Version Control for Structuring Collaborative Decision-Making

BJÖRN EBBINGHAUS and MARTIN MAUVE, Heinrich Heine University Düsseldorf, Germany

The collaborative development of decision alternatives is a complex challenge. Our approach is to apply mechanisms from a domain that faces similar problems—version control systems used in software development. These mechanisms, designed to manage contributions and avoid conflicts, offer valuable solutions for structuring collaborative decision-making. By adapting these techniques, we propose ways to ensure effective participation while keeping the process manageable. This paper demonstrates their application and the necessary modifications for decision-making contexts.

Additional Key Words and Phrases: group decision-making, social computing, online-participation, version control systems

1 Introduction

The widespread availability of Internet access has given rise to a variety of collaborative decision-making processes. Entire communities, cities and even countries are trying to involve their citizens in decisions that would affect them. What was previously only possible in small groups of experts is now, at least theoretically, possible on a large scale.

One aspect of collaborative decision-making processes is the ability of participants to contribute their own ideas and proposals. In previous collaborative decision-making processes, this was limited to participants posting their ideas and others commenting or voting on them. While there have been some ideas about making the generation of proposals itself collaborative[1], there has been no widely accepted approach to providing this functionality in collaborative decision-making systems. What is traditionally done by committees of a few experts now poses a major communication challenge when done by participants themselves. Proposals generated by untrained participants need to be vetted and structured for refinement, not by said experts, but by a crowd of co-participants.

Supporting collaborative proposal generation requires some rules and support for what participants can do. How can a large group of untrained participants systematically create options and develop them iteratively without the whole process descending into chaos? While it is desired that *all* participants have the opportunity to participate, they must not get in each other's way. If a participant wants to improve an option, it must not simply be overwritten. At the same time, many participants may have many ideas for improving a proposal, so how can these be coordinated? No version of a proposal should be lost, but at the same time there may be many similar proposals. Participants in an online process interact asynchronously with the system, effectively forming a distributed system in which it is not certain that all participants are on the same page at all times.

This work explores the usability of version control system (VCS) techniques that have been successfully used for collaborative software development. The approach of incremental improvement through iterative steps, combined with a structure that allows almost anyone to participate with minimal risk, seems a good fit, even if some improvements are needed.

1.1 Related Work

There has been work on collaborative writing tools using VCS, such as *Upwelling*[4] and *Patchwork*[3]. These tools combine real-time collaboration with version control, allowing multiple authors to work on the same document at the same time. While popular software such as Google Docs also

Authors' Contact Information: Björn Ebbinghaus, ebbinghaus@hhu.de; Martin Mauve, mauve@hhu.de, Heinrich Heine University Düsseldorf, Group Computer Networks, Computer Science Department, Faculty of Mathematics and Natural Sciences, Düsseldorf, Germany.

uses versions, which can be described as other VCS features, this is kept simple and is not the main focus of their design. Most closely related to this work is the proposal by Weidner[5], which outlines a design for versioned collaborative documents. The proposal combines Google Docs-style real-time collaboration with Git-style fork-merge collaboration. The author outlines the architecture of a local-first platform for versioned collaborative documents, outlines the programming model that programmers would use to support new document types on top of the platform, and discusses the tools needed to implement the proposal. It focuses more on the technical aspects of the local-first nature of the system, while this work explores the usability implications of such a system for use in large-scale collaborative decision-making processes.

2 Version Control Systems

Version control systems are widely used by software developers to manage and track changes to the files in their code base. It allows developers to collaborate on a project simultaneously while keeping track of every change made. The most widely used VCS for code today is git[1].

Version control systems such as git track changes to files in a project. Developers decide when it is time to save the latest changes to a set of files, usually when a coherent change to the code base has been completed. Once these changes have been recorded, developers share them with their collaborators by publishing the changes to a central platform.

Once a change has been recorded, it is uniquely identifiable and becomes a permanent part of the history, i.e. a *version*, of the project. The entire history of all files can be replayed or rewound by applying or removing these changes from the files one by one. This allows people to follow the evolution of the code, for example to see when a bug crept in, or perhaps to see in what context those changes were needed at the time. Both of these features are important not only for software development, but for many other collaborative projects.

VCSs allow multiple developers to work on the same files at the same time. These systems minimise conflicts by automatically merging changes when edits are made to separate parts of a file or to different files. This feature ensures a scalable workflow, allowing projects to involve large teams working simultaneously without constant manual coordination.

However, these mechanisms are not foolproof when changes overlap in the same part of a file. In such cases, conflicts arise, and automatic merging becomes impossible. To overcome this, these systems provide advanced conflict resolution tools.[2] Developers can review conflicting changes, understand their context, and manually reconcile differences to maintain a consistent project history.

These tools have evolved from working with known collaborators to tools that facilitate external collaboration, inviting people from all over the Internet to contribute their changes and move a project forward together. And while this depends on a few developers in privileged positions reviewing and accepting those changes, with the entire history of changes visible to everyone, it is trivial to make a private copy of a project and continue working on it, while getting updates from the main project when needed.

Using these features, VCSs like git have proven to be powerful tools for collaboration at scale, allowing thousands of developers to work together on developing complex software.[3]

[1] https://git-scm.com/
[2] https://git-scm.com/docs/merge-strategies
[3] https://github.com/torvalds/linux

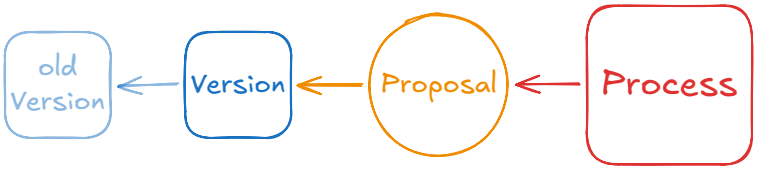


Fig. 1. Iconography. Note that the real relation from child to parent is shown here, while in later figures, the relations between versions are reversed, i.e. they display them chronologically.

3 Basic Mechanics

Collaborative decision-making processes require a structure that accommodates multiple stakeholders with different interests and perspectives. For example, in a public participation process, a local government may propose a new zoning code. The initial proposal outlines changes to land use in a district, which stakeholders—including residents, business owners and urban planners—review and discuss. As they make suggestions, such as adjusting building heights or adding green space, each idea is recorded as a new version, providing a clear history of revisions. If a group of residents proposes an alternative zoning plan, a new branch is created so that their input can be considered without changing the original proposal.

During this process, several versions may develop in parallel, each reflecting a different perspective. For example, while one branch may focus on adjustments to business districts, another branch may prioritise green spaces in residential areas. Later, if there is agreement, these branches can be merged into a single proposal that combines both perspectives.

3.1 Proposals and Versions

Proposals are mutable entities in the system and the main entry point for user participation. As they change, their history should be traceable. For this purpose, proposals consist of a chain of *versions* leading to the current *version*, called the proposal's *head*.

A *version* of a proposal contains its content as plain text or really any appropriate data type and a set of $0 - n$ parents. While proposals are mutable, versions are immutable, meaning that once created nothing about them can change. Neither the contents nor the references. This implies that these references can only be to existing versions, resulting in a causally directed acyclic graph.

Proposals always explicitly refer to exactly one version, and implicitly contain their history through the ancestors of that version. A new proposal does not have to be created with a new version; it can refer to an existing version. Several proposals can refer to the same version at the same time. To keep proposal development reasonably simple, proposals are only allowed to move to a version that is a descendant of the current version, i.e. a previous version can never be removed from a proposal's history.

The creation of a new proposal, while transparent to the user, creates a new version containing the content and a new proposal referencing that version. The proposal is then in the set of proposals for the process.

Later, if the user wants to change the proposal, a new version is created with the new content, referencing the previous version. The proposal is then updated to the new version. The resulting state is shown in Fig. 1.

3.2 Branching and Change Requests

The concepts introduced so far work for single users iterating sequentially on proposals. But the aim of the system is for multiple users to collaborate simultaneously.

Once one user has created a proposal, another user can come in and request a change to move it forward. To do this, they create a new version as before, but instead of updating the proposal, they create a new type similar to a proposal called a branch.

Branches and proposals are exactly the same, except that proposals are added to the process's set of proposals, and branches are not. Everything that applies to proposals also applies to branches. A *branch* can be elevated to a proposal at any time.

After the new version and branch have been created, the user can make a request to the original proposal to accept this change (see Fig. 2). If this request is accepted, the proposal will update its head to the head of the requesting branch. The requesting branch can then be deleted.

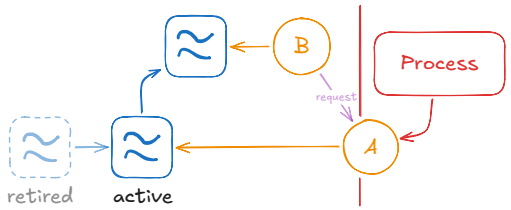


Fig. 2. Branch B requests a change from proposal A. If A accepts, it would update its reference to the head of B.

Although there are no predetermined roles in this system, someone has to be in control to drive the proposals forward. Control is somewhat meritocratic. The user who originally created a branch/proposal becomes its *maintainer*, and is responsible for the development of that branch/proposal. The maintainers are therefore the ones who open, accept or reject change requests to and from other branches.

If a change request is permanently rejected, the maintainer of the requesting branch may decide to elevate that branch to an independent proposal by adding it to the process. This means that there may be several proposals with a common history. If the situation allows, the two proposals may be merged in the future.

3.3 Blockage of Change Request / Divergence of Branches

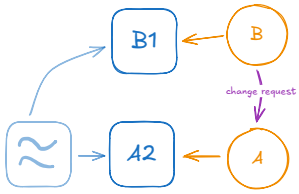
Consider the scenario in Fig. 3a. B created a change request for A. Subsequently, A progressed to version A_2. In that case, the history of the two branches A and B is no longer linear, but *divergent*. One has versions in its history that the other does not, and vice versa. This is a potential conflict, and it cannot be trivially merged with a click, it is *blocked*, because changing the head of the proposal to the head of the requesting branch would abandon the current head (Fig. 3b). Someone needs to decide how to add the missing versions to the respective branches.

Normally it is the responsibility of the requesting maintainer to resolve the situation, but technically both maintainers are able to merge the content of the two versions. There are ways of automatically merging texts[2], but either way the maintainer needs to check that the semantics of the new version matches the intent.

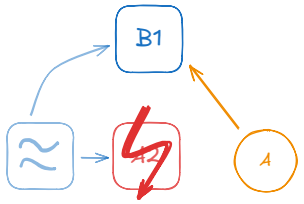
The requesting maintainer provides a trivially mergeable version again, by including the missing, divergent history in a new merged version (Fig. 3c). After this, the proposal's history is a strict subgraph of the change request, and the receiving maintainer simply has to accept it (Fig. 3d).

3.4 Discussion

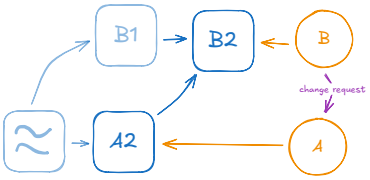
The mechanisms presented so far give rise to different ways of reflecting real decision-making processes. It allows all participants to participate equally, without fixed roles. The role of *maintainer*



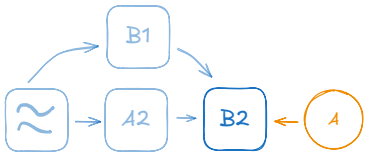
(a) B opened a change request for A.



(b) A can not accept the change request as is, because then A_2 would be lost.



(c) B creates a new merge version B_2, that includes A_2.



(d) A is now able to merge B by just advancing its head to B_2.

Fig. 3. Scenario in which a change request gets blocked and needs to be resolved before A can easily accept the change request.

is not assigned globally during the process, but is earned when the proposal is created and is then restricted to that role. Responsibility for this proposal is taken on in addition to control.

It is important to note that maintainership is not the same as ownership. Created versions of proposals are publicly available to all participants and can be reused by them at any time, even retaining their full history. If there is a problem with a maintainer, for example because they are no longer involved in the process or are unwilling to accept the desired changes, other participants can easily create a new proposal with the same version. This allows participants to moderate the process themselves. However, the underlying immutable data structure does not allow anyone in the process to perform destructive actions such as deleting versions.

However, the underlying complexity of the system has natural drawbacks. Forks in closely related proposals can be confusing for contributors. Duplication and possibly ambiguous navigation can quickly lead to contributor fatigue, especially if they are unfamiliar with the data schema used. This, combined with the nature of the system where content changes regularly, can lead to disorientation. A mechanism for prioritising proposals is therefore needed to keep the focus on the relevant proposals. More on this in the next section.

3.5 Voting

The system can facilitate the creation and iteration of proposals, as well as basic collaboration between users. As a quick and easy way to participate, users can express their opinions as votes, just like in other participation systems.

Primarily votes are cast on versions, as they are immutable. Each vote has the same weight, and it is possible to cast or withdraw votes at any time during the process. The value of a vote is not relevant to the general mechanics, the system supports multi-value votes, e.g. *approve* and *reject*.

An explicit vote cast by a user for a version remains with that version until it is withdrawn, even if the version is obsolete. A vote for the leader of a branch or proposal implies a vote for every branch or proposal that refers to it. Thus, a vote for a single version may imply a vote for several branches/proposals at once.

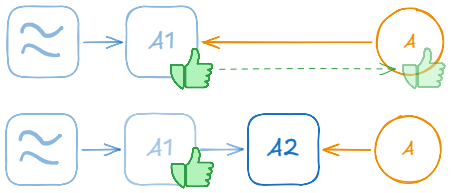


Fig. 4. A vote for A_1 implies a vote for A, as long as it is the head of the branch. If the branch advances to A_2, the implication is lost, and the user needs to agree to As new head again.

By associating votes with versions rather than proposals, the system allows votes to be cast on potential change requests. Users can vote on the version of a change request before it is accepted. Once a change request has been accepted, the version in question becomes the head of the proposal, and thus a vote for the proposal. This approach can also be thought of as a *conditional vote*. This concept is illustrated in Fig. 5.

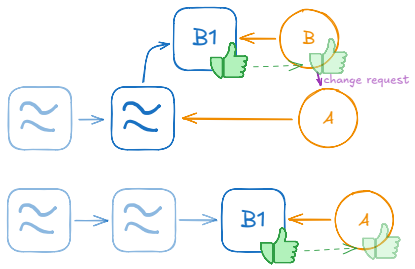


Fig. 5. Branch B has an open change request towards A with version B_1, which has a vote. Once A accepts the change request the vote automatically implies a vote to A.

3.5.1 Automatic Following of Votes. As well as explicitly voting for immutable versions, users can also allow their vote to follow a branch or proposal. This is a usability enhancement to voting. A branch that has a follow vote sets an explicit vote for the current head and any future heads until the vote for the branch is withdrawn. These votes are not just implicit, but explicit votes for versions as explained above. They can be changed individually, and versions that existed before the follow vote was enabled are not affected. Existing votes are not overwritten either.

This automatic following of votes is branch-specific, so it will not follow diverging branches of change requests. After a merge, the history of the requesting branch will not receive any votes, as they were never the head of the branch, although they are now in its history

3.5.2 Discussion. The ability to vote in the system is not primarily intended as a means of making a final decision, but as a means of enabling rapid communication. Without it, users would have no clear indication of how many other users are involved in a proposal, other than perhaps the number of change requests.

The ability to vote for a version in a change requesting branch can be important in several ways:

Firstly, for the maintainers of the receiving branch, as they can use it to prioritise their efforts towards more popular requests. They can also see how many votes they could immediately gain with a merge.

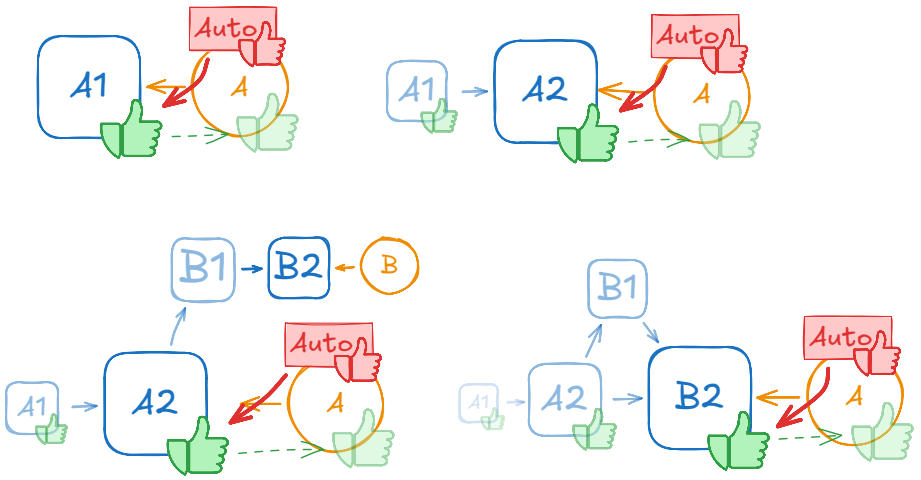


Fig. 6. Auto votes follow the head branch A, but not version B_1 of branch B. Once B is merged into A, does A point to B_2, and it gets the vote.

Second, for the maintainers of the requesting branch: they have leverage over the requested branch, and if the request is not accepted, and they add their branch as a new proposal, they keep the votes, giving them a head start and reducing the risk involved.

Third for the users themselves. They can shift their vote to something that *might* happen, reducing the time they have to be actively have to keep up with the development of the process.

As participants interact with the system and express their preferences, the system can provide feedback on the popularity of proposals, and the direction of the community. This can help to guide the development of the process, and to identify areas that need more attention. Users can be notified of changes in the status of proposals they are interested in, and can be encouraged to participate further in the process.

4 Comments & Issues

Participants can express themselves in three ways: proposals, change requests and votes. Proposals and change requests require more time and effort, while voting is quick, requiring only a single click to express support or opposition.

A common form of communication in the system is comments. Comments are short pieces of text attached to any entity, including other comments. This allows discussions to develop naturally into threads or trees.

A challenge arises when discussions occur in branches that evolve or are reused in forks. A comment made on a version of a branch may no longer be relevant to a fork, but it may still be relevant to an active change request based on that version. To solve this problem, comments in the system are contextualised by the version and branch they were made on. This creates an overlay of comments on top of the version graph (see Fig. 7), allowing users to track discussions across different versions and branches. Each comment is linked to a specific version within a branch, so it is possible to filter and group comments based on when and where they were added.

Filtering allows users to hide comments that are not relevant to their current branch or version, while grouping makes it easier to see comments from specific branches or drafts, depending on what the user wants to see. For example, in a fork that is branching away from the source, users

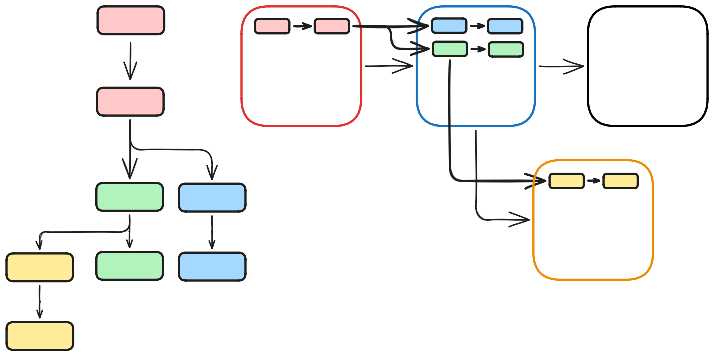


Fig. 7. A comment tree forms an overlaying graph of comments on top of the version graph.

could hide all comments that are not part of the fork, or that are not predecessors of a comment within the fork.

4.1 Issues

There is a more formal variation of the comment, the issue, which draws inspiration from the issue systems found in version control systems such as GitHub or GitLab, but with modifications to fit the context of participation system, such as a tighter relationship with a specific version and branch. Issues allow participants to raise concerns, suggest improvements, or flag areas that need attention.

Issues are like an actionable, top-level comment. In addition to their text content, they can have a status, such as open, closed, or resolved. Participants can use those if they have a specific issue, that they need to track or resolve, but can not or do not want to create a full change request for it, yet. For example, an issue might highlight an unclear section of a proposal, or suggest an area for further research.

Since issues are more restricted in their position, and have more formal semantics, they act as anchor points for further discussions, utilizing common comments. When a new branch is created, the maintainer can decide whether to carry over existing issues, or discard them if they are no longer relevant.

As work on the proposal progresses, issues can be resolved by new versions, containing changes that address the issue. Once an issue has been addressed, it can be marked as closed. Closed issues, along with their comment threads, can be hidden from the current branch view, keeping the focus on the active areas of the proposal, while being archived for future reference.

Issues fill the gap between lightweight comments and more heavyweight change requests. The spread of options for users to participate (see Fig. 8) aims at providing tools for collaboration for different situations.

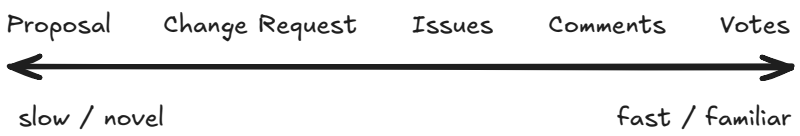


Fig. 8. Scale of ways for participants to interact.

5 Conclusion

We explored how version control systems (VCS) can support collaborative decision-making. Using techniques commonly used in software development, we explored how similar mechanisms could be adapted to support collaborative proposal generation.

By introducing concepts such as immutable versions, branching and structured change requests, the system provides a scalable approach to collaborative proposal development. It ensures that all participants' contributions are preserved, while maintaining a coherent and traceable history. By providing structure to the process, it gives participants tools and guidance on how to contribute effectively. This structure also helps to manage the complexity of large-scale collaboration, making it easier to navigate and understand each other's contributions.

A major challenge in using the features presented here for collaborative decision-making is to ensure that they are easy to use for everyone, not just tech-savvy users. Tools such as Git, while well known in software development, can be confusing to people who are not familiar with them. This complexity can be a barrier to participation for less experienced participants. Therefore, features have been designed so that novice participants do not have to rely on them to participate. A novice participant does not need to know about versions in order to browse, vote and comment on proposals.

The proposed system will now need to be tested in user trials. This will help to identify usability issues and assess the effectiveness of the system in real-world scenarios. The system should be tested in different scenarios, from small expert groups to large public participation, to see how well it scales and is used in different contexts.

References

[1] Jan Behrens, Axel Kistner, Andreas Nitsche, Björn Swierczek, and Interaktive Demokratie e.V (Eds.). 2014. *The Principles of LiquidFeedback* (1st edition ed.). Interaktive Demokratie e.V, Berlin.

[2] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter Van Hardenberg. 2022. Peritext: A CRDT for Collaborative Rich Text Editing. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW2 (2022), 1–36.

[3] Geoffrey Litt, Paul Sonnentag, Max Schöning, Adam Wiggins, Peter van Hardenberg, and Orion Henry. 2024. Patchwork lab notebook: Version control for everything. https://www.inkandswitch.com/patchwork/notebook/. [Online; accessed 26-June-2024].

[4] Karissa Rae McKelvey, Scott Jenson, Eileen Wagner, Blaine Cook, and Martin Kleppmann. 2023. https://www.inkandswitch.com/upwelling/

[5] Matthew Weidner. 2023. Proposal: Versioned Collaborative Documents. In *Programming Local-first Software Workshop*. https://mattweidner.com/assets/pdf/versioned_collaborative_documents.pdf