство, можно ввести отношение порядка на множестве веток.

Для начала определим, что каждый коммит принадлежит ровно одной ветке. Такой подход используется в mercurial, но не в git [1; 2]. Коммиты в одной ветке должны представлять линейную структуру. Каждый коммит, не являющийся слиянием (кроме первого), имеет одного родителя. Соответственно ветка, в которой находится такой коммит, должна быть либо веткой родителя, либо новой веткой, ещё не встречающейся в графе коммитов. Коммит со слиянием имеет двух родителей, соответственно ветка, в которой он располагается, обязательно должна быть веткой одного из родителей. Нетрудно видеть, что при таком определении началом ветки не может быть коммит со слиянием.

Будем говорить, что ветка А является родительской веткой для ветки Б, если первый коммит в ветке Б имеет родителя в ветке А. Будем обозначать это отношение как Б<А.

Разрешим коммиты слияния в ветке А с родителями в ветках А и Б только в том случае, если ветка А является родительской для Б.

На рис. 8 показано, как будет выглядеть общая структура графа коммитов. На указанном изображении между ветками выстроен следующий порядок: Б<А, В<А, Г<В, Д<В.

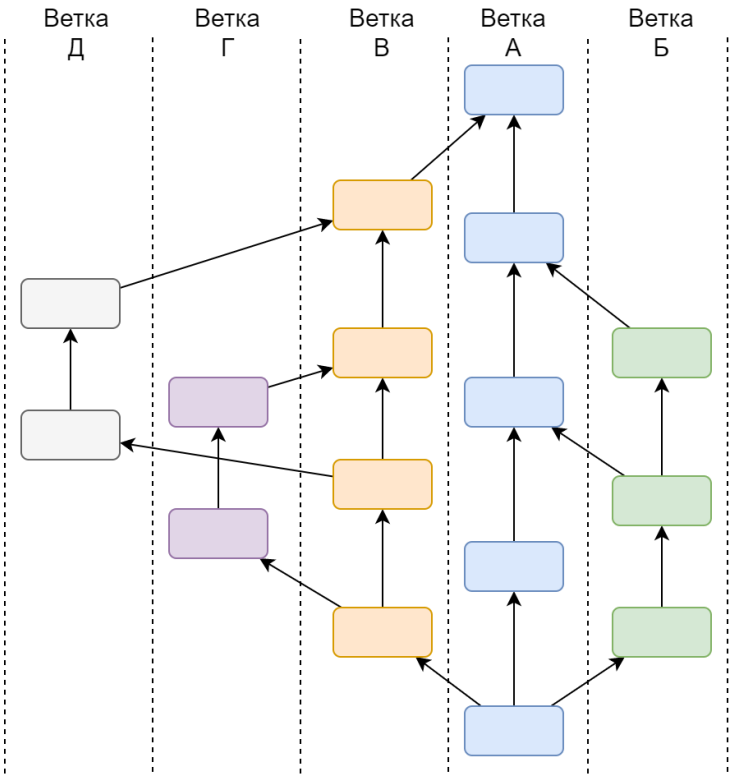

Рис. 8. Общая структура графа коммитов
Fig. 8. General structure of commits graph

Заметим, что такое ограничение на управление ветками не нарушает принятые способы работы с системами контроля версий и позволяет организовать параллельную работу над моделью данных [8]. Такая система позволяет вынести определённую работу в отдельную ветку и влить результат в родительскую ветку.

Покажем, что в такой модели множество коммитов линеаризуемо. Начнём обход с первого коммита главной ветки в модели и будем восстанавливать состояние по одному коммиту. В случае, когда следующий коммит этой ветки не является слиянием, линейность очевидна. Рассмотрим ситуацию, когда коммит является слиянием:

- так как разрешены слияния только с непосредственными родителями, то ветка, которая вливается в текущую, растёт из неё же;
- вливаемая ветка могла быть уже влита в текущую ранее по истории, таким образом есть часть коммитов от последнего вливания (либо от начала ветки, если его не было), которая ещё не была влита в текущую ветку.

В таком случае достаточно взять операции из последней части прилегающей ветки и трансформировать их относительно соответствующих операций своей ветки в соответствии с алгоритмом, приведённым ранее. В случае, когда в коммитах прилегающей ветки есть

коммиты слияния, необходимо аналогично повторить указанные действия для прилегающей ветки рекурсивно.

Нетрудно увидеть, что при таком подходе все перемещения операций для трансформаций происходят только в одну сторону в натуральном порядке, задаваемом множеством веток. Помимо того, что такой подход исключает возможность разрешить противоречия пользовательских намерений по-разному, он также исключает возможность, когда нужно будет трансформировать операции друг относительно друга и наоборот. Иными словами, если при слиянии для операций O_1 и O_2 была применена трансформация $O_2' = T(O_1, O_2)$, то данный подход гарантирует, что трансформации $O_1' = T(O_2, O_1)$ применено не будет.

Заключение

В результате работы была сформирована модель системы контроля версий, которая базируется на подходе Operational Transformation. Анализ применения подхода в явном виде показал, что требуется введение дополнительных ограничений для поддержания консистентности модели. Чтобы избежать сужения возможных моделей данных, для которых можно будет применить полученное решение, были добавлены ограничения на работу с ветками в системе контроля версий. Введённые ограничения не нарушают принятых принципов управления ветками и позволяют организовать многопользовательскую работу.

Таким образом, применимость данного решения для организации многопользовательской работы над какой-либо моделью данных ограничена только возможностью описать работу над этой моделью в виде последовательного применения операций и их трансформации относительно друг друга. В случаях, когда намерения пользователей противоречат друг другу и такая трансформация не может быть выполнена, возможно разрешение конфликта самим пользователем без потери консистентности модели данных.

Список литературы / References

1. **Chacon S., Straub B.** Pro git. Apress, 2014.
2. **Mackall M.** Towards a better SCM: Revlog and mercurial. Proc. Ottawa Linux Sympo, 2006, vol. 2, p. 83–90.
3. **Apel S. et al.** Semistructured merge: rethinking merge in revision control systems. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011, p. 190–200.
4. **Cavalcanti G., Borba P., Accioly P.** Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, 2017, vol. 1, no. OOPSLA, p. 1–27.
5. **Sun C., Ellis C.** Operational transformation in real-time group editors: issues, algorithms, and achievements. In: Proceedings of the 1998 ACM conference on Computer supported cooperative work. 1998, p. 59–68.
6. **Sun C. et al.** Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 1998, vol. 5, no. 1, p. 63–108.
7. **Cart M., Ferrie J.** Asynchronous reconciliation based on operational transformation for p2p collaborative environments. In: International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2007). IEEE, 2007, p. 127–138. DOI 10.1109/COLCOM.2007.4553821

8. **Krusche S., Berisha M., Bruegge B.** Teaching code review management using branch based workflows. In: Proceedings of the 38th International Conference on Software Engineering Companion. 2016, p. 384–393. DOI 10.1145/2889160.2889191

Материал поступил в редколлегию
Received
27.04.2020

Сведения об авторе

Салтук Константин Евгеньевич, студент, Новосибирский государственный университет (Новосибирск, Россия)
k@saltuk.ru

Information about the Author

Konstantin E. Saltuk, student, Novosibirsk State University (Novosibirsk, Russian Federation)
k@saltuk.ru