SplitLess: A Peer-to-Peer Expense Sharing Application

Floyd Peiszan
Department of Computer Science
University of Basel
floyd.peiszan@unibas.ch
Examiner: Prof. Dr. Christian Tschudin
Supervisor: Dr. Erick Lavoie

Master's Project Report January 19, 2026

Abstract

Expense sharing applications require strong correctness guarantees, especially when implemented in a decentralized and concurrent setting. Peer-to-peer designs avoid central coordination but introduce subtle concurrency challenges that are difficult to reason about informally. We present SplitLess, a decentralized expense sharing system formally specified in TLA+. The model supports dynamic group membership, editable expenses, and asynchronous replication while preserving consistency through restricted modification rights, share acknowledgment, and well-defined merge semantics. We verify safety and liveness properties using the TLC model checker on bounded but representative configurations. Based on the specification, we implement a Python command-line application as a proof of concept and evaluate it using large-scale probabilistic testing. The results show no correctness violations and provide strong evidence that the TLA+-driven design generalizes beyond the verified models, demonstrating the effectiveness of formal methods for building correct peer-to-peer applications.

Acknowledgments

I would like to thank Prof. Dr. Christian Tschudin and Dr. Erick Lavoie for providing the opportunity to work on this project.

I am especially grateful to my supervisor Dr. Erick Lavoie for his continuous support throughout the project. Our discussions were consistently helpful, and his guidance was particularly valuable when I was getting started with TLA+, which I had not used before. I benefited from his detailed feedback on early drafts of this report, as well as from his suggestions on the overall design of the application. His help in structuring TLA+ records and actions, especially in the early stages when my specifications were still overly complex, provided a clear framework that I could rely on throughout the project.

Contents

1 Introduction

Modern digital applications increasingly mediate everyday social and financial interactions, whether it is arranging a night out with friends through social media, buying a new pair of socks from Amazon, or looking up the opening hours of a restaurant on Google Maps. Digital communication governs much of our daily lives. Applications of this kind are commonly built around centralized service providers that collect, store, and process user data. While centralization simplifies system design, it also introduces structural concerns. Users must trust a single entity with often sensitive data, system behavior can be non-transparent, and operational costs and failures are concentrated in one place. In response, there has been growing interest in decentralized and peer-to-peer systems as a way to distribute trust and reduce reliance on centralized infrastructure.

Expense sharing applications are a common example of socially embedded software that manages sensitive financial information. Typical use cases include shared housing, group travel, and social activities among friends. Most existing expense sharing applications rely on centralized servers to store transaction data and compute balances. This raises privacy concerns, such as providers having direct access to detailed transaction histories or monetizing user data.

In this report, we explain the design and implementation of a peer-to-peer expense sharing application for groups of friends, which we call SplitLess. The core goal is to support expense sharing without relying on a central server. We allow dynamic group membership and editable expenses that can be dynamically assigned to groups. Redesigning the application as a peer-to-peer system introduces several challenges. We need to maintain correct balances under concurrent updates, handle dynamic group membership and evolving expense states, and ensure consistency across replicated states. In contrast to centralized approaches, solving these challenges is more complex in a peer-to-peer environment. There is no central authority defining the correct system state, and eventual consistency among peers must be achieved through robust mechanisms. Addressing these challenges requires careful system design and validation of correctness properties. We focus on correctness and consistency rather than on security mechanisms such as encryption or authentication. Design choices and assumptions are motivated by the intended use case of a trusted group of friends.

This project combines formal modeling with practical implementation and testing. We contribute a TLA+ specification of the SplitLess system, including key safety and liveness properties. The specification is model checked up to three users and three replicas, with one expense and one group, giving us strong confidence in model correctness. We also verified two users on two replicas, with two expenses and two groups. Furthermore, we present a Python implementation of the application, including a command-line interface consistent with the TLA+ specification. We verify key safety and liveness properties of the Python implementation using probabilistic testing.

The remainder of this report is organized as follows. We begin in Chapter 2 by introducing the background and core concepts underlying decentralized and concurrent systems. In Chapter 3 we then explain the design of the SplitLess application with its formal TLA+ specification, covering the core data structures, actions, concurrency challenges, merge semantics, and correctness properties. We next

turn to the implementation in Chapter 4, where we explain how the TLA+ specification is translated into a usable Python system. In Chapter 5, we evaluate both the design and the implementation using TLC model checking and probabilistic testing of the Python code. In Chapter 6, we place SplitLess in context by discussing related work and comparing it to other decentralized financial approaches. We conclude the report in Chapter 7 by summarizing the results and outlining directions for future work.

2 Background

2.1 TLA+ and TLC

In this section, we give a brief overview of the Temporal Logic of Actions (TLA+) language and its model checker, the Temporal Logic of Actions Checker (TLC). TLA+ was developed by Leslie Lamport and is described both in a book [4] and on an accompanying website with video lectures [5]. We use TLA+ to model the SplitLess application and to reason about its key functionality and properties. This report does not aim to explain how to use TLA+ in general. Instead, we introduce the necessary syntax when presenting excerpts from our specification.

TLA+ is a formal specification language for describing concurrent and distributed systems. It focuses on what a system does rather than how it is implemented, abstracting away unnecessary implementation details. The language is grounded in mathematics, in particular set theory and temporal logic. A TLA+ specification models system execution as a sequence of discrete snapshots, called states. Each state is formally defined as an assignment of values to variables. State transitions are defined by actions, which describe how variables change to produce a new state. Actions may include preconditions that restrict the set of states in which they can be applied. A sequence of states generated in this way is called a behavior of the specification. TLC is a tool that explores finite behaviors of a TLA+ specification, starting from an initial state. By checking properties and invariants over states and behaviors, TLC can systematically detect logical flaws in the specification.

Using TLA+ to model distributed applications offers several key benefits. It encourages a state-based view of system behavior and requires precise definitions of system assumptions and guarantees. This shifts effort from debugging implementation code to validating the system design, reducing conceptual complexity in later development stages. Combined with TLC, this approach provides strong confidence in the correctness of the model within the explored bounds.

In this project, we use TLA+ to formally model the core behavior of the SplitLess system. The specification defines system states, allowed operations, and correctness properties. We use TLC to validate key safety and liveness properties under concurrent execution. The TLA+ specification also serves as the foundation for our command-line implementation of the application.

2.2 Conflict-Free Replicated Data Types (CRDTs)

In this section, we describe conflict-free replicated data types (CRDTs), which we use to ensure correctness under concurrent state modifications on different replicas, that is, local copies of the application state maintained on each user's device.

CRDTs are data structures designed for replicated systems, allowing multiple replicas to be updated concurrently without coordination. Replicas may temporarily diverge, but, assuming updates are eventually communicated, they are guaranteed to converge again. CRDTs achieve convergence without relying on a central authority or a total ordering of operations. Instead of resolving conflicts after

they occur, merge conflicts are avoided by construction, based on carefully defined mathematical merge semantics.

The most important property of CRDTs is **strong eventual consistency**. Informally, this property ensures that all replicas eventually converge to the same state.

Definition 2.2.1 (Strong Eventual Consistency)**.** from [3], slightly rephrasing [10]

1. Eventual update: If an update is applied by a correct replica, then all correct replicas will eventually apply that update.

2. Convergence: Any two correct replicas that have applied the same set of updates are in the same state, even if the updates were applied in a different order.

Furthermore, state-based CRDTs require monotonic state growth, which informally means, that more recent states can only evolve within carefully constrained possibilities. Together, these properties make CRDTs tolerant to message reordering, duplication, and delay, and therefore well suited for decentralized, asynchronous communication. For this reason, we apply CRDT-based data types throughout SplitLess for all replicated storage.

2.2.1 Causal-Length Sets

A causal-length set (CLS) is a CRDT used to represent a set that supports both additions and removals. It is designed to correctly handle concurrent add and remove operations. In SplitLess, we use CLS to model group membership.

Each element in the set is associated with a monotonically increasing integer counter. The parity of this counter determines whether the element is present: even values represent absence, while odd values represent presence. Adding or removing an element increments the counter. However, an operation only modifies the counter if it changes the element's state. For example, an add operation increments the counter only if the element is currently absent. Replicas converge by taking the maximum counter value for each element. As a result, the final state depends on causal history rather than on the order of operations. A more detailed explanation of the intuition behind CLS can be found in [8].

A CLS can be viewed as a dictionary in which keys represent set elements and values store the corresponding integer counters.

Algorithm 1 CLS Add

1: **Input:** element e
2: **if** $CLS[e]$ is even **then**
3: $\quad \texttt{CLS}[e] \leftarrow CLS[e] + 1$
4: **end if**

Algorithm 2 CLS Remove

1: **Input:** element e
2: **if** $CLS[e]$ is odd **then**
3: $\quad \texttt{CLS} \leftarrow CLS[e] + 1$
4: **end if**

Algorithm 3 CLS Contains

1: **Input:** element e
2: **return** $\texttt{CLS}[e] \bmod 2 = 1$

Algorithm 4 CLS Merge

1: **Input:** $\texttt{CLS}_1, \texttt{CLS}_2$
2: **for all** e **do**
3: $\quad \texttt{CLS}_1[e] \leftarrow \max(\texttt{CLS}_1[e], \texttt{CLS}_1[e])$
4: **end for**

This data structure allows us to model dynamic group membership with concurrent updates across replicas, while ensuring that the most recent information eventually reaches every replica, assuming updates are propagated.

3 Design

In this chapter, we describe the design of SplitLess and the main ideas behind its decentralized architecture. The system is modeled using TLA+, which we use to specify states, actions and correctness properties. The complete specification is given in Appendix A.

SplitLess is a decentralized, peer-to-peer expense sharing application, that allows users to create groups and track shared expenses without relying on a central server. The system supports dynamic group membership and editable expenses, allowing users to correct mistakes after expenses have been created or assigned to a group. Users operate on local replicas and may perform updates concurrently, even while temporarily offline.

This decentralized setting introduces concurrency challenges that do not arise in centralized applications. Independent updates on replicated states can lead to ambiguity or inconsistent intermediate states if not handled carefully. The design of SplitLess therefore focuses on correctness under concurrency. Wherever possible, updates follow CRDT-inspired principles and require no coordination. Some corner cases rely on additional synchronization mechanisms.

The chapter is structured as follows. We first give a high-level overview of the supported functionality and the main concurrency challenges. We then introduce the correctness properties of the system, followed by the state model and the basic action set. Finally, we describe how the identified concurrency challenges are resolved through action restrictions, synchronization and merge semantics.

The design assumes usage by small groups of friends with a basic level of trust. As a result, encryption, authentication and digital signatures are out of scope, and the focus is placed on correctness and consistency of replicated states.

3.1 Overview and Supported Functionality

At a high level, users can create groups, invite other users and leave groups again. Users can create expenses, specify how costs are shared and later modify or delete expenses to correct mistakes. Expenses exist independently from groups and can be assigned to or removed from a group. Group balances are derived from the set of valid expenses associated with the group.

Each user operates on a single local replica and performs actions without immediate coordination. Updates are exchanged asynchronously and replicas are merged to restore a consistent view of the system state. Merges are deterministic and require no administrator to resolve potential conflicts.

3.2 Concurrency Challenges

Our execution model allows replicas to temporarily diverge. When multiple users perform operations on the same logical entities, such as groups or expenses, concurrent updates can interact in subtle ways. Without additional constraints, these interactions can lead to ambiguous or inconsistent states after replicas are merged.

In this section we identify the main concurrency challenges that arise in SplitLess. We focus on how concurrent actions affect the system state and what issues arise during merging. The mechanisms used to resolve these challenges are discussed in Chapter 3.6.

3.2.1 Concurrent Expense Modifications

Expenses are editable to allow users to correct mistakes after creation. In a decentralized setting, this raises the possibility that the same expense is modified concurrently on different replicas. If two replicas apply conflicting modifications to the same expense, multiple incompatible versions of that expense may exist at the same time.

Without additional constraints, merging such replicas would require resolving which version of the expense should be considered the most recent or correct. This ambiguity directly affects balance calculations and can lead to inconsistent group states, as different replicas may include different versions of the same expense when computing balances.

3.2.2 Concurrent Expense Group Assignment

Expenses exist independently from groups and can be assigned to or removed from a group. If concurrent actions assign the same expense to different groups on different replicas, the system may temporarily contain states where a single expense appears to belong to multiple groups.

This situation is problematic because it could cause double-spending. Group balances are derived from the expenses in it. When the same expense is assigned to multiple groups it will count towards the balances in both groups.

3.2.3 Leaving Groups During Expense Updates

Group membership is dynamic, allowing users to leave and rejoin groups. A concurrency challenge arises when a user leaves a group while an expense involving that user is concurrently added or modified on another replica.

In such a scenario, a user may accumulate additional balance changes after they have already left the group. This undermines the expectation that leaving a group finalizes a user's balance.

3.3 Correctness Properties

The concurrency challenges described in the previous section make it insufficient to reason about SplitLess purely in terms of individual actions or local states. Instead, we need a precise notion of correctness that holds even when replicas evolve independently and are merged asynchronously.

Because SplitLess manages shared financial data, small inconsistencies can lead to unfair outcomes or loss of trust among users. This risk is amplified by decentralization, where concurrent updates and delayed communication are the norm. Correctness therefore has to be treated as a first-class concern in the system design.

To make correctness explicit, we define formal correctness properties in the TLA+ specification. These properties restrict both the set of reachable states and the allowed state transitions, allowing us to precisely state what behaviors are acceptable and to verify them using model checking.

In TLA+, correctness properties are expressed using three complementary classes. *Invariants* restrict the set of valid system states and must hold in every reachable state. *Safety properties* restrict

how state variables evolve over time. *Liveness* properties ensure that progress is eventually possible, despite concurrency and delayed communication.

We group the correctness properties of SplitLess into three conceptual categories. First, financial correctness properties ensure that expenses and balances behave consistently. Second, CRDT-based properties guarantee convergence under replica merging by enforcing monotonicity of versioned states. Finally, additional synchronization properties capture correctness requirements that go beyond pure CRDT semantics. These properties are necessary to solve the previously identified concurrency challenges. We introduce the synchronization mechanisms in Chapter 3.6.

In the following, we introduce the correctness properties at a conceptual level. The full set and formal definitions of these properties are given in Appendix A, and their verification is discussed in Chapter 5.

3.3.1 Financial Correctness Properties

The primary purpose of SplitLess is to track shared expenses and balances correctly. Financial correctness properties ensure that money is neither created nor lost and that balances are consistent across group membership changes. These properties are primarily expressed as *invariants*, as they must hold in every reachable system state.

At the expense level, correctness requires that the individual shares of an expense sum up to the total expense amount. This prevents inconsistent or malformed expense records. At the group level, the sum of balances of all active members must always be zero. This reflects the closed nature of a group: every amount paid by one user is owed by the others. These properties ensure that balance calculations remain meaningful even as users join or leave groups and expenses are modified.

Financial invariants form the foundation of correctness in SplitLess. They are independent of concurrency but must remain preserved under all concurrent behaviors introduced by replication and merging.

3.3.2 CRDT-Based Correctness Properties

SplitLess relies on replicated states and asynchronous communication between replicas. To guarantee eventual consistency, the system adopts CRDT-inspired design principles. The associated correctness properties ensure that replica states can be merged deterministically and that replicas eventually converge to the same result.

These guarantees are expressed through a combination of *Safety* and *Liveness* properties. We define safety properties so that version and membership counters are required to be monotonically non-decreasing. This ensures that newer information always dominates older information during merges and that replicas never revert to outdated states. By enforcing monotonicity, merges can be implemented by taking component-wise maxima. This guarantees convergence without requiring coordination or global ordering.

Liveness properties complement this by ensuring that updates eventually propagate. They assert that the most recent versions of expenses and group membership changes will eventually be reflected on all replicas, assuming continued communication. Together, these properties guarantee that replicas not only remain internally consistent but also converge to the same state over time.

3.3.3 Additional Synchronization Properties

While many aspects of SplitLess can be modeled using CRDT principles, some concurrency scenarios require additional synchronization to preserve correctness. For example concurrent expense modifications cannot be resolved by monotonic merging alone (see Chapter 3.2.1).

To address these cases, SplitLess introduces explicit synchronization mechanisms such as share acknowledgments and invitation acceptance, which are described in detail in Chapter 3.6. The associated correctness properties ensure that these mechanisms enforce progress and prevent inconsistent states. These properties are expressed as a combination of *safety* and *liveness* properties.

Safety properties ensure that acknowledgments, once given, are not revoked within the same expense version. Liveness properties ensure that it is always possible for an expense to eventually become valid, even if participants temporarily leave a group. These properties guarantee that the system does not deadlock in partially completed states.

3.4 State Model

In the next Sections we introduce the formal state model and the basic action set, before we return to each challenge and show how the design resolves it. We now introduce the state model of SplitLess from a conceptual perspective. We first describe the main data structures that make up the system state and their roles in the application. We then explain how these concepts are represented in TLA+ by using records and giving a concrete example. The complete and precise specification of all state variables can be found in Appendix A.

3.4.1 Conceptual Data Structures

The state of SplitLess is composed of three main kinds of entities: replicas, groups, and expenses. Together they capture all information required to track shared expenses in a decentralized setting.

An *expense* represents a single financial transaction. It records who paid for it, how the total amount is split among participants, and the currently associated group. Since expenses can be modified, each expense carries versioning information that allows replicas to resolve conflicts deterministically.

A *group* represents a set of users who share expenses. Groups define the social and financial context in which expenses are interpreted. They track user membership with a causal length set (see Chapter 2.2.1). Each user has their own grow-only counter that determines whether they are currently an active member or not.

A *replica* represents the local copy of the system state. Each user operates on exactly one replica and applies actions locally without coordination with others. Replicas store all groups and expenses known to the user at a given point in time. Because replicas evolve independently and communicate asynchronously, their states may temporarily diverge. Reconciling these differences is handled through merge operations, which are discussed later.

Together, these three data structures form the complete system state. All user actions operate by transforming this state, either by creating new entities or by modifying existing ones.

3.4.2 Modeling States with TLA+ Records

The TLA+ language supports the definition of *records*. You can think of them as complex data structures similar to `struct`s in C, as they allow grouping multiple related variables into a single

value. In our specification we use records to model the core entities of the system. Records in TLA+ are defined using square brackets. Each field is associated with a name and a type.

As an example, the following record models a single expense:

```
Expense ==
    [ id : POSSIBLE_EXPENSE_IDs,
      group : POSSIBLE_GROUP_IDs \cup {NO_GROUP},
      version : Nat,
      payer : USERS,
      amount : Nat,
      shares : [USERS -> Nat],
      acknowledged_shares : [USERS -> BOOLEAN],
      deleted : BOOLEAN
    ]
```

Each expense is identified by a unique identifier. The `group` field indicates whether the expense is currently assigned to a group or not. The `version` field is used to order modifications of the same expense across replicas. The `payer`, `amount`, and `shares` fields capture the financial content of the expense, while `acknowledged_shares` records whether participants have accepted their share. More about this in Chapter 3.6.3. The `deleted` flag allows expenses to be removed without physically erasing them from the state.

Capitalized names such as `USERS` or `POSSIBLE_EXPENSE_IDs` refer to constant sets that are defined when configuring a specific TLC model. This separation between specification and configuration allows us to reason about the system independently of concrete system size.

Similar records are defined for groups and replicas. Their full definitions, together with all auxiliary state variables, are provided in Appendix A.

3.5 Allowed Actions

SplitLess evolves through user-triggered actions that modify the system state. Actions correspond to the operations users intuitively expect from an expense sharing application, such as creating groups, adding expenses or leaving a group. Each action is executed locally on a replica and the result may later be propagated to other replicas through merging.

In this section we describe the *basic actions* of the system from a high-level perspective. Most of these actions are designed to remain CRDT-friendly. In later sections we introduce a small number of additional synchronization mechanisms to handle concurrency corner cases. The basic actions capture the intended functionality of SplitLess but do not yet address all concurrency-related issues. In Chapter 3.6 we add restrictions and describe the synchronization mechanisms.

3.5.1 High-Level Overview of Actions

The actions of SplitLess can be divided into group- and expense-related actions. Together, they define the core functionality of the application.

Group Actions

We define the following actions to manage groups and their membership:

1. **CreateGroup:** This action allows a user to create a new group. The creator becomes the initial and only member of the group. Each user may create an arbitrary number of groups.

2. **InviteMember:** This action allows an active group member to invite another user to the group. Invitations do not immediately change group membership but signal intent to add a new member.

3. **AcceptInvitation:** This action allows an invited user to accept an invitation and become an active member of the group. Explicit acceptance ensures that users are not added to groups without consent.

4. **LeaveGroup:** This action allows an active member to leave a group. Leaving is only permitted if the user does not have a negative balance in the group. If a user leaves with a positive balance, this balance is gifted to the group. More about this in Chapter 3.6.4.

These actions enable dynamic group membership and reflect common real-world usage patterns of expense sharing applications.

Expense Actions

We define the following actions to manage expenses. Every action increments the expense's version.

1. **CreateExpense:** This action allows a user to create a new expense. The creator of the expense is marked as the payer. The payer specifies the shares of all participants, and the total amount is derived from these shares. Initially, the expense is not assigned to any group.

2. **AddExpenseToGroup:** This action assigns an existing expense to a group. All users participating in the expense (i.e who have positive shares) and the payer need to be members of the group.

3. **RemoveExpenseFromGroup:** This action removes an expense from its assigned group. After removal, the expense no longer affects the group's balances.

4. **ModifyExpenseParameters:** This action enables expense modifications, in particular the participants' shares. The set of participants may change between versions.

5. **DeleteExpense:** This action marks an expense as deleted. Deleted expenses do not contribute to group balances and deletion cannot be undone.

At this level, expense actions describe only their intended effects. Additional constraints on who may execute these actions under which conditions are introduced in Chapter 3.6 when we address concurrency challenges.

3.5.2 Modeling Actions in TLA+

In the TLA+ specification, actions define all allowed state transitions of the system. An action specifies a set of parameters, the conditions under which the action is enabled, and the state changes that occur when the action is executed.

Most actions follow the same structural pattern. We describe this pattern using the `CreateGroup` action as an example:

```
CreateGroup ==
      \E actor \in USERS :
      \E gid \in POSSIBLE_GROUP_IDs :
      \E rid \in POSSIBLE_REPLICA_IDs :
        /\ ASSIGNED_REPLICA[actor] = rid
```

```
/\ \A otherRid \in POSSIBLE_REPLICA_IDs :
      replicas[otherRid].groups[gid] = NO_GROUP
/\ LET newGroup ==
      [ id |-> gid,
        members |-> [u \in USERS |-> IF u = actor THEN 1 ELSE 0],
        totalGifted |-> 0,
        individualGiftsSent |-> [u \in USERS |-> 0]]
   IN /\ replicas' =
        [replicas EXCEPT ![rid].groups[gid] = newGroup]
      /\ actionCounter' = actionCounter + 1
```

In the first part, actions introduce a set of variables that must exist for the action to be enabled. These variables are declared using the existential quantifier (\E). In this example, the action requires the existence of an actor, a group identifier and a replica identifier.

In the second part, the action specifies conditions that must hold for the chosen variables. The action is enabled only if all these conditions are satisfied. Multiple conditions are combined using conjunction. For example, `CreateGroup` requires that the actor operates on their assigned replica and that no other replica already contains a group with the same identifier.

The final part of the action defines the resulting state transition. The `LET` clause is used to construct new records, and the `EXCEPT` construct specifies how existing records are updated. Variables annotated with a prime symbol (') represent the next-state values. All state variables defined in the initial state of the specification must be assigned in every action.

All other actions in the specification follow this general structure. While their preconditions and state updates differ, this pattern provides a clear and explicit mapping between user-level operations and state transitions. The complete set of actions and their detailed definitions are provided in Appendix A.

3.6 Resolution of Concurrency Challenges

Independent operations on replicated states introduce situations where concurrent actions can lead to ambiguous or inconsistent system states. In this section we describe how we resolve the main concurrency challenges introduced in Chapter 3.2. We add restrictions and extend the action set with synchronization where a plain CRDT style is not sufficient.

3.6.1 Concurrent Expense Modifications

If the same expense is modified concurrently on different replicas, two different expense records with the same identifier and version number, but different contents can exist. At the state level, this results in multiple incompatible representations of the same expense. We use version numbers to determine which record is most recent. Identical versions make it impossible to deterministically decide which expense state should dominate during merging.

SplitLess prevents this situation by restricting expense modifications to a single user. Only the payer of an expense is allowed to modify it. Because each user operates on exactly one replica, this restriction ensures that modifications of a given expense are serialized. As a result, there can never be two concurrent modifications of the same expense.

Each modification increments the expense version counter. During merging, replicas are combined expense-wise by selecting, for each expense, the record with the highest version number. Since versions are monotonically increasing and concurrent modifications are prevented, merging is deterministic.

Correctness is ensured by financial invariants that validate expense consistency, safety properties that enforce monotonically increasing expense versions, and liveness properties that guarantee the most recent version eventually reaches all replicas.

3.6.2 Concurrent Expense Group Assignment

Another challenge occurs when the same expense is assigned to different groups on different replicas. At the state level, this results in an expense that contributes to multiple group balance calculations. Since group balances are derived from the expenses assigned to them, this leads to double counting of the same expense and violates basic financial correctness.

This situation is prevented by restricting group assignment actions in the same way as expense modifications. Only the payer of an expense is allowed to add or remove it from a group. Because the payer acts on a single replica, concurrent group assignments for the same expense cannot occur.

Adding or removing an expense from a group increments its version number and stores the current group in the expense record. During merging, the expense record with the highest version is selected, ensuring that after merging the expense is associated with the same group.

Correctness is enforced by invariants that ensure referenced groups exist and that group balances sum to zero. Safety properties guarantee monotonic version growth, and liveness properties ensure that group assignment updates propagate to all replicas.

3.6.3 Leaving Groups During Expense Updates

A more subtle issue arises when a user leaves a group, while concurrently on another replica, an expense involving that user is added, modified or removed. At the state level, this can lead to situations where a user accumulates additional balance after they have already left the group.

To address this, SplitLess introduces share acknowledgments. An expense only becomes valid and contributes to group balances once all users with positive shares have explicitly acknowledged their participation. This ensures that users are aware of and agree to the expenses they are involved in, before those expenses affect balances.

Acknowledged shares are modeled such that they cannot be revoked for the same expense version. When an expense is modified or when its' group membership changes, all acknowledgments except those of the payer are reset. This forces users to re-acknowledge updated information.

During merging, replicas first select the expense record with the highest version number. If two records have the same version, acknowledged shares are merged by taking the union of acknowledgments from both replicas. This preserves progress while avoiding inconsistencies.

To ensure expenses can always become valid, a liveness property guarantees that all positive shares can eventually be acknowledged. In cases where a user leaves and never acknowledges their share, an additional action allows the payer to absorb the leaving user's share. For completeness, the model also includes an action that allows a user to rejoin and acknowledge their share.

3.6.4 Users with Positive Balances Leaving a Group

While not a concurrency challenge, a further correctness issue arises when a user with a positive balance leaves a group. A positive balance means the group owes money to that user. At the state level, this would leave remaining users owing money to someone who is no longer a group member, causing group balances to no longer sum to zero.

SplitLess resolves this by requiring users who leave with a positive balance to gift their remaining funds to the group. The group stores the total gifted amount received and distributes it equally among the active members. For each individual user, the group also tracks the amount gifted towards the group. If the user later rejoins the group, they do not reclaim the gifted balance but receive an equal share like all other members.

This mechanism introduces an additional dependency between expenses and gifted balances. If an expense is later modified or deleted, previously gifted amounts derived from that expense may no longer be valid. To prevent inconsistencies, gifted balances are not stored as fixed values. Instead, they are recalculated whenever an expense changes in a way that affects group balances.

The gifted amount by a user, who left the group at some point, is computed as the total amount they paid towards the group minus the total amount they owed. By recalculating gifts from the current expense state, the model ensures that gifted balances always remain consistent with the underlying financial data. The gifts are included in the balance calculation by subtracting what a user gifted to a group and adding an equal share of the total amount gifted towards the group.

During merging, gifted amounts need to be recalculated. We first merge all expense records and groups and only afterwards recompute gifted balances based on the merged expenses.

Financial invariants ensure that group balances sum to zero. Together with the merging semantics for expenses and groups, this preserves correctness.

4 Implementation

The Design described in Chapter 3 and specified in TLA+ defines the intended behavior of the SplitLess system and forms the basis for concrete implementations of the application. We now present a Python command line interface (CLI) tool that allows users to interact with the system and execute the defined actions. The implementation preserves the semantics of the formal model while introducing practical system considerations. It serves as a proof of concept rather than a production-ready application. We use simple mechanisms for replica storage and communication to demonstrate that SplitLess is usable in practice. The implementation is publicly available on GitHub[1].

In this section, we describe the Python CLI implementation, with a focus on how it differs from the TLA+ model. We provide an overview of the implementation structure and explain how it aligns with the formal specification.

4.1 Differences to the TLA+ Specification

The TLA+ model abstracts away all implementation details that are not required to specify and reason about the system. Implementing the model introduces additional concerns that are not present in the specification. In particular, we must add persistent storage for replicated states, incorporate real network communication between processes, support dynamic identifier generation, and provide a user interface for interacting with the application. Furthermore, we introduce functionality to map opaque identifiers to human-readable names in order to improve usability. We discuss each of these aspects in turn.

4.1.1 Persistent Replica Storage

We use simple JSON files to persistently store replica states on disk, allowing states to survive application restarts. Because we assume peers are not malicious, we store data in plain form, without encryption or authentication. The file structure follows directly from the TLA+ record definitions. We store one file per user, representing that user's replica state.

In contrast to the TLA+ model, we allow exactly one user per replica. Each file contains one field for groups and one field for expenses known to the replica. Group and expense records are organized according to the TLA+ specification. For improved user experience, we additionally attach human-readable names to records. These names are not used as unique identifiers but serve only as mappings to the underlying identifiers. If multiple records share the same name, the system prompts the user to provide the explicit identifier.

Unlike the TLA+ specification, we also store a set of users known to the local machine. This enables a simple login mechanism, allowing multiple users to interact with their replicas on the same installation.

[1] https://github.com/floydpei/SplitLess

The current implementation does not cache replicas in memory. Instead, it performs a disk read or write for every user action. This simplifies the implementation. The resulting latency is negligible, as state updates occur infrequently and replica sizes are small on modern systems.

4.1.2 Communication

The TLA+ specification abstracts away communication entirely, modeling it implicitly through multiple replica variables. A real implementation must instead provide an explicit mechanism for exchanging states between machines, or between processes on the same machine.

In our Python implementation, we use UDP communication to transfer replica states. This approach makes it straightforward to exchange information within a local network. Since the application is a proof of concept and assumes a trusted group of users, we avoid complex or secure communication mechanisms. Messages are therefore sent without encryption or authentication.

4.1.3 Identifier Generation

In the TLA+ specification, identifiers for groups and expenses are defined as constants in the TLC model. Uniqueness is guaranteed because the model checker has a global view of all replicas. A real application cannot rely on predefined identifiers, as users act independently and cannot know whether an identifier is already in use.

To address this, we generate pseudo-random identifiers using Python's `uuid`[2] library. It provides cryptographically secure identifier generation according to RFC 9562, §5.4 and, most importantly for our purposes, is simple to use.

4.2 From TLA+ Specification to Executable Code

In this section, we describe the process of implementing the TLA+ specification in Python. Rather than introducing new architectural concepts, the code structure closely follows the structure of the TLA+ model. We split functionality into logical components, each responsible for a specific concern, such as handling groups. This keeps the implementation organized and simplifies reasoning about it.

The Python implementation is a direct operationalization of the TLA+ specification. Core TLA+ records are represented as Python dictionaries, and TLA+ actions are implemented as Python functions. No additional application logic is introduced beyond the system-level concerns discussed in Chapter 4.1. All logic functions follow the same structure. First, we explicitly check TLA+ preconditions using `if` statements, enforcing the same safety conditions as in the specification. If a precondition fails, the function exits with an error message. Once all preconditions are satisfied, the state is updated by modifying only a small number of fields in the replica dictionary. This closely reflects the use of `EXCEPT` clauses in TLA+.

4.2.1 Benefits of the TLA+ First Approach

Using the model-checked TLA+ specification as the starting point significantly reduces implementation complexity. It also prevents ad-hoc decision making during development. By closely following the specification, the implementation becomes shorter, easier to read, easier to reason about, and simpler to verify against the defined correctness properties. Without the specification, the logic would likely

[2] https://docs.python.org/3/library/uuid.html

be more intertwined, making it harder to identify and reason about corner cases. In addition, we would lack the carefully designed correctness properties that guide the implementation.

4.2.2 Command Line Interface

We provide a command line interface (CLI) to expose the application's functionality to users. The CLI builds on the implementation structure as a thin interface layer. The actions available to users correspond one-to-one with the TLA+ actions. In addition, we provide commands to display replica state and inspect balances.

The CLI prevents users from entering arguments that do not match the required types for a given action. However, users may still provide arguments that violate action preconditions. These checks are handled by the core logic, not the interface. The CLI is responsible only for invoking actions and displaying success or error messages.

We use a simple login mechanism that allows multiple users to work with their replicas on the same machine. This also simplifies manual testing, as we can create multiple processes for different users within the same directory.

5 Evaluation

In the context of expense sharing applications, correctness is critical, as users rely on the system to manage financial data. Even small inconsistencies can lead to unfair outcomes. As a distributed, peer-to-peer system, SplitLess is particularly prone to subtle bugs. Concurrency and replication significantly increase system complexity compared to a centralized approach. Informal reasoning alone is therefore insufficient to gain high confidence in correctness.

We define correctness properties for our TLA+ specification in Chapter 3, ensuring that all reachable system states remain within the expected bounds. In addition, we apply probabilistic testing to the Python CLI implementation. In this chapter, we discuss the practical evaluation of the correctness properties using TLC for the TLA+ specification and explain how we apply the testing approach to the Python CLI tool.

5.1 TLC Model Checking

TLA+ allows us to state precise correctness properties about a specification. TLC then exhaustively explores all reachable states within a finite model, starting from an initial state. This approach is particularly effective at uncovering corner cases that might otherwise be overlooked. In the following, we describe the models we tested, report the results, and discuss their implications.

5.1.1 Finite Models and Depth Bounding

The full SplitLess specification has an infinite state space. There are unbounded sequences of actions, as operations such as adding and removing entities can repeat indefinitely. In addition, there is no inherent upper bound on the number of users, replicas, groups, or expenses. TLC therefore requires bounding both the number of modeled entities and the exploration depth.

The depth limit prevents infinite alternation of actions while still allowing the exploration of meaningful concurrent behaviors. All our models were checked up to a depth of seven, meaning that at most seven actions are applied before the exploration stops. We do not count merge actions or share acknowledgment actions towards this depth. These actions must remain enabled even after reaching the depth limit to preserve the liveness properties. After the main exploration phase, TLC can still apply these actions to resolve potential liveness issues, such as replicas not receiving updates or a non-deleted expense never becoming validated. In TLA+, this is modeled using weak fairness assumptions over the relevant actions.

5.1.2 Model Checking Configuration

We tested several models with slightly different configurations. In this section, we focus on the largest models that we were able to verify successfully. In TLA+, identifiers for replicas, users, groups, and

expenses are specified as constants. For each TLC model, concrete values for these constants must be provided.

Model 1: The largest fully checked model consists of three users and three replicas, where each user operates on a distinct replica, one group, and one expense. With three replicas and users, we capture transitive concurrency. A single group is sufficient to verify concurrent behaviors, as groups are handled independently. With one expense, we can explore scenarios where users leave a group while the expense is modified concurrently. However, this model does not cover cases where an additional expense is added to a group while a user involved in that expense leaves concurrently.

Model 2: We were unable to extend Model 1 with a second expense, as the available machines did not have sufficient memory to complete liveness checking. Nevertheless, we include the partially explored results in this report.

Model 3: To capture interactions involving multiple expenses, we also tested a model with two users and two replicas, where each user operates on a distinct replica, two groups, and two expenses. In this configuration, there is no transitive concurrency. However, it allows us to verify scenarios where a user leaves a group, while concurrently, a second expense involving that user is added to the group. Because this model is significantly smaller, we were also able to verify configurations with two groups, ensuring that no unintended interactions arise between groups.

In addition, we evaluated smaller configurations with reduced exploration depth during development to validate basic behavior and invariants.

5.1.3 Model Checking Results

All models that could be fully explored with TLC on our machines showed no violations of the specified correctness properties. No counterexamples were found. For reference, we include a table summarizing the results, including the number of states explored. As mentioned earlier, Model 2 could not be fully checked. TLC ran for three days before the machine ran out of memory. Even so, we were able to explore and verify a substantial number of states for this model.

Table 5.1: TLC model checking results for the largest verified configurations (depth limit = 7).

	Users	**Replicas**	**Expenses**	**Groups**	**States**	**Distinct States**
Model 1	3	3	1	1	171.771.434	11.771.143
Model 3	2	2	2	2	35.866.123	2.826.758
Model 2	*3*	*3*	*2*	*2*	*12.034.193.738*	*1.149.903.300*

The large number of explored and distinct states highlights the combinatorial complexity, even for small configurations, and underlines the importance of exhaustive model checking. The fact that all checked models satisfy the specified safety and liveness properties confirms the correctness of our specification under the defined constraints.

5.1.4 Implications and Confidence

TLC guarantees correctness within the explored bounds. While we did not verify the specification exhaustively for all possible system sizes, the tested models exercise all relevant interaction patterns. The fundamental structure of concurrency does not change with scale.

Although Model 2 could not be fully explored due to memory constraints, Model 3 captures the same interaction patterns without transitive concurrency. Together with the probabilistic testing

results discussed in Section 5.2, this provides additional confidence that the unobserved behaviors do not introduce correctness violations.

Combined with the careful system design, these results give us high confidence that the TLA+ specification correctly represents the intended behavior of the SplitLess application. Assuming that the checked models are representative of larger and more complex scenarios, we expect the design to generalize beyond the verified instances.

5.1.5 Limitations

TLC can only verify a specification against explicitly defined correctness properties. For TLC to be effective, these properties must accurately restrict all possible state sequences to the intended range of behaviors. We carefully designed the properties to characterize legal states and transitions. While we are confident that the state space is constrained in line with the application's goals, it is always possible that a relevant property was overlooked.

Although the evaluated models cover the key interaction patterns within a bounded exploration depth, longer action sequences may exist that are not exercised within these bounds. Due to state space explosion, larger models and deeper explorations are infeasible on our hardware. As a result, TLC cannot provide absolute guarantees beyond the verified bounds, but it does offer strong evidence of correctness within them.

5.2 Probabilistic Testing of the Python CLI

With high confidence in the correctness of the TLA+ specification, we now need a way to verify that the Python CLI implements it correctly. To do so, we test the same correctness properties on concrete states generated by sequences of actions executed through the CLI. In this section, we describe our test setup and present the resulting outcomes.

5.2.1 General Concept

At a high level, we test the implementation by repeatedly selecting random actions with random parameters and validating the resulting state against the correctness properties defined in TLA+. If this process runs for a sufficiently long time without finding violations, we gain confidence in the correctness of the implementation. We do not exhaustively explore all possible states, as the state space is orders of magnitude larger than what our hardware could handle. However, by randomly sampling actions and resulting states, we can still cover a representative portion of the state space. This approach is inspired by the testing methodology used for the Pando personal volunteer computing tool in Erick Lavoie's PhD thesis [6].

5.2.2 Test Setup

We first define a number of test runs and a maximum number of steps per run. Each step represents a randomly chosen CLI action that may modify the replica state. Every test run starts from a fresh initial state and uses a new random seed. In each run, multiple replicas and users are involved.

After each step, we check all correctness properties. If a violation is detected, we stop the execution and report the error, the relevant replica states, and the test run seed. Using the seed, we can reproduce the failing execution and verify that subsequent bug fixes resolve the issue.

For testing, we also implemented a fully in-memory version of replica storage, as persistence is not required in this context. Using memory instead of disk I/O significantly speeds up test execution. While the latency of individual actions is negligible for CLI users, testing involves many more actions and therefore benefits substantially from this optimization.

5.2.3 Action Execution

For each step, a random action together with random parameters is selected. However, many actions require several parameters that must be consistent for the action to be enabled. For example, to add an expense to a group, the payer of the chosen expense must match the selected actor, and all users with positive shares must be members of the chosen group. As a result, such actions are unlikely to be enabled and to modify the state. In our experiments, the success rate for the `AddExpenseToGroup` action was around 0.4%.

To address this and improve test efficiency, we introduce *smart actions*. A smart action selectively chooses parameters that satisfy the action's preconditions and therefore allow it to succeed. We define smart variants for all actions with complex parameter constraints that would otherwise rarely pass. To still test precondition checking, we randomly choose between executing a smart or a fully random version of an action. We achieved good results with a mix of 80% smart and 20% random executions. Using smart actions significantly increases success rates and improves overall test efficiency.

We also do not assign equal probabilities to all actions. Some actions occur more frequently than others in realistic usage scenarios. For example, once groups are established, users typically perform mostly expense-related operations. We therefore structure each test run into three phases. In the first, very short phase, we prioritize group creation by assigning higher probabilities to the corresponding actions. The second, also short phase focuses on group membership, giving invitation and acceptance actions higher probability. In the final, longer phase, expense-related actions receive the highest probability, reflecting typical real-world usage.

We experimented with different probability distributions until we achieved a good balance between group and expense creation and modification, with acceptable success rates for most actions, typically around 20%.

5.2.4 Correctness Property Validation

Similar to TLC checking, after each step, that is, after each executed action, we verify all invariants defined in the TLA+ specification to ensure structural consistency of the replica states. In addition, we keep a copy of the previous state to check the remaining safety properties, such as monotonicity of counters.

If a violation occurs, we immediately stop the test execution and record the error, the relevant replica states, and the random seed required for reproduction.

After all steps of a single test run have been executed, we merge all replicas and verify that they contain the same and most recent state. This mirrors the liveness checking performed in the TLA+ specification. We also attempt actions that allow users to acknowledge pending shares, in order to validate the liveness property that all non-deleted expenses assigned to a group can eventually become valid. This confirms that our progress assumptions hold in practice.

5.2.5 Test Scale and Results

After completing smaller-scale tests and fixing all detected bugs, we performed a final large-scale test to gain high confidence in system correctness.

In total, we executed 80,000 test runs with 1,000 steps each. We tested with five users, ensuring sufficient concurrent behavior. We chose 80,000 runs to align with 10,000 runs per core on the available eight-core machine. This configuration provides a reasonable number of test runs to build confidence in correctness without requiring excessive execution time. The aggregated test output, including the number of expenses and groups created as well as the action pass rates, is provided in Appendix C.

No errors were reported after the tests completed. This result gives us high confidence that the CLI implementation correctly adheres to the TLA+ specification and that the additional implementation complexity does not introduce bugs in the final version.

5.2.6 Bugs Found During Testing

Probabilistic testing proved essential for uncovering hidden bugs. We identified several small issues in the implementation that would most likely not have been detected through manual testing. All bugs discovered through probabilistic testing were corner cases that rarely occur during normal usage. For example, in the unlikely case where a user had a share of zero in an expense, the implementation did not always reset the acknowledgment of this share after the expense was removed from a group, resulting in inconsistent states. Another noteworthy issue was related to identifier generation. Initially, we used identifiers of eight characters to keep replica states readable during development. In one test run, two expenses were created with the same random identifier, leading to a version decrease and a safety property violation. To address this, we now use identifiers of 32 characters.

5.2.7 Limitations and Discussion

As mentioned earlier, our approach does not exhaustively validate every possible state. This would be infeasible, as the state space is very large, and since each expense share can take any `float` value, it is effectively infinite. As a result, it is always possible that some relevant state or action sequences were not exercised during testing and that undiscovered errors remain. However, given the large number of test runs performed, we consider this unlikely and have high confidence in the implementation.

State coverage largely depends on randomness. We improve coverage and efficiency by introducing smart actions that select parameters likely to enable an action, and by tuning the probabilities of actions to better reflect realistic usage patterns.

Finally, our testing validates the alignment between the TLA+ model and the Python implementation. Since we observed no violations of the TLA+ correctness properties, we assume that the CLI is a faithful and valid implementation of the specification.

5.3 Performance Characteristics of the CLI Implementation

In addition to correctness, we performed a lightweight evaluation of the performance characteristics of the Python CLI implementation. The goal was not to benchmark the system in depth, but to verify that latency and resource usage are acceptable for the intended use case.

We reused the probabilistic testing framework to measure action latency, with correctness checks disabled and the persistent JSON-based storage backend enabled. Using smart actions reflects typical user behavior, as most actions in practice satisfy their preconditions.

For 1,000 actions with five and ten users, the average latency per action was between 8 and 10 milliseconds. This latency is negligible for interactive use. For comparison, the in-memory backend achieved an average latency of approximately 0.2 milliseconds per action, highlighting the overhead of disk I/O.

Storage requirements are low and grow linearly with the number of groups and expenses. An expense with two users occupies approximately 400 bytes, while an expense with ten users requires around 825 bytes. A group with ten users typically occupies between 700 and 800 bytes, depending on group history. These values are well within reasonable limits for small groups.

Because all states are stored as JSON and accessed via dictionary lookups, performance will eventually degrade as the number of records grows. While this does not affect the intended scale of use, a more efficient storage backend would be required for larger deployments.

6 Related Work

Expense sharing is a well-established application domain. Existing solutions emphasize ease of use and user experience. Most popular applications rely on centralized architectures to manage transaction data. This approach is less complex and easier to scale than the decentralized methods used in SplitLess. One widely used centralized application is Splitwise [1]. We use Splitwise ourselves, and the idea for this project was inspired by this application.

In this chapter, we discuss Splitwise as a representative example of centralized expense sharing applications and position SplitLess as a decentralized alternative. We also relate our work to decentralized data management approaches for financial applications, including CRDT-based systems and blockchain-based designs. These systems address similar challenges related to consistency and concurrency, but they make different trade-offs with respect to coordination.

We already discuss the basics of replicated data types and TLA+ in Chapter 2. Examining these approaches in full detail would be beyond the scope of this report. Instead, we focus on a high-level comparison that helps position SplitLess within the broader landscape of existing work.

6.1 Splitwise [1]

In February 2011, Splitwise was launched as SplitTheRent, intended for rent splitting, by Ryan Laughlin, Jon Bittner, and Marshall Weir. In 2013, it was integrated with Venmo to allow users to settle payments through the platform. Since 2024, it has supported bank transfers directly within the app through a partnership with a Visa payment service provider. Today, the application is widely used for various expense sharing scenarios, ranging from shared housing to trips with friends.

The core functionality of Splitwise is similar to that of SplitLess. Users can create groups and track shared expenses. The application simplifies transaction histories and recommends exact payments to settle outstanding balances using a minimal number of transactions. We did not implement this feature in SplitLess during the project and instead focused on correct, decentralized, and concurrent expense tracking. However, the CLI of SplitLess can recommend a user to pay the next expense to equalize balances.

Splitwise stores all user and transaction data on a central server. Each request that modifies the state of a group is authoritatively handled by this server. Compared to our approach, this greatly simplifies concurrency management, as the server always maintains an up-to-date view of the system, and users must synchronize with it for every change. Users are not allowed to perform local modifications without contacting the server. As a result, users must trust the server to handle transaction data correctly. They must also trust the organization to respect data privacy.

SplitLess aims to provide similar core functionality by tracking shared expenses among groups of users. However, it removes the reliance on a central authority and instead focuses on correctness under decentralization and concurrency.

6.2 CRDT-based Financial Ledgers

Beyond application-specific expense sharing tools like Splitwise, there are also more fundamental studies on distributed ledgers and replication mechanisms for financial data. Some research has explored the use of CRDTs and related conflict-free replication mechanisms for financial ledger applications. One recent example is the GOC-Ledger, a state-based replicated ledger built from grow-only counters [7]. Instead of using consensus protocols that totally order all updates, GOC-Ledger models each account's state as a composition of monotonically growing counters that, when merged, converge to a consistent view of balances across replicas. The use of grow-only counters means that the internal state only increases; derived values such as account balances are computed from combinations of counters (e.g. created, transferred, burned) rather than directly stored. This design allows the system to achieve eventual convergence under weaker delivery assumptions than reliable broadcast and to avoid reasoning over operation histories for safety.

The GOC-Ledger work demonstrates that replicated ledgers do not necessarily require consensus and that simple monotonic CRDT primitives can serve as the foundation for replicated balance tracking. However, its guarantees differ from those required by an expense sharing application. In particular, GOC-Ledger does not prevent double spending at the level of individual operations. Conflicting transfers may temporarily lead to inconsistent derived balances, with violations only being detected after replicas are merged. This behavior is acceptable in the context of a low-level ledger abstraction, but it shifts responsibility for handling such anomalies to higher layers.

Another important difference lies in how debts are represented and settled. GOC-Ledger models value as transferable tokens: a user may transfer a token representing an obligation to another party, who can then further transfer or eventually redeem it. This provides a flexible mechanism for modeling payments and delegation of claims. In contrast, SplitLess follows a more restrictive but user-oriented model. Debts are settled by creating new expenses that explicitly reference the original payer, and obligations cannot be freely transferred between users. This reduces flexibility but aligns more closely with common real-world expense sharing practices.

Finally, SplitLess requires additional synchronization beyond pure CRDT merging to ensure application level correctness. Mechanisms such as share acknowledgments and explicit invitation acceptance introduce coordination points that are largely absent in GOC-Ledger. While this means SplitLess is not purely a CRDT at the state level, it allows the system to enforce stronger guarantees about when balances are finalized and visible to users. In this sense, SplitLess operates at a higher semantic level, trading some of the generality of ledger-style transfers for clearer and more predictable user-facing behavior.

6.3 Blockchain-based Approaches

Another line of work related to decentralized financial applications is based on blockchain and distributed ledger technologies. Blockchain systems maintain a globally replicated transaction log that is updated through consensus protocols. This approach was popularized by Bitcoin [9] and has since been widely explored for financial applications due to its strong guarantees regarding immutability, auditability and resistance to tampering.

In contrast to CRDT-based approaches, blockchains rely on global consensus and a total ordering of all state changes. This provides strong consistency guarantees at the system level, but it also introduces significant latency and computational overhead. For applications like expense sharing,

where interactions are typically limited to small groups of users, such global guarantees are often stronger than necessary. In particular, SplitLess only requires consistency within individual groups, not agreement on a single global history of all expenses.

SplitLess therefore avoids global consensus and instead relies on local actions, replica merging and explicitly stated correctness properties. This makes the system lighter-weight and better suited for collaborative expense tracking among small, trusted groups. At the same time, the design does not preclude the use of blockchain infrastructure altogether. In principle, SplitLess could be implemented on top of an existing blockchain platform, such as Ethereum [2], using the blockchain as a communication and persistence layer.

Taken together, these approaches form a spectrum. GOC-Ledger explores minimal synchronization using monotonic CRDTs for replicated balances, while blockchain-based systems provide strong global consistency through consensus. SplitLess occupies a middle ground: it avoids global ordering but introduces limited application-level synchronization to enforce stronger user-facing correctness guarantees than a pure ledger abstraction.

7 Conclusions

The goal of this project was to design and implement a peer-to-peer expense sharing system, with a focus on correctness under concurrency rather than centralized coordination. We aimed to support editable expenses and dynamic group membership. In this chapter, we summarize our achievements, reflect on the practical applicability of peer-to-peer expense sharing applications, and outline possible directions for future work.

We designed a complete formal specification of a decentralized expense sharing system in TLA+. The specification captures dynamic group membership, editable expenses, and concurrent updates across replicas. We explicitly modeled concurrency challenges instead of treating them as implementation artifacts. Potential conflicts are resolved through explicit restrictions on which users are permitted to modify expenses and group assignments. In addition to modification rights, the model captures gifted balances and requires share acknowledgments for expenses to guarantee correctness under concurrency. We verified the specification against key correctness properties aligned with the application requirements using the TLC model checker. TLC exhaustively explores all possible behaviors of a finite model starting from an initial state. No violations were found, giving us strong confidence in the correctness of the specification.

Furthermore, we implemented a concrete Python command-line interface that closely mirrors the TLA+ specification, where state-changing actions are direct translations of the corresponding TLA+ actions. To bridge the abstraction gap of the specification, we introduced persistent storage, communication, and dynamic identifier generation. Nevertheless, using TLA+ as a foundation significantly reduced implementation complexity. The core logic was already well defined, without concern for concrete system details such as communication, and many subtle concurrency bugs were identified and prevented early. We complemented this with probabilistic testing of the Python implementation, mirroring key aspects of the TLC model checker. Together, these approaches give us strong confidence that the design generalizes beyond the bounded models checked by TLC and behaves correctly under realistic usage patterns.

Our findings suggest that a fully decentralized expense sharing application is technically feasible. The model shows that, with certain restrictions, strong consistency guarantees are possible even without global coordination. However, the current system remains a proof of concept. Throughout this project, we assumed users to be a trusted group of friends and did not implement or design security mechanisms. While these assumptions simplify correctness, they limit applicability in real-world environments. For practical use, update availability must also be considered. Users should be online frequently enough to avoid long propagation delays, as late updates may affect user experience even though correctness is preserved. Unlike a centralized application, our approach requires at least one other known and online user to fetch updates. To eventually obtain the most recent state, a user must receive updates from all other users.

There are several avenues for extending this work. As noted earlier, security and trust are not addressed. Future work could introduce authentication, update signatures, and encrypted commu-

nication and storage. User experience could be significantly improved by implementing a mobile application on top of the existing design, providing intuitive group and expense management, as well as automatic synchronization. Functionality could also be extended by storing richer metadata, such as timestamps and descriptions. Finally, further formal analysis is possible. Larger or more diverse TLC models could be checked on more powerful hardware, or formal proofs could be derived to establish the correctness of individual actions.

This project combined formal methods for designing distributed and concurrent systems with practical system design. TLA+ enabled early reasoning about concurrency and helped prevent design errors before implementation. The project demonstrates that formal methods are not merely theoretical, but can meaningfully inform and enhance real-world distributed applications.

Bibliography

[1] Splitwise. Available at: https://www.splitwise.com (accessed: 2026-01-07).

[2] Vitalik Buterin et al. Ethereum whitepaper, 2013. Available at: https://ethereum.org/en/whitepaper/ (accessed: 2026-02-09).

[3] Martin Kleppmann and Heidi Howard. Byzantine eventual consistency and the fundamental limits of peer-to-peer databases. *arXiv preprint arXiv:2012.00472*, 2020. Available at: https://doi.org/10.48550/arXiv.2012.00472 (accessed: 2026-02-09).

[4] Leslie Lamport. *Specifying systems*, volume 388. Addison-Wesley Boston, 2002. ISBN: 9780321143068. Available at: https://lamport.azurewebsites.net/tla/book-02-08-08.pdf (accessed: 2026-02-09).

[5] Leslie Lamport. The TLA+ toolbox. 2021. Available at: https://lamport.azurewebsites.net/tla/toolbox.html (accessed: 2025-12-30).

[6] Erick Lavoie. *Personal Volunteer Computing*. PhD thesis, McGill University, Montreal, Canada, December 2019. Advisor: Prof. Laurie Hendren. Available via Internet Archive Wayback Machine at: https://web.archive.org/web/20220108160000/http://ericklavoie.com/phd/dissertation.pdf (accessed: 2026-02-09).

[7] Erick Lavoie. Goc-ledger: State-based conflict-free replicated ledger from grow-only counters. *arXiv preprint arXiv:2305.16976*, 2023. Available at: https://doi.org/10.48550/arXiv.2305.16976 (accessed: 2026-02-09).

[8] Erick Lavoie. State-based ∞ p-set conflict-free replicated data type. *arXiv preprint arXiv:2304.01929*, 2023. Available at: https://doi.org/10.48550/arXiv.2304.01929 (accessed: 2026-02-09).

[9] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. Available at: https://bitcoin.org/bitcoin.pdf (accessed: 2026-02-09).

[10] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011. Available at: https://doi.org/10.1007/978-3-642-24550-3_29 (accessed: 2026-02-09).

A TLA+ Specification

This appendix contains the full TLA+ specification of the SplitLess system, as referenced throughout Chapter 3. The specification does not model the group invitation logic. Instead, members can directly add other users to a group. This choice reduces overhead for TLC model checking, as one fewer step is required to include a new user in a group, namely adding the user directly instead of first sending an invitation and then requiring its acceptance. As a result, we can reduce the exploration depth by at least one while still exploring the same relevant states, which allows us to verify larger models. The invitation logic is modeled in a separate TLA+ specification in Appendix B.

```
1  ---------- MODULE SplitLess_replica_group_expenses ----------
2  EXTENDS Naturals, Sequences, FiniteSets

4  CONSTANTS
5      USERS,
6      POSSIBLE_SHARES,
7      POSSIBLE_EXPENSE_IDs,
8      POSSIBLE_GROUP_IDs,
9      NO_EXPENSE,
10     NO_GROUP,
11     POSSIBLE_REPLICA_IDs,
12     ASSIGNED_REPLICA

14 VARIABLES replicas, actionCounter

16 ----------------------------
17 Records
18 ----------------------------

20 Expense ≜
21         [id : POSSIBLE_EXPENSE_IDs,
22          group : POSSIBLE_GROUP_IDs ∪ {NO_GROUP},
23          version : Nat,    grow only counter to track current state of expense
24          payer : USERS,
25          amount : Nat,
26          Shares can be other than POSSIBLE_SHARES as paye can absorb shares
27           shares : [USERS → Nat],
28           acknowledged_shares : [USERS → BOOLEAN ],
29           deleted : BOOLEAN ]

31 Group ≜
32         [id : POSSIBLE_GROUP_IDs,
33          members : [USERS → Nat],   Causal length counter for each user
34          totalGifted : Nat,
35          individualGiftsSent : [USERS → Nat]]

37 Replica ≜
38        [id : POSSIBLE_REPLICA_IDs,
39         recordedExpenses : [POSSIBLE_EXPENSE_IDs → (Expense ∪ {NO_EXPENSE})],
40         groups : [POSSIBLE_GROUP_IDs → (Group ∪ {NO_GROUP})]
41        ]

43 ----------------------------
44 Initialization
45 ----------------------------
46 Init ≜
47      ∧ replicas =
48          [rid ∈ POSSIBLE_REPLICA_IDs ↦
49             [id ↦ rid,
50              recordedExpenses ↦ [eid ∈ POSSIBLE_EXPENSE_IDs ↦ NO_EXPENSE],
51              groups ↦ [gid ∈ POSSIBLE_GROUP_IDs ↦ NO_GROUP]
52             ]
53          ]
54      ∧ actionCounter = 0
```

```
56  ------------------------------
57  Helper Functions
58  ------------------------------

60  Get expenseIds that are added to a specific group
61  GroupExpenseIds(gid, recordedExpensesIn) ≜
62    {eid ∈ POSSIBLE_EXPENSE_IDs :
63        ∧ recordedExpensesIn[eid] ≠ NO_EXPENSE
64        ∧ recordedExpensesIn[eid].deleted = FALSE
65        ∧ recordedExpensesIn[eid].group = gid
66        ∧ ∀ u ∈ DOMAIN recordedExpensesIn[eid].shares :
67            (recordedExpensesIn[eid].shares[u] > 0)
68              ⇒ recordedExpensesIn[eid].acknowledged_shares[u] = TRUE}

70  IsMember(memberCounter) ≜
71    memberCounter%2 = 1

73  WasEverMember(memberCounter) ≜
74    memberCounter > 0

76  RECURSIVE SumFunction(_)
77  SumFunction(F) ≜
78        IF DOMAIN F = {} THEN 0
79         ELSE LET d ≜ CHOOSE x ∈ DOMAIN F : TRUE
80              IN   F[d] + SumFunction([y ∈ DOMAIN F \ {d} ↦ F[y]])

83  Balance(u, gid, replica) ≜
84          LET groupExpenses ≜ GroupExpenseIds(gid, replica.recordedExpenses)
85          IN   SumFunction([eid ∈ groupExpenses ↦
86                                IF replica.recordedExpenses[eid].payer = u
87                                 THEN replica.recordedExpenses[eid].amount ELSE 0])
88               − SumFunction([eid ∈ groupExpenses ↦ replica.recordedExpenses[eid].shares[u]])
89               − replica.groups[gid].individualGiftsSent[u]

92  ComputeBalances(grp, recordedExpensesIn) ≜
93        [u ∈ USERS ↦
94          LET groupExpenses ≜ GroupExpenseIds(grp.id, recordedExpensesIn)
95          IN   SumFunction([eid ∈ groupExpenses ↦
96                                IF recordedExpensesIn[eid].payer = u
97                                 THEN recordedExpensesIn[eid].amount ELSE 0])
98               − SumFunction([eid ∈ groupExpenses ↦ recordedExpensesIn[eid].shares[u]])
99        ]

102 ComputeGifts(grp, balances) ≜
103       LET giftingUsers ≜
104             {u ∈ USERS : ¬IsMember(grp.members[u]) ∧ balances[u] > 0}
105           newTotalGifted ≜ SumFunction([u ∈ giftingUsers ↦ balances[u]])
106           newIndividualGifts ≜ [u ∈ USERS ↦ IF u ∈ giftingUsers THEN balances[u] ELSE 0]
107       IN  [grp EXCEPT !.totalGifted = newTotalGifted,
108                       !.individualGiftsSent = newIndividualGifts]

111 RecalcGifts(groupsIn, recordedExpensesIn) ≜
```

```
        [gid ∈ POSSIBLE_GROUP_IDs ↦
          IF groupsIn[gid] = NO_GROUP THEN NO_GROUP
           ELSE LET grp ≜ groupsIn[gid]
                    balances ≜ ComputeBalances(grp, recordedExpensesIn)
                IN  ComputeGifts(grp, balances)
        ]

    ----------------------------
    Group actions
    ----------------------------

CreateGroup ≜
    ∃ actor ∈ USERS :
    ∃ gid ∈ POSSIBLE_GROUP_IDs :
    ∃ rid ∈ POSSIBLE_REPLICA_IDs :
      ∧ ASSIGNED_REPLICA[actor] = rid
        Ensure each gid is only used once across replicas
      ∧ ∀ otherRid ∈ POSSIBLE_REPLICA_IDs : replicas[otherRid].groups[gid] = NO_GROUP
      ∧ LET newGroup ≜
            [id ↦ gid,
             members ↦ [u ∈ USERS ↦ IF u = actor THEN 1 ELSE 0],
             totalGifted ↦ 0,
             individualGiftsSent ↦ [u ∈ USERS ↦ 0]]
          newReplica ≜
            [replicas[rid] EXCEPT !.groups = [@ EXCEPT ![gid] = newGroup]]
        IN  ∧ replicas' = [replicas EXCEPT ![rid] = newReplica]
            ∧ actionCounter' = actionCounter + 1

AddMember ≜
    ∃ actor, newMember ∈ USERS    :
    ∃ gid ∈ POSSIBLE_GROUP_IDs :
    ∃ rid ∈ POSSIBLE_REPLICA_IDs :
      ∧ ASSIGNED_REPLICA[actor] = rid
      ∧ replicas[rid].groups[gid] ≠ NO_GROUP
      ∧ IsMember(replicas[rid].groups[gid].members[actor])
      ∧ ¬IsMember(replicas[rid].groups[gid].members[newMember])
      ∧ LET newReplica ≜
            [replicas[rid] EXCEPT !.groups =
              [@ EXCEPT ![gid].members[newMember] = @ + 1]]
        IN  ∧ replicas' = [replicas EXCEPT ![rid] = newReplica]
            ∧ actionCounter' = actionCounter + 1

LeaveGroup ≜
    ∃ actor ∈ USERS :
    ∃ gid ∈ POSSIBLE_GROUP_IDs :
    ∃ rid ∈ POSSIBLE_REPLICA_IDs :
      ∧ ASSIGNED_REPLICA[actor] = rid
      ∧ replicas[rid].groups[gid] ≠ NO_GROUP
      ∧ IsMember(replicas[rid].groups[gid].members[actor])
      ∧ Balance(actor, gid, replicas[rid]) ≥ 0
      ∧ LET updatedGroups ≜
            [replicas[rid].groups EXCEPT
              ![gid].members[actor] = @ + 1]
```

```
166             newGroups ≜ RecalcGifts(updatedGroups, replicas[rid].recordedExpenses)
167             newReplica ≜
168               [replicas[rid] EXCEPT !.groups = newGroups]
169         IN    ∧ replicas' = [replicas EXCEPT ![rid] = newReplica]
170               ∧ actionCounter' = actionCounter + 1

173 ----------------------------
174 Expense actions
175 ----------------------------

177 CreateExpense ≜
178     ∃ actor ∈ USERS :
179     ∃ shares ∈ POSSIBLE_SHARES :
180     ∃ eid ∈ POSSIBLE_EXPENSE_IDs :
181     ∃ rid ∈ POSSIBLE_REPLICA_IDs :
182       ∧ ASSIGNED_REPLICA[actor] = rid
183       ∧ SumFunction(shares) > 0
184       ∧ ∀ otherRid ∈ POSSIBLE_REPLICA_IDs : replicas[otherRid].recordedExpenses[eid] = NO_EXPENSE
185       ∧ LET newExpense ≜
186             [id ↦ eid,
187              group ↦ NO_GROUP,
188              version ↦ 0,
189              payer ↦ actor,
190              amount ↦ SumFunction(shares),
191              shares ↦ shares,
192              acknowledged_shares ↦ [u ∈ USERS ↦ IF u = actor ∧ shares[u] > 0 THEN TRUE ELSE FALSE],
193              deleted ↦ FALSE]
194           newReplica ≜
195             [replicas[rid] EXCEPT !.recordedExpenses = [@ EXCEPT ![eid] = newExpense]]
196         IN    ∧ replicas' = [replicas EXCEPT ![rid] = newReplica]
197               ∧ actionCounter' = actionCounter + 1

199 AddExpenseToGroup ≜
200     ∃ actor ∈ USERS :
201     ∃ eid ∈ POSSIBLE_EXPENSE_IDs :
202     ∃ gid ∈ POSSIBLE_GROUP_IDs :
203     ∃ rid ∈ POSSIBLE_REPLICA_IDs :
204       ∧ ASSIGNED_REPLICA[actor] = rid
205       ∧ replicas[rid].groups[gid] ≠ NO_GROUP
206       ∧ replicas[rid].recordedExpenses[eid] ≠ NO_EXPENSE
207       ∧ IsMember(replicas[rid].groups[gid].members[actor])
208       ∧ replicas[rid].recordedExpenses[eid].payer = actor
209       ∧ replicas[rid].recordedExpenses[eid].group = NO_GROUP
210       ∧ {u ∈ USERS : replicas[rid].recordedExpenses[eid].shares[u] > 0}
211             ⊆ {u ∈ USERS : IsMember(replicas[rid].groups[gid].members[u])}
212       ∧ LET newExpense ≜
213             [replicas[rid].recordedExpenses[eid] EXCEPT !.group = gid, !.version = @ + 1]
214           newExpenses ≜
215             [replicas[rid].recordedExpenses EXCEPT ![eid] = newExpense]
216           newReplica ≜
217             [replicas[rid] EXCEPT !.recordedExpenses = newExpenses]
218         IN    ∧ replicas' = [replicas EXCEPT ![rid] = newReplica]
219               ∧ actionCounter' = actionCounter + 1
```

```
RemoveExpenseFromGroup ≜
    ∃ actor ∈ USERS :
    ∃ eid ∈ POSSIBLE_EXPENSE_IDs :
    ∃ gid ∈ POSSIBLE_GROUP_IDs :
    ∃ rid ∈ POSSIBLE_REPLICA_IDs :
      ∧ ASSIGNED_REPLICA[actor] = rid
      ∧ replicas[rid].groups[gid] ≠ NO_GROUP
      ∧ replicas[rid].recordedExpenses[eid] ≠ NO_EXPENSE
      ∧ IsMember(replicas[rid].groups[gid].members[actor])
      ∧ replicas[rid].recordedExpenses[eid].group = gid
      ∧ replicas[rid].recordedExpenses[eid].payer = actor
      ∧ LET newExpense ≜ [replicas[rid].recordedExpenses[eid] EXCEPT !.group = NO_GROUP,
                                                                     !.version = @ + 1,
                                                                     !.acknowledged_shares =
                                                                     [u ∈ USERS ↦ IF u  = actor ∧ replicas[rid].recordedExpenses[eid].shares[u] > 0
                                                                       THEN TRUE ELSE FALSE]]
            newExpenses ≜ [replicas[rid].recordedExpenses EXCEPT ![eid] = newExpense]
            newGroups ≜ RecalcGifts(replicas[rid].groups, newExpenses)
            newReplica ≜ [replicas[rid] EXCEPT !.recordedExpenses = newExpenses,
                                               !.groups = newGroups]
        IN  ∧ replicas′ = [replicas EXCEPT ![rid] = newReplica]
        ∧ actionCounter′ = actionCounter + 1

ModifyExpenseParameters ≜
    ∃ actor ∈ USERS :
    ∃ shares ∈ POSSIBLE_SHARES :
    ∃ eid ∈ POSSIBLE_EXPENSE_IDs :
    ∃ rid ∈ POSSIBLE_REPLICA_IDs :
      ∧ ASSIGNED_REPLICA[actor] = rid
      ∧ replicas[rid].recordedExpenses[eid] ≠ NO_EXPENSE
      ∧ replicas[rid].recordedExpenses[eid].payer = actor
      ∧ SumFunction(shares) > 0
      ∧ ¬replicas[rid].recordedExpenses[eid].deleted
      ∧ IF replicas[rid].recordedExpenses[eid].group ≠ NO_GROUP
        THEN {u ∈ USERS : shares[u] > 0}
             ⊆ {u ∈ USERS : IsMember(replicas[rid].groups[replicas[rid].recordedExpenses[eid].group].members[u])}
        ELSE TRUE
      ∧ LET newExpenses ≜
              [replicas[rid].recordedExpenses EXCEPT
                ![eid].shares = shares,
                ![eid].acknowledged_shares = [u ∈ USERS ↦ IF u = actor ∧ shares[u] > 0 THEN TRUE ELSE FALSE],
                ![eid].amount = SumFunction(shares),
                ![eid].version = @ + 1]
            newGroups ≜
              IF replicas[rid].recordedExpenses[eid].group = NO_GROUP
              THEN replicas[rid].groups
              ELSE RecalcGifts(replicas[rid].groups, newExpenses)
            newReplica ≜
              [replicas[rid] EXCEPT !.recordedExpenses = newExpenses,
                                    !.groups = newGroups]
        IN  ∧ replicas′ = [replicas EXCEPT ![rid] = newReplica]
            ∧ actionCounter′ = actionCounter + 1
```

```
DeleteExpense ≜
    ∃ actor ∈ USERS :
    ∃ eid ∈ POSSIBLE_EXPENSE_IDs :
    ∃ rid ∈ POSSIBLE_REPLICA_IDs :
      ∧ ASSIGNED_REPLICA[actor] = rid
      ∧ replicas[rid].recordedExpenses[eid] ≠ NO_EXPENSE
      ∧ replicas[rid].recordedExpenses[eid].payer = actor
      ∧ replicas[rid].recordedExpenses[eid].deleted = FALSE
      ∧ IF replicas[rid].recordedExpenses[eid].group ≠ NO_GROUP
        THEN ∧ IsMember(replicas[rid].groups[replicas[rid].recordedExpenses[eid].group].members[actor])
        ELSE TRUE
      ∧ LET newExpenses ≜
              [replicas[rid].recordedExpenses EXCEPT ![eid].deleted = TRUE, ![eid].version = @ + 1]
            newGroups ≜
              RecalcGifts(replicas[rid].groups, newExpenses)
            newReplica ≜
              [replicas[rid] EXCEPT !.recordedExpenses = newExpenses,
                                    !.groups = newGroups]
        IN  ∧ replicas' = [replicas EXCEPT ![rid] = newReplica]
            ∧ actionCounter' = actionCounter + 1

AcknowledgeShare ≜
    ∃ actor ∈ USERS :
    ∃ eid ∈ POSSIBLE_EXPENSE_IDs :
    ∃ rid ∈ POSSIBLE_REPLICA_IDs :
      ∧ ASSIGNED_REPLICA[actor] = rid
      ∧ replicas[rid].recordedExpenses[eid] ≠ NO_EXPENSE
      ∧ replicas[rid].recordedExpenses[eid].deleted = FALSE
      ∧ replicas[rid].recordedExpenses[eid].group ≠ NO_GROUP
      ∧ IsMember(replicas[rid].groups[replicas[rid].recordedExpenses[eid].group].members[actor])
      ∧ replicas[rid].recordedExpenses[eid].shares[actor] > 0
      ∧ replicas[rid].recordedExpenses[eid].acknowledged_shares[actor] = FALSE
      ∧ LET newExpenses ≜
              [replicas[rid].recordedExpenses EXCEPT ![eid].acknowledged_shares[actor] = TRUE]
            newGroups ≜
              RecalcGifts(replicas[rid].groups, newExpenses)
            newReplica ≜
              [replicas[rid] EXCEPT !.recordedExpenses = newExpenses,
                                    !.groups = newGroups]
        IN  ∧ replicas' = [replicas EXCEPT ![rid] = newReplica]
            ∧ UNCHANGED actionCounter


 Payer absorbs share of a member who left and never acknowledged
PayerAbsorbsLeftMemberShare ≜
    ∃ actor ∈ USERS :
    ∃ eid ∈ POSSIBLE_EXPENSE_IDs :
    ∃ leftMember ∈ USERS :
    ∃ rid ∈ POSSIBLE_REPLICA_IDs :
      ∧ ASSIGNED_REPLICA[actor] = rid
      ∧ actor ≠ leftMember
      ∧ replicas[rid].recordedExpenses[eid] ≠ NO_EXPENSE
      ∧ replicas[rid].recordedExpenses[eid].deleted = FALSE
```

```
        ∧ replicas[rid].recordedExpenses[eid].payer = actor
        ∧ replicas[rid].recordedExpenses[eid].shares[leftMember] > 0
        ∧ replicas[rid].recordedExpenses[eid].acknowledged_shares[leftMember] = FALSE
        ∧ replicas[rid].recordedExpenses[eid].group ≠ NO_GROUP
        ∧ LET gid ≜ replicas[rid].recordedExpenses[eid].group
          IN  ∧ replicas[rid].groups[gid] ≠ NO_GROUP
              ∧ IsMember(replicas[rid].groups[gid].members[actor])
              ∧ ¬IsMember(replicas[rid].groups[gid].members[leftMember])
              ∧ WasEverMember(replicas[rid].groups[gid].members[leftMember])
              ∧ LET oldShares ≜ replicas[rid].recordedExpenses[eid].shares
                    leftShare ≜ oldShares[leftMember]
                    newShares ≜ [u ∈ USERS ↦
                        IF u = actor THEN oldShares[u] + leftShare
                         ELSE IF u = leftMember THEN 0
                         ELSE oldShares[u]]
                    newExpenses ≜
                      [replicas[rid].recordedExpenses EXCEPT
                        ![eid].shares = newShares,
                        ![eid].version = @ + 1]
                    newGroups ≜ RecalcGifts(replicas[rid].groups, newExpenses)
                    newReplica ≜
                      [replicas[rid] EXCEPT !.recordedExpenses = newExpenses,
                                            !.groups = newGroups]
                IN  ∧ replicas′ = [replicas EXCEPT ![rid] = newReplica]
                    ∧ UNCHANGED actionCounter

    Alternative conflict resolution: member rejoins to acknowledge
    Follows the add member protocoll by having a group member reentering the rejoining one
RejoinToAcknowledge ≜
    ∃ inviter, rejoiner ∈ USERS :
    ∃ gid ∈ POSSIBLE_GROUP_IDs :
    ∃ rid ∈ POSSIBLE_REPLICA_IDs :
        ∧ ASSIGNED_REPLICA[inviter] = rid
        ∧ replicas[rid].groups[gid]          ≠ NO_GROUP
        ∧ IsMember(replicas[rid].groups[gid].members[inviter])
        ∧ ¬IsMember(replicas[rid].groups[gid].members[rejoiner])
        ∧ WasEverMember(replicas[rid].groups[gid].members[rejoiner])
        ∧ ∃ eid ∈ POSSIBLE_EXPENSE_IDs :
            ∧ replicas[rid].recordedExpenses[eid] ≠ NO_EXPENSE
            ∧ replicas[rid].recordedExpenses[eid].group = gid
            ∧ replicas[rid].recordedExpenses[eid].shares[rejoiner] > 0
            ∧ replicas[rid].recordedExpenses[eid].acknowledged_shares[rejoiner] = FALSE
        ∧ LET newReplica ≜
              [replicas[rid] EXCEPT !.groups =
                [@ EXCEPT ![gid].members[rejoiner] = @ + 1]]
          IN  ∧ replicas′ = [replicas EXCEPT ![rid]   = newReplica]
              ∧ UNCHANGED actionCounter


    ----------------------------
    Merge action helpers
    ----------------------------

MergeExpense(expOwn, expOther) ≜
```

```
381         IF expOwn = NO_EXPENSE ∧ expOther = NO_EXPENSE
382          THEN NO_EXPENSE
383          ELSE IF expOwn ≠ NO_EXPENSE ∧ expOther = NO_EXPENSE
384          THEN expOwn
385          ELSE IF expOwn = NO_EXPENSE ∧ expOther ≠ NO_EXPENSE
386          THEN expOther
387          ELSE IF expOwn.version > expOther.version
388          THEN expOwn
389          ELSE IF expOwn.version < expOther.version
390          THEN expOther
391          ELSE
392           LET mergedAcknowledgedShares ≜
393                 [u ∈ USERS ↦
394                   expOwn.acknowledged_shares[u] ∨ expOther.acknowledged_shares[u]]
395           IN  [expOwn EXCEPT !.acknowledged_shares = mergedAcknowledgedShares]

398 MergeGroup(grpOwn, grpOther, mergedExpenses, gid) ≜
399         IF grpOwn = NO_GROUP ∧ grpOther = NO_GROUP
400          THEN NO_GROUP
401          ELSE IF grpOwn ≠ NO_GROUP ∧ grpOther = NO_GROUP
402          THEN grpOwn
403          ELSE IF grpOwn = NO_GROUP ∧ grpOther ≠ NO_GROUP
404          THEN grpOther
405          ELSE
406           LET mergedMembers ≜
407                 [u ∈ USERS ↦ CHOOSE n ∈ {grpOwn.members[u], grpOther.members[u]} :
408                                 n ≥ grpOwn.members[u] ∧ n ≥ grpOther.members[u]]
409               mergedGroup ≜ [grpOwn EXCEPT !.members = mergedMembers]
410               balances ≜ ComputeBalances(mergedGroup, mergedExpenses)
411           IN  ComputeGifts(mergedGroup, balances)

413 ----------------------------
414 Merge action
415 ----------------------------

417 MergeReplicas ≜
418  ∃ ownRid, otherRid ∈ POSSIBLE_REPLICA_IDs :
419    ∧ ownRid ≠ otherRid
420    ∧ LET
421        mergedExpenses ≜
422          [eid ∈ POSSIBLE_EXPENSE_IDs ↦
423             MergeExpense(replicas[ownRid].recordedExpenses[eid],
424                          replicas[otherRid].recordedExpenses[eid])]

426        mergedGroups ≜
427          [gid ∈ POSSIBLE_GROUP_IDs ↦
428             MergeGroup(replicas[ownRid].groups[gid],
429                        replicas[otherRid].groups[gid],
430                        mergedExpenses,
431                        gid)]

433        newReplica ≜
434          [replicas[ownRid] EXCEPT
```

```
                !.groups = mergedGroups,
                !.recordedExpenses = mergedExpenses]
        IN    replicas' = [replicas EXCEPT ![ownRid] = newReplica]
              ∧ UNCHANGED actionCounter

-----------------------------
Next relation
-----------------------------
Next ≜
    ∨ CreateGroup
    ∨ AddMember
    ∨ LeaveGroup
    ∨ CreateExpense
    ∨ AddExpenseToGroup
    ∨ RemoveExpenseFromGroup
    ∨ ModifyExpenseParameters
    ∨ DeleteExpense
    ∨ AcknowledgeShare
    ∨ PayerAbsorbsLeftMemberShare
    ∨ RejoinToAcknowledge
    ∨ MergeReplicas
    ∨ UNCHANGED ⟨replicas, actionCounter⟩


-----------------------------
Invariants
-----------------------------
TypeOK ≜
  ∀ rid ∈ POSSIBLE_REPLICA_IDs :
    ∧ replicas[rid].recordedExpenses
        ∈ [POSSIBLE_EXPENSE_IDs → (Expense ∪ {NO_EXPENSE})]
    ∧ replicas[rid].groups
        ∈ [POSSIBLE_GROUP_IDs → (Group ∪ {NO_GROUP})]

Inv_Conservation_of_amount ≜
  ∀ rid ∈ POSSIBLE_REPLICA_IDs :
    ∀ eid ∈ POSSIBLE_EXPENSE_IDs :
      replicas[rid].recordedExpenses[eid] ≠ NO_EXPENSE ⇒
        LET e ≜ replicas[rid].recordedExpenses[eid]
        IN    e.amount = SumFunction(e.shares)

Inv_ExpenseGroupExists ≜
  ∀ rid ∈ POSSIBLE_REPLICA_IDs :
    ∀ eid ∈ POSSIBLE_EXPENSE_IDs :
      ∧ replicas[rid].recordedExpenses[eid] ≠ NO_EXPENSE
      ∧ replicas[rid].recordedExpenses[eid].group ≠ NO_GROUP
      ⇒
        replicas[rid].groups[replicas[rid].recordedExpenses[eid].group] ≠ NO_GROUP


Inv_GroupBalanceZero ≜
  ∀ rid ∈ POSSIBLE_REPLICA_IDs :
    ∀ gid ∈ POSSIBLE_GROUP_IDs :
```

```
        replicas[rid].groups[gid] ≠ NO_GROUP ⇒
          LET allUsers ≜
                include every user that was a member of the group at some part,
                as they always could accumulate negative balances
              {u ∈ USERS : WasEverMember(replicas[rid].groups[gid].members[u])}
            total ≜
              SumFunction([u ∈ allUsers ↦ Balance(u, gid, replicas[rid])])
          IN  total + replicas[rid].groups[gid].totalGifted = 0


Inv ≜
  ∧ TypeOK
  ∧ Inv_Conservation_of_amount
  ∧ Inv_ExpenseGroupExists
  ∧ Inv_GroupBalanceZero


  ------------------------
  Liveness Helper
  ------------------------
AllReplicasHaveAtLeastExpenseVersion(eid, version) ≜
    ∀ rid ∈ POSSIBLE_REPLICA_IDs :
        ∧ replicas[rid].recordedExpenses[eid] ≠ NO_EXPENSE
        ∧ replicas[rid].recordedExpenses[eid].version ≥ version

AllReplicasHaveAtLeastGroupMemberCounter(gid, user, counter) ≜
    ∀ rid ∈ POSSIBLE_REPLICA_IDs :
       ∧ replicas[rid].groups[gid] ≠ NO_GROUP
       ∧ replicas[rid].groups[gid].members[user] ≥ counter

AllPositiveSharesAcknowledged(eid, replica) ≜
    ∀ u ∈ USERS :
       replica.recordedExpenses[eid].shares[u] > 0
       ⇒ replica.recordedExpenses[eid].acknowledged_shares[u] = TRUE

  ------------------------
  Liveness
  ------------------------

  Expense version captures modifications and group membership
Liveness_ExpensePropagates ≜
    ∀ rid ∈ POSSIBLE_REPLICA_IDs :
       ∀ eid ∈ POSSIBLE_EXPENSE_IDs :
          □◇(replicas[rid].recordedExpenses[eid] ≠ NO_EXPENSE
                ⇒ AllReplicasHaveAtLeastExpenseVersion(eid, replicas[rid].recordedExpenses[eid].version))


Liveness_GroupMemberShipPropagates ≜
    ∀ rid ∈ POSSIBLE_REPLICA_IDs :
       ∀ gid ∈ POSSIBLE_GROUP_IDs :
          ∀ user ∈ USERS :
             □◇(replicas[rid].groups[gid] ≠ NO_GROUP
                   ⇒ AllReplicasHaveAtLeastGroupMemberCounter(gid, user, replicas[rid].groups[gid].members[user]))

  Expenses in groups eventually resolve acknowledgment status
  Either all shares are acknowledged or the expense is deleted/removed, or all members leave
```

$$
\begin{array}{l}
\mathit{Liveness\_ExpenseSharesEventuallyAcknowledged} \triangleq \\
\quad \forall\, \mathit{rid} \in \mathit{POSSIBLE\_REPLICA\_IDs} : \\
\quad\quad \forall\, \mathit{eid} \in \mathit{POSSIBLE\_EXPENSE\_IDs} : \\
\quad\quad\quad \Box\Diamond(\\
\quad\quad\quad\quad \text{LET } \mathit{exp} \triangleq \mathit{replicas}[\mathit{rid}].\mathit{recordedExpenses}[\mathit{eid}] \\
\quad\quad\quad\quad \text{IN } \quad \land\, \mathit{exp} \neq \mathit{NO\_EXPENSE} \\
\quad\quad\quad\quad\quad\quad \land\, \mathit{exp.group} \neq \mathit{NO\_GROUP} \\
\quad\quad\quad\quad\quad\quad \land\, \neg \mathit{exp.deleted} \\
\quad\quad\quad\quad\quad\quad \Rightarrow (\, \lor\, \mathit{AllPositiveSharesAcknowledged}(\mathit{eid},\, \mathit{replicas}[\mathit{rid}]) \\
\quad\quad\quad\quad\quad\quad\quad\quad \lor\, \mathit{exp.deleted} \\
\quad\quad\quad\quad\quad\quad\quad\quad \lor\, \mathit{exp.group} = \mathit{NO\_GROUP} \\
\quad\quad\quad\quad\quad\quad\quad\quad \lor\, \forall\, u \in \mathit{USERS} : \neg \mathit{IsMember}(\mathit{replicas}[\mathit{rid}].\mathit{groups}[\mathit{exp.group}].\mathit{members}[u])))
\end{array}
$$

Safety Helper

$$
\begin{array}{l}
\mathit{NoDecreaseExpenseVersion} \triangleq \\
\quad \forall\, \mathit{rid} \in \mathit{POSSIBLE\_REPLICA\_IDs} \ : \\
\quad \forall\, \mathit{eid} \in \mathit{POSSIBLE\_EXPENSE\_IDs} : \\
\quad\quad (\mathit{replicas}[\mathit{rid}].\mathit{recordedExpenses}[\mathit{eid}] \neq \mathit{NO\_EXPENSE}) \\
\quad\quad\quad \Rightarrow \\
\quad\quad\quad \land\, \mathit{replicas}'[\mathit{rid}].\mathit{recordedExpenses}[\mathit{eid}] \neq \mathit{NO\_EXPENSE} \\
\quad\quad\quad \land\, \mathit{replicas}'[\mathit{rid}].\mathit{recordedExpenses}[\mathit{eid}].\mathit{version} \\
\quad\quad\quad\quad\quad \geq \mathit{replicas}[\mathit{rid}].\mathit{recordedExpenses}[\mathit{eid}].\mathit{version}
\end{array}
$$

$$
\begin{array}{l}
\mathit{NoDecreaseGroupMembersCounter} \triangleq \\
\quad \forall\, \mathit{rid} \in \mathit{POSSIBLE\_REPLICA\_IDs} : \\
\quad \forall\, \mathit{gid} \in \mathit{POSSIBLE\_GROUP\_IDs} : \\
\quad \forall\, u \in \mathit{USERS} : \\
\quad\quad (\mathit{replicas}[\mathit{rid}].\mathit{groups}[\mathit{gid}] \neq \mathit{NO\_GROUP}) \\
\quad\quad\quad \Rightarrow \\
\quad\quad\quad \land\, \mathit{replicas}'[\mathit{rid}].\mathit{groups}[\mathit{gid}] \neq \mathit{NO\_GROUP} \\
\quad\quad\quad \land\, \mathit{replicas}'[\mathit{rid}].\mathit{groups}[\mathit{gid}].\mathit{members}[u] \\
\quad\quad\quad\quad\quad \geq \mathit{replicas}[\mathit{rid}].\mathit{groups}[\mathit{gid}].\mathit{members}[u]
\end{array}
$$

$$
\begin{array}{l}
\mathit{NoDecreaseAcknowledgedSharesSameVersion} \triangleq \\
\quad \forall\, \mathit{rid} \in \mathit{POSSIBLE\_REPLICA\_IDs} \ : \\
\quad \forall\, \mathit{eid} \in \mathit{POSSIBLE\_EXPENSE\_IDs} : \\
\quad \forall\, u \in \mathit{USERS} : \\
\quad\quad \land\, \mathit{replicas}[\mathit{rid}].\mathit{recordedExpenses}[\mathit{eid}] \neq \mathit{NO\_EXPENSE} \\
\quad\quad\quad \land\, \mathit{replicas}'[\mathit{rid}].\mathit{recordedExpenses}[\mathit{eid}] \neq \mathit{NO\_EXPENSE} \text{ this always exists by } \mathit{NoDecreaseExpenseVersion} \text{ property} \\
\quad\quad \land\, \mathit{replicas}[\mathit{rid}].\mathit{recordedExpenses}[\mathit{eid}].\mathit{version} = \mathit{replicas}'[\mathit{rid}].\mathit{recordedExpenses}[\mathit{eid}].\mathit{version} \\
\quad\quad \land\, \mathit{replicas}[\mathit{rid}].\mathit{recordedExpenses}[\mathit{eid}].\mathit{acknowledged\_shares}[u] = \text{TRUE} \\
\quad\quad \Rightarrow \\
\quad\quad\quad \mathit{replicas}'[\mathit{rid}].\mathit{recordedExpenses}[\mathit{eid}].\mathit{acknowledged\_shares}[u] = \text{TRUE}
\end{array}
$$

Safety

$$
\begin{array}{l}
\mathit{Safety\_ExpenseVersionsNonDecreasing} \triangleq \\
\quad \Box[\mathit{NoDecreaseExpenseVersion}]_{\{\langle \mathit{replicas},\, \mathit{actionCounter} \rangle\}}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Safety\_GroupMembersCounterNonDecreasing} \triangleq \\
\quad \Box[\mathit{NoDecreaseGroupMembersCounter}]_{\{\langle \mathit{replicas},\, \mathit{actionCounter} \rangle\}}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Safety\_AcknowledgedSharesNonDecreasingForSameVersion} \triangleq \\
\quad \Box[\mathit{NoDecreaseAcknowledgedSharesSameVersion}]_{\{\langle \mathit{replicas},\, \mathit{actionCounter} \rangle\}}
\end{array}
$$

607 ----------

608 Specification

609 ----------

610 $Spec \triangleq Init \land \Box[Next]_{\langle replicas,\ actionCounter\rangle}$

612 $FairSpec \triangleq$

613 $\land\ Spec$

614 $\land\ \mathrm{WF}_{\langle replicas,\ actionCounter\rangle}(MergeReplicas)$

615 $\land\ \mathrm{WF}_{\langle replicas,\ actionCounter\rangle}(AcknowledgeShare)$

616 $\land\ \mathrm{WF}_{\langle replicas,\ actionCounter\rangle}(PayerAbsorbsLeftMemberShare)$

617 $\land\ \mathrm{WF}_{\langle replicas,\ actionCounter\rangle}(RejoinToAcknowledge)$

619 ==========

* Modification History

* Last modified Sun *Jan* 18 11:16:46 *CET* 2026 by *floyd*

* Created *Fri Oct* 24 11:14:17 *CEST* 2025 by *floyd*

B TLA+ Specification of the Invitation Logic.

This appendix contains the TLA+ specification of the group invitation logic. It includes correctness properties that ensure the intended behavior. These properties state that users can become active members only after they have been invited and have accepted the invitation, as well as CRDT-related safety properties (monotonic counters) and liveness properties ensuring that invitations and acceptances eventually reach all replicas. The specification was model checked to a depth of ten with three users on three replicas, and no correctness property violations were found.

```
MODULE SplitLess_group_join_leave
EXTENDS Naturals

CONSTANTS
    USERS,
    POSSIBLE_REPLICA_IDs,
    ASSIGNED_REPLICA,
    INITIAL_MEMBER

VARIABLES
    replicas, actionCounter

----------------------------
Records
----------------------------
Group ≜ [ members : [USERS → Nat],
          invited_members : [USERS → Nat]]

Replica ≜ [id : POSSIBLE_REPLICA_IDs,
           group : Group]


----------------------------
Initialization
----------------------------
Init ≜
    ∧ LET InitialGroup ≜
          [members ↦ [u ∈ USERS ↦ 0],
           invited_members ↦ [u ∈ USERS ↦ IF u = INITIAL_MEMBER THEN 1 ELSE 0]]
      IN  ∧ replicas = [rid ∈ POSSIBLE_REPLICA_IDs ↦ [group ↦ InitialGroup]]

    replicas = [rid ∈ POSSIBLE_REPLICA_IDs
                ↦ [id ↦ rid,
                   group ↦ INITIAL_GROUP] ]
    ∧ actionCounter = 0


----------------------------
Helper Actions
----------------------------
IsMember(memberCounter) ≜
  memberCounter%2 = 1

IsMemberContender(memberCounter) ≜
    memberCounter%2 = 1


----------------------------
Group Actions
----------------------------
InviteMember ≜
    ∃ actor, newMember ∈ USERS :
    ∃ rid ∈ POSSIBLE_REPLICA_IDs :
        ∧ ASSIGNED_REPLICA[actor] = rid
        ∧ IsMember(replicas[rid].group.members[actor])
        ∧ ¬IsMember(replicas[rid].group.members[newMember])
```

```
57          ∧ ¬IsMemberContender(replicas[rid].group.invited_members[newMember])
58          ∧ LET newReplica ≜
59              [replicas[rid] EXCEPT !.group.invited_members[newMember] = @ + 1]
60              IN
61                  ∧ replicas′ = [replicas EXCEPT ![rid] = newReplica]
62                  ∧ actionCounter′ = actionCounter + 1

64  AcceptInvitation ≜
65      ∃ actor ∈ USERS :
66      ∃ rid ∈ POSSIBLE_REPLICA_IDs :
67          ∧ ASSIGNED_REPLICA[actor] = rid
68          ∧ IsMemberContender(replicas[rid].group.invited_members[actor])
69          ∧ LET newReplica ≜
70              [replicas[rid] EXCEPT
71                  !.group.invited_members[actor] = @ + 1,
72                  !.group.members[actor] = @ + 1]
73              IN
74                  ∧ replicas′ = [replicas EXCEPT ![rid] = newReplica]
75                  ∧ actionCounter′ = actionCounter + 1

77  LeaveGroup ≜
78      ∃ actor ∈ USERS :
79      ∃ rid ∈ POSSIBLE_REPLICA_IDs :
80          ∧ ASSIGNED_REPLICA[actor] = rid
81          ∧ IsMember(replicas[rid].group.members[actor])
82          ∧ LET newReplica ≜
83              [replicas[rid] EXCEPT !.group.members[actor] = @ + 1]
84              IN
85                  ∧ replicas′ = [replicas EXCEPT ![rid] = newReplica]
86                  ∧ actionCounter′ = actionCounter + 1


89  ----------------------------
90  Merge action
91  ----------------------------
92  MergeReplicas ≜
93      ∃ ownRid, otherRid ∈ POSSIBLE_REPLICA_IDs :
94          ∧ ownRid ≠ otherRid
95          ∧ LET
96              merged_members ≜
97                  [u ∈ USERS ↦
98                    CHOOSE n ∈ {replicas[ownRid].group.members[u], replicas[otherRid].group.members[u]} :
99                              n ≥ replicas[ownRid].group.members[u] ∧ n ≥ replicas[otherRid].group.members[u]]
100             merged_persumed_membrs ≜
101                 [u ∈ USERS ↦
102                   CHOOSE n ∈ {replicas[ownRid].group.invited_members[u], replicas[otherRid].group.invited_members[u]} :
103                             n ≥ replicas[ownRid].group.invited_members[u] ∧ n ≥ replicas[otherRid].group.invited_members[u]]
104             merged_group ≜
105                 [replicas[ownRid].group EXCEPT !.members = merged_members,
106                                                !.invited_members = merged_persumed_membrs]
107           IN    ∧ replicas′ = [replicas EXCEPT ![ownRid].group = merged_group]
108                 ∧ UNCHANGED actionCounter
```

```
111  ----------------------------
112  Next relation
113  ----------------------------
114  Next ≜
115      ∨ InviteMember
116      ∨ AcceptInvitation
117      ∨ LeaveGroup
118      ∨ MergeReplicas
119      ∨ UNCHANGED ⟨replicas, actionCounter⟩


122  ----------------------------
123  Invariants
124  ----------------------------
125  TypeOK ≜
126    ∀ rid ∈ POSSIBLE_REPLICA_IDs :
127      ∧ replicas[rid].group
128            ∈ Group

130  ----------------------------
131  Liveness Helper
132  ----------------------------
133  AllReplicasHaveAtLeastGroupMemberCounter(user, member_counter) ≜
134      ∀ rid ∈ POSSIBLE_REPLICA_IDs :
135          ∧ replicas[rid].group.members[user] ≥ member_counter

137  AllReplicasHaveAtLeastGroupPersumedMemberCounter(user, persumend_member_counter) ≜
138      ∀ rid ∈ POSSIBLE_REPLICA_IDs :
139          ∧ replicas[rid].group.invited_members[user] ≥ persumend_member_counter

141  ----------------------------
142  Liveness
143  ----------------------------
144  Liveness_GroupMemberShipPropagates ≜
145      ∀ rid ∈ POSSIBLE_REPLICA_IDs :
146          ∀ user ∈ USERS :
147                □◇AllReplicasHaveAtLeastGroupMemberCounter(user, replicas[rid].group.members[user])

149  Liveness_PersumedGroupMemberShipPropagates ≜
150      ∀ rid ∈ POSSIBLE_REPLICA_IDs :
151          ∀ user ∈ USERS :
152                □◇AllReplicasHaveAtLeastGroupPersumedMemberCounter(user, replicas[rid].group.invited_members[user])


    Safety Helper
158  MemberOnlyAfterInvitation ≜
159    ∀ rid ∈ POSSIBLE_REPLICA_IDs :
160    ∀ u ∈ USERS :
161      ∧ ASSIGNED_REPLICA[u] = rid
162      ∧ ¬IsMember(replicas[rid].group.members[u])
163      ∧ IsMember(replicas′[rid].group.members[u])
164      ⇒
165        ∧ IsMemberContender(replicas[rid].group.invited_members[u])

167  NoDecreaseMembershipCounters ≜
```

$\forall\, rid \in POSSIBLE\_REPLICA\_IDs :$
$\forall\, u \in USERS :$
$replicas'[rid].group.members[u]$
$\geq replicas[rid].group.members[u]$
$\land\ replicas'[rid].group.invited\_members[u]$
$\geq replicas[rid].group.invited\_members[u]$

Safety

$Safety\_MemberOnlyAfterInvitation \triangleq$
$\Box[MemberOnlyAfterInvitation]_{\langle replicas,\, actionCounter\rangle}$

$Safety\_NoDecreaseMembershipCounters \triangleq$
$\Box[NoDecreaseMembershipCounters]_{\langle replicas,\, actionCounter\rangle}$

Specification

$Spec \triangleq Init \land \Box[Next]_{\langle replicas,\, actionCounter\rangle}$
$FairSpec \triangleq Spec \land \mathrm{WF}_{\langle replicas,\, actionCounter\rangle}(MergeReplicas)$

* Modification History
* Last modified *Thu Jan* 15 10:42:22 *CET* 2026 by *floyd*
* Created *Mon Nov* 24 10:30:46 *CET* 2025 by *floyd*

C Output of the Probabilistic Testing of the Python Implementation

This appendix presents the output of the probabilistic testing of the Python implementation. It reports the total execution time of the tests (aggregated over eight CPU cores) as well as the average runtime per test run. For each action, we report the total number of times it was triggered, together with its failure rate (at least one precondition prevents the action from changing the system state) and its success rate (all preconditions are satisfied and the action results in a state change).

```
Average Results over 80028 test runs with temporal checks: True
======================================================================

 Summary:
  Average expenses created: 77.91
  Average groups created:   14.80

 Timing:
  Total time:   35H_28M_16S
  Average/test: 1.596s

 Action Results (Total across all runs):
```

Action	Passed	Failed	Total	Pass Rate
accept	5012245174	7334904615	12347149789	40.6%
acknowledge_share	8179992131	68595790319	76775782450	10.7%
add_expense_to_group	14796715374	46836644730	61633360104	24.0%
delete_expense	5343829776	23650345664	28994175440	18.4%
edit_expense	4182667463	25139974476	29322641939	14.3%
invite	5688001577	18061981880	23749983457	23.9%
leave	3962291040	12593533774	16555824814	23.9%
merge	80064995314	0	80064995314	100.0%
payer_absorbs_left_member	285040520	9215591594	9500632114	3.0%
remove_expense_from_group	3727908500	4212059655	7939968155	47.0%

```
======================================================================
 All test runs finished.
======================================================================
```

Disclosure of AI Use

AI tools were used to assist in the preparation of this report. All AI usage is purely of assistive nature where no new content was generated. AI was used in two distinct ways:

1. **LaTeX formatting:** ChatGPT-5 was used to assist with LaTeX-specific formatting tasks, such as the use of LaTeX commands, environments, and document layout elements (e.g., title page and formatting boilerplate). It was not used to design or influence the conceptual, logical, or organizational structure of the report's content, nor to generate any technical material, results, or arguments.

2. **Language assistance:** ChatGPT-5 was used to assist with English phrasing and stylistic improvements throughout the report. No new content was created by the AI tool. All ideas, structure, and content originate from me (with feedback and discussions with Dr. Erick Lavoie). The AI tool was used exclusively to improve clarity, flow, and idiomatic English expression. All AI-generated suggestions were carefully reviewed, edited where necessary, and compared against the original text. We take full responsibility for the final wording, accuracy, and content of the document.