9 First Specifications and Design

KEYWORDS
VersionControl, MusicProduction

To identify the critical parts of the needed system before beginning its implementation, the first step is to design and implement smaller, more manageable components with a lower scope and complexity. Version control, together with collaboration, is the most critical aspect of the final design. Therefore, to better understand the system architecture, we initially design and specify two small prototypes. These will be used to acquire insights and inform the development of the overall system.

Due to its relatively lower complexity, the first prototype focuses on collaboration through project synchronization. This design proposes a simple peer-to-peer synchronization tool for projects. Following this, the version control component is designed to be limited to local projects to maintain a controlled scope. The design of both components provides the foundation for combining these methodologies, resulting in an initial prototype that can serve as an initial design to prototype from.

9.1 Specification I: File Sync

As outlined, the initial prototype focuses on file synchronization between systems. This idea occurred while I was temporarily sharing a studio workspace with a colleague of mine to have a place to share equipment and collaborate on some ideas. At that time, I had a partially damaged MacBook with a non-functional keyboard and display, which was repurposed into a stationary studio hub with all relevant equipment connected and left in the studio at all times.

This meant we didn't need to bring our personal computers into the studio for every project session, making things much more convenient. However, we still wanted to keep building on ideas at home and return to the studio to continue iterating. I found myself wishing for a workflow comparable to git push/pull to sync our work. That way, I could avoid relying on manual transfers via external storage media. I considered using established command-line tools such as ssh and rsync. SSH (Secure Shell) provides a secure way to connect to another computer over a network and access its files remotely. RSYNC, by contrast, is a synchronization utility that compares files across two locations and transfers only the differences. These tools are suitable for transferring and updating files between systems. However, they do not, by themselves, provide a clear strategy for handling conflicting versions, and they also require access to the other machine's user account.

I knew I wanted to avoid version control for now, but I still needed a solution. While researching the current state of version control in software development, I encountered the following approach. In a podcast by Primeason et al. (2025), Casey Muratori described his custom version control system for his game development studio. Instead of only allowing a merged state, his system collects all changes and saves the resulting project state to a central repository. If there are conflicting changes, the system collects all variants and lets the developer resolve the conflict manually later. Until resolution, multiple versions coexist in parallel, described as a temporary "superposition" of file states.

This approach appears suitable for project synchronization. Let's try to define a protocol. Imagine two users working on the same project, each keeping a local copy. User A adds a new session. User B deletes a session. Both users modify an existing session. When user A compares the local state with user B's remote state, the result shows that user B lacks two sessions present in user A's version, and that one session is modified. In turn, when user B compares the local state with user A's remote state, the result shows that user A contains two sessions missing from user B's version. It also shows that one session is modified. First, the

synchronization protocol will be constrained so that the local state can be modified, but the remote will serve only as a source of truth. No direct modifications are made to the remote system. If a session is missing locally, it can be added. If a session exists locally but not in the remote state, it can be removed. In cases of conflicting versions, resolution will use the superposition principle described previously.

To identify changes and differences between two iterations of a project, the relative path and the hash value of the sessions are utilized. The path encodes the location and name of a session within the project folder, while the hash, being a deterministic value generated from the session's data, enables detecting changes or identifying the same content across sessions. Formally, this relationship can be defined as follows.

9.1.1 System Model

Let $\mathbf{Contents}$ denote the set of all valid session contents and $\mathbf{RelativePaths}$ denote the set of all valid relative paths of session files within the project folder. The set of all sessions is defined as

$$\mathbf{Sessions} \triangleq [\text{relativePath} : \mathbf{RelativePaths}, \text{content} : \mathbf{Contents}].$$

For a session $s \in \mathbf{Sessions}$, let $p(s)$ denote its relative path and let $c(s)$ denote its content. Thus, $p(s) = s.\,\mathbf{relativePath}$ and $c(s) = s.\,\mathbf{content}$.

Furthermore, let $\mathbf{Hashes}$ denote the set of all valid hash values, and let $h : \mathbf{Contents} \to \mathbf{Hashes}$ be the hash function that assigns a hash value to the content of a session. The hash value of a session s is then defined as $h(s) \triangleq h(c(s))$.

Under the assumption of an ideal collision-resistant hash function, equal session contents yield equal hash values, and equal hash values imply equal session contents. Formally, for all $s_1, s_2 \in \mathbf{Sessions}$,

$$c(s_1) = c(s_2) \Rightarrow h(s_1) = h(s_2) \land h(s_1) = h(s_2) \Rightarrow c(s_1) = c(s_2).$$

To simplify the model, the content of a session is not represented explicitly. Instead, the model represents session content by its hash value. This abstraction assumes that equal hash values correspond to equal session contents. Consequently, assigning a hash value to a session models the assignment of the corresponding content. Under this abstraction, the set of sessions is redefined as

$$\mathbf{Sessions} \triangleq [\text{relativePath} : \mathbf{RelativePaths}, \text{content} : \mathbf{Hashes}].$$

Let $\mathbf{Systems}$ denote the set of all participating systems. Each system maintains a set of sessions. This relationship is captured by a function

$$\text{sessionsBySystem} : \mathbf{Systems} \to \mathcal{P}(\mathbf{Sessions}),$$

where $\mathcal{P}(\mathbf{Sessions})$ denotes the power set of $\mathbf{Sessions}$.

Within each system, session paths must be unique. This constraint is formalized by the predicate

$$\text{UniquePaths}(S) \triangleq \forall s_1, s_2 \in S :; s_1 \neq s_2 \Rightarrow p(s_1) \neq p(s_2).$$

Furthermore, the existence of a session with a given path in a system is defined by

$$\text{PathExists}(system, p) \triangleq \exists s \in \text{sessionsBySystem}(system) : p(s) = p.$$

The well-formedness condition of the system state is then given by

$$
\begin{aligned}
&\textbf{TypeOK} \triangleq \\
&\quad \land\ \text{sessionsBySystem} \in [\textbf{Systems} \to \mathcal{P}(\textbf{Sessions})] \\
&\quad \land\ \forall \mathit{system} \in \textbf{Systems} :\ \text{UniquePaths}(\text{sessionsBySystem}(\mathit{system}))
\end{aligned}
$$

and the initial state of the system is defined, such that every system starts with an empty set of sessions:

$$
\textbf{Init} \triangleq \forall \text{system} \in \textbf{Systems} :; \text{sessionsBySystem}(\mathit{system}) = \emptyset.
$$

9.1.2 User Actions

User actions are modeled as state transitions on the function **sessionsBySystem**. Each action affects only one system, while the session sets of all other systems remain unchanged.

A session may be added to a system only if no session with the same relative path already exists in that system. The corresponding operation is defined as follows:

$$
\begin{aligned}
&\text{AddSession}(\mathit{system}, \mathit{session}) \triangleq \\
&\quad \land\ \mathit{system} \in \textbf{Systems} \\
&\quad \land\ \mathit{session} \in \textbf{Sessions} \\
&\quad \land\ \neg\text{PathExists}(\mathit{system}, p(\mathit{session})) \\
&\quad \land\ \forall s \in \textbf{Systems} :\ \text{sessionsBySystem}'(s) = \\
&\qquad \begin{cases}
\text{sessionsBySystem}(s) \cup \{\mathit{session}\}, & \text{if } s = \mathit{system}, \\
\text{sessionsBySystem}(s), & \text{otherwise.}
\end{cases}
\end{aligned}
$$

A session may be deleted by removing it from the corresponding system. This operation is defined as follows:

$$
\begin{aligned}
&\text{DeleteSession}(\mathit{system}, \mathit{session}) \triangleq \\
&\quad \land\ \mathit{system} \in \textbf{Systems} \\
&\quad \land\ \mathit{session} \in \text{sessionsBySystem}(\mathit{system}) \\
&\quad \land\ \forall s \in \textbf{Systems} :\ \text{sessionsBySystem}'(s) = \\
&\qquad \begin{cases}
\text{sessionsBySystem}(s) \setminus \{\mathit{session}\}, & \text{if } s = \mathit{system}, \\
\text{sessionsBySystem}(s), & \text{otherwise.}
\end{cases}
\end{aligned}
$$

A session may be edited by modifying its content, which produces a new hash value while preserving its relative path. Formally, the edit operation is defined as follows:

$$
\begin{aligned}
&\text{EditSession}(\mathit{system}, \mathit{session}, \mathit{newHash}) \triangleq \\
&\quad \land\ \mathit{system} \in \textbf{Systems} \\
&\quad \land\ \mathit{session} \in \text{sessionsBySystem}(\mathit{system}) \\
&\quad \land\ \mathit{newHash} \in \textbf{Hashes} \\
&\quad \land\ \forall s \in \textbf{Systems} :\ \text{sessionsBySystem}'(s) = \\
&\qquad \begin{cases}
\big(\text{sessionsBySystem}(s) \setminus \{\mathit{session}\}\big) & \\
\quad \cup \{\, [\text{relativePath} : p(\mathit{session}), \text{hash} : \mathit{newHash}] \,\} & \text{if } s = \mathit{system}, \\
\text{sessionsBySystem}(s) & \text{otherwise.}
\end{cases}
\end{aligned}
$$

A session may be moved by changing its relative path while preserving its content. This operation replaces the session with a new session that has the same hash value but a different relative path. The target path must not already exist in the system. Formally, the move operation is defined as follows:

$$
\begin{aligned}
&\textbf{MoveSession}(\mathit{system}, \mathit{session}, \mathit{newPath}) \triangleq \\
&\quad \land\ \mathit{system} \in \textbf{Systems} \\
&\quad \land\ \mathit{session} \in \textbf{sessionsBySystem}(\mathit{system}) \\
&\quad \land\ \mathit{newPath} \in \textbf{RelativePaths} \\
&\quad \land\ \neg\textbf{PathExists}(\mathit{system}, \mathit{newPath}) \\
&\quad \land\ \forall s \in \textbf{Systems} :\ \textbf{sessionsBySystem}'(s) = \\
&\qquad \begin{cases}
\big(\textbf{sessionsBySystem}(s) \setminus \{\mathit{session}\}\big) & \\
\quad \cup\ \{\, [\textbf{relativePath} : \mathit{newPath},\ \textbf{hash} : h(\mathit{session})] \,\} & \text{if } s = \mathit{system}, \\
\textbf{sessionsBySystem}(s) & \text{otherwise.}
\end{cases}
\end{aligned}
$$

The predicate **SessionOps** summarizes all possible user induced state transitions on sessions. It is defined as the disjunction of the primitive session operations.

$$
\begin{aligned}
&\textbf{SessionOps} \triangleq \\
&\quad \exists\, \mathit{system} \in \textbf{Systems} : \\
&\qquad \lor\ \exists\, \mathit{session} \in \textbf{Sessions} :\ \textbf{AddSession}(\mathit{system}, \mathit{session}) \\
&\qquad \lor\ \exists\, \mathit{session} \in \textbf{sessionsBySystem}(\mathit{system}) : \\
&\qquad\quad \lor\ \exists\, \mathit{newHash} \in \textbf{Hashes} :\ \textbf{EditSession}(\mathit{system}, \mathit{session}, \mathit{newHash}) \\
&\qquad\quad \lor\ \exists\, \mathit{newPath} \in \textbf{RelativePaths} :\ \textbf{MoveSession}(\mathit{system}, \mathit{session}, \mathit{newPath}) \\
&\qquad\quad \lor\ \textbf{DeleteSession}(\mathit{system}, \mathit{session})
\end{aligned}
$$

9.1.3 Sync Actions

To compare and synchronize two projects, the difference between sessions must first be determined. The difference between two sessions is determined by comparing their relative paths and hash values. Let $s_1, s_2 \in \textbf{Sessions}$, four distinct cases are possible.

Two sessions are considered identical if both their relative paths and their hash values are equal. This case is defined as

$$\textbf{Equal}(s_1, s_2) \triangleq p(s_1) = p(s_2) \land h(s_1) = h(s_2).$$

Two sessions are considered to be different versions of the same session if their relative paths are equal but their hash values differ. This case is defined as

$$\textbf{DifferentVersion}(s_1, s_2) \triangleq p(s_1) = p(s_2) \land h(s_1) \neq h(s_2).$$

Two sessions are considered to represent the same content under different locations if their hash values are equal but their relative paths differ. This case is defined as

$$\textbf{DifferentLocation}(s_1, s_2) \triangleq h(s_1) = h(s_2) \land p(s_1) \neq p(s_2).$$

Finally, two sessions are considered unrelated if both their relative paths and their hash values differ. This case is defined as

$$\textbf{DifferentSession}(s_1, s_2) \triangleq p(s_1) \neq p(s_2) \land h(s_1) \neq h(s_2).$$

Based on the type of difference between sessions, different actions can be taken to synchronize two projects. These four cases can be summarized in the following truth table,

$p(s_1) = p(s_2)$	$h(s_1) = h(s_2)$	DiffType	Action
True	**True**	Equal	None
True	**False**	Different Version	Superposition
False	**True**	Different Location	Move

False	**False**	No Correlation	Copy / Deletion

In several cases, more than one action is possible. This is because the selected action does not depend solely on the relationship between two individual sessions. Rather, it depends on their relationship to all other sessions in the two systems and on which system performs the comparison. As stated above, only the data on the local system are modified, whereas the remote system remains unchanged. Whether a specific operation is applicable can be determined by the following predicates.

A session can be deleted if it exists on the local system but has no corresponding session on the remote system with either the same relative path or the same hash value. In this case, deleting the local session contributes to making the local project state consistent with the remote state.

$$\begin{aligned}&\textbf{CanDelete}(\mathit{local}, \mathit{remote}, s) \triangleq \\ &\quad \land\ s \in \textbf{sessionsBySystem}(\mathit{local}) \\ &\quad \land\ \forall r \in \textbf{sessionsBySystem}(\mathit{remote}) : \textbf{DifferentSession}(s, r)\end{aligned}$$

A session can be copied from the remote system to the local system if no session on the local system has either the same relative path or the same hash value. In this case, the remote session represents content that is absent from the local project.

$$\begin{aligned}&\textbf{CanCopy}(\mathit{local}, \mathit{remote}, r) \triangleq \\ &\quad \land\ r \in \textbf{sessionsBySystem}(\mathit{remote}) \\ &\quad \land\ \forall s \in \textbf{sessionsBySystem}(\mathit{local}) : \textbf{DifferentSession}(s, r)\end{aligned}$$

A session can be move if a local session and a remote session have the same hash value but different relative paths. This condition is captured by the following predicate:

$$\begin{aligned}&\textbf{CanMove}(\mathit{local}, \mathit{remote}, s, r) \triangleq \\ &\quad \land\ s \in \textbf{sessionsBySystem}(\mathit{local}) \\ &\quad \land\ r \in \textbf{sessionsBySystem}(\mathit{remote}) \\ &\quad \land\ \textbf{DifferentLocation}(s, r) \\ &\quad \land\ \neg\textbf{PathExists}(\mathit{local}, p(r))\end{aligned}$$

A conflict may occur when a local session and a remote session share the same relative path but differ in their hash values, this condition is defined as follows:

$$\begin{aligned}&\textbf{CanConflict}(\mathit{local}, \mathit{remote}, s, r) \triangleq \\ &\quad \land\ s \in \textbf{sessionsBySystem}(\mathit{local}) \\ &\quad \land\ r \in \textbf{sessionsBySystem}(\mathit{remote}) \\ &\quad \land\ \textbf{DifferentVersion}(s, r) \\ &\quad \land\ \neg\textbf{PathExists}(\mathit{local}, \textbf{LocalRenamePath}(p(s))) \\ &\quad \land\ \neg\textbf{PathExists}(\mathit{local}, \textbf{RemoteRenamePath}(p(r)))\end{aligned}$$

To resolve conflicts, relative paths are modified by appending a version specific suffix before the file extension. Let **LocalRenamePath** and **RemoteRenamePath** denote functions that transform a relative path by inserting a suffix before the `.als` extension.

Formally, for a path $p \in \textbf{RelativePaths}$, the renamed paths are defined as

$$\textbf{LocalRenamePath}(p) \triangleq \textbf{InsertSuffix}(p, \text{"vl"})$$

and

$$\textbf{RemoteRenamePath}(p) \triangleq \textbf{InsertSuffix}(p, \text{"vr"}).$$

The function **InsertSuffix** inserts the given suffix immediately before the `.als` extension of the path. For example,

$$\text{InsertSuffix}(\text{"track.als"}, \text{"vl"}) = \text{"track.vl.als"}.$$

This preserves the base name and file extension while ensuring that the resulting paths are distinct. The suffix **"vl"** denotes the local version, whereas the suffix **"vr"** denotes the remote version.

Using these predicates, the actually synchronization actions can be defined as state transitions on the local system, where the delete synchronization step removes a local session that satisfies the deletion condition:

$$\begin{aligned}&\text{DeleteSync}(local, remote) \triangleq \\ &\quad \exists s \in \text{sessionsBySystem}(local) : \\ &\qquad \land\ \text{CanDelete}(local, remote, s) \\ &\qquad \land\ \text{DeleteSession}(local, s)\end{aligned}$$

While a copy synchronization step adds a session from the remote system to the local system:

$$\begin{aligned}&\text{CopySync}(local, remote) \triangleq \\ &\quad \exists r \in \text{sessionsBySystem}(remote) : \\ &\qquad \land\ \text{CanCopy}(local, remote, r) \\ &\qquad \land\ \text{AddSession}(local, r)\end{aligned}$$

A move synchronization step aligns the location of a local session with that of a corresponding remote session:

$$\begin{aligned}&\text{MoveSync}(local, remote) \triangleq \\ &\quad \land\ \exists\, s \in \text{sessionsBySystem}(local),\ r \in \text{sessionsBySystem}(remote) : \\ &\qquad \land\ \text{CanMove}(local, remote, s, r) \\ &\qquad \land\ \text{MoveSession}(local, s, p(r))\end{aligned}$$

A conflict synchronization step preserves both conflicting versions locally by assigning distinct conflict resolution paths to the local and remote sessions. This action is applicable if a local and a remote session have the same relative path but different hash values.

$$\begin{aligned}&\text{ConflictSync}(local, remote) \triangleq \\ &\quad \land\ \exists\, s \in \text{sessionsBySystem}(local),\ r \in \text{sessionsBySystem}(remote) : \\ &\qquad \land\ \text{CanConflict}(local, remote, s, r) \\ &\qquad \land\ \text{MoveSession}(local, s, \text{LocalRenamePath}(p(s))) \\ &\qquad \land\ \text{AddSession}(local,\ [\text{relativePath} : \text{RemoteRenamePath}(p(r)),\ \text{hash} : h(r)])\end{aligned}$$

The predicate **SyncOps** summarizes all possible synchronization operations between two systems. It is defined as follows:

$$\begin{aligned}&\text{SyncOps} \triangleq \\ &\quad \exists\, l, r \in \text{Systems} : \\ &\qquad l \neq r \land (\text{DeleteSync}(l, r) \lor \text{CopySync}(l, r) \lor \text{ConflictSync}(l, r) \lor \text{RenameSync}(l, r))\end{aligned}$$

9.1.4 System Specification

The system specification

$$\text{Spec} \triangleq \text{Init} \land \square[\text{Next}]_{\text{sessionsBySystem}},$$

designated as Spec, defines SessionOps (the set of valid session-related operations) and SyncOps (the set of valid synchronization operations) as the valid steps the systems can take through the predicate $\text{Next} \triangleq \text{SessionOps} \lor \text{SyncOps}$.

While **Init** is the initial state, and $\text{Next}_{\text{sessionsBySystem}}$ assures that each state transition follows the defined valid operations. Additionally, so-called stuttering steps are allowed, enabling the

system to remain unchanged. This is captured by $\mathit{StutteringStep} = \mathit{Next} \lor \mathit{True}$, where True represents no change, and the system can either progress or stutter in its current state.

9.1.5 Refinements

The current definition of the synchronization protocol presents certain inherent limitations. Specifically, in some instances, it is not possible to determine the appropriate synchronization operation, resulting in ambiguity. Consider the following scenario involving the state of a local and a remote system:

$$\begin{aligned}&\textbf{local} : (\textbf{session1.als}, h1)\\ &\textbf{remote} : (\textbf{session2.als}, h1),\ (\textbf{session1.als}, h2)\end{aligned}$$

In this case, two potential synchronization operations can be performed. Since $(\textbf{session1.als}, h1)$ on the local system shares the same hash but differs in its path compared to $(\textbf{session2.als}, h1)$ on the remote system, a `MoveSync` operation becomes feasible. However, because $(\textbf{session1.als}, h1)$ on the local system has the same path as $(\textbf{session1.als}, h2)$ on the remote system, a `ConflictSync` operation is also possible.

If the `MoveSync` operation is performed on $(\textbf{session1.als}, h1)$ on the local system, the correlation to $(\textbf{session1.als}, h2)$ on the remote system is removed, allowing $(\textbf{session1.als}, h2)$ to be copied to the local system. This results in the state:

$$\textbf{local} : (\textbf{session2.als}, h1), (\textbf{session1.als}, h2)$$

Alternatively, performing the `ConflictSync` operation first would update the local state to:

$$\textbf{local} : (\textbf{session1.vl.als}, h1), (\textbf{session1.vr.als}, h2)$$

Subsequently, both sessions on the local system can be moved to align with the remote state, resulting in:

$$\textbf{local} : (\textbf{session1.als}, h1), (\textbf{session2.als}, h2)$$

Although both approaches ultimately lead to the same final state, they introduce ambiguity into the synchronization process. It would be preferable to restrict the synchronization process to a single, predefined path. One way to achieve this is by extending the `CanConflict` predicate to exclude cases where the `MoveSync` operation would be possible.

$$\begin{aligned}&\textbf{CanConflict}(\mathit{local}, \mathit{remote}, s, r) \triangleq\\ &\quad \land\ s \in \textbf{sessionsBySystem}(\mathit{local})\\ &\quad \land\ r \in \textbf{sessionsBySystem}(\mathit{remote})\\ &\quad \land\ \textbf{DifferentVersion}(s, r)\\ &\quad \land\ \neg\exists\ \textbf{r2} \in \textbf{sessionsBySystem}(\mathit{remote}) : \textbf{DifferentLocation}(s, \textbf{r2})\end{aligned}$$

9.1.6 Implementation outline

9.2 Specification II: Creative Version Control