Specification of team software project

Department of Software Engineering

Faculty of Mathematics and Physics, Charles University

Solvers:	Bc. Ondřej Krsička
Study program:	Computer Science - Software and Data Engineering
Project title:	Using CRDTs to enable collaborative editing in Denicek
Project type:	Research project
Supervisor:	Mgr. Tomáš Petříček, Ph.D.
Expected start:	1.9.2025
Expected end:	31.3.2026

1 Introduction

MyDenicek is a CRDT-based local-first document editing substrate that enables collaborative editing of tree-structured documents with automatic conflict resolution. The project reimplements the synchronization layer of the original *Denicek* system [1], replacing Operational Transformation (OT) with Conflict-free Replicated Data Types (CRDTs) to satisfy the requirements of local-first software [2].

The original *Denicek* system provides end-user programming experiences including collaborative editing, programming by demonstration (recording and replaying user actions), incremental recomputation, and schema change control. However, its OT-based synchronization is complex, error-prone, and requires a central server. This project creates an alternative built on CRDTs that enables true peer-to-peer collaboration without a central authority.

MyWebnicek is a web-based programming system built on top of *MyDenicek*, providing a user interface for document navigation, editing, and collaborative features. It demonstrates the end-user programming capabilities through formative examples such as counter apps, todo lists, and document refactoring scenarios.

The project is inspired by *MyWebstrates* [3], a local-first CRDT-based alternative to Webstrates. Both *MyDenicek* (the core library) and *MyWebnicek* (the web application) are implemented in TypeScript.

1.1 Relation to Denicek

This project directly extends the research presented at *ACM UIST 2025* [1]. While the original Denicek uses Operational Transformation for synchronization, MyDenicek replaces this with a CRDT-based approach using Loro [4]. Loro provides a native movable tree CRDT (LoroTree) that handles concurrent structural edits, node moves, and reparenting automatically. The key architectural difference is that nodes are indexed by unique IDs rather than paths, avoiding the "shifting index" problem that occurs during concurrent structural edits.

The project maintains a clear separation between the CRDT substrate (`@mydenicek/core`) and the user interface (`mywebnicek`), enabling future development of alternative front-ends similar to how the original Denicek supports both Webnicek and Datnicek.

2 Project description

The project consists of two main components: the core CRDT library (*MyDenicek*) and the web application (*MyWebnicek*). The core library provides a TypeScript API for document manipulation with automatic conflict resolution, while the web application provides a user interface for end-user programming experiences.

2.1 User journey

The user journey through MyWebnicek consists of the following phases:

1. **Document Creation/Loading:** Users can create a new document or load an existing one. Documents can be shared via URL for collaborative editing.
2. **Document Navigation:** Users navigate the document tree using keyboard shortcuts or mouse clicks. The selected node is highlighted in both the rendered view and the JSON inspector.
3. **Document Editing:** Users perform edit operations through toolbar controls. Operations include adding nodes, editing values, wrapping nodes, and deleting nodes.
4. **Recording/Replay:** Users can record a sequence of edit operations and replay them on different parts of the document, enabling programming by demonstration.
5. **Collaboration:** Multiple users can edit the same document simultaneously. Changes are synchronized via WebSocket and merged automatically using CRDTs.
6. **Undo/Redo:** Users can undo and redo operations. The undo history is local to each user.

2.2 Functional requirements

The following functional requirements specify the capabilities of the system. Requirements labeled [Core] relate to the `@mydenicek/core` library, while [Web] requirements relate to the `mywebnicek` application.

2.2.1 Core Library Requirements

FR-01: Document Representation [Core] The system shall represent documents as trees of nodes, where each node has a unique ID, a kind (element or value), and additional properties based on its kind.

FR-02: Element Nodes [Core] Element nodes shall have a tag name, a dictionary of attributes, and an ordered list of child node IDs.

FR-03: Value Nodes [Core] Value nodes shall contain a text string that can be modified through splice operations.

FR-04: Node Operations [Core] The system shall support the following operations on nodes:

- `addChild`: Add a new child node (element, value, formula, ref, or action) to a parent
- `addSibling`: Add a new sibling node before or after an existing node
- `deleteNode`: Remove a node and its descendants from the document
- `moveNode`: Move a node to a different parent or position
- `copyNode`: Create a copy of a node under a target parent
- `updateTag`: Change the tag name of an element node
- `updateAttribute`: Set or modify an attribute on an element node
- `spliceValue`: Insert, delete, or replace text within a value node
- `wrapNode`: Wrap a node in a new parent element

FR-05: Conflict Resolution [Core] The system shall automatically resolve conflicts when concurrent operations are applied:

- Concurrent wrap operations on the same node shall result in a single wrapper (using deterministic wrapper IDs)
- Concurrent attribute updates shall use Last-Writer-Wins (LWW) semantics
- Concurrent child additions shall preserve both children
- Concurrent tag updates shall use LWW semantics

FR-06: Undo/Redo [Core] The system shall maintain an undo stack and redo stack for each local session. Undo shall reverse the most recent local operation, and redo shall reapply the most recently undone operation.

FR-07: Recording [Core] The system shall support recording a sequence of operations starting from a specified node. Recorded operations shall use generalized node IDs (`$0`, `$1`, etc.) that can be bound to different nodes during replay.

FR-08: Replay [Core] The system shall support replaying a recorded script starting from a specified node. The generalized node IDs in the script shall be bound to actual node IDs based on the starting node.

FR-09: Transformations [Core] The system shall support defining transformations on element nodes that are automatically applied to new children. This enables pattern-based document manipulation.

FR-10: Formula Nodes [Core] Formula nodes compute values from an `operation` and child arguments (value nodes, ref nodes, or nested formulas). Evaluation detects circular references and enforces max recursion depth.

FR-11: Formula Operations [Core] Built-in operations: string (`concat`, `uppercase`, etc.), math (`plus`, `multiply`, etc.), array (`atIndex`, `splitString`), and tree (`countChildren`).

FR-12: Ref Nodes [Core] Ref nodes reference other nodes via a `target` ID. In formulas, they resolve to the target's value.

FR-13: Action Nodes [Core] Action nodes store recorded operations as generalized patches. They have a `label`, `target` (bound to `$0`), and `actions` list. Triggering replays patches on the target.

2.2.2 Web Application Requirements

FR-14: Document Rendering [Web] The application shall render the document tree as HTML, allowing users to see the final document appearance.

FR-15: Node Selection [Web] Users shall be able to select nodes by clicking on them in the rendered view or navigating with keyboard shortcuts (arrow keys, Tab).

FR-16: JSON Inspector [Web] The application shall display a JSON view of the document structure, highlighting the currently selected node.

FR-17: Element Details Panel [Web] When an element node is selected, the application shall display its tag, attributes, and available operations.

FR-18: Add Node Interface [Web] Users shall be able to add child nodes through a popover interface that allows specifying the tag name or text content.

FR-19: Recording Controls [Web] The application shall provide controls to start and stop recording, and display the recorded script.

FR-20: Replay Controls [Web] The application shall provide controls to replay a recorded script on the currently selected node.

FR-21: Keyboard Shortcuts [Web] The application shall support keyboard shortcuts for common operations:

- Arrow keys: Navigate between nodes
- Delete/Backspace: Delete selected node
- Ctrl+Z: Undo
- Ctrl+Y / Ctrl+Shift+Z: Redo

FR-22: WebSocket Synchronization [Web] The application shall synchronize document changes with other clients via WebSocket connection to a sync server.

FR-23: Formula Rendering [Web] Formula nodes display computed results or formula structure (for debugging).

FR-24: Action Node Rendering [Web] Action nodes render as buttons; clicking triggers replay on the target.

2.3 Non-functional requirements

The following non-functional requirements specify quality attributes and system constraints.

NFR-01: Convergence All replicas of a document shall eventually converge to the same state after receiving all operations, regardless of the order in which operations are received.

NFR-02: Offline Support The system shall support offline editing. Local changes shall be preserved and synchronized when connectivity is restored.

NFR-03: Response Time User interface operations (node selection, editing) shall complete within 100ms to provide responsive feedback.

NFR-04: Browser Compatibility The web application shall function correctly on current versions of Chrome, Firefox, Safari, and Edge.

NFR-05: Type Safety The codebase shall use TypeScript with strict type checking. The use of `any` type is prohibited; `unknown` or specific types shall be used instead.

NFR-06: Code Quality The codebase shall pass ESLint checks with no errors. Unused imports shall be removed automatically, and imports shall be sorted consistently.

NFR-07: Test Coverage Core library functionality shall be covered by unit tests (Vitest). User interface functionality shall be covered by end-to-end tests (Playwright).

NFR-08: Documentation Public APIs shall be documented with JSDoc comments. The README shall contain setup instructions and usage examples.

NFR-09: Reproducibility The development environment shall be reproducible using npm. All dependencies shall be specified in package.json with locked versions.

2.4 Architecture

The system follows a monorepo structure with clear separation between packages:

```
apps/
  mywebnicek/                # React 19 + Fluent UI web application
  mydenicek-sync-server/     # WebSocket sync server
packages/
  mydenicek-core/            # Core CRDT logic (Loro wrapper)
  mydenicek-react/           # React hooks and context
```

2.4.1 Core Library Architecture

The core library (`@mydenicek/core`) provides two main classes:

DenicekDocument is the entry point. It wraps a Loro document and provides:

- `change(fn)`: Execute mutations via a DenicekModel callback
- `undo()` / `redo()`: Undo/redo operations
- `replay(script, startNodeId)`: Replay a recorded script
- `connectToSync()` / `disconnectSync()`: WebSocket synchronization
- `export()` / `import()`: Serialization for sync

DenicekModel provides read/write operations within `change()` callbacks:

- Read: `getNode()`, `getParentId()`, `getChildren()`
- Write: `updateAttribute()`, `updateTag()`, `wrapNode()`, `deleteNode()`, `moveNode()`, `spliceValue()`, `addChild()`, `addSibling()`, `copyNode()`

2.4.2 Data Flow

1. User performs an action in the UI (e.g., clicks “Add Child”)
2. React component calls `document.change(model => { ... })`
3. DenicekModel applies the change to the Loro document
4. Loro generates patches; undo manager captures inverse

5. If subscribed, patches are recorded for replay
6. Loro synchronizes changes to connected peers via WebSocket
7. Remote peers receive patches and update their local document
8. React re-renders based on the updated document state

2.5 Formula Engine

The formula engine enables spreadsheet-like reactive computation. Nodes derive values from other nodes through operations.

2.5.1 Node Types

Formula Nodes compute values. They have an `operation` attribute (e.g., “plus”) and children serving as arguments.

Ref Nodes reference other nodes via a `target` attribute. In formulas, they resolve to the target’s value.

Value Nodes provide literal arguments.

2.5.2 Evaluation

The evaluator traverses the formula tree recursively:

1. Check recursion depth (max 100) and circular references
2. Evaluate each child: value nodes yield literals, ref nodes resolve targets, formula nodes recurse
3. Look up operation, validate arity, invoke with arguments
4. Return result or error (`#ERR:` prefix)

2.5.3 Built-in Operations

String: `lowerText`, `upperText`, `capitalize`, `concat` (variadic), `trim`, `length`, `replace`

Math: `plus`, `minus`, `multiply`, `divide`, `mod`, `round`, `floor`, `ceil`, `abs`

Array: `atIndex`, `splitString`, `arrayLength`

Tree: `countChildren`

2.5.4 Errors

- `#ERR: max depth exceeded`
- `#ERR: circular reference`
- `#ERR: node deleted`
- `#ERR: {op} not found`
- `#ERR: {op} expects {X} args, got {Y}`
- `#ERR: division by zero`

2.5.5 Example

Nested formulas compute a formatted name. Given value nodes `"john"` and `"doe"`, the formula `concat(capitalize(ref->john)` `" ", capitalize(ref->doe))` evaluates to `"John Doe"`.

2.6 Programmable Buttons (Action Nodes)

Action nodes enable programming by demonstration: users record operations and save them to a button for replay.

2.6.1 Structure

An action node stores: `label` (display text), `target` (node bound to `$0`), and `actions` (list of `GeneralizedPatch` objects).

2.6.2 Generalized Patches

Patches use variable placeholders instead of concrete node IDs:

```
GeneralizedPatch {
    action: "put" | "del" | "insert" | "splice" | "move" | "copy"
    path: (string | number)[]   // Contains $0, $1, etc.
    value?: unknown
    length?: number
}
```

Variables: `$0` is the target node; `$1`, `$2`, ... are nodes created during replay.

2.6.3 Recording

1. Select target node, click "Start Recording"
2. Perform operations (add, edit, delete nodes)
3. Recorder captures patches, generalizing node IDs to variables
4. Click "Stop Recording"; patches are stored in an action node

2.6.4 Replay

1. Click button or use replay controls
2. System maps `$0` to target node
3. For each patch: substitute variables, apply patch, assign new variables to created nodes

2.6.5 Rendering

Action nodes render as buttons. Click triggers replay; Ctrl+click selects for editing.

2.7 Formative Examples

These examples from the original Denicek paper demonstrate the end-user programming model.

2.7.1 Counter

Demonstrates formulas and action buttons. The document contains a value node storing the count (e.g., `"5"`), a ref node displaying it, and two action buttons targeting the value.

The "+1" button records wrapping the target in `formula(plus)` with argument `"1"`. Clicking transforms `"5"` into a formula `plus(ref, "1")` that evaluates to `"6"`. The "-1" button similarly wraps with `minus`.

2.7.2 Todo List

Demonstrates recording an "add item" pattern. The document contains a `ul` with existing `li` items (each with a checkbox and text), plus an "Add" action button targeting the list.

To create the button: select the `ul`, record adding a new `li` with checkbox and text value, save to the action node. Each click replays the pattern, appending a new item.

2.7.3 Conference List (Schema Refactoring)

Demonstrates concurrent editing with CRDT conflict resolution. The document contains a list of conferences, each with name and location.

Scenario: User A adds a "year" field to existing conferences (records wrapping each `li`'s content). User B concurrently adds a new conference.

Result: Both operations succeed. A's changes apply to existing items; B's new item appears without the year field. A can replay the transformation on B's new item to add the field.

2.7.4 Hello World (Bulk Transformation)

Demonstrates applying transformations across multiple items. The document contains a `ul` with several `li` items ("Hello", "World", "Denicek") and an "Emphasize" action button.

Record wrapping content in `<strong>` and appending "!". Replay on each `li` transforms "Hello" to "Hello!" wrapped in `strong`.

2.7.5 Price Calculator (Formulas)

Demonstrates reactive computation. The document contains value nodes for quantity (`"3"`) and price (`"25"`), plus a `formula(multiply)` node referencing both. The formula evaluates to `"75"` and updates automatically when either value changes.

Additional formulas like `countChildren` can count items in a list dynamically.

3 Platform and technologies

The project uses the following technologies:

- **Programming Languages:**
  - TypeScript – all packages and applications
- **Frameworks and Libraries:**
  - React 19 – user interface framework
  - Fluent UI (`@fluentui/react-components`) – UI component library
  - Loro – CRDT library with native movable tree support
  - Vite – build tool and development server
- **Testing:**
  - Vitest – unit testing for core library
  - Playwright – end-to-end testing for web application
- **Code Quality:**
  - ESLint – linting with typescript-eslint
  - eslint-plugin-simple-import-sort – import sorting
  - eslint-plugin-unused-imports – unused import removal
- **Infrastructure:**
  - npm workspaces – monorepo management
  - GitHub Pages – web application hosting
  - Azure App Service – sync server hosting
  - WebSocket – real-time synchronization

4 Evaluation

The evaluation approach focuses on demonstrating that the CRDT-based synchronization correctly implements the Denicek editing model and supports the formative examples from the original paper.

Unit Tests The core library includes unit tests (Vitest) that verify:

- Correctness of node operations (add, delete, move, copy, wrap)
- Conflict resolution behavior for concurrent operations
- Undo/redo functionality
- Recording and replay of operation sequences
- Synchronization between multiple documents

End-to-End Tests The web application includes Playwright tests that verify:

- User interface interactions (selection, navigation, editing)

- Recording and replay workflows
- Undo/redo through the UI
- Bulk operations on multiple nodes

Formative Examples Evaluation against examples from the Denicek paper (Section 2.7): counter (reactive values, action buttons), todo list (recording patterns), conference list (concurrent schema changes), hello world (bulk transformations), price calculator (formula engine).

Comparison with Original Denicek A qualitative comparison will assess:

- Feature parity with the original system
- Differences in conflict resolution behavior
- Developer ergonomics of the API

5 Risks

CRDT Limitations Some Denicek operations may not map cleanly to CRDT semantics. Mitigation: Identify problematic operations early and design workarounds or document limitations.

Loro Performance Large documents may cause performance issues. Mitigation: Implement lazy loading and pagination if needed; profile performance with realistic document sizes.

Conflict Resolution Semantics Users may expect different conflict resolution behavior than what CRDTs provide. Mitigation: Document the conflict resolution rules clearly; provide UI feedback when conflicts are resolved.

Browser Compatibility WebAssembly (used by Loro) may have compatibility issues in some browsers. Mitigation: Test on all target browsers; provide fallback or polyfills if needed.

Scope Creep The desire to implement all Denicek features may exceed available time. Mitigation: Prioritize core functionality (editing, sync, undo/redo) over advanced features (formula evaluation, debugging).

6 Milestones / Deliverables

MyDenicek Library: TypeScript library with CRDT-based documents (element, value, formula, ref, action nodes), edit operations, formula engine (19 operations), action nodes with recording/replay, Loro-based conflict resolution, and undo/redo.

MyWebnicek Application: React web app with document rendering, toolbar-based editing, recording/replay UI, and WebSocket collaboration.

Documentation: README, API docs, architecture overview, conflict resolution rules.

Tests: Unit tests (Vitest) for core library; E2E tests (Playwright) for web app.

Demo: Live demonstration of editing, recording/replay, collaboration, and conflict resolution.

Source Code: Open-source GitHub repository.

7 Time Schedule

The project spans September 2025 to March 2026. The proposal was submitted on 16.10.2025 and approved on 11.11.2025. Work began with attendance at the DARE2025 summer school on CRDTs and local-first software, followed by analysis of the original Denicek system and prototyping. The following table shows completed work (DARE2025 + git history, counting each day with commits as 1 full working day) and estimated remaining effort.

Milestone	**Activity**	**Time**	**MD**	**Status**
DARE2025	Summer school on CRDTs and local-first	Sep 2025	5	Done
Analysis	Study Denicek, CRDTs, prototype	Oct–Nov 2025	4	Done
Core + Undo	DenicekModel, UndoManager, basic UI	Nov–Dec 2025	14	Done
Recording/Replay	Recorder, generalized patches	Dec 2025	1	Done
Monorepo + Tests	Package structure, unit/E2E tests	Jan 2026	6	Done
Loro Migration	Replace Automerge with Loro CRDT	Jan 2026	4	Done
Programmable Buttons	Action nodes, recording to buttons	Jan 2026	2	Done
Formula Engine	Evaluation, operations, ref nodes	Jan 2026	2	Done
Completed			**38**	
Sync Server Deployment	Deploy to Azure, public collaboration	Jan–Feb 2026	4	Planned
Web Application Polish	UI improvements, conflict feedback	Feb 2026	6	Planned
Formative Examples	Counter, todo, conference list demos	Feb–Mar 2026	5	Planned
Testing & Bug Fixes	E2E tests, edge cases, bug fixes	Mar 2026	5	Planned
Documentation	README, API docs, video demo	Mar 2026	4	Planned
Finalization	Polish, presentation preparation	Mar 2026	3	Planned
Remaining (estimated)			**27**	
Total			**65**	

Table 1: Project timeline: completed work (DARE2025 summer school + git history) and estimated remaining effort.

The completed work includes migration from Automerge to Loro CRDT (January 2026), formula engine, and programmable buttons. WebSocket synchronization works locally; remaining work focuses on deploying the sync server to Azure App Service for public collaboration, UI polish, and formative example demonstrations.

8 Form of collaboration

Collaboration with the supervisor, Mgr. Tomáš Petříček, Ph.D., is essential for ensuring alignment with the original Denicek design and research objectives.

8.1 Consulting plan

Regular Meetings Weekly meetings with the supervisor, 60–90 minutes each. Topics: progress updates, technical challenges, design decisions, alignment with original Denicek semantics.

Ad-hoc Communication Discord for quick questions between meetings.

Code Reviews The supervisor may review code and provide feedback on architecture and implementation decisions.

8.2 Repository management

The project is hosted on GitHub at `https://github.com/krsion/MyDenicek`. Development follows standard Git workflow with feature branches and pull requests. The main branch is deployed to GitHub Pages for the live demo.

References

[1] T. Petříček, et al. *Denicek: Computational Substrate for Document-Oriented End-User Programming.* In Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25). no. 32, pp. 1–19.

[2] M. Kleppmann, et al. Local-first software: you own your data, in spite of the cloud. Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. 2019.

[3] Clemens Nylandsted Klokmose, James R. Eagan, and Peter van Hardenberg. 2024. MyWebstrates: Webstrates as Local-first Software. In Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24). Association for Computing Machinery, New York, NY, USA, Article 42, 1–12.

[4] Loro: Reimagine state management with CRDTs. `https://loro.dev/`

[5] N. Preguiça. *Conflict-free Replicated Data Types: An Overview.* 2018. `https://arxiv.org/abs/1806.10254`.